**Mastercard Interview Question**

**1 beancontext and applicationcontext. Can we get data from beancontext to application context.**

**2 Same phone no how to validate encrypt**

**3 what will happen if we do not do xss.**

**4 acurator spring boot**

**5 create the pipeline**

**6 SSO configuration**

**7 Bifunction Java 8**

**8 composition and**

**9 Roles**

**10 Jenkins**

**Explain roles and responsibilities**

**Why java is not pure object oriented language.?**

- Java is not fully object oriented because it supports primitive data type like it,byte,long

**Auto boxing and unboxing.**

- Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called unboxing

**Anonymous inner class**

- a class can contain another class known as nested class. It's possible to create a nested class without giving any name.

- A nested class that doesn't have any name is known as an anonymous class.

**Able to answer on approach for creating and testing singleton class.**

- private constructor

- static class

- static method

**Factory**

**-** retrieve shape base on name.

**Design Patterns**

Aware about factory, abstract factory and singleton only.

**Micro Services and Cloud PrincipleMicroservices** help in breaking the boundaries of large applications and build logically independent smaller systems inside the system. E.g. using Amazon AWS you can build a cloud application with minimum effort. ... By doing so, changes to one microservice do not impact others.

Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are. Highly maintainable and testable. Loosely coupled. Independently deployable. Organized around business capabilities.

The Most Complete Platform for Microservices. AWS has integrated building blocks that support any application architecture, regardless of scale, load, or complexity.

**Only learning experience in micro services.**

**Not aware about 12 factor applications.**

**1. Codebase**

One codebase tracked in revision control, many deploys

Your codebase should be tracked in a central version control system that is easily accessible to all your developers. We prefer to use Git and share all our repositories through GitHub.

**2. Dependencies**

Explicitly declare and isolate dependencies

Not aware about micro service patterns.

Have done PoC for deploying a sample application on PCF.

Security

Configuration

Store config in the environment

Configuration options should never be hardcoded, for two reasons:

Backing Services

Treat backing services as attached resources

A backing service is one that requires a network connection to run, like MySQL or Memcached. If the location or connection details of such a service changes, you should not have to make code changes. Instead, these details should be available in the configuration.

Build, release, run

Strictly separate build and run stages

Build, release, and run stages should be treated as completely distinct from one another:

Processes

Execute the app as one or more stateless processes

Stateless applications are designed to degrade gracefully. That means if a part of your application stack fails, the app itself does not become a failure. In other words, the state of your system is completely defined by your databases and shared storage, and not by each individual running application instance.

**Port Binding**

Export services via port binding

Your application service should also be accessible via a URL, just as your backing services are. This enables your application to be fully self-contained.

**Concurrency**

Scale out via the process model

Every process inside your application should be treated as a first-class citizen.

**Disposability**

* Maximize robustness with fast startup and graceful shutdown
* When you deploy new code, you want the new version to start right away and be able to deal with incoming traffic.

**Dev/prod parity**

* Keep development, staging, and production as similar as possible
* Your development environment should resemble production as closely as possible.

**Logs**

* Treat logs as event streams
* Logging is important for debugging and checking up on the general health of your application. However, your application should not be concerned with the storage and management of these logs. Instead, log entries should be treated as an event stream that is routed to a separate service for archival and analysis.

Examples of such external services are New Relic or Logentries.

**Admin Processes**

* Run admin/management tasks as one-off processes
* Once your application is running in production, you'll want to do a lot of simple administrative tasks from time to time

**Difference between authentication and authorization.**

* Authentication and authorization might sound similar, but they are distinct security processes in the world of identity and access management (IAM).
* Authentication confirms that users are who they say they are. Authorization gives those users permission to access a resource.

