ISA-DEVOPS – Contrôle écrit

- Date : avril 2024
- Durée : 3 heures
- Aucun document autorisé ; barème donné à titre indicatif.

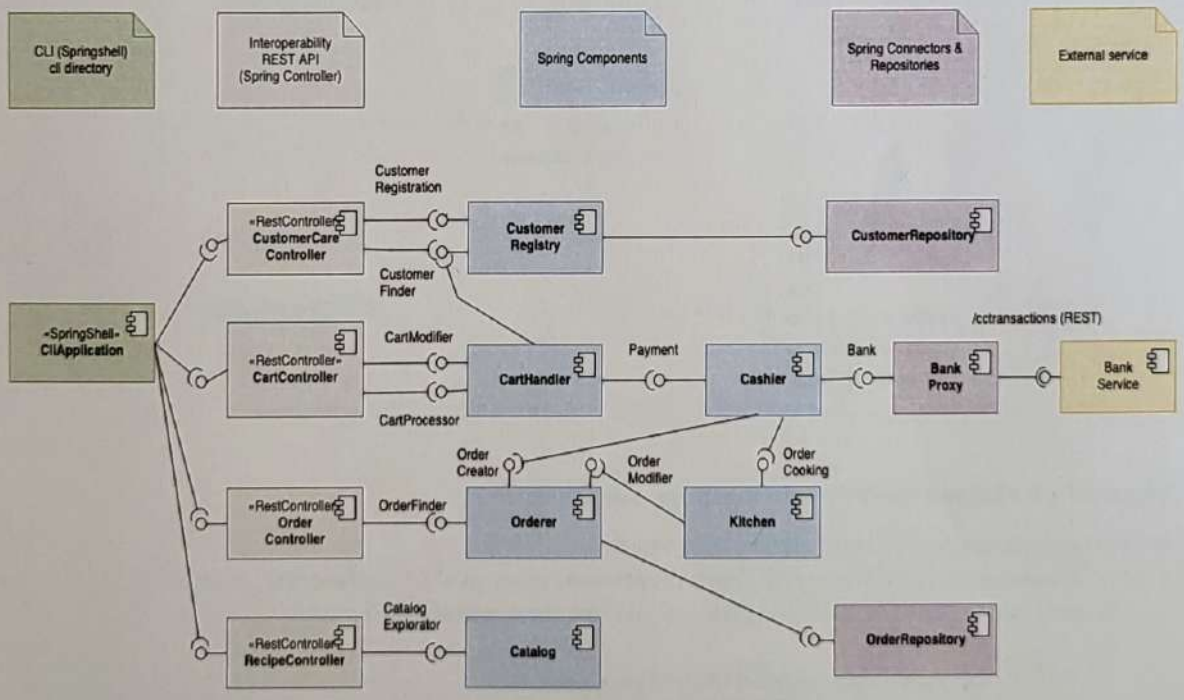
Lisez tout le sujet avant de commencer à répondre aux questions ; les questions sont identifiées en gras dans le texte ; les différentes parties sont indépendantes.

En cas de doute ou d'inconnues, posez vos hypothèses en tête de votre réponse.

Toute fraude identifiée sera systématiquement transmise au conseil de discipline de l'Université

