**INFORMATIONS :**

On étudie les design patterns, et la compréhension des concepts est ce qu'il y a de plus important dans ce module (examen oral).

ATTENTION : Tout ne va pas être dit dans le cours. Nous devons faire des recherches. L'activité en classe (10% de la note) consistera en l'explication d'un pattern après une préparation de 30-40 minutes par exemple. On doit rendre 4 TP sur 5 pour pouvoir se présenter à l'exam.

Pour la présentation, on ne cherche pas à comprendre ce que fait une application, mais l'architecture/implémentation. On trouve une application qui implémente "**NFR**" (performance, disponibilité, modificabilité, survivabilité), et on doit expliquer l'archi de haut/bas niveau. On doit aussi montrer un **design pattern** qui est utilisé dans l'application.

Une autre activité classe notée consistera en le fait d'assister aux sessions DevOps du 12/13 mai et prendre des notes : promo code : HEG2250StudentsDevOpsDays

Le CC sera sur le contenu des TP.

**COURS :**

On traire l'architecture de haut comme de bas niveau. Si on réalise un nouveau design, on doit considérer les éléments de la NFR mentionnés plus hauts.

L'architectural tactic est une stratégie de design, alors que l'architectural pattern c'est un style. Ce sont deux concepts très importants. Les "tactics" engendrent une architecture de haut niveau, et rajouter le design pattern fait passer en bas niveau (beaucoup plus détaillé).

Une tactique pourrait être une amélioration de l'efficacité du programme (en utilisant un meilleur algorithme), la cadence des évènements (pour les besoins de computation), l'utilisation du cache, etc. On devra traiter les tactiques de l'application choisie dans la présentation.

Comme technique pour la disponibilité, il y a typiquement le ping/echo, le heartbeat (log du système chaque milliseconde).

La modificabilité est un gros problème. Il faut s'assurer que les composants et leurs dépendances soient au point. Il faut augmenter la cohésion dans un composant et diminuer le couplage. Un changement dans le programme ne doit pas affecter l'intégrité du programme.

Ce qu'on appelle l'architecture, ce sont les décisions de design du système. À la base du système, il y a un architecte et le concept du logiciel. L'architecture n'est pas une phase de développement, il y a besoin d'un architecte pendant toute la vie du projet. Quand on crée une architecture, on commence par les NFR, on utilise les tactiques pour créer une architecture de haut-niveau puis les design pattern pour le bas-niveau.

Les principes de design sont notamment l'abstraction (rassemblement du QUOI (NFR), du COMMENT (design) et du POURQUOI), l'encapsulation et la modularisation.

*MVC (model-view-controller)* est l'un des patterns les plus connus. Il permet de résoudre plusieurs problèmes de conception et difficulté (vues multiples/synchrones par exemple) mais a comme répercussions la complexité et le couplage 🡪 *voir diagrammes sur les slides.* Le modèle est responsable du processing des données, la vue de l’output pour l’utilisateur, et le contrôleur de l’input et des appels associés à ces inputs.

Si on est développeur de backend et qu'on doit créer la documentation des API (par exemple pour ceux du frontend pour savoir comment connecter), on peut utiliser *l'API console documentation*.

Une API, c'est un connecteur. Si on crée une nouvelle plateforme, on doit créer une API pour connecter les sources externes à la plateforme. En Python, on utilise Django pour créer une API pour les services web.

Le problème des systèmes, c'est qu'ils sont très complexes. On crée donc une interface (appelée *facade*), alors laquelle les clients peuvent se connecter. C'est aussi utile pour cacher l'information ou pour améliorer la lisibilité par exemple. Parfois ce pattern est combiné à un proxy.

On monte parfois son architecture en couches. Cela donne une meilleure structure. Si on a beaucoup de choses à montrer, on a besoin d'une couche de présentation. Dans chaque couche, on a des sous-couches. Choisir le nombre de couches est important, c'est un équilibre à trouver mais généralement on n'en veut pas beaucoup. L'interface fonctionne comme une façade pour chaque couche, et chacune d'elles utilise des technologies différentes.

On a des composants, des objets et des modules dans un logiciel. Quand on crée une architecture, on doit d'abord créer le composants (classe, interface, …). Il y a de l'input et de l'output.

Voici des exemples de technologies (PAS À APPRENDRE PAR CŒUR) :

Une image contenant texte

Description générée automatiquement

Monsieur Behfar a créé une architecture qui propose les meilleurs prix de plusieurs entreprises de vêtements tels que H&M et Zara. Le problème c'est que les API ne sont pas les mêmes, il doit d'abord créer un connecteur qui se connecte à toutes les API. Il a donc créé une facade, ainsi que des composants de types adapter, ainsi que wrapper (qui donne la possibilité de connecter plusieurs API en même temps.

