Introduction

Les applications dites d’entreprise commencaient à se complexifier de plus en plus donc Rod Johnson un development australien créé en 2004 le framework Spring et cofound la société spring source

-> l’idée pour lui c’est de standardiser une façon de programmer

-> quelque chose de bas niveau

Au début l’idée est mal percu surtout due a la config xml importante sur un projet spring

Il a intégré les nouveautés du monde java (Web services, rest, no SQL cloud computing..)

Pourtant il va sagit d’une révolution sans précédent, avec le temps spring a concurrencer de plus en plus ouvertement JAVA EE sur plusieurs points car toujours 2-3 ans en avance par rapport à JEE.

Aujourd’hui incontournable ! Et 10 ans après la création de spring, autre révolution, spring boot !

-> pareil au début vu comme gadget, aujourd’hui utilisé dans plus 60% des technologies qui utilisent java

Pourquoi?

Car springboot facilite les dépoloiements des applications, et surtout sur le cloud

Si vous avez trouvé le code java compliqué et vous avez du mal a construire vos architecture?

Bonne nouvelle car spring va rendre votre code beau !

———————————————————————————————————————————

———————————————————————————————————————————

OBJECTIF DE CE COURS : ON VA CRÉER UNE APPLICATION D’ENTREPRISE

Avec une application d’entreprise on va avoir besoin de pas mal de chose classique, des vues, un BDD etc..

Imaginez un outil qui vous crééer comme par magie toutes les réponses à ces besoins !

-> C’EST LE ROLE DE SPRING INITIALIZER https://start.spring.io/

Il va vous aider à définir ce que vous voulez.

Pour construire une application on a besoin de dizaines de librairies, spring initializer va vous proposer des lots de librairies selon vos besoin appelés ‘SPRING BOOT STARTERS’.

SI ON VEUT UNE API REST, ON A UN STARTER POUR ÇA !

////

/// DÉFINITION API REST

////

Une API REST (également appelée API RESTful) est une interface de programmation d'application (API ou API web) qui respecte les contraintes du style d'architecture REST et permet d'interagir avec les services web RESTful. L'architecture REST (Representational State Transfer) a été créée par l'informaticien Roy Fielding.

Une API est un ensemble de définitions et de protocoles qui facilite la création et l'intégration de logiciels d'applications. Elle est parfois considérée comme un contrat entre un fournisseur d'informations et un utilisateur d'informations, qui permet de définir le contenu demandé au consommateur (l'appel) et le contenu demandé au producteur (la réponse). Par exemple, l'API conçue pour un service de météo peut demander à l'utilisateur de fournir un code postal et au producteur de renvoyer une réponse en deux parties : la première concernant la température maximale et la seconde la température minimale.

En d'autres termes, lorsque vous souhaitez interagir avec un ordinateur ou un système pour récupérer des informations ou exécuter une fonction, une API permet d'indiquer au système ce que vous attendez de lui, afin qu'il puisse comprendre votre demande et y répondre.

// DÉMONSTRATION

Sur le site https://start.spring.io/, montrer que si l’on fait add Dependencies, tu tapes web et on a un lot de librairies qui nous sont proposés pour créer une application avec SPRING MVC par exemple

// pour créer notre application on va donc sélectionner plusieurs starters

* Spring WEB
* Spring Data JPA avec Hibernate pour gérer les transactions de la BDD
* H2 Database pour faciliter la mise en place de la BDD

On aurait pu évidement prendre d’autres choses :

* si on avait besoin de security on aurait pu prendre SPRING SECURITY
* De faire tourner des taches planifier (STARTER QUARTZ SCHEDULER)
* De sockets on a un strarter pour ça
* De déployer tout ça dans un cloud genre azur ou aws j’ai ça aussi

/// CONFIGURATION

On peut également choisir de configurer le projet avec Maven ou Gradle nous on va rester sur Maven.

On va sélectionner la version 2.7.0

On utilise JAVA dans la version 11

-> options

-> java 11

Comme c’est MAVEN on peut spécifier le triplet maven : com.mycompany.formationspring

///

/// GENERATE APP

///

Une fois la config effectuée on appuie sur le bouton generate !

-> cela vous génère un zip

CELA VOUS A GÉNÉRER UNE APPLICATION COMPLETEMENT FONCTIONNEL AVEC LES BESOINS EXPRIMES AUPARAVANT

Evidément si on veut créer une applicaiton qui controle les train de la Sncf il reste encore à la coder !

Spring Boot apporte également la capacité de rendre executable une application qui utilise le framework spring.

On va pouvoir démarrer l’application avec springBoot.

//

// Lancer l’application télécharger avec SpringBoot

//

Aller dans votre ligne de commande

Se positionner à l’endroit ou est votre application

La déziper

Si votre JAVA\_HOME N’EST PAS CONFIGURER IL VA FALLOIR LE FAIRE le configurer à la volée :

* sur PC : set JAVA\_HOME = C:\Program Files\Java\jdk-11/0/2

- Pour faire la config sur Mac et s’assurer que tout va bien :

-> vim ~/.zshrc

-> echo $JAVA\_HOME (resultat : /Library/Java/JavaVirtualMachines/jdk-17.0.1.jdk/Contents/Home)

//

// Config en plus sur Mac

//

brew install maven

La commande sur Mac pour lancer le projet : mvn spring-boot:run

Ensuite vous lancer la commande suivante : mvnw spring-boot:run (en étant positionné dans le dossier demo)

Quand on lance l’application maven va télécharger et installer pas mal de chose

mais on voit qu’à la fin c’est écris Tomcat started on port 8080

L’application est donc disponible sur localhost:8080

//

// http://localhost:8080/

//

Quand vous arrivez sur <http://localhost:8080/> on voit qu’on a une sorte d’erreur qui s’affiche car dans notre projet on a rien

//

// Hello World

//

On arrêter le serveur (ctrl+c) on a va ajouter une première page dans notre projet avec un hello world.

Src > main > java > ressources > static > index.html

Dans le dossier static on va avoir les contenus web

//// essayons maintenant d’ajouter des fonctionnalités à notre application

//

// PREMIER FONCTIONNEL

//

Pour l’instant notre application affiche juste un hello world

Essayons d’afficher quelque chose de fonctionnel

///

/// Démonstrattion 1

///

Afficher par exemple une liste de contacts en faisant appel à un web service rest

Pour cela donner à la classe le dossier ziper source-files.zip

- Copier coller le fichier index.html, .js dans le dossier ressources > static de notre projet démo (remplacer donc le fichier.html)

* un cran au dessus dans le dossier ressources on va y glisser data.sql et le fichier application.properties (faudra donc remplacer data.sql)
* Dans le repertoire java > com > mycompany > demo copier coller le dossier controller/entity et repository

Une fois les fichiers/dossiers copiés dans notre projet démo

Arrêter le serveur

Recompiler le projet

//

// Rebuilder le projet

//

Mvnw clean install depuis le dossier démo

L’appliation a été rebuildé

Je relance l’application

Si on raffraichis la page, on obtient une liste de contact

//

// Ais-je rééllement affaire à une page dynamique et le contenu n’est pas en dure? démonstration

//

Clique droit inspecter

Aller dans Network

Cocher XHR

Raffraichir la page index.html

On verra à un moment que l’url /list a été appelé et qu’un format Json a été retourné

-> on a donc contacté le serveur pour récupérer des données !

//

// IntelliJ

//

Bonne nouvelle pour vous, on va utiliser IntelliJ

Une fois le projet chargé dans IntelliJ, on va lancer l’application

//

// Run the application from IntelliJ

//

Src > main > java > com > mycompany > formationspring > DemoApplication.java

//

// Si jamais il ne comprends pas tout de suite comment runner l’application : CONFIGURATION

//

EDIT RUN CONFIGURATION

NEW RUN CONFIGURATION

MAVEN

COMMANDE LINE : spring-boot:run

OK

-> maintenant vous pouvez runner l’application !

//

// Des questions ? Je vous comprends !

//

Pour ceux qu’on suivit le cours JEE, vous allez vous demander : mais ou est mon serveur applicatif?

COmment on accède à ma BDD, comment on configure tout ça?

C:’est plus nécessaire ou quoi? Si biensur, mais spring va faire plein de choses pour nous !

On va voir maintenant comment tout ça fonctionne et comment créer nous même ces différents projets/fichiers

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

**1ERE PARTIE - POURQUOI A-TON BESOIN DE SPRING?**

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

L’idée ici c’est que l’on va se mettre dans l’idée de créer une application pour différents clients, soulever des problématiques, voir comment avec nos connaissances en objet on peut résoudre ces problématiques, en comprendre les limites et du coup comprendre pourquoi Spring est apparu et comment il répond de manière simple à ces problématiques.

Expliquer aux élèves qu’on pourrait 3 clients :

* un client de l’automobile, qui veut gérer ses factures en console, les facture vont de 1 à x.
* Un client d’un supermarché qui veut gérer ses factures via une page web, et ses factures sont préfixes de INV\_\* pour chaque facture
* Un client qui vend de la peinture, et lui il a besoin d’une page web, mais il génère ses factures avec une douchette (un bip comme dans les magasins)

L’idée générale ça va être de montrer comment répondre à ces besoins en essayant de centraliser le code et ça va s’avérer compliqué sans utiliser spring car le code de base n’est pas flexible.

Pour infos :

Les classes InvoiceController, InvoiceRepository et InvoiceService vont représenter le premier client.