ISA Partie #1 : Principes appliqués à TheCookieFactory /9

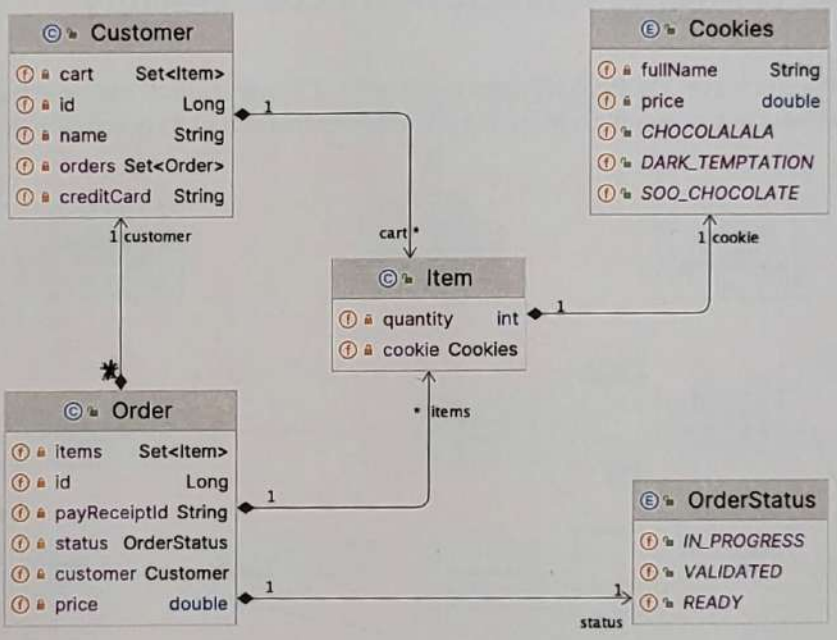
On considère l'application TheCookieFactory qui sert d'illustration à cet enseignement. Vous trouverez ci-dessous le schéma d'architecture, la liste des interfaces et le modèle des entités.



Functional interfaces

To deliver the expected features, the coD system defines the following interfaces:

- `CartModifier` : operation to modify a given customer's cart, by updating the number of cookies in it, and to retrieve the contents of the cart
- `CartProcessor` : operations for computing the cart's price and validating it to process the associated order;
- `CustomerFinder` : a *finder* interface to retrieve a customer based on her identifier (here simplified to her name);
- `CustomerRegistration` : operations to handle customer's registration (users profile, ...)
- `CatalogueExploration` : operations to retrieve recipes available for purchase in the CoD;
- `OrderCreator` : operations to create an order from the customer;
- `OrderFinder` : a *finder* interface to retrieve an order based on its identifier, to follow its status;
- `OrderModifier` : operations to modify the order's status
- `OrderCooking` : operations to process an order (kitchen order lifecycle management);
- `Payment` : operations related to the payment of a given cart's contents;
- `Bank` : operations that act as proxies to the external bank service.



Répondez à chaque question de manière synthétique :

```
@PostMapping(consumes = APPLICATION_JSON_VALUE)
public ResponseEntity<CustomerDTO> register(@RequestBody @Valid CustomerDTO cusdto) {
    // Note that there is no validation at all on the CustomerDto mapped
    try {
        return ResponseEntity.status(HttpStatus.CREATED)
                .body(convertCustomerToDto(registry.register(cusdto.name(), cusdto.creditCard())));
    } catch (AlreadyExistingCustomerException e) {
        // Note: Returning 409 (Conflict) can also be seen a security/privacy vulnerability, exposing
        return ResponseEntity.status(HttpStatus.CONFLICT).build();
    }
}
```

1. L'extrait de code ci-dessus utilise le pattern DTO (*Data Transfert Object*). Expliquez son principe, et donnez selon vous son principal avantage et son principal inconvénient.

2. Dans les extraits des classes Customer et Order données, expliquez le mapping de relation entre Order et Customer en détaillant quels annotations et attributs d'annotation permettent de faire le lien.

3. Dans l'annotation @OneToMany de la classe Customer, expliquez le comportement obtenu par l'attribut fetch défini à LAZY. Expliquez aussi quelles sont les implications vis-à-vis du code dans une transaction et hors d'une transaction.

