Knowledge and questions for backender candidates

Backend senior greenfield project

Knowledge (bold lines are mandatory)

- **Ability to express complex ideas about previous projects**
- **Commitment, ability and aptitude for teamwork**
- **SOLID principles and clean code**
- **Experience with Java 8**
- **Strong knowledge Unit testing (JUnit, Mockito)**
- Static code analysis tools (Sonar, PMD, Checkstyle, etc)
- **Design patterns**
- **TDD**
- Knowledge of operating systems (Linux)
- **Git** and shell bash
- **Agile methodologies**
- **Spring, Maven**, Gradle
- Microservice architecture (service discovery, distributed traceability, APIs composition, ...)

Nice to have

- DDD
- Side-projects
- Jenkins, Ansible, Docker
- Experience with Akka, Spring Boot
- Experience with relational databases and NoSQL

Questions

- What's new in Java 8? Explain some of them.
    - *Lambda expressions, Java Stream API for bulk data operations on collections, package Java Time ...*
    - *Lambda expressions:*
        - *Functional programming uses lambda expressions to write code.*
        - *A lambda expression is a block of code that gets passed around. It has parameters and a body just like methods do, but it doesn't need a name.*
        - *Syntax:*

          parameter -> {body}
          (parameter1, parameter2) -> {body}

        - *Lambda expressions are anonymous methods and they work with interfaces that have only one method. These are called functional interfaces, and they can be used with functional programming. Optionally, these interfaces can be annotated like @FunctionalInterface.*

    - *Java Stream API:*
        - *It has been added to perform filter/map/reduce like operations with the*

        collection.
    - *There are two types of execution: sequential and parallel.*
        - *stream() → sequential stream*
        - *parallelStream() → parallel stream (parallel processing values are not in order).*
- Given the following list implement a solution in order to get even numbers using Java 8 Streams

```
List<Integer> list = Arrays.asList(1,2,3,4);
Stream<Integer> evenList = list.stream().filter(val -> val % 2 == 0);
```

- What do you notice when you do code review?
    - *When I review the code, I check that it doesn't have errors, and that the implemented code is clean and understandable. Also, I double-check that the code is as optimized as possible.*

- Have you ever worked with Scrum? Tell us what it is, what events do you remember and what roles are involved?
    - *Yes, I have worked with Scrum.*
    - *Scrum is a work methodology for agile software development.*
    - *Events: Daily, Sprint Planning, Sprint Review, Sprint Restrospective, Refinement*
    - *Roles: Product Owner, Scrum Master, Business Analyst, Development team.*

- What access modifiers (or visibility) do you know in Java?
    - *public: the method can be called from any class.*
    - *private: the method can only be called from within the same class.*
    - *protected: the method can only be called from classes in the same package or subclasses.*
    - *Default: the method can only be called from classes in the same package. This one is tricky because there is no keyword for default access.*

- Differences between an abstract class and an interface. When would you use one or the other?

→ *We would use an abstract class when we want to share code among several related classes. We expect that classes that extend your abstract class require access modifiers other than public or have many common methods or fields.*

→ *We would use an interface when we need that unrelated classes would implement our interface. We define the methods but we don't implement them.*

→ *Differences:*

- *An abstract class doesn't allow multiple inheritance and an interface allows multiple inheritance.*
- *An abstract class can have abstract and non-abstract methods and an interface can have only abstract methods.*
- *An abstract class can supply the implementation of an interface and an interface can't supply the implementation of an abstract class.*
- *The "abstract" keyword is used to define an abstract class and the keyword "interface" is used to define an interface.*
- *An abstract class can be extended using keyword "extends" and*

an interface can be implemented using keyword "implements".
    - *An abstract class can extend another Java class and implement multiple Java interfaces and an interface can extend another Java interface only.*
    - *An abstract class can have class members like private, protected, etc., and members of an interface are public by default.*