Les classes InvoiceControllerNewClient, InvoiceRepositoryNewClient, et InvoiceServiceNewClient le deuxième client etc…

//// ILLUSTRATION

Illustrons les problèmes soulevés et que solve Spring

Imaginons que l’on est un logiciel de facuration et que l’on veut implémenter la fonctionnalité de création de facture

//

// créer nouveau projet : CORE

//

Créer un projet depuis IntelliJ

- New project

- Maven

- Sélectionner la JDK 17

- Cocher create froject from archetype

- Chercher org.apache.maven.archetype.maven.archetype.quickstart

- Appeler le project core

- Group id : com.mycompany.invoice

* Next > next (vous pouvez choisir ou stocker le projet)

///// Configurer la bonne version de la JDK dans le projet

Dans le pom.xml remettre la version 17

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>17</maven.compiler.source>

<maven.compiler.target>17</maven.compiler.target>

</properties>

///////// Vérifier la version de votre JDK dans IntelliJ :

File > Settings > Build execution deployement Compiler > Java Compiler version la version 17 \*

(Target bytecode version Selectionner 17)

Également dans :

File > Project Structure > Project > version17

// Pour Mac c’est dans preferences\*

//

// Création de l’entité metier

//

Dans com.company.invoice créer un package entity avec une classe Invoice

//

// Création du service

//

Dans com.company.invoice créer un package service avec une classe InvoiceService

// l’idée générale

-> stocker la nouvelle facture quelque part

-> on va utiliser le dévelopement rapide en mettant les données en mémoire

-> on verra comment mettre les données en mémoires

-> on va quand même créer un système de persistance

//

// Création du repository

//

Dans com.company.invoice créer un package repository avec une classe InvoiceRepository

L’idée c’est que plutôt que d’avoir une BDD opérationnelle on va créer un ArrayList static dans cette classe on aura une instance de la liste et on l’alimentera à chaque fois.

//

// Alimentation de la classe principale App

//

Dans cette classe on va demander à l’utilisateur le nom du client

Et on va générer notre première facture pour cela, clique droit sur le fichier App dans com.company.invoice et faire Run (avant on peut faire un petit clean and build avec le petit marteau au dessus)

///

/// Prototype créé

///

On a créé un prototype mais qui soulève pas mal de problème !

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 1 - Movie > Core (part 1)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Donner aux élèves l’exercice et corriger ensemble

////////

//////// Le problème de la flexibilité des applications (Revenir au première module core créé)

////////

Note : App1 représente l’évolution des services

App2 représente le fait qu’un client veut gérer ses factures en ligne de commande et l’autre avec un formulaire

L’idée générale ici est d’expliquer, que si on développe une application de base elle est très rigide et ne va pas correspondre à toutes les demandes de nos clients, comment la rendre flexible?

Une application n’est pas naturellement flexible on va essayer de répondre à ce problème en donnant des examples :

Comme on l’a vu, on a des factures qui sont énumérés de 1 à X de manière croissante, et tout est gérée en ligne de commande.

Que se passe-til si demain un client nous dis, moi je veux pas gérer ça en ligne de commande mais je veux gérer ça via un formulaire, et moi mes factures elles commencent par INV\_

On va devoir dupliquer notre service InfoService et gérer le nommage de nos fichiers différements

Dans App on va devoir demander dans quelle configuration on est pour savoir quel service appelé.

Si il veut une interface graphique il va falloir créer un controlleur qui va gérer les saisies

On va devoir ensuite un controller responsable de récupérer le saisies dans la vue (créer un package controller)

InvoiceController représentera le traitement classique en ligne de console

InvoiceControllerNewClient une interface web graphique permettant une saisie via un formulaire

Imaginons maintenant qu’on a un client qui a applications et l’une doit gérer ça en console et l’autre avec une interface graphique, ses factures commencent bien par 1 mais préfixer par l’année en cours 2022\_1….

En plus de cela on aimerait stocker tout ça dans une base de données, pour cela faudra peut-être mettre en place une structure JDBC….

On voit bien que de nature notre application n’est pas flexible et que l’on rencontre rapidement de rééls soucis, Spring va répondre à toute ces problématiques.

Pour gérer l’application de facturation avec une BDD faudra dupliquer InvoiceRepository et mettrre du code JDBC

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 2 - Movie > Core (core part 2)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Donner aux élèves l’exercice et corriger ensemble

/////

///// PROGRAMMATION PAR CONTRAT (démonstrations > core part 2)

/////

Notre application n’est pas flexible. Pk?

Partout ou un composant fait appel à un autre composant, comme par exemple quand on instancie implicitement dans notre controller une dépendance vers notre service qui lui même est dépendant de notre repository.

Pour résoudre ce problème on va utiliser un design pattern, (modèle de conception) que l’on appel la programmation par contrat.

-> pour relacher les contraintes entre les composants via des interfaces ou classes abstraites

-> on passe d’un couplage fort à un couplage faible

-> ces interfaces/classe abstraites disent comment le composant du haut va communiquer avec le composant du bas.

Entre notre service et nos repository on va créer une interface.

//

// CRÉATION DE NOS INTERFACE (Démonstrations core part 2)

//

InvoiceRepositoryInterface dans le package repository

InvoiceControllerInterface dans le package controller

L’objectif c’est que nos deux controllers implémentent InvoiceControllerInterface donc il va falloir qu’il fasse appel à la même méthodes (la renommer)

Ce qu’il faut faire :

* Dans InvoiceControllerNewClient et InvoiceController, implémenter la nouvelle interface InvoiceControllerInterface, avec le même nom de méthode
* Dans InvoiceRepository et InvoiceRepositoryNewClient implémenter InvoiceRepositoryInterface
* Dans InvoiceService et InvoiceServiceController implémenter InvoiceServiceInterface

-> désormais je peux référence à des interfaces mais plutôt à des interfaces

Dans inVoiceServiceNewClient ne plus faire référence à InvoiceRepositoryNewClient mais à l’interface InvoiceRepositoryInterface, et même principe pour la partie controller.

D’une manière général, toutes les références aux classes directes doivent être enlevé !

Bon tout ça c’est bien beau mais le code ne fonctionne plus, va falloir réparer ce qu’il se passe dans la classe principale App.

Pk? Car pour l’instant on utilise pas les implémentations mises en place

Montrer que si je lance le code j’ai un null pointer exception

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 3 - Movie > Core (core part 3)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Appliquer à notre projet fil tout ce qu’on avons fait ensemble dans le projet core de démonstrations

/////

///// INJECTION DE DÉPENDANCES (Démonstrations > Core (core part 4))

/////

L’idée ici c’est que on déclare dans nos classes, on a déclaré nos interfaces mais elles sont null, on va devoir passer par des getter/setters pour les instancier, cela s’appel de l’injection de dépendance, on va avoir besoin d’injecter les liens vers ces interfaces pour pouvoir les utiliser, et du coup on va devoir le faire à la Mano, via des getters/setters.

Partout où il y a une référence à une interface dans une classe, mettre en place des getters/setters.

Au niveau de App, on va ensuite surtout utiliser les setters mis en place pour setter des valeurs à interface encapsulées dans nos classes.

Pour générer les getters et les setters (click droit + generate + getters/setters).

Une fois que c’est fait, je vais gérer à la Mano la gestion des dépendances dans App.

Peut importe le scénario, le controller à besoin du service, donc je set le Service dans le controller pour le scénario 1.

Ensuite le service à besoin du repository, donc je set le repository dans le service.

De la le controller peut créer son invoice.

Je répète les mêmes injections de dépendances pour le scénario 2.

/////

///// Revenons sur notre toisième client et gérons le fait que lui veut générer des factures avec une douchette

/////

Il va nous falloir un nouveau controller !

-> créer InvoiceControllerThirdClient et appeler les méthodes qu’il faut

Ici la configuration pour ces troisième client est la suivante :

Il utilise son propre controller car scanne ses factures

Pour la génération des factures, c’est le même que le premier client car ce sont des factures de 1 en 1

Pour la BDD c’est le même repository aussi que le client 1

-> gérer donc c’est partie ce nouveau scénario dans App

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 4 - Movie > Core (core part 4)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

/////

///// Inversion de contrôle (core part 4)

/////

Et si on se concentrer sur les interfaces à implémenter et non pas aux clients?

Pour cela va falloir renomer pas mal de chose !

InvoiceController -> InvoiceControllerKeyBoard

InvoiceControllerNewClient -> InvoiceControllerWeb

InvoiceControllerThirdClient -> InvoiceControllerScanner

InvoiceRepositoryInMemory -> InvoiceRepository

InvoiceRepositoryNewClient -> InvoiceRepositoryDataBase

InvoiceService -> InvoiceServiceNumber

InvoiceServiceNewClient -> InvoiceServicePrefix

Dans le App on va aussi opérer des changements

On va poser trois questions : (voir App)

Va falloir maintenant ajouter aux interface InvoiceControllerInterface et InvoiceServiceInterface les méthodes pour setter le service et le repository.

On fait ça car maintenant on passe par des interfaces donc on veut avoir accès aux méthodes qui permettent de setter les services et repository.

Rester l’applicationa avec la nouvelle config

Autre solution ?

//

// Réflexivité

//

On pourrait utiliser les principes de reflexivité !

(Spring utilise ces stratégies, nous on le fait à la main pour expliquer son fonctionnement)

Cela correspond au fait d’instancier un objet sur la bas d’un nom de classe, voir App.

Donc plus besoin de switch case !

On va donc demander à l’internaute quelles sont les classes qu’il souhaite utiliser?

Pour taper le nom de classe du controller par exemple, on va taper :

com.mycompany.invoice.controller.InvoiceControllerWeb

com.mycompany.invoice.service.InvoiceServiceNumber

com.mycompany.invoice.repository.InvoiceRepositoryDataBase

Conclusion :

Si demain je créé un nouveau repository, service ou controller app ne change pas.

Le fait d’avoir un endroit centralisé du code qui se charge d’instancier les composants de l’architecture et de les mettre en relation grâce à l’injection de dépendance et bien cela s’appel l’inversion de contrôle ou EOC (Inversion of Control)

C’est l’un des pilliers de spring et on va voir comment il le met en oeuvre

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 5 - Movie > Core (core part 4)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

**2EME PARTIE - LES FONDAMENTAUX DU FRAMEWORK SPRING**

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//

// CONTENEUR LEGER (Démonstrations > core part 5)

//

Bon dans la vraie vie on aurait pas intérêt à demandé à l’utilisateur de rentrer manuellement les classes à invoquer, cela n’aurait pas de sens.

Et puis soyons honnête, la config d’un projet ne va pas non plus changer tous les 4 matins.