4. Dans l'annotation @OneToMany de la classe Customer, expliquez le comportement obtenu par l'attribut cascade défini à REMOVE.

```
@Entity
@Table(name= "orders")
public class Order {

    @Id
    @GeneratedValue
    private Long id;

    @ManyToOne
    @NotNull
    private Customer customer;

    @ElementCollection
    private Set<Item> items;

    @Positive
    private double price;

    @NotBlank
    private String payReceiptId;

    @Enumerated(EnumType.STRING)
    @NotNull
    private OrderStatus status;
```

```
@Entity
public class Customer {

    @Id
    @GeneratedValue
    private Long id;

    @NotBlank
    @Column(unique = true)
    private String name;

    @Pattern(regexp = "\\d{10}+", message = "Invalid creditCardNumber")
    private String creditCard;

    @OneToMany(cascade = {CascadeType.REMOVE}, fetch = FetchType.LAZY, mappedBy = "customer")
    private Set<Order> orders = new HashSet<>();

    @ElementCollection
    private Set<Item> cart = new HashSet<>();
```

5. Le code ci-dessous est extrait du composant CartHandler qui implémente les interfaces CartModifier et CartProcessor. Expliquez le comportement de @Transactional sur la méthode *validate* en détaillant en particulier quels objets sont effectivement mis à jour dans la base de données.

```
@Override
@Transactional
public Order validate(Long customerId) throws PaymentException, EmptyCartException, CustomerIdNotFoundException {
    Customer customer = customerFinder.retrieveCustomer(customerId);
    if (customer.getCart().isEmpty())
        throw new EmptyCartException(customer.getName());
    Order newOrder = payment.payOrderFromCart(customer, cartPriceFromCustomer(customer));
    customer.clearCart();
    return newOrder;
}
```

6. Dans le code ci-dessous extrait du composant Cashier qui implémente l'interface Payment, expliquez pourquoi il est important de placer l'attribut propagation de l'annotation @Transactional à la valeur MANDATORY.

```
@Override
@Transactional(propagation = Propagation.MANDATORY)
public Order payOrderFromCart(Customer customer, double price) throws PaymentException {
    String paymentReceiptId  = bank.pay(customer, price).orElseThrow(()
            -> new PaymentException(customer.getName(), price));
    Order order = orderer.createOrder(customer, price, paymentReceiptId);
    return kitchen.processInKitchen(order);
}
```

ISA Partie #2 : Étude de cas étendant TheCookieFactory /4

On considère l'extension suivante du projet TheCookieFactory en partant de la version qui vous était fournie :

- Chaque commande de cookies fait gagner des points de fidélité lorsque la commande est payée (on peut imaginer un gain de 1 point par tranche de 10 euros comme règle du MVP, mais de toute façon, il ne vous est pas demandé d'écrire de code métier).
- Les cookies peuvent être achetés individuellement avec des points (chaque cookie a une valeur en points potentiellement différente).
- Lorsqu'on achète un cookie dans une commande, on le paie intégralement en argent ou en points. On peut, en revanche, faire une commande dont certains cookies sont payés avec des points et d'autres avec de l'argent.
- Les points utilisés dans une telle commande sont évidemment retirés du compte une fois la commande validée et cette commande fera uniquement gagnée des points sur le montant payé en argent.

Vous devez proposer une extension de l'architecture TheCookieFactory.

Si vous ne vous rappelez pas exactement les signatures de méthodes dans les interfaces et les routes REST de TheCookieFactory, définissez-les et considérez-les comme des hypothèses de travail.

1. Identifiez les différents composants (composants métier, contrôleurs, repositories) à mettre en jeu dans votre architecture ;
   - **Décrivez les modifications de l'assemblage** sous la forme d'un diagramme de composants partiels. Vous pouvez ne montrer que les modifications ou un sous-ensemble impacté des composants à votre convenance.
   - **Décrivez les interfaces modifiées ou nouvelles** des composants (signatures typées des interfaces en pseudo-code).
   - **Décrivez les routes REST modifiées ou nouvelles** servies par les contrôleurs (donnez juste la route avec ses paramètres, pas d'annotation Spring ou d'autres définitions lourdes).
2. Identifiez les objets métiers **nouveaux ou modifiés** au sein de votre architecture ;
   - Décrivez ces objets sous la forme d'un diagramme de classes. Vous pouvez ne montrer que les modifications ou un sous-ensemble impacté des entités.
3. **Justifiez vos choix de conception** pour cette architecture : il s'agit d'argumenter pourquoi vous avez découpé ou étendu les composants de la manière présentée (qu'est-ce que cela permet en termes de découpage des traitements, des responsabilités, etc.), en quoi les interfaces permettent de bien gérer le métier et les responsabilités, en quoi les objets métiers que vous proposez permettent de capturer et structurer les données importantes du métier dont le comportement est géré par les composants.