**symmetric and asymmetric key.**

* Symmetric encryption uses a private key to encrypt and decrypt an encrypted email.
* Asymmetric encryption uses the public key of the recipient to encrypt the message. Then if the recipient wants to decrypt the message the recipient will have to use his/her private key to decrypt.

https://www.geeksforgeeks.org/difference-between-symmetric-and-asymmetric-key-encryption/

**Digital signature**

* A digital signature is a mathematical scheme for verifying the authenticity of digital messages or documents
* . A valid digital signature, where the prerequisites are satisfied, gives a recipient very strong reason to believe that the message was created by a known sender (authentication), and that the message was not altered in transit

**What is Hashing.**

* Hashing means using some function or algorithm to map object data to some representative integer value.
* This so-called hash code (or simply hash) can then be used as a way to narrow down our search when looking for the item in the map.

**Spring and Spring boot**

**difference between IoC and DI.**

**RDBMS Concepts**

RDBMS stands for relational database management system. A relational model can be represented as a table of rows and columns. A relational database has following major components:

1. Table

2. Record or Tuple

3. Field or Column name or Attribute

4. Domain

5. Instance

6. Schema

7. Keys

**· Normalization**

* Normalization is a database design technique that reduces data redundancy and eliminates undesirable characteristics like Insertion, Update and Deletion Anomalies
* Normalization rules divides larger tables into smaller tables and links them using relationships

**· NoSQL DB types of it.**

NoSQL databases (aka "not only SQL") are non tabular, and store data differently than relational tables. NoSQL databases come in a variety of types based on their data model. The main types are document, key-value, wide-column, and graph. They provide flexible schemas and scale easily with large amounts of data and high user loads.

· Aware about ACID properties.

**Maven:**

- Build life cycle, phases, goal and tasks

Maven Build Lifecycle

**The Maven build follows a specific life cycle to deploy and distribute the target project.**

**There are three built-in life cycles:**

**default: the main life cycle as it's responsible for project deployment**

**clean: to clean the project and remove all files generated by the previous build**

**site: to create the project's site documentation**

**Maven phases**

**validate: check if all information necessary for the build is available**

**compile: compile the source code**

**test-compile: compile the test source code**

**test: run unit tests**

**package: package compiled source code into the distributable format (jar, war, …)**

**integration-test: process and deploy the package if needed to run integration tests**

**install: install the package to a local repository**

**deploy: copy the package to the remote repository**

**Git**

**Aware about Git flow and can explain the same.**

**The overall flow of Gitflow is: A develop branch is created from master. ... When a feature is complete it is merged into the develop branch. When the release branch is done it is merged into develop and master. If an issue in master is detected a hotfix branch is created from master.**

**CI/CD**

**Continuous integration (CI) and continuous delivery (CD) embody a culture, set of operating principles, and collection of practices that enable application development teams to deliver code changes more frequently and reliably. The implementation is also known as the CI/CD pipeline**

**Having experience in creating pipeline in Jenkins.**

**What is a Jenkins Pipeline?**

**Jenkins Pipeline (or simply "Pipeline") is a suite of plugins which supports implementing and integrating continuous delivery pipelines into Jenkins.**

**A continuous delivery pipeline is an automated expression of your process for getting software from version control right through to your users and customers.**

**Jenkins Pipeline provides an extensible set of tools for modeling simple-to-complex delivery pipelines "as code". The definition of a Jenkins Pipeline is typically written into a text file (called a Jenkinsfile) which in turn is checked into a project’s source control repository. [1]**

**For more information about Pipeline and what a Jenkinsfile is, refer to the respective Pipeline and Using a Jenkinsfile sections of the User Handbook.**

**REST –**

**Representational state transfer**

**Representational state transfer (REST) is a software architectural style which uses a subset of HTTP.[1] It is commonly used to create interactive applications that use Web services. A Web service that follows these guidelines is called RESTful. Such a Web service must provide its Web resources in a textual representation and allow them to be read and modified with a stateless protocol and a predefined set of operations. This approach allows interoperability between the computer systems on the Internet that provide these services. REST is an alternative to, for example, SOAP as way to access a Web service.**

**API Call –**

**What is an API Call?**

**Now that we have a gist of what an API is and what it does, let’s switch our attention to an API call. Simply put, the moment you add an endpoint to a URL and send a request to a server, this is what counts as making an API call. For example, when you log on to any app or ask a question via a browser, you are actually making an API call.**