Pour gérer les différentes instanciations des classes, on pourrait plutôt créer une fichier de configuration dans lequel on mettrait les classes à instancier, et ce fichier varierait et serait configuré en fonction du client !

On le lierait, et il instancierai les bonnes classes.

C’est le coeur de SPRING, on appel cela le conteneur léger de spring (SPRING LIGHTWEIGHT CONTAINER)

Ce conteneur va effectuer l’inversion de contrôle et garder en mémoire tous les composants instanciées durant toute la vie de l’app et ceci sous forme de graph pour matérialiser les dépendances qui existent entre ces composants.

L’idée d’avoir un fichier de configuration c’est l’idée originelle de spring, le plus souvent les informations sont fournies au format xml.

Cette méthode est toujours valide et illustre très bien ce qu’il se passe, on va commencer par celle-ci mais il y en a d’autres.

////

//// applicationContext.xml

////

Fournir aux élèves le fichier applicationContext.xml

Copier coller ce fichier dans un dossier resources que nous n’avons pas donc faudra le créer

(attenion au nom, c’est un standard maven)

Le nom applicationContext.xml pourrait être changé mais nous allons le garder, car c’est souvent ce que l’on retrouve.

Pour le créer :

Main > resources (Attention quand tu le créer à ne mettre qu’un seul s)

Tout ce que l’on va avoir dans ressources sera disponible dans le class path de l’application

Classpath est un paramètre passé à une machine virtuelle Java qui définit le chemin d'accès au répertoire où se trouvent les classes et les packages Java afin qu'elle les exécute.

Dans le fichier applicationContext.xml, dans beans on a toutes les classes à instancier, l’ordre n’a pas d’importance lors de la déclaration.

Les déclarer.

Une fois le fichier de configuration configuré voir applicationContext.xml, la question c’est comment démarrer l’application avec ce conteneur leger?

-> il va nous falloir quelques librairies

-> donner aux élèves le fichier spring-context-pom et ajouter la dépendance à notre pom.xml

Ajouter cette dépendance à votre pom.xml

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.20</version>

</dependency>

Normalement l’IDE (IntelliJ) va te mettre à disposition un bouton pour télécharger les dépendances manques. Ce petit bouton c’est Load Maven Changes.

Dans External Libraries tu retrouves tout ce qui était indispensable pour faire tourner spring sur notre projet

Mavent va télécharger la librairie spring context et toutes les dépendances qui lui sont associées.

Tout a été téléchargé dans External Libraries, on voit que plusieurs ont été téléchargées.

// L’idée générale, c’est qu’on a plus besoin de demander à l’utilisateur quelles classes sont à instancier, donc dans App on va pouvoir tout supprimer.

//

// Utilisation du ApplicationContext

//

Une fois le fichier App mis en place (voir commentaires dans le fichier) relancer l’application tout est fonctionnel :)

Ce qui est cool une fois tout ça mis en place, c’est que si maintenant je veux utiliser InvoiceControllerScanner j’ai juste à modifier mon applicationContext.xml, mon App n’a pas changé et le code fonctionne (faire la démonstrations)

Je peux de la même manière changer le service et le repository.

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 6 - Movie > Core (core part 6)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Voir correction

////

//// Affectation de valeur à une propriété ( on reste dans Démonstrations > core (part 5)

////

Faut savoir que dans notre applicationContext.xml , la classe InvoiceServicePrefix par exemple ne sera instanciée qu’une seule.

Du coup à l’intérieur on pourrait virer les attributs statiques car de toute façon il n’y aura qu’une seule instance de cette classe;

(Voir modifications dans InvoiceServicePrefix)

L’idée ici c’est que dans InvoiceServicePrefix on avait un attribut lastNumber ou on peut faire démarrer la facture à un certain nombre.

On va également ajouter un attribut qui représentera le prefix à ajouter sur nos factures, de bases c’est INV\_ mais demain ça se trouve ce sera NUM\_;

On va donc mettre des getters et des setters pour ces deux attributs et setter leur valeur via le fichier applicationContext.xml via la balise property.

Montrer qu’on peut passer d’autre types d’objet comme un File mais que par exemple on ne pourrait pas lef aire avec un objet Date en passant la valeur 10/10/2022 car il ne serait pas a quoi correspond le mois et le jour par exemple.

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 7 - Movie > Core (toujours core part 7)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// maintenant le nom du fichier et le type du fichier à générer est paramètrable dans applicationContext.xml

//

// Autowiring byName et byType (Démonstrations > core part 6)

//

L’idée c’est que si par exemple dans ApplicationContext.xml, name et ref avait la même valeur au niveau d’une property :

<property name="repository" ref="repository"/>

Ce serait bien de pouvoir dire à Spring si t’as un name qui vaut toto ba regarde si tas pas une référence à une classe qui s’appel pareil, et ça c’est possible :)

Pour cela assurez-vous que vos id dans vos beans de applicationContext.xml ont le même nom que vos attributs de classe. (Voir core part 6)

Si c’est le cas tu peux ajouter autowire=« byName » sur tes beans, si ils suivent tous ces configuration tu peux directement l’ajouter au parent.

Si on le mets directement sur le parent et bien pour chaque Bean il va scanner les propriété et voir si j’ai un autre objet déclaré dans ce fichier qui pourrait être injecté dans cette propriété.

//

// Autowire byType (Démonstrations > core part 7)

//

Le problème c’est que l’autowire byName nous impose d’avoir une synthaxe particulière et d’avoir un id sur nos beans dans applicationContext.xml

L’autowiring peut fonctionner avec un autre mode, le type byType

C’est donc pas le nom des composants qui va être exploité mais leur type

Par exemple pour InvoiceControllerWeb, la propriété service est de type InvoiceServiceInterface.

Spring va rechercher si il existe un composant d’un type compatible et donc va trouver InvoiceServicePrefix, quelque soit l’id de l’objet injecté.

Le fichier applicationContext.xml devient alors beaucoup plus léger mais pas forcément plus simple à comprendre, car les injections de dépendances ne sont plus explicites.

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 8 - Movie > Core (toujours core part 7)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//

// Configuration par annotation (Démonstrations > core part 8)

//

Depuis Spring 2.5 on peut exploiter les annotations pour la configuration pour éviter des fichiers xml trop volumineux

Pour l’autowiring, il existe des annotations par exemple.

Dans le fichier applicationContext.xml, on peut utiliser <context:annotation-config/> on va avoir une errreur on va l’expliquer.

La balise annotation-config est précédé des deux points, cela signifie qu’elle nous vient d’un espace de nommage specifique , pas celui par defaut

Deja le context: signifie que annotation-config nous vient de l’espace de nommage context.

Utiliser ces espaces de nommage permet d’éviter des conflits quand des balises auraient le même nom pour des choses compléments différents

C’est le cas quand on utilise des librairies qui utilisent des balises de même nom.

annotation-config est exploitée via la librarie spring-context mais elle est aussi via la librairie Spring beans, donc pour préciser que cela nous vient de context on utilise cet espace de nommage.

D’ailleurs si vous regarder la balie parent beans :

- Xmlns siginifie que toutes les balises qui n’ont pas de préfix seront référés à cet espace de nommage identifié par Xmlns.

-xsi reprend le shéma que vous avez au dessus, mais y ajoute un fichier xsd qui va expliquer la grammaire de nommage, l’ensemble des balises que l’on peut utiliser et comment elle fonctionne

Comment le retrouver?

Ouvrir librairire Spring beans > MetaInf > Spring schemas > et vous retrouverez la référence à ce fichier xsd et a côté ou se trouve se fichier

https\://www.springframework.org/schema/beans/spring-beans.xsd=org/springframework/beans/factory/xml/spring-beans.xsd

Pour utiliser la balise annotation-config il va falloir dans l’entête du fichier xml dire qu’il y a un autre espace de nommage Avec d’autres balises

Copier la ligne de xmlns= et ajouter :context et préciser l’espace de nommage (voir dossier core part 8)

Et copier coller le contenu dans xsi:schemaLocation et modifier dans la copie spring-beans par spring-context (voir dossier core part 8)

Ensuite votre ligne <context:annotation-config/> fonctionnera

// @Autowired

-> on va pouvoir ajouter cette annotation partout ou mes classes on peut besoin de l’autoWired

Ajouter l’annotation dans InvoiceControllerWeb

Dans InvoiceServicePrefix

Le fait d’être passé par l’annotation @Autowire, les setters ne seront plus utilisés, si on veut forcer Spring a passer par les setters faudra le faire l’ajouter seulement sur les setters.

@Autowired évidement est à ajouter sur les propriété que je veux lier à mes injections de dépendances

@Autowired

private InvoiceRepositoryInterface repository;

Dans InvoiceServicePrefix

@Autowired

private InvoiceServiceInterface service;

Dans InvoiceControllerWeb

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 9 - Movie > Core (toujours core part 8)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Config sur applicationcontext.xml et modif sur DefaultMovieService et MovieController

//

// Valorisation des propriétés par annotation @Value (Démonstrations > toujours core part 8)

//

Il reste plus grand chose dans notre fichier xml, mais peut-on se débarasser des propriétés qui transmettent des données à nos attributs? -> et bien oui avec l’attribut @Value

En fait, on utilise jamais @Value avec une valeur en dure

On pourrait enlever les property de notre fichier xml et noter au dessus de la propriété lastNumber @Value(‘112’); mais en fait ça n’aurait aucun intérêt autant initialiser notre attribut directement dans le code.

On va plutôt créer pour cela un fichier application.properties (qu’on aurait pu appeler différemment), le référencer côté Spring dans notre fichier .xml et faire référence dans l’attribut @Value à ce fichier application.properties.

Pour créer ce fichier simplement clique droit sur le dossier resources new file faire application.properties

Ps : attention a bien enlever le static à l’attribut lastNumber car les attributs static ne sont pas pris en compte par Spring

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 10 - Movie > Core (toujours core part 8)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//

// Détection automatique des beans (démonstrations > Core part 9)

//

Peut-on se passer des balises beans pour instancier nos classes?