- What is Maven and why is it used? What is Maven life cycle?
    - *Maven is a software tool to manage and to build Java project. It uses a file called Project Object Model (POM) to describe the project, its dependencies on other external modules and components, and the order of building of elements.*
    - *Maven life cycle is a sequence of phases, to define the order in which the goals are to be executed. It has three built-in build lifecycles: default, clean and site.*

- What is Git and what is it used for? List all Git commands that you know.
    - *Git is a version control software to manage projects.*
    - *Git commands: git init, git clone, git add (dir), git commit –a –m, git checkout, git remote –v, git fetch <remote>, git push, git push origin, git branch, git merge*

- What is a mock? What would you use it for?
    - *A mock object is an object that simulate a certain behavior of a class or an interface. In this way, we can perform an unit test and validate that the class reacts correctly.*
    - *I would use it to perform unit tests and to check my software.*

- How would you explain to someone what Spring is? What can it bring to their projects?
    - *Spring is a framework for dependency injection and allows building a Java application faster and decoupled systems.*
    - *Spring can bring low coupling, modularity, scalability, and it helps to avoid dependency between classes. It allows to have a cleaner and optimized code.*

- What's the difference between Spring and Spring Boot?
    - *Spring Boot is not considered a framework as such, it is an extension. In the same way as Spring, it helps in the creation of projects but in a more agile way, and saving us from having to waste time in carrying out heavy configurations on files.*

- Do you know what CQRS is? And Event Sourcing?
    - *CQRS is an architecture pattern that considers separately the models for reading and writing data.*

- Differences between IaaS and PaaS. Do you know any of each type?
    - *IaaS → infrastructure as a service, for the management and administration. (Example: Amazon Web Services).*
    - *PaaS → platform as a service, to only worry about the construction of the app.*

- Explain what a Service Mesh is? Do you have an example?
    - *It is a dedicated software infrastructure to handle communication between microservices.*
- Explain what is TDD? What is triangulation?
    - *TDD: Test-Driven development. It is a development practice that consists of first writing the tests, then writing the code that passes the test correctly and, finally,*

refactoring code.
  - *Triangulation: It is an implementation strategy that consists of first choosing the simplest case, then applying TDD and finally, repeating the previous steps with different cases.*

- Apply the Factory pattern with lambda expressions

- Reduce the 3 classes (*OldWayPaymentStrategy*, *CashPaymentStrategy* and *CreditCardStrategy*) into a single class (*PaymentStrategy*). You do not need to create any more classes or interfaces. Also, tell me how you would use *PaymentStrategy*, i.e. the different payment strategies in the Main class

```
public interface OldWayPaymentStrategy {
    double pay(double amount);
}

public class CashPaymentStrategy implements OldWayPaymentStrategy {

    @Override
    public double pay(double amount) {
        double serviceCharge = 5.00;
        return amount + serviceCharge;
    }
}

public class CreditCardStrategy implements OldWayPaymentStrategy {

    @Override
    public double pay(double amount) {
        double serviceCharge = 5.00;
        double creditCardFee = 10.00;
        return amount + serviceCharge + creditCardFee;
    }
}
```

→ I create the interface with the common method pay, and later, in the Main, for example, for an amount equal to 100.00, the operations are performed for the cases CashPaymentStrategy and CreditCardStrategy

```
public interface PaymentStrategy {

      //write here your solution
      //method pay
      public double pay (double amount);
}

public class Main {

    public static void main(String[] args) {

      double serviceCharge = 5.00;
      double creditCardFee = 10.00;
      //example amount = 100.00
      double inputAmount = 100.00;
      System.out.println("CashPaymentStrategy, pay: " + pay(amount ->
amount + serviceCharge, inputAmount));
```

```
            System.out.println("CreditCardStrategy, pay: " + pay(amount ->
amount + serviceCharge + creditCardFee, inputAmount));
          }

          private static double pay(PaymentStrategy payment, double
inputAmount) {
            return payment.pay(inputAmount);
          }
      }
```