**How routing works**

**Routing Logic**

**How do networked hosts route traffic to its destination? For most hosts, routing comes down to knowing just enough to make it someone else's problem. Hosts do not have complete routing tables describing the entire Internet. They generally know just enough to distinguish between "directly connected", meaning "on the same LAN", and "somewhere else", which makes it some router's problem. Routers know a little more about the topology a few hops away.**

**Observable –**

**Angular makes use of observables as an interface to handle a variety of common asynchronous operations. For example: You can define custom events that send observable output data from a child to a parent component. The HTTP module uses observables to handle AJAX requests and responses.**

**Good attitude to learn**

**Microservice**

**The services embrace micro-level concerns like single responsibility, separation of concerns, modularity, etc.**

**How to breakdown monolith to microservices**

### **Warm Up with a Simple and Fairly Decoupled Capability**

**Starting down a microservices path requires a minimum level of operational readiness. It requires on demand access to deployment environment, building new kinds of continuous delivery pipelines to independently build, test, and deploy executable services, and the ability to secure, debug and monitor a distributed architecture**

### **Minimize Dependency Back to the Monolith**

**As a founding principle the delivery teams need to minimize the dependencies of newly formed microservices to the monolith. A major benefit of microservices is to have a fast and independent release cycle.**

### **Split Sticky Capabilities Early**

* **Session in ecommerce**

### **Decouple Vertically and Release the Data Early**

* **The main driver for decoupling capabilities out of a monolith is to be able to release them independently.**

### **Decouple What is Important to the Business and Changes Frequently**

**Decoupling capabilities from the monolith is hard. I’ve heard** [**Neal Ford**](http://nealford.com/) **use the analogy of a careful organ surgery. In the online retail application, extracting a capability involves carefully extracting the capability’s data, logic, user facing components and redirecting them to the new service.**

### **Decouple Capability and not Code**

**Whenever developers want to extract a service out of an existing system, they have two ways to go about it: extract code or rewrite capability.**

### **Go Macro First, then Micro**

**Finding the domain boundaries in a legacy monolith is both an art and science. As a general rule applying domain driven design techniques to find the** [**bounded contexts**](https://martinfowler.com/bliki/BoundedContext.html) **defining microservices boundaries is a good place to start.**

### **Migrate in Atomic Evolutionary Steps**

**The idea of vanishing a legacy monolith into thin air by decoupling it into beautifully designed microservices is somewhat of a myth and arguably undesirable.**

**Concern to keep in mind while breaking down monoliths to microservices?**

* **More granularity means more moving parts. Refactoring a monolithic application to microservices creates many small components that constantly communicate; the complexity is shifted around to the interconnections between services.**
* **Difficult to find root cause of issue - More Logging**
* **More Failure Points - Failure of service plans**
* **Tracing Performance :-**
* **Operational complexity is also increased due to the increased demands on managing these services and monitoring them. The ability to quickly deploy small independent services is a win for development, but it puts additional strain on operations as half-a-dozen applications now turn into hundreds of little microservices. Coordinating a large number of rapidly changing services necessitates automated continuous integration and continuous delivery**

**How to do centralized logging?**

## **Centralized Logging Using Log Streams**

**Using Log Streams is one way to implement centralized logging. The common way to implement it is to stream microservice logs to a common queue. Distributed logging server listens to the queue and acts as log store. It provides search capabilities to search the trace.**

## **Popular Implementations**

**Some of the popular implementations include**

* **the ELK stack (Elastic Search, Logstash and Kibana) for Centralized Logging.**
* **Zipkin, Open Tracing API, and Zaeger for Distributed Tracing.**

**-Centralize api for log.**

**How to handle async transaction for async services?**

**What kind of testing require for microservices architecture?**

## **Unit Testing for Microservices**

**Microservices are itself build on the notion of splitting the smallest unit of business logic. These services then communicate with each other over a network. Therefore Unit testing is all the more important in this context to validate each business logic aka microservices separately.**

## **Microservice Integration testing**

**Integration testing verifies the communication path and interactions between the components to detect interface defects. How it’s done?**