@Component va indiquer que la classe est un composant de votre architecture (c l’annotation la plus générique)

Mais on ne va pas utiliser celle-ci, on va utiliser une annotation fille qui est @Controller (techniquement pas de différences entre les deux , Spring va les traite de la même manière mais c’est plus explicite comme ça.

On aura donc @Service pour nos services et @Repository pour nos repository

Pas de différence entre ces annotations si ce n’est @repository qui a quelques subtilités en plus.

La promesse de Spring par rapport à ces annotations, c’est qu`à l’avenir l’interpretation de ces annotations pourraient changer et devenir plus spécifique, donc autant les utiliser.

Alors est-ce que @Component reste utile? Oui car mise à part ces 4 annotations il n’en existe pas d’autres annotations, si demain si on a des composants complémentaires @Component sera utile. (Pas d’autres stéréotypes qui héritent de @Component)

Attention, annoter toutes ces classes ne suffit pas, il faut dire A Spring explicitement a partir de quel package cette detection doit commencer.

Pour ça dans notre fichier xml on a une nouvelle balise à utiliser , on va préciser a Spring ou le Scan de l’application démarre :

On pourrait préciser le démarrage du Scan sur le chemin le plus haut <context:component-scan base-package=« com.company.invoice"/> mais c’est pas une bonne pratique.

Relancer l’application après

//

// Sommes-nous allé trop loin avec les annotations?`

//

Si vous regardez bien le code, avant dans le fichier xml on avait une configuration 1, on savait ce qui allait être instanciée, maintenant qu’on utilise du full annotation, si je voulais lancer la configuration 2 pour un client 2 par exemple? Comment pourrais-je faire? Faudrait la déclarer cette configuration 2

Mais la du coup avec les annotations, comment pourrais-je faire? Si par exemple je voulais utiliser le service InvoiceServiceNumber? Ba si je rajoute @Service à ce service je vais avoir une erreur que vous allez souvent voir en Spring : UnsatisfiedDependencyException -> no uniqe Bean definition : expection single matching Bean but found 2

Que s’est-il passé?

Deux services ont été instanciée, le problème c’est que InvoiceControllerWeb l’attribut service a instancié un objet compatible avec l’interface InvoiceServiceInterface et du coup ba ya deux classes services qui le sont maintenant, du coup Spring ne sait pas quel composant injecter !

On aurait le même problème si on avait annoté deux repository, ou deux controller au niveau de App, on se saurait quel controller récupérer du context.

Conclusion :

POur pouvoir gérer deux scénarios deux configuration faudrait que chaque classe soit annoté, mais l’autowiring étant basé sur le type, cela rend impossible l’injection de dépendance.

Est-ce impossible de répondre à ce besoin avec les annotations? Répondre aux besoin du client 1, 2 et 3 en mettant nos annotations partout sur nos controller, services et repository? Bon ce serait dommage quand même car c’est ce besoin qui nous a conduis jusqu’ici.

//

// Gérer les conflits de dépendances

//

Mettre @Service sur tous les services

Mettre @Controller sur tous les controllers

Mettre @Repository sur tous les repository

Ici on va voir comment résoudre le problème mentionné ci-dessus avec les annotations.

Pas de remède miracle, plusieurs solutions :

* Répartition par package
* Répartition par librairie
* Répartition par Annotation @primary
* Autowiring byName
* @profile ou @Conditionnal

On va en voir une maintenant, la répartition par package, on en verra ici et la également.

Pour cela on va duppliquer la balise <context:component-scan base-package=« com.company.invoice"/> et ne plus préciser le package principal, c’est une mauvaise pratique

Il faut scanner au niveau des types de composants :

<context:component-scan base-package="com.company.invoice.repository"/>

<context:component-scan base-package="com.company.invoice.controller"/>

<context:component-scan base-package="com.company.invoice.service"/>

Bon faire ça ne résoud pas le problème mais il permet de créer des sous package et regrouper ensemble nos configuration

//

// Regrouper nos configuration en package (Core part 10)

//

Réorganiser le différentes classes dans leur package voir projet

Penser à refactoriser les classes quand tu les mets dans des packages

Maintenant que tout organiser en package plus spécifique, malgrès qu’on est déclaré tous nos controllers et services et repository avec des annotations, il suffira de préciser

<context:component-scan base-package="com.company.invoice.repository.web"/>

<context:component-scan base-package="com.company.invoice.controller.prefix"/>

<context:component-scan base-package="com.company.invoice.service.memory"/>

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 11 - Movie > Core (toujours core part 9)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//

// Se passer du fichier xml (démonstrations > core part 11)

//

INFO pour comprendre pourquoi on peut dégager context:annotation-config car maintenant on a les scans

<context:annotation-config> is used to activate annotations in beans already registered in the application context (no matter if they were defined with XML or by package scanning).

<context:component-scan> can also do what <context:annotation-config> does but <context:component-scan> also scans packages to find and register beans within the

Notre fichier xml est presque vide, mais pourrait-on aller encore plus loin et s’en passer totalement? Ou l’avoir en complément si besoin?

Déja, <context:annotation-config/> on peut la dégage de notre projet.

Mais pour les lignes suivants :

<context:property-placeholder location="classpath:application.properties"/>

<context:component-scan base-package="org.mycompany.bluraystore.controller"/>

<context:component-scan base-package="org.mycompany.bluraystore.repository.file"/>

<context:component-scan base-package="org.mycompany.bluraystore.service"/>

On fait comment pour dire a Spring qu’il existe un fichier Properties etc…

Et bien on va devoir le mettre dans le code java, on parle de java config (configuration java)

On va créer pour cela AppConfig et utiliser l’annotation @Configuration et @ConfigurationScan.

A partir de la on pourrait supprimer le fichier xml mais comme il sert a rien on peut le laisser.

Il reste à traduire le context-property-placeholder

Du coup maintenant si on ne veut plus utiliser le xml on va devoir modifier le fichier App en utilisant la classe AnnotationConfigApplicationContext.

Et voila on peut se passer du fichier xml :)

//

// Le fichier xml en complément?

//

On peut aussi se dire, je veux utilise le fichier xml en complément, comment faire cela? Pas très compliqué, au niveau de AppConfig utiliser l’annotation

@ImportRessource

Par exemple, on pourrait choisir d’importer le application.properties dans le xml et pas au niveau des annotations de AppConfig

AU FINAL, NOUS ON VA SE PASSER COMPLÉTEMENT DE NOTRE XML

Conclusion :

AppConfig n’a pas de méthode pour l’instant, mais pourrait-elle en contenir ? Et bien oui et cela va s’avérer d’ailleurs très intéressant

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 12 - Movie > Core (toujours core part 10)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//

// Nos classes de configuration plus en détail (démonstrations > core part 12

//

On pourrait avoir d’autres classes de configuration

On pourrait isoler dans un autre fichier de configuration l’annotation @PropertySource que l’on avait dans AppConfig dans une nouvelle classe AppConfigPropertySource par exemple.

Du coup dans App, cela va avoir des conséquences, faut juste rajouter AppConfigPropertySource à AnnotationConfigApplicationContext

On pourrait avoir 10 fichiers de config mais pourquoi pas zéro? :)

C’est tout à fait possible !

On a vu que l’annotations @Configuration pouvait être dans n’importe quelle fichier java, alors pourquoi pas dans notre App ! Ba c’est ce que l’on va faire !

Les deux autres classes de configuration vont donc servir à rien maintenant !

Donc montrer ici que dans AppV1WithTwoConfig on peut faire référence à nos deux fichiers de configuration et qu’après on peut s’en passer en mettant tout dans App.

//

// Ajouter aux classes annotés @configuration

//

En fournissant à ces méthodes @bean ces méthodes auront vocations a fourni des objets supplémentaires au conteneur léger.

Instancions dans App un service grace a une méthode

Pour cela, nous allons retourner nous même un service donc enlever com.company.invoice.service.prefix de component Scan

C’est notre méthode qui va retourner ce service maintenant, on l’appelera par exemple configureInvoiceService.

L’annotation @Bean nous a permis de retourner un service, mais on aurait même pu retourner notre PropertySource en retournant nptre application.properties.

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 13 - Movie > Core (toujours core part 11)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

**3EME PARTIE - Mise en place de Spring Boot (démonstration > core part 13**

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Maintenant qu’on a mis tout ça en place, pourquoi est né Spring boot ?

Imaginons pour cette partie que je n’avais qu’un seul client donc qu’une seule config

Je souhaite garder InvoiceControllerWeb - InvoiceServicePrefix - InvoiceRepositoryMemory

Ba dans un premier temps je vais enlever les annotations @Controller et @Service et @Repository dans les classes que je n’ai pas mentionnée au dessus.

Vu que je n’utilise maintenant plus q’un seul controller service et repository je vais pouvoir modifier @ComponentScan(basePackages et mettre le package principal

Je peux également supprimer la méthode qui me retourner mon service dans App

-> si je relance mon code normalement ca fonctionne (CONFIG 1)

//

// @SpringBootApplication

//

Pour utiliser SpringBoot je vais utiliser une première annotation mais je vais avoir besoin pour cela d’ajouter la librairie Spring boot appelée SPRING-BOOT-AUTOCONFIGURE

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-autoconfigure</artifactId>

<version>2.7.0</version>

</dependency>

À ajouter dans le pom.xml

@SpringBootApplication vient remplacer @Configuration et @ComponentScan, springboot va automatiquement detecté les controllers et services et repository déclarés.

Attention on garde tout de même @PropertySource

-> Load new Maven Dependencies

@SpringBootApplication vient donc alléger le code en remplacant nos deux annotations mais pour l’instant il ne s’agit pas vraiment du application Spring boot à proprement dit.

-> lancer l’application

SpringBoot ne fait pas que l’ancer notre classe executable on va le voir ensemble

D’ailleurs la façon dont le context va être récupéré va chaner aussi voir modification dans App

En ayant rajouté ApplicationContext context = SpringApplication.run(App.class); si on relance l’application on voit que les logs sont différents, on a maintenant une application pilotée par springBoot

On a d’ailleurs des logs similaires au log que l’on avait eu quand on a lancé pour la première fois notre application générée par Spring initializer

D’ailleurs on comparera avec le temps l’application que l’on a généré avec ce que nous avait généré Spring initializer

//

// @PropertySource("classpath:application.properties")

//

Il faut savoir que quand vous allez lancer votre application Avec springBoot, Spring va regarder si a la racine de votre classpath, vous avez un fichier application.properties, plus besoin donc de le préciser avec une annotation, springBoot le detectera automatiquement !

