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.

Default implementation for interface methods: you can implement a method (using default reserved keyword) on an interface so that every subclass will inherit this default implementation.

Java Lambdas and Streams API: giving the possibility of programming using lambdas and iterating over collections using the different Streams methods.

A new Garbage Collector.

New Date classes: LocalDate, LocalTime, LocalDateTime, to manage times and dates.

• 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);

list.stream().filter(i -> i % 2 == 0).collect(Collectors.toList());

• What do you notice when you do code review?

You can notice bugs, code issues, but also code smells, bad programming practices (in terms of clean code, performance and/or security), missing tests, etc….

• Have you ever worked with Scrum? Tell us what it is, what events do you remember and

what roles are involved?

Yes I’ve been working with Scrum (or similar) most time in my career:

Roles:

* Product Owner (or even Proxy product owner): this role is the responsible of letting the developers team know what’s needed and its priority.
* Scrum Master: manages all the scrum events and tools to help the team with the organization until the team becomes autonomous.
* The rest of the team: developers, testers, and any other… Take part in every meeting.

Events:

* Daily: short meeting (should last 15 minutes aprox.) in which every developer (or any other team member) share with the rest of the team what he did the previous day, what’s he doing actually, possible blockings, doubts, …
* Sprint Planning: first meeting that marks the beginning of a sprint. In this sprint the stories to be developed and the goals to be achieved are set.
* Sprint Refinement: Before the sprint planning, it is required that every story has all the information required and has a detailed definition. This event is just to achieve this for every story. Also for voting the story points (sometimes done in the planning) if using Scrum Poker Estimation.
* Sprint Review: Last meeting for every sprint. A summary of what’s done and delivered and what not is done. It is possible also to make a demo about every delivered functionality.
* Sprint Retrospective: In this optional meeting/event the idea is to take a look to the previous sprint (the last completed) to think in common (the whole team) about what the team did good and what could be improved.

• What access modifiers (or visibility) do you know in Java?

Public: public for all classes

Private: private for all clases.

Protected: public for the same package and subclasses, private for the rest.

Default: public for the same package and private for the rest.

• Differences between an abstract class and an interface. When would you use one or the

other?

Both are not instantiable but the main difference is:

* Interfaces are focused to be as contracts, any of its methods can be implemented (don’t think about default methods since java 8 now). A subclass can implement (“inherit” from) one or many interfaces
* Abstract classes are focused for inheritance: so they can have some implemented methods. A subclass can inherit from one and only one abstract class (java does not tolerate multiple inheritance).

When to use any or other? Depending on many considerations, but one key to know which one to use is the number of inheritance (a subclass can implement many interfaces but extend only one abstract class)

• What is Maven and why is it used? What is Maven life cycle?

Maven is a tool that manages the dependencies in your project libraries for you. So you don’t need to dive in the dependency hierarchy (many libraries depend on others, and those on others… creating a huge dependency tree).

The maven life cycle is the “pack” of phases that passes every project building

• What is Git and what is it used for? List all Git commands that you know.

Git is a version control tool. So lets say a tool to manage the code deliver (just in a brief summary).

Some git commands: clone, init, push, pull, merge, commit, checkout, add.

• What is a mock? What would you use it for?

Is a virtualization or simulation of something used (usually) in testing. So for unit tests some mocking tools (like powerMock or Mockito) let you “simulate” another class that its being called inside the class you’re testing. In integration tests there can be other kind of tools to virtualize some services or elements in order to ignore those specifications in the test.

• How would you explain to someone what Spring is? What can it bring to their projects?

A framework aiming to the pattern of IoC (Inversion of Control).

It can bring to your projects some powerful utilities (its too long to explain in such a short question).

• What's the difference between Spring and Spring Boot?

Lets say that Spring Boot is a kind of extension of Spring but adding extra help for the usually required configurations (applications server, dependencies, …)

• Do you know what CQRS is? And Event Sourcing?

CQRS (I don’t remember the complete meaning of every letter) is a pattern that splits the traditional CRUD to the Database in two different services (or microservices): one responsible of reading (Query) and the other responsible of the rest (Write, Delete, Update, Validations, …).

Event sourcing (I just know a bit of theory about this one) is a pattern to ensure every change done on anything is ensured and handled as an event.

• Differences between IaaS and PaaS. Do you know any of each type?

IaaS: Infrastructure as a Service. The Infrastructure (let’s say the hardware, the servers) are given as a service. Amazon Web Services, Microsoft Azure, IBM Bluemix.

PaaS: Platform as a Service. The whole platform (included the infrastructure) is given as a service by a third party. It means, the infrastructure and additional stuff like operating system, configurations, … are given. So the client of this services can ignore all this focusing only in the application development and configuration.

• Explain what a Service Mesh is? Do you have an example?

I don’t know.

• Explain what is TDD? What is triangulation?

TDD is Test Driven Development, so the tests are developed before the functionality in order to ensure that every line of code implemented is required and aimed to make a test work.

I don’t know what’s triangulation.

• Apply the Factory pattern with lambda expressions.

Let’s consider we got an Interface (Vehicle) and three classes implementing it (Car, Truck, Bike).

I would create a factory in which we would populate a Map (in factory constructor or in a static block of code) like this:

Map<String, Vehicle> vehiclesMap = new HashMap<>();   
vehiclesMap.put(“CAR”, Car::new);   
vehiclesMap.put(“BIKE”, Bike::new);   
vehiclesMap.put(“TRUCK”, Truck::new);

So, in the Factory.getVehicle(String vehicleType) method we could have this line:

Return this.vehiclesMap.get(vehicleType);

• 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

I will take the following assumption: the hardcoded numbers (serviceCharge and creditCardFee never change, or almost never) to the following solution (and I don’t like it so much). Also I would make PaymentStrategy a class and not an interface (but it may make sense for future payment methods to keep it as interface) so ill change the pay method to static:

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;

}

}

public interface PaymentStrategy {

public static double pay(boolean paidWithCreditCard, double amount){

double serviceCharge = 5.00;

double creditCardFee = 10.00;

double payment = serviceCharge + amount;

if(paidWithCreditCard)

{

payment += creditCardFee;

}

return payment;

}

}

public class Main {

public static void main(String[] args) {

double amountCash = PaymentStrategy.pay(false,6.00);

System.out.println("This customer paid with cash: " + amountCash);

double amountCard = PaymentStrategy.pay(true,6.00);

System.out.println("This customer paid with credit card: " + amountCard);

}}