DevOps Partie #1 : Principes /3

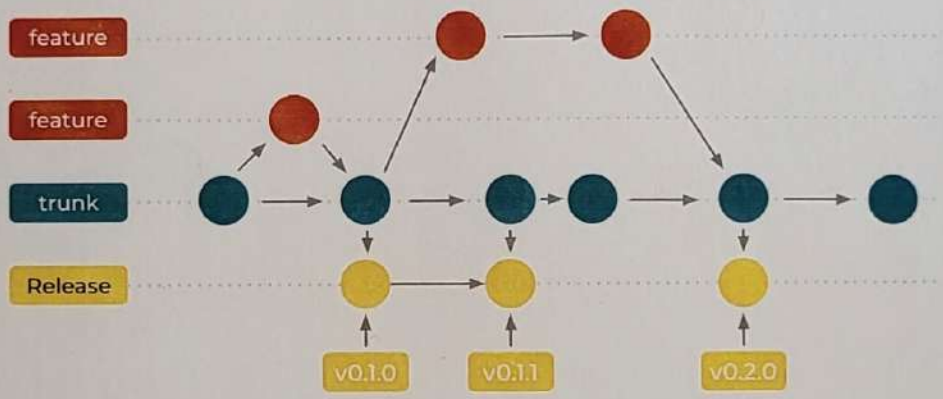
Répondez à chaque question de manière synthétique :

1. Donnez trois types de test (on parle des catégories de test et pas d'une implémentation en particulier liée à une technologie. Des exemples peuvent cependant être donnés), la définition associée et la potentielle priorité entre eux.

2. L'acronyme CI/CD est largement utilisé dans un contexte « DevOps ». À quoi correspond-il ? Donnez une définition distincte pour chacune des deux parties.

DevOps Partie #2 : Étude de cas /4

Le trunk-based development est une méthode de développement logiciel où tous les développeurs travaillent sur une seule branche principale (le tronc commun ou « trunk »). Contrairement aux approches où chaque fonctionnalité ou tâche est développée sur sa propre branche dont la durée de vie est longue et dont l'intégration se fait sur une branche « develop », cette méthode incite les développeurs à créer des branches *feature* qui durent peu de temps, avec peu de commits, et à merger très fréquemment de petits changements sur la branche principale. Afin de déployer une nouvelle version en production, l'équipe peut tirer du trunk une branche de type *Release*, taguée avec le nouveau numéro de version.



L'objectif de cette méthode est d'obtenir plusieurs avantages :

- Peu de conflits de merge et lorsqu'il y en a, ils sont faciles à résoudre.
- Correction de bugs facilitée car les PR (Pull Requests) sont petites et il est donc plus facile de trouver l'origine d'un bug.
- Fluidité même quand la taille de l'équipe et la complexité du code augmentent.

Le trunk-based development est indissociable d'une approche DevOps par CI/CD. En prenant une architecture git/jenkins/artifactory/docker , **expliquez** comment vous adapteriez toute la chaine de CI/CD pour mettre en œuvre cette méthode de développement et assurer la qualité de manière automatisée. Votre réponse doit couvrir :

- Les évènements sur le git (push, PR, tag).
- Les types de test enclenchés à quel moment.
- La reconstruction et le stockage d'artefact.
- La reconstruction et la publication d'images Docker.