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 14 - Movie > Core (toujours core part 13)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//

// Spring Boot & Maven (Démonstrations > core part 13)

//

Dans cette partie, l’idée est d’expliquer comment configurer notre projet actuel pour arriver à une config similaire à celle générée par Spring initializer et comment déployer l’application avec maven.

On a suivi deux voix jusqu’ici avec Spring initializer, on a eu application clef en main, puis on est passé par les concepts fondamentaux Spring pour créer notre application et arriver vers Spring boot.

**Mais finalement, pourrait-on mettre en place à la Mano la même architecture que Spring initializer nous avait donné?**

La réponse est oui, presque.

On pourrait comparer un peu (ouvrir les deux projets, demo et la dernière version de la démonstrations (core part 11) et comparer les deux.

/// Différence au niveau de l’executable

La première différence à noter, c’est dans l’executable App, dans notre version on a pu récupérer le context, ce n’est pas le cas dans l’application de démo mais on aurait tout à fait pu le faire en variabilisant le retour de SpringApplication.run()

D’ailleurs c’est parceque nous on part sur l’usage de la console et non d’une page web, et ça c’est une différence majeure car si on lance l’application de démo, on voit que le serveur ne s’arrête pas.

ELle ne s’arrêtera que lorsqu’on souhaitera arrêter le service alors que notre appli de facturation s’arrête après avoir générer la facture.

Si vous regardez bien dans notre code, on a pas les arguments de la méthode main passer SpringApplication.run(App.class) contrairement à la démo !

Au niveau des dossiers controller/service/repository c’est plus ou moins la même chose;

//// différence dans le pom.xml

Il subsiste une différence majeure ce sont la gestion des cycles de vie de l’application avec Maven configurer dans le pom.xml.

Dans notre config actuelle, on avait une dépendace pour spring-context, Spring-boot et junit pour les tests unitaires.

Dans notre version actuelle, on aurait pu supprimer la dépendance spring-context car elle est une dépendance transitive de spring-autoconfigure. Celle-ci a été ignorée car on l’a déclarée dans nos dépendances.

La question c’est comment spring initializer s’est débrouillé pour gérer les versions de spring-context et spring-bootautoconfigure pour ne pas avoir de conflit?

//Aucune précision sur les versions

Si on tcheck les dépendances installées par Spring initializer dans demo ont voit qu’il n’y a aucune précision sur les versions des dépendances alors ou est-ce spécifié?

Pour cela il faut remonter plus haut dans le pom.xml au niveau de la balise parent qui indique a maven que le projet que nous réalisons hérite d’un projet plus vaste. (Spring-boot-starter-parent)

Dans notre application, nos dépendances n’héritent de rien,

Donc quand on hérite d’un projet maven on bénéifice de tout ce qui est déclaré dans le projet parent

///

/// Allons voir ce qu’il y a dans spring-boot-starter-parent

///

Aller voir sur mvn [repository.com](http://repository.com)

Cliquez sur la version de la dépendance

Si on tcheck on voit que cela ne fournit aucune dépendance à la compilation mais des tones de dépendances gérées (managed dependenciess)

On retrouve la librairie de Spring-boot que l’on utilise spring-boot-auto-configure, la partie web et jpa que l’on a vait demandé sur Spring initializer

Les managed dependencies ne sont pas directement téléchargées mais pour lesuqelles ont donne des informations au cas ou elles seraient nécessaires

spring-boot-starter-parent fournis donc toutes les versions des dépendances au cas ou elles seraient utiles pour votre projet.

Voir la version de nos dépendances en fonction de notre version de spring-boot-parent.

//

// Configuration de notre projet (démonstrations > core part 12)

//

Copier coller spring-boot-starter-parent de démo dans notre projet actuel et enlever les version de spring-context et autoconfigure qui ne sont plus utiles, d’ailleurs Spring-context on peut le dégager.

Héritez de spring-boot-parent c’est cool on a pas à se soucier de la verison des dépendances.

//

/// spring-boot-maven-plugin

//

Dans démo, on a aussi un plugin maven de Type spring-boot-maven-plugin. Rappelez quand on avait fait mvnw Spring-boot:run et bien cette commande nous venait de ce plugin.

Il apporte pas mal de chose comme démarrer l’application java on verra d’autres exemple après et pourquoi il est essentiel.

On va donc copier ce plugin. D’ailleurs sur notre projet la partie plugin-management on pourrait la dégager.

Pour le moment on aura pas besoin de plus de choses dans notre application actuelle.

On ne va donc pas récupérer les dépendaces pour le web ou encore pour jpa et h2 Database.

//

// spring-boot-starter

//

Mais a-t-on du coup quand même besoin d’un starter pour créer notre application?

Pour le savoir, retournons sur Spring initializer et créons une application sans dépendances, et vouons un peu si on a un starter de départ ou pas.

AU lieu de télécharger tu peux faire explore, aller dans le pom.xml et vous verrez qu’on a un starter de départ qui est spring-boot-starter qui a une dépendance vers spring-boot-context et autoconfigure.

On va donc changer ma dépendance spring-boot-autoconfigure de notre projet par la dépedance spring-boot-starter, les versions seront gérés par le parent.

On a maintenant une application qui est similaire a celle que l’on aurait créé si on était passé par Spring initializer.

(Voir pom.xml)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 15 - Movie > Core (toujours core part 14)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//

// Fat archive - déployer une application

//

A ce stade que vous apporte Spring boot? A ce stade pas grande chose en vrai !

Bon ok spring-boot nous apporte :

* une annotation @SpringBootApplicaiton ncd ous simplifie un peu l’écriture du code
* SpringAPplication.run une novuelle facon de démarre une app mais on sait pas vraiment pk on utilise celle la plutot que celle davant avec Spring
* Allez le plugin spring-boot-maven nous offre une nouvelle facon de lancer une application avec Spring-boot:run mais on pourrait s’en passer non?

// lancer l’application sans passer par spring-boot:run

Pour lancer l’application sans passer par cette commande on va ouvrir un terminal, et aller dans les fichiers .class généré par IntelliJ.

=> Target > classes

Moi ça se trouve dans /Users/samihhabbani/Documents/spring\_boot/démonstrations/core part 13/target/classes

Si je lance :

samihhabbani@macbook-pro-de-samih classes % java com.company.invoice.App

Error: Unable to initialize main class com.company.invoice.App

Caused by: java.lang.NoClassDefFoundError: org/springframework/context/ApplicationContext

Je vais avoir une erreur car j’ai besoin de pas mal de dépendances !

Faudrait pouvoir éxecuter java -cp avec la dépendance à spring-context

Si compare ce que l’on faisait avant yavait pas besoin de faire tout ca et d’aller dans Target classes

Et pas besoin de préciser les librairies dont on avait besoin et on avait pas besoin de préciser la classe de l’executable Spring boot la détecter automatiquement (App)

///

/// Peut-on executer l’application avec le jar?

///

En production, on aurait besoin d’un .jar pour executer notre application par exemple et bien essayons de lancer l’application avec le .jar qui sera généré par mvn clean Install

Pour cela commenter le plugin spring-boot-maven-plugin

Se positionnner sur l’entrée du projet /Users/samihhabbani/Documents/spring\_boot/démonstrations/core part 13 et lancer mvnw (ou mvn) clean install

Dans le target maintenant cela vous a généré un jar

On va rentrer dedans et voir ce qu’il y à l’intérieur (tu peux explorer ou Unarchiver), on verra que dans le dossier META-INF on a un manifest mais aucun classpath avec toutes les librairies nécessaires ni même l’indication de la classe à executer

Allons dans le Target et essayons d’éxécuter le jar généré, aller pour cela dans target

samihhabbani@macbook-pro-de-samih target % pwd

/Users/samihhabbani/Documents/spring\_boot/démonstrations/core part 13/target

Et lancer la commande suivante : java -jar core-1.0-SNAPSHOT.jar

Et malheureusement il ne se passe rien !

Alors avec maven on pourrait configurer le fichier manifest de façon à pointer vers la classe exécutable App et configurer le classpath mais il resterait toujours le problème liées au dépendances :

// problématiques

Mes librairies se trouvent sur notre poste local dans mon repository maven local, le répertoire .m2, si je veux executer ce jar dans une autre machine, il faudrait que le classpath soit portable on peut plus porter sur le m2 de ma machine personnelle faudrait livrer le jar et toutes les librairies nécessaires dans un repertoire et faire pointer le classpath vers ce répertoire.

Le plugin spring-boot-maven-plugin peut-il nous aider à générer ce jar executable ? Et bien oui !

Aller décommenter le plugin dans vos dépendances de sorte a ce qu’on est accès à ce plugin.

// relancer un clean install avec ce plugin activé dans nos dépendances

///// -> cela nous a généré un jar beaucoup plus gros !

D’ailleurs si on dézip ce fichier on voit qu’on arrive pas tout de suite dans le code mais on a trois dossier :

* BOOT-INF
* META-INF
* org

Dans BOOT-INF on retrouve nos classes et nos fichiers de configuration et un répertoire lib dans lequel on va retrouver toutes nos librairies.

Dans META-INF on retrouve notre manifest qui est plus complet dans lequel on retrouve :

* Start-class : App
* Les librairies qui font parties de notre class path :

Spring-Boot-Lib: BOOT-INF/lib/

Spring-Boot-Classpath-Index: BOOT-INF/classpath.idx

//

// Executons maintenant notre fat archive

//

Maintenant on peut executer notre jar :

Cd /Users/samihhabbani/Documents/spring\_boot/démonstrations/core part 13/target

java -jar core-1.0-SNAPSHOT.jar

Il nous a donc fallu plusieurs éléments pour executer cette application :

* plugin maven spring boot
* @SpringBootApplicaiton
* SpringApplication.run()

Cela nous a permis :

* faciliter l’écriture en Spring
* Faciliter le build de l’application
* Pour la déployer et l’executer !