On doit aussi, pour créer une architecture, des vues. Il y a trois possibilités, mais celle qui importe c'est le modèle Kruchten 4+1 : une vue logique (*décompose le système en composants et connecteurs et mappe les fonctionnalités*), une vue de développement (*on mappe le code et les éléments de la vue logique ?)*, une vue de process (*on explique la communication entre les processus*) et la vue physique (*réseau, stockage, etc.)*.

Le concept de statelessness signifie que les requêtes doivent être indépendantes les unes des autres du point de vue du serveur. C'est important car cela permet de partager les requêtes entre différents serveurs et, ainsi, la tactique de "concurrence" des serveurs. **Il est interdit d’utiliser des variables statiques ou globales** dans une architecture « stateless ».

Sur le serveur, on peut avoir des fichiers (par exemple XML), qui sont des ressources. Généralement, on partage la représentation d'une ressource (par exemple une URL). On utilise les hypermédias (lien) pour connecter la représentation des ressources qui définit l'adresse des ressources sur le serveur.

Lorsqu'en tant que client, on navigue sur Amazon, les données nous concernant sont stockées localement dans les cookies (état client). Lorsqu'on ajoute un article dans le panier, en revanche, c'est stocké sur le serveur (état ressource) dans un header avec un token permettant d'identifier un utilisateur.

Ajouter une query dans la représentation de ressource (url) est une possibilité, mais le serveur peut également retourner une deuxième url dans le header.

L'un des problèmes importants est la gestion de longs services où les serveurs ne répondent pas immédiatement : *request/acknowledge/poll & request/acknowledge/callback* :

Poll : une réponse 303 ((see other) url : parce qu'il ne peut pas immédiatement prendre le résultat : ce n'est pas prêt) est retournée immédiatement et le client doit envoyer une autre requête. Puis une réponse 202 (accepté mais pas prêt), puis 200 (lorsque c'est prêt). Pour faire ça, on calcule l'URL de base, on la trim et on rajoute la deuxième url pour retourner au client. Le service est préparé dans la deuxième url. La fonction getBaseUri() est celle qui permet de trim.

Callback : la requête GET contient l'URL où la réponse doit être envoyée, et le serveur envoie directement une réponse 201 (créée) lorsqu'elle est prête avec une requête POST.

Avec le poll, c'est le client qui envoyait beaucoup de requêtes, mais avec le callback, le client n'en a pas besoin.

Dans l'architecture ATOM (formatage de XML), la couche de business et de domaine, c'est la même chose : pour les objets métier. On a une couche de transport (sur le client et le serveur) puis les API et/ou les entrypoints dans la couche de services. La couche de business permet de tout encapsuler, mais on sépare ce qui existe dans cette couche en les annotant dans la couche de transport qui sert à diviser ces couches : *on peut avoir un set dans la couche business mais un subset/un feed qui contient une collection d'objets/quelques entries dans la couche transport*.

Créer une collection de représentations permet de réduire le trafic. La collection, c'est un feed, avec un link "self", et l'entry est un id dans la collection avec un link "alternative". Les représentations sont des des liens, des adresses qui représentent les ressources.

Jaxb est utilisé pour le marshalling/unmarshalling (sérialiser/désérialiser) qui est utilisé pour les transports entre le client et le serveur : *changement en XML, en JSON, etc…* @XmlRootElement permet de changer les objets Java en XML et vice-versa. @XmlAccessType permet de définir quels éléments doivent être transportés, comme les FIELD par exemple avec .FIELD.

*DTO (data transport object)* ne contient que les éléments que l'on souhaite transporter : *l'auteur et le titre d'un livre mais pas tout le livre*. Cela sert aussi à mimiser le trafis sur le réseau. Ce n’est pas un objet métier et ne peut donc pas être annoté.

Dans l'entrypoint, on a un POST et un GET. On définit le path en tête de classe, en tête de méthode, et dans le fichier web.xml de Tomcat (pour le servlet). Dans cette balise <servlet-mapping>, on trouvera une autre <url-pattern> qui contiendra un élément de path qui viendra après l'application pour différencier des autres chemins.

Les classes BookGetClient et BookPostClient de l'exemple servent à envoyer la requête.

*Timeout* : s'il y a un problème avec le réseau, on envoie une requête à partir du client mais ne recevons aucune réponse.