**Integration test collects microservices together to verify that they collaborate as intended to achieve some larger piece of business logic.**

## **Component Testing in Microservices**

**A component or microservice is a well-defined coherent and independently replaceable part of a larger system. Once we execute unit tests of all the functions within microservices, it’s time to test microservice itself in isolation.**

## **Contract Testing in Microservices**

**When some consumer couples to the interface of a component to make use of its behavior, a contract is formed between them.**

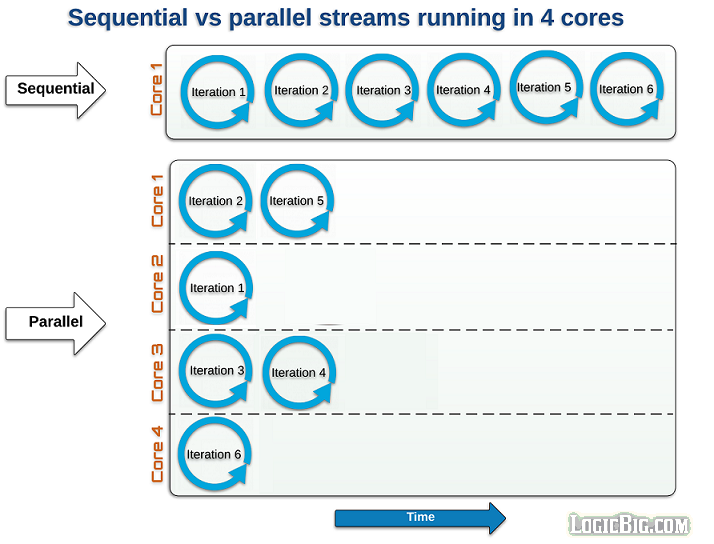
## **End-to-End Testing in Microservices**

**We usually treat the system as a black box while performing end to end tests. Because, of course the intention differs from other tests, it verifies that the system as a whole meets business goals irrespective of the component architecture in use.**

**What kind of automated test you write if your service used by others?**

**Difference between normal and parallel stream? When to use parallel stream?**

* **Parallel streams divide the provided task into many and run them in different threads, utilizing multiple cores of the computer.**
* **On the other hand sequential streams work just like for-loop using a single core.**

****

**import java.time.LocalTime;**

**import java.util.Arrays;**

**import java.util.stream.Stream;**

**public class SequentialParallelComparison {**

**public static void main (String[] args) {**

**String[] strings = {"1", "2", "3", "4", "5", "6", "7", "8", "9", "10"};**

**System.out.println("-------\nRunning sequential\n-------");**

**run(Arrays.stream(strings).sequential());**

**System.out.println("-------\nRunning parallel\n-------");**

**run(Arrays.stream(strings).parallel());**

**}**

**public static void run (Stream<String> stream) {**

**stream.forEach(s -> {**

**System.out.println(LocalTime.now() + " - value: " + s +**

**" - thread: " + Thread.currentThread().getName());**

**try {**

**Thread.sleep(200);**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**});**

**}**

**}**

**How stream are performing better then for each?**

**Solid principles**

* [**Single Responsibility Principle**](https://stackify.com/solid-design-principles/)
* **Open/Closed Principle**
* [**Liskov Substitution Principle**](https://stackify.com/solid-design-liskov-substitution-principle/)
* [**Interface Segregation Principle**](https://stackify.com/interface-segregation-principle/)
* [**Dependency Inversion**](https://stackify.com/dependency-inversion-principle/)

**How will you write test case when someone break open close principles?**

**How to call restservice async?**

**Future<MyResponseObject> future = asyncInvoker.get(MyResponseObject.class);**

***// or***

**Future<MyResponseObject> future = asyncInvoker.get( new InvocationCallback<MyResponseObject>() {**

**@Override**

**public void completed(MyResponseObject o) {**

***// do something with o***

**}**

**@Override**

**public void failed(Throwable t) {**

***// handle the failed request/response***

**}**

**});**

**Enable async**

**Can we have 2 classes as spring boot applications?**

**Can we have same name bean in springboot application?**

**Db design customer give some currency exchange will get some other currency**