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

**4EME PARTIE - Les applications web avec Spring boot (démonstrations > core part 13)**

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

On va se concentrer sur les solutions les plus couramment utilisées.

Pré-requis :

* avoir déjà fait du web avec Java
* Savoir ce qu’est un serveur applicatif
* Savoir ce qu’est une api rest et exploiter des données avec javascript

On va parler design pattern MVC !

//

// Spring Boot Web Starter

//

Rappelez vous en JEE on avait besoin d’un conteneur de serveur, un serveur applicatif et on avait utilisé Tomcat qui executer nos servlets, la compilation java nous fournissait un fichier war pour web archive

Cette config semble incompatible avec Spring-boot car ce que l’on démarre ici c’est l’application elle-meme pas un serveur applicatif

On va pouvoir le faire aussi avec notre apli Spring boot mais plutot que de déposer notre application dans tomcat on va déposer tomcat dans notre application :)

//

// Spring Boot web starter

//

En vrai c’est quelque chose que l’on a déjà expérimenté avec le starter spring-boot-start-web qu’il va falloir ajouter à vos dépendances, et on verra que lorsque l’on démarre l’application maintenant et bien on a localhost 8080 qui a démarré.

On a une erreur car on a pas d’index.html dans notre projet

Il nous présente un msg d’erreur pas très jolie, si on souhaitais avoir une page d’erreur plus jolie faudrait lui fournir /error

//

// Créer notre page index.html

//

Dans resources créer un dossier static avec une page index.html à l’intérieur

Rédémarrer ensuite l’application

Le Hello World devrait s’afficher correctement mais on pourrait se poser plusieurs questions à ce stade :

1. Pourquoi ais-je tomcar? Pourquoi suis-je sur le port 8080 d’ou nous vient cette configuration?
2. Suis-je obliger de rédémarrer l’application pour voir les changements?

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 16 - Movie > Core (toujours core part 15)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//

// Auto-configuring Spring

//

Pour comprendre pourquoi on a Tomcat c’est assez simple il suffit d’aller dans la vue maven, dans les dépendances et vous verrez que le plugin spring-boot-starter a une dépendance direct avec tomcat

Au passage on se rend compte que spring-boot-starter a une dépendance transitive vers spring-boot-starter donc on pourrait dégager spring-boot-starter

Pourquoi on a une config par défaut ?

Car l’annotation @SpringBootApplication (dans App) en fait est un raccourcis vers 3 choses :

@Configuration

@ComponentScan

@EnableAutoConfiguration et lui va chercher tous les fichiers de configuration annoté par @Configuration et on a des fichiers de confg en dehors de nos package installé par Spring-boot-starter web qui vont être detecté

// Comment changer cette configuration?

Et bien on va devoir configurer certaines choses dans notre application.properties, il faut pour cela connaitre à l’avance le nom des propriétés que l’on peut retrouve sur la documentation officielle de Spring

<https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties.server>

Pour trouver ce lien vous pouvez taper common application Properties, quand tu arrives sur la docs il faut aller dans server Properties

Chercher la propriété : [server.port](https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#application-properties.server.server.port) on voit qu’elle a une valeur par défaut

Chercher aussi server.servlet.context-path (le chemin qui mène vers l’entrée de mon applicaiton, index.html) pour changer le context path de l’application

Les modifications sont a apporter dans application.properties

Relancer le serveur après avoir modifié tout ca

Le port 80 est le port par défaut sur localhost

Je peux donc taper :

<http://localhost/invoice> et j’arriverai sur ma page d’accueil

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 17 - Movie > Core (toujours core part 15)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//

// Ressources web statiques (démonstrations > core part 13)

//

Pour redéployer automatiquement vos modifications quand on modifie les pages web

Tu peux cocher une option sur IntelliJ

File > settings > Build, Execution, deployment > compiler

Cocher l’option build project automatically

(Aller dans preferences sur Mac pour arriver ici)

Evidement cette solution fonctionne que en local, en production ya pas d’IDE c’est un jar qui est executé donc faudra recompiler l’application

Autres solutions :

Les spring dev tools mais ce sont des outils de dev, donc en production le problème sera le même.

On peut donc pour cela retourner da ns les Properties de Spring, rubrique web Properties et chercher la property Spring.resources.static-locations on retrouve l’emplacement de nos éléments statiques par défaut

L’idée c’est de dire par exemple, mon fichier index.html je vais l’externaliser par rapport à mon projet donc il ne sera plus dans mon jar.

Je pourrais par exemple le mettre dans C\\temp\\static et préciser avec la propriété spring.resources.static-locations que mon index.html est dedans.

Du coup si que je sois en production ou pas, si je modifie mon fichier index.html j’aurais pas besoin de recharger le serveur un simple rechargement de page suffira.

(Voir application.properties)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Exercice 17 - Movie > Core (toujours core part 15) OPTIONNEL

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//

// Spring MVC démonstrations > core part 13)

//

Controleur frontal qui va faire appel à toutes les requêtes qui lui sont destinées, comme la plus part des servers mvc, il va s’agir d’une servlet, fourni par spring-web-mvc

Notre servlet va attraper les requêtes http qui sont destinées au serveur

Cette valeur par défaut est fournie par une propriété (web Properties) que l’on peut modifier dans notre application.properties : Spring.mvc.servlet.path et sa valeur par défaut est ‘/‘;

Conclusion :

LE SIMPLE FAIT D’UTILISER LE STARTER-WEB NOUS FAIT PROFITER D’UN MVC FONCTIONNEL

//

// Comment l’utiliser ? démonstrations > core part 13)

//

Les requêtes http vont s’adresse à la couche de controller que l’on a déjà mis en place via les annotations @Controller.

Comment la servlet fait-elle pour contacter un controller plutôt qu’un autre?

C’est l’url qui va l’orienté vers le controller de destination

Créer la méthode displayHome() dans InvoiceControllerWeb et montrer que quand je tape l’url <http://localhost/invoice/invoice-home> j’ai bien mon system.out qui s’affiche.

//

// Les vues avec Thymeleaf démonstrations > core part 14)

//

//

Petite note à moi : même penser à bien importer la dépendance thymeleaf dans le pom.xml pour que les fichiers html soit reconnu par le controller quand je fais return’index’;

Peut-on utiliser jsp comme modèle de templating? Techniquement oui avant c’était un standard mais cela support pas mal de limites.

Spring nous recommande un autre moteur de templating, Thymeleaf

(Thymeleaf est une techno java)

On peut conserver la page html d’origine avec Thymeleaf mais faudra adapter l’entête de notre fichier avant.

Comme la page html maintenant avec Thymeleaf n’est plus statique mais dynamique, elle va se retrouver plus dans le dossier static mais dans un nouveau dossier ‘templates’ (standar spring-boot)

A partir de la, pour utiliser index.html que l’on a déplacer dans templates, au niveau de noter controller on va retourner ‘index’.

Ajouter un span a votre index.html :

À l’entête de votre page html : xmlns:th="http://wwww.thymeleaf.org"

<span th:text=« ${#locale.country}">US</span>

${#locale.country} » : langue par défaut du navigateur

Thymeleaf n’est pas fourni par le starter web donc va falloir l’ajouter à notre dépendance

On peut aller sur Spring initialzer et chercher la dépendance liée à Thymeleaf

Faire explore et récupérer la dépendance.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

On peut modifier la langue par défaut de l’application via les Properties :

Pour passer de FR à US vous pouvez modifier de cette façon :

spring.mvc.locale=US

//

// Affichage de données dans la vue (toujours part 14)

//

Modifier le index.html pour permettre des actions et d’afficher une liste de factures.

Pour cela, modifier dans InvoiceRepositoryDataBase (à créer si inexistant) une méthode qui me retournera ma liste de facture.

Pour cela créer InvoiceRepositoryDataBase et mettre dans l’interface la méthode list() qui nous retourna une liste d’invoice

Faudra ensuite ajouter une méthode dans l’interface Service pour appeler le répository qu’on implémentera dans les classes qui l’implemente et qui appeleront tout simplement la méthode list() de leur repository.

Depuis le controlleur je vais appeler le service du coup pour récupérer ma liste.

La question maintenant c’est comment faire pour afficher la liste dans ma vue.

Pour afficher la liste de facture, modifier le index.html

Pour renvoyer notre liste à la vue on pourrait utiliser pas mal de chose, dont le httpServletRequest que l’on connait bien, on pourrait le passer en paramètre de la fonction displayHome et faire :

request.setAttribute(‘invoices’, invoices)

Mais on a un objet plus adapté surtout quand on veut renvoyer qu’un truc à la vue :

Faut juste renvoyer la liste en sortie de la méthode c’est tout.

Le truc c’est que si l’on renvoit directement la liste dans la méthode list() de InvoiceCOntrollerweb, on ne dit plus quelle vue Spring doit retourner.

Et bien spring va mapper l’url capter et chercher un fichier index.html du même nom diretement.

Quand est-il de l’attribut setAttributes qu’on a passé a la request (‘invoices’)? Et bien on peut utiliser une nouvelle annotation qui est @ModelAttribute pour cela !

//

// Modularité (Démonstrations > 16 - invoise web) (nouveau projet web)

//

Rappelez-vous pour éviter les problèmes au niveau des injections des dépendances on utilisait pas les annotations @Controller et @Service et @Repository partout, mais maintenant il faudrait que Spring continue d’assurer ses promesses et gère les différentes configurations de nos clients !

On va pas tout de suite répondre au problème mais apporter des réponses

Pour l’instant on a un qu’un seul controller qui a sa place, le InvoiceControllerWeb. Keyboard et Scanner ne correspond pas au web.

/// Pour les futures démonstrations on va créer un nouveau projet wen

La générer via Spring initializer.