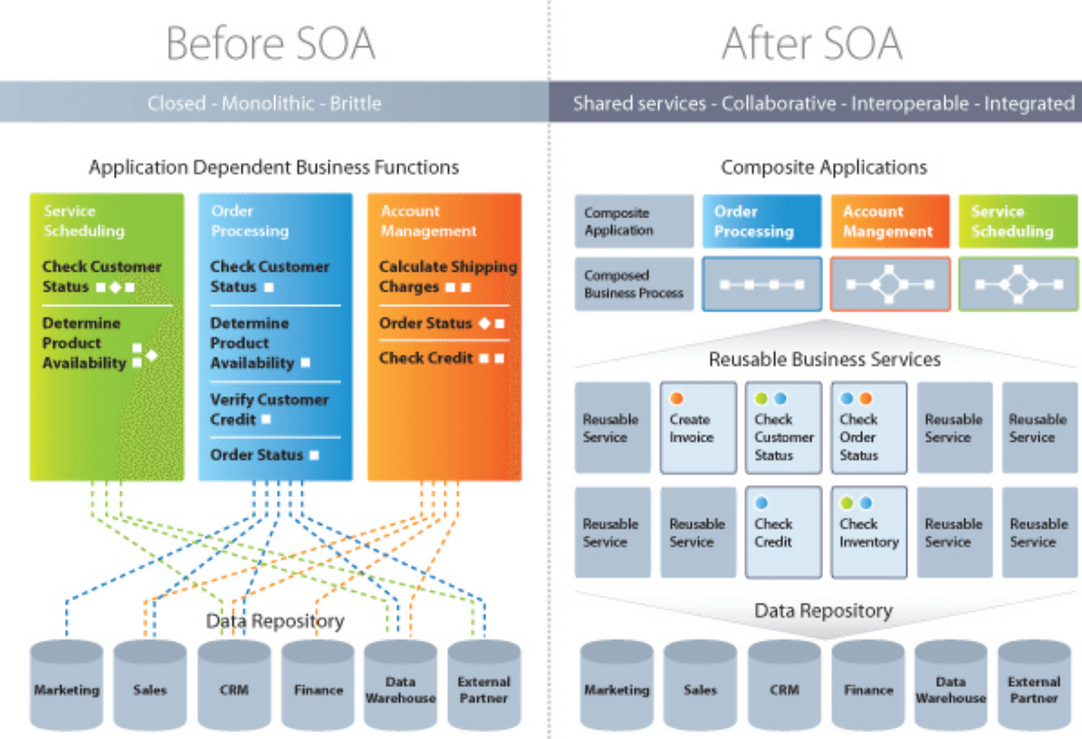
**On peut nous demander les trois timeouts à l'examen** : connection timeout (lorsque le client demande une connexion), connect timeout (jusqu’à ce que la connexion en question soit établie) et socket timeout (attente de données post-connexion). Pour les deux derniers, il y a des exceptions qui sont levées. Le concept est très simple: si la réponse (response time) est moins que le timeout, on continue, sinon, on fait une autre procédure (principe de fault tolerance). RequestConfig est un objet utilisé pour setter les timeouts.

*Circuit breaker* est utilisé dans les mêmes cas de figure : on envoie une requête mais la ressource est indisponible ou il y a une erreur sur le réseau. Le principe du timeout, c'est qu'on utilise une URL2 (fault tolerance) à la place de l’URL1, mais avec le pattern Circuit Breaker, c'est différent : on arrête complètement d'envoyer la requête sur le serveur. C'est le même concept que dans les circuits électroniques : si la ressource est indisponible : on ouvre le circuit. Sinon, on le ferme.

En termes de code, c'est le même principe que pour le timeout, mais au lieu d'un try/catch, on fait une boucle où on essaie 3 fois d'atteindre la ressource et on ouvre/arrête le breaker si ça ne marche pas.

Cela clôt les resource patterns. Ils interagissent avec la ressource. En fonction du pattern implémenté, la réponse/le comportement est différent.

On passe donc au chapitre *"service composition" : SoA (service-oriented architecture).* Supposons que dans une entreprise on ait des applications qui contiennent tous des business services (*service scheduling, order processing, etc.*) qui interagissent tous avec des BDD différentes. C'est une architecture dite monolithique qui n'est que très peu scalable. La différence que SoA apporte est une couche supplémentaire pour y encapsuler les business services. Les applications peuvent ainsi être développées indépendamment les unes des autres et sont donc évolutives. La SoA est donc basée sur les microservices, mais le niveau de granularité est moins fin que dans une architecture pure de microservices. Elle ajoute un nouveau critère de qualité (QA) extrêmement important : l'adaptabilité. Cela signifie que si on a un changement dans l'architecture, elle doit être adaptable/scalable : *si l'architecture est monolithique et un développeur part, le projet est très fortement compromis, c'est pourquoi il faut une architecture suivant SoA, notamment quand plusieurs personnes travaillent dessus.*



**Attention** **: ces différences (mono/SA/microservices) ont de forte chance de constituer une question d'examen.**

Supposons le processus d'immatriculation d'étudiants. On crée une app qui contient quelques services de métier. On a besoin d'un orchestrateur pour appeler tous les services. On pourrait faire ça avec une classe manager (orchestration), mais on peut aussi le faire avec un bus de service (comme pour publish/subscribe : choreography) où les services sont strictement indépendants (losely coupled) et doivent "s'inscrire" aux events adéquats en communiquant avec le bus.