Triplet

com.mycompany.invoice

Artifact : invoice-web

Dépendances :

Starter web

Starter Thymeleaf

-> generate

////

//// On va ouvrir les deux projets sur IntelliJ et bouger certaines classes dans le nouveau qu’on a créé

////

Pour bien comprendre ce que j’ai fait, j’ai copié collé le projet core part 14 car cela ne va plus être un projet spring-boot mais un simple projet Spring.

16 - invoice web va devenir notre nouveau projet web et on va séparer la partie web de la partie servie métiers.

POur ouvrir le deuxième projet dans IntelliJ, ouvrir la vue Maven, ya un bouton +, tu peux selectionner le projet que l’on vient de créer.

Dans Intellij faut donc avoir core part 15 d’ouvert et 16 - invoice web

On va transiter pas mal de chose de 15 - core à 16 - invoice web depuis IntelliJ on ayant les deux projets ouverts :

* les propriétés de application.properties de 15-core on été copié dans celui-ci (ne prend que à partir du port pour notre nouveau projet)
* Déplacer InvoiceControllerWeb dans un dossier controller que vous aller créer dans com.mycompany.invoice.invoiceweb.controller
* Supprimer le package web qui ne servira plus à rien dans le package controller de 15-core
* supprimer parent, les starters et le plugin dans le pom.xml de 15-core et remettre la dépendance vers Spring context (ne pas oublier de préciser la version)

On a maintenant deux projets qui ont le même package racine : com.mycompany.invoice

15 - core représente une librairie core

On va rajouter un niveau de package (appelé core) pour tout ce qu’il y a dans 15 - core, pour cela créer un package core dans le package com.mycompany.invoice new package et déplacer tous les dossiers à l’intérieur du nouveau package.

Notre librairie n’a plus vocation a être executable donc on peut supprimer App dans 15 - core

Application.xml ne nous sert plus depuis un bail donc on peut le supprimer aussi

A ce stade le dossier template aussi et static on disparu

-> faire un build de ma librairie

//

// Ajouter une dépendance de 16 - invoice web à notre librairie 15 -core

//

Pour ajouter la dépendance à notre librairie tu peux aller dans le pom.xml de 16 - invoice web et click droit generate > dependency (chercher l’artifact de notre librairie : com.mycompany.invoice.core)

Après avoir mis la dépendance relancer le build (ne pas oublier de faire maven load changes avant)

On voit qu’on a une erreur car @SpringBootApplication va lancer un componentscan au la racine de InvoiceWebApplication or la dépendance à l’interface dans notre controller se trouve dans notre librairie, on aurait plusieurs solutions pour résoudre ce problème :

La première solution va être de remonter d’un cran InvoiceWebApplication et le placer au même rang que le package com.mycompany.invoice pour qu’il Scan toutes les classes de ce package y compris celui-ci de notre librairie.

Pour le monter d’un cran, vous aller devoir changer l’affichage du projet en project file et déplacer InvoiceWebApplication au niveau du package invoice.

On a toujours une erreur car on a pas accès au propriété lastNumber et prefix qui sont dans le application.properties de notre librairie core.

Pourquoi? Car @SpringBootApplication ne va lire que le application.properties de notre projet 16 - invoice, c’est liée à l’autoconfiguration. On pourrait modifier cette auto configuration, mais le plus va être de copier les propriétés de application.properties de notre librairie core dans notre projet web. D’ailleurs on pourrait maintenant le supprimer de notre librairie core  
  
Voyons maintenant si l’application tourne toujours.

// Conclusion

On a séparé les responsabilités avec une librairie core qui contient les services les entités métier et l’interface qui permet de définir le contrat de chaque controller et une app dédiée avec les ressources web exploité par notre moteur de templating.

Objectif :

Ne pas tout mélanger !

//

// Paramètre de requête et préparation à REST (15-core et 16-invoiceweb)  
//

Objectif accéder à plus d’informations sur une facture au niveau du tableau html

Modifier du coup le td de notre tableau dans invoice-home.html

Dans l’url que l’on va généré en général on met dans l’url l’entité concerné donc je vais essayer de générer une url qui fera /invoice/lenumerodelafacture

<a th:href="@(/invoice/(number)(number = ${invoice.number})" th:text="${invoice.number}"></a>

Rajouter une nouvelle méthode pour accéder au détail de la facture dans InvoiceControlelrWeb

Pour aller dans le sens d’une architecture REST et bien les urls de InvoiceControllerWeb devrait commencer par /invoice car liée à l’entité Invoice, pour cela on va ajouter une annotation à InvoiceControllerWeb qui va préfixer les url par /invoice et capter du coup toutes les urls commencant par cela. On pourra simplifier le mapping des urls dans nos méthodes.

Comme maintenant je suis déjà dans /invoice maintenant je peux simplifier l’url que je constituait pour arriver au détail d’une facture :

<a th:href="@(/(number)(number = ${invoice.number})" th:text="${invoice.number}"></a>

Mettre en place une nouvelle méthode dans notre librairie core getInvoceByNumber() à ajouter dans l’interface InvoiceServiceInterface

getInvoceByNumber() appelera une méthode getById() dans notre repository qu’il faudra implémenter aussi.

La méthode que l’on appelera sera donc dans InvoiceRepositoryDataBase.

On veut retourner un peu plus d’infos donc on va ajouter à notre facture un nouvel attribut orderNumber (modifier l’entité)

//

// ModelAndView

//

A ce stade on a un problème c’est qu’on a renommé nos urls du coup par exemple /home ne pointe plus vers invoice-home.html et comment on va faire avec notre nouvelle méthode displayInvoice() qui a un url /invoice/numerodefacture?

Faudrait pouvoir renvoyer une vue et un model, et bien on a retourner un nouveau type d’objet :

ModelAndView

Une fois nos méthode modifiées, on va créer la vue invoice-details.html

//

// On aurait une solution alternative au ModelAndView : argument de type Model model

//

Model model quand passé en argument doit toujours être le dernier argument

Voir correction avec les versions de InvoiceControllerWeb

//

// Gestion de formulaire (Démonstrations - core part 15)

//

Créer page pour l’ajout de facture invoice-create-form.html

Ajouter dans InvoiceCOntrollerWeb la méthode permettant de faire la redirection vers invoice-create-form.html

Mettre en place le formulaire

Ajouter @RequestMapping à la méthode createInvoice() qui nous permettait de créer une facture

Pour respecter le format application Rest on aimerait juste capter un post sur l’url /invoice

On va utiliser pour cela @PostMapping(‘’) (on peu ne rien mettre dans les parenthèses)

Pour info @RequestMapping par défaut sont en Get donc on pourrait les remplacer par des @GetMapping partout

Créer ensuite invoice-created.html pour renvoyer un msg de succès.

Faudrait donc createInvoice() renvoit un string pour renvoyer ‘invoice-created’ mais ca pose un problème car faudrait modifier l’interfaceController et donc les autres controller qui implémente cette interface.

On pourrait utiliser ModelAndView mais ca imposerait à InvocieControllerInterface et aux méthodes qui l’implémentent et donc au projet complet d’avoir accès à la config Spring-boot donc modifier tout ca pour retourner une string c’est pas si mal.

Et puis dans la vraie vie, on aura rarement une interface pour nos controllers.

Ne pas oublier la méthode post dans votre formulaire

//

// Backing Bean

//

Maintenant qu’on a fait la page qui accueil le formulaire, et qui renvoit le msg de succès faudrait une classe qui représente notre formulaire : on parle de Backing Bean

=> la classe Invoice fera l’affaire (numéro de facture auto-incrémenter)

Reprendre la méthode createInvoice dire qu’on attend une Invoice en entrée

-> ajuster cette méthode au niveau des classes et interfaces

Comment associer mes inputs et la classe Invoice ?

Voir invoice-create-form

//

// A bien faire

//

Va falloir copier @ModelAttribute Invoice invoice de createInvoice dans la méthode qui affiche le formulaire displayInvoiceCreateForm.

Pourquoi est-ce qu’on a mis ca la, car au moment de charger le formilaire, on a besoin d’instancier un objet dans la page invoice-create-form pour potentiellement pré-chargé des infos dans le formulaires, ce serait par exemple le cas dans le cadre d’une mise à jour !

//

// Validation des beans : BackingBean

//

Backing Bean c’est quoi?

The client-side HTML page fields are mapped with back end Java bean class. This is called as Form Backing Bean Object.

Pour valider des propriétés d’un Bean on utilise les JSR 303 et 380 aussi appelé java x Validation ou Jakarta validation

Le standard Bean Validation (standard) ouvre la porte a des validations personnalisés

-> faudra ajouter cette dépendance à notre pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

// Annotations

Ya des annotations qu’on peut mettre sur nos proriétés pour controller les valeurs saisies mais souhaite-on le faire sur Invoice? Pas vraiment car Invoice représente une entité en base, et la les règles du formulaire nous viennent plus du côté UI… et dans un autre contexte, Invoice pourrait avoir d’autres règles

On va donc créer un nouveau package Form copier Invoice à l’intérieur et ajouter les annotations dont on a besoin. (La renommer InvoiceForm

Changer la méthode createInvoice dire qu’elle attend InvoiceForm ajouter l’attribut @Valid

On va devoir enlever l’implémentation à l’interface InvoiceControllerInterface car on modifie souvent le code et c’est galère.

Dans la vraie vie on aura pas plusieurs implémentations de controller

Attention on a modifié l’argument de createInvoice, penser à modifier invoice-create-form.html au niveau de l’objet récupéré par Thymeleaf

Penser également à modifier displayInvoiceCreateForm avec InvoiceForm

Maintenant dans les logs du runs, on te montre que tu as deux erreurs liées à ton formulaire, l’idée ici c’est de se demander comment les afficher?

////

/// BindingResult

///

L’annotation @Valid va stocker les erreurs dans un objet BindingResult, les erreurs sont collectées à l’intérieur, faut le mettre derrière le dernier attribut que l’on souhaite valider dans les arguments

Mettre à jour la méthode createInvoice en ajoutant BindingResult en argument

On va voir maintenant comment afficher les erreurs

//

// Affichage des erreurs

//

///// OGNL

Faut savoir que quand on utilise Thymeleaf tout ce qu’on met entre les acolades et ce qu’on appel de l’OGNL (Object Graph Navigation Library) sous projet Apache -> sorte de standar java

////// SpEL

Dès que Spring est utilisé avec OGNL cela devient du SpEL (plus du OGNL) : expression extrement puissant qui peut s’utiliser partout dans le code java

On peut exprimer des conditions, utiliser des regex, des variables d’environnement etc

On peut faire référence au context Spring (le context d’execution) :

* httpSession
* Dates
* Boolean
* …

Pour faire référence au contexte faut utiliser le #

Voir les divs mises en place dans invoice-create-form pour voir comment afficher les erreurs

th:if="${#fields.hasErrors('customerName')}" th:errors="\*{customerName}"

On a accès à un objet Fields et on peut préciser l’erreur que l’on cherche

Pour afficher l’erreur on utilisera pas th:text mais th:errors d’ailleurs dans notre InvoiceForm on peut préciser les erreurs que l’on souhaite renvoyer.

/// Que se passe-t-il si je ne mets pas de message par défaut?

Et bien on a des msg par défauts qui sont pas mal d’ailleurs qui nous vienenent de Hibernate Validator

Aller dans External Libraries

Hibernate-validator

org.hibernate.validator

Tout en bas ‘Resource Bundle ‘ValidationMessages’ -> vous avez toute sorte de msg prédéfinies dans plein de langues

//

/// Comment modifier le texte par défaut?

//

-> copier coller ce fichier l’appeler de la même manière

-> le même dans ses resources et la tu peux changer les msg d’erreurs :)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

**5EME PARTIE - Développer une API Rest avec Spring (06 - Développer une API ReST avec Spring)**

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

On parle de service web REST lorsque l’URL utilisé pour contacter le serveur s’écrit sous une forme standard qui indique le changement d’état que va subir la ressource ciblée.

Ressource : l’information sur laquelle on souhaite agir

/voiture GET -> liste de voitures

/voiture/12 GET -> retourne une voiture

/voiture POST -> création de voiture

/voiture/12 PUT -> modifier la voiture

/voiture/12 DELETE -> supprimer une voiture

//

// Standard JAX-RS et implémentation de référence Jersey

//

En java pour créer des web-service REST on utilise ces références.

En spring on va utiliser la structure spring-mvc donc pas besoin d’installer autre chose.

//

// Créer un package api

//

On va voir comment transformer InvoiceControllerlWen en ressource

Créer un package api et copier coller InvoiceControllerlWeb en InvoiceResource

//

// responseBody, RequestBody, RestController

//

On va modifier displayInvoice dans la copie

On va retourner une contenu textuel

PAr défaut ce sera du format json car si tu ouvre spring-boot-starter-web on voit une dépendance vers spring-boot-starter-json

On a deux fois displayInvoice , Pour faire fonctionner cette méthode faut commenter displayInvoice de InvoiceConrollerWeb

//// s’occuper ensuite de list()

//// et s’occuper ensuite de createInvoice() (le service devra nous retourner un invoice donc penser à modifier la partie core et les services/interface liées à ce controller

Si on regarde InvoiceResource toutes les mthodes de cette classe sont dédiés aux échanges rest, quand on a une resource REST on peut éviter de dire que toutes les sorties doivent être convertis en enlevant les ResponseBody et en ajoutant l’annotation @RestController

Maintenant on va voir comment exploiter ces services rest

//

// Expérimenter les services REST

//

Objectif ici, appelez notre url avec POSTMAN

A quoi va service notre api REST?

On va utiliser du javascript et json pour éviter le rechargement de la page

Une API peut avoir d’autres objectifs comme permettre à d’autres apps d’interagir avec notre service de facturation.

// [postman.com](http://postman.com) (INSTALLATION SUR L’ORDINATEUR)

-> pour télécharger https://www.postman.com/downloads/

On va utiliser postman pour expérimenter nos services REST

Aller dans postman et tester l’url suivante :

<http://localhost/invoice>

On va retrouver une combinaison de clef-valeurs

Tester maintenant :

<http://localhost/invoice>/NUM\_2

On voit qu’ici on a récupéré l’order Number (on avait configuré ca comme ca dans l’application on pouvait l’avoir qu’ici)

On va maintenant essayer d’ajouter une facture :

L’url c’est <http://localhost/invoice>/ en POST (selectionner POST)

Cliquer sur le body dans postman et copier coller :

{

"customerInvoice": ",My Company ",

"orderNumber": "ON\_003"

}

Et choisir dans raw le format JSON

On voit bien que la requête POST nous renvoit notre objet crée (Invoice)

Maintena qu’on a testé nos données, on va les exploiter dans l’application

//

// Exploiter les services ReST dans l’application

//

On va mixer des web services rest asynchrones et synchrone

/// création de la facture : pour la création de la facture on va rester du synchrone et du coup on passera par InvoiceControllerWeb on modifiera l’url @PostMapping(‘create’) pour pouvoir l’appeler

Pour l’affichage des données on le fera à travers notre api rest

On ne gère pas le formulaire en rest car ca ferait pas mal de taff côté javascript et on a pas le temps.

* modifier l’url de la méthode createInvoice
* Charger un fichier js dans un dossier js(fournir le code javascript à la classe)
* Modifier la page invoice-home.html pour charger le javascript

//// Objectif :

Sur la page d’accueil quand je clique sur une facture j’affiche son détail sur la même page avec javascript, notre service rest et même du style.css

Donner à la classe :

* la dernière version de invoice-home, style.css et script.js
* Penser à décommenter la méthode displayHome de InvoiceControllerWeb pour charger la page d’accueil, le script js sera alors appelé automatiquement dès la page chargée.

Ajouter le style à notre page invoice-home.html

NOTE :

@{/css/style.css}

L’@ ici veut dire qu’on part de la racine du context root

Conclusion :

* j’ai utilisé les méthodes displayHome et createInvoice de InvoiceControllerWeb que j’ai décommenté
* J’ai ajouté fichiers js et css à mes ressources
* J’ai modifié le contenu de invoice-home.html

Montrer à la classe dans l’inspection du navigateur que des requêtes sont bien faites dans network !

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

**6EME PARTIE - Spring et les bases de données relationnelles (07 - Spring et les bases de données relationnelles / JDBC)**

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

On va voir les persistances de données avec Spring Data

On va pas utiliser H2 Database mais MySQL

///// Installation de MySQL (moteur mysql)

Lien pour télécharger mysql : https://dev.mysql.com/downloads/mysql/

Quand tu installes tu sélectionnes custom et tu installes juste mysql server

Dans la configuration : use legacy authentification Method

Créer un compte utilisateur avec mdp : root et user : root l’ajouter

All hosts

Le moteur de bdd sql est installé sur ma machine et est démarré

Il est installé sous forme de service pour le trouver : dans le menu démarrer :

services.msc

Chercher MySQL80 tu peux changer le type de démarrage en cliquant dessus et arrêter et redémarrer le serveur

///// Installation de DBeaver

Le lien pour installer : https://dbeaver.io/download/

Client de base de données (on aurait pu utiliser workbench ou autre

Pour intéragir avec le serveur de base de données il faut un client (on aurait pu le faire en ligne de commande, mais je vous propose un client universel qui fonctionnera avec oracle mysql, sql server etc..

Il en existe beaucoup : squirell ou dbeaver qui sont populaires

/// Se connecter a notre server de bdd mysql

Cliquer sur le bouton + avec la prise dans DBeaver

Selectionner mysql

Pour vérifier que la connexion fonctionne appuyer sur Tester la connexion

Un driver JDBC va être requis téléchargez le

//

// Création de la BDD

//

Donner le dossier BDD à la classe

Créer une bdd depuis DBeaver avec la ligne suivante :

(Clique droit sur la connexion, editeur sql

CREATE DATABASE `INVOISE` DEFAULT CHARACTER SET utf8 COLLATE utf8\_general\_ci;

Pour executer le script, troisième petit onglet à gauche de l’encadré de script

USE INVOISE;

DROP TABLE IF EXISTS INVOICE;

CREATE TABLE INVOICE (

INVOICE\_NUMBER BIGINT NOT NULL AUTO\_INCREMENT,

CUSTOMER\_NAME VARCHAR(50) NOT NULL,

ORDER\_NUMBER VARCHAR(13),

PRIMARY KEY(INVOICE\_NUMBER)

) ENGINE=InnoDB;

INSERT INTO INVOICE (CUSTOMER\_NAME,ORDER\_NUMBER) VALUES

('John Doe','ON\_001'),

('Mike Myers','ON\_002'),

('Benny Hill’,'ON\_003');

Executer le script.

On va voir ensemble maintenant comment exploiter cette base de données avec Spring

// dépendances requises

MySQL driver

JDBC API

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jdbc</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

Pour effectuer une connection on va utiliser une data source avec un pull de connexion c’est spring qui va instanciée le data source pour toute l’application

On pourrait créer une classe de configuration qui nous renvoit un nouveau Bean avec un DataSource, récupéré ce Bean dans le context et faire un premier select par exemple mais pas besoin car Spring attends juste certaines propriétés dans le applicationContext.xml

// donner à la classe les propriétés à configurer (jdbc.txt)

Le pool de connexion qui est derrière la connexion c’est Hikari CP, quand on tire Spring jdbc on titre une dépendance vers Spring Hikari CP, et l’autoconfiguration de spring-jdbc-start va automatiquement prendre en charge cette librairie

Hikari CP : Implémentation de pool de connexion la plus performante sur le marché en java

Après avoir ajouté les propriétés, développer la partie dans l’executable InvoiceWebApplication pour tester la connection Avec un premier select en console.

//

// Ecrire et lire avec JDBC template

//

Le fait d’avoir annoté notre controller avec @Repository nous donne accès à JDBC template

Encapsuler jdbctemplate dans InvoiceRepositoryDatabase

Ajouter la dépendance spring-jdbc dans votre module core

On va d’abord s’occuper de récupérer la liste des invoices en BDD, voir méthode list()

Dans InvoiceRepositoryDatabase (penser à rajouter des constructeurs dans votre entity Invoice)

Ensuite getById

Ensuite createInvoice