**Attention : on peut nous demander la différence entre orchestration et choreography.**

Dans l'architecture de microservices, on sépare le modèle de la base de données. Cela est nécessaire pour avoir une architecture déployable, scalable. Ce n'est toutefois pas toujours possible.

Supposons que la BDD d'une banque soit actualisée (write) 2-3 fois par jour mais que des requêtes de clients y accèdent (read), toute la journée. Ce sont des configurations très différentes, et read est un service et write également, qui peuvent tous deux demander des microservices complètement différents. *CQRS* est un pattern de microservice, qui divise ces deux services write/read et leur donne des accès et représentations différents. Plus globalement, il vise à éviter du couplage non désiré entre des services. On y trouve une base de données sur laquelle on effectue la mise à jour (master), et ces changements vont être utilisés pour une base de données en lecture (slave). Les permissions pour les personnes qui lisent les données sont différentes que celles de ceux qui les mettent à jour.

*Si deux personnes achètent en même temps (plusieurs services de read) un billet de cinéma pour la même place, il y a un fichier de texte qui prend et garde la séquence d'évènements et où on peut la lire, ceci pour éviter les conflits. C'est le concept d'event sourcing.*

Dans la démo CQRS, on a un write service qui vient écrire les livres dans un fichier texte. Une fois que l'on peut lire, on rajoute un DTO (*data transfer object*). Rappelons que nous ajoutons une couche de transport pour y mettre toutes les annotations Jaxb ((un)marshalling : formatage acceptable pour le transfert tel que xml, json, csv) au lieu de dans la couche business. De plus, si par exemple on a 100 attributs et qu'on veut en transférer 10. C'est dans le DTO qu'on décide ce qu'on marshall/unmarshall.

*PubSubHub* : lorsqu'on utilise le reader précédemment mentionné, le writer est bloqué est vice-versa. On ajoute donc un service hub par lequel tout passe/où tous les services sont enregistrés, et tout fonctionne de façon asynchrone.

On peut avoir, par exemple en Java, plusieurs processus qui tournent en même temps, soit sur une machine, soit plusieurs. On utilise dans tous les cas le multithreading, et on associe chaque service/tâche à un thread. On peut implémenter une classe de thread, soit on étend l'interface Thread/Runnable. Supposons qu'on ait deux URLs. On peut appeler deux threads différents pour effectuer les requêtes qui sont alors synchrones. Pour ce faire, on implémenterait un thread spécifique aux clients (*clientThread*).

**Pour l'examen final, on pourrait nous demander d'expliquer une architecture, de donner les différentes entre un pattern ou un autre, *pourquoi on utilise CQRS, pourquoi on a mis un service hub ici, pourquoi…* Pour le CC, c'est sur les TP et il y aura une question de concept.**

**RETOURS PRESENTATIONS :**

On doit parler de QA, et le/les design pattern(s) doi(ven)t être basé(s) sur les NFR/QA (on doit les expliquer).

Pour l'architecture de haut-niveau, cela doit être basé sur les NFR également.

Les designs patterns doivent être des patterns de bas niveau idéalement et doivent être inclus dans l'architecture.

La définition/présentation de l'application n'est pas importante dans ce cours. On doit aussi implémenter les QA (en plus des patterns) et de ce qui est marqué sur le plan.

**LABO** :

D'abord exécuter le service, puis le client en tant que Java Application.

TP2 : d'abord, le code est fermé, on indique son statut. On teste et retourne 4 fois le CircuitBreakerException. Si le retour n'est pas 200, on sait que ça ne marche pas donc on peut open le circuit. On compte le nombre de fois où l'exception est retournée. Au 5ème appel, il ouvre à moité est un timeout est appelé. S'il y a 3 timeouts (compteur aussi), on reset le circuitBreaker à ouvert. Si on obtient 200 (ça marche), on ferme le circuit et on a terminé. Il faut coder la méthode execute du circuitBreaker. On devrait avoir un 404 en lançant le serveur car la ressource n'est pas disponible. Il n'y a aucune API ou service sur le serveur (c'est le rôle du client). Quand on choisit la classe pour exécuter le client en tant que Java Application, on sélectionne la classe qui contient le main (ici, WaitingClient)

Début de code :

* récupérer l'uri
* récupérer le path
* checker si l'uri existe