**SPRINGBOOT QUESTIONS**

**Springboot properties:-**

**starter dependency, auto-configuration, embedded server, Spring Boot CLI, Spring Actuator, Spring Initializer**

**It removes most of the pain associated with dealing with Spring e.g. a lot of configuration and dependencies and a lot of manual setups.**

**Starter dependency**

**This feature aggregates common dependencies together. For example, if you want to develop Spring MVC based RESTful services then instead of including Spring MVC JAR and Jackson JAR file into classpath you can just specify spring-boot-web-starter and it will automatically download both those JAR files. Spring Boot comes with many such starter dependencies to improve productivity.**

**1- What is springboot?/5- What are the main features of Springboot?/2- Why should I use springboot?**

**Ans:- Springboot is framework that provides easy of development by providing features such as initializer, starter-dependenies,**

**auto-configuration, Spring Actuator. Microwservice development is easy spring boot.**

**a- Initializer:- A web application which can create initial project structure for us and our project will be subproject of this parent project.**

**b- Starter-dependencies:- It is springboot provided maven or gradle artifacts that provides bunch of jars to our project without**

**worying about version conflicts and compatiblities issues. for example if we use spring-boot-starter-web artifact then it will include**

**all jars related spring web, spring restful webservice and jackson jars and we are not worried about version conflicts and compatiblities**

**issues.**

**c- Auto Configuration:- Springboot automatically configures required beans by scanning artifacts in maven or gradle file.**

**What is @SpringBootApplication?**

**What is @SpringBootApplication annotation in spring boot?**

**Many Spring Boot developers always have their main class annotated with @Configuration, @EnableAutoConfiguration and @ComponentScan. Since these annotations are so frequently used together (especially if you follow the best practices above), Spring Boot provides a convenient @SpringBootApplication alternative.**

**The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration and @ComponentScan**

**with their default attributes:**

**The following are the parameters accepted in the @SpringBootApplication annotation:**

**exclude: Exclude the list of classes from the auto configuration.**

**excludeNames: Exclude the list of fully qualified class names from the auto configuration. This parameter added since spring boot 1.3.0.**

**scanBasePackageClasses: Provide the list of classes that has to be applied for the @ComponentScan.**

**scanBasePackages Provide the list of packages that has to be applied for the @ComponentScan. This parameter added since spring boot 1.3.0.**

**3- Proc and cons of Springboot?**

**4- @Configuration vs @SpringBootApplication**

**5- How to configure web server for springboot**

**Ans: Spring Boot web applications include a pre-configured, embedded web server by default. We can modify the default configuration as per our requirements using application.properties**

**server.port=80**

**server.address=my\_custom\_ip**

**server.error.whitelabel.enabled=false**

**server.tomcat.max-threads=200**

**server.connection-timeout=5s**

**6- How to configure Jdbc driver in springboot.**

**#==== connect to mysql ======#**

**## MySQL**

**spring.datasource.url=jdbc:mysql://192.168.1.4:3306/test**

**spring.datasource.username=mkyong**

**spring.datasource.password=password**

**# Oracle**

**spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl**

**spring.datasource.username=system**

**spring.datasource.password=Password123**

**1- What Is Microservices?**

**ANS: Microservices is an architectural style that enable us to sperate monolithic application into autonomous services(sup projects).**

**These services(subprojects) communicate with each other and work autonomously. If one service is down then other services will**

**also function. we are able to deploy each service independently. we can use different technologies to create other servicea(subprojects).**

**for example one micserviceservice in Java, other is on DotNet and another one is in Python. The size of team of one microservice should**

**be 3 to 6 employees.**

**Spring cloud is an Integration software that integrates with external systems. It allows microservices framework to build applications which perform restricted amounts of data processing. We can develop microservices using Spring Boot but they are all stand alone applications. But suppose we now have to connect the various applications and build a distributed system then we face complexities such as Latency overhead, Bandwidth issues, security issues.**

**Complexity associated with distributed systems-**

**Service Discovery-**

**Service discovery tools manage how processes and services in a cluster can find and talk to one another. It involves a directory of services,**

**registering services in that directory, and then being able to lookup and connect to services in that directory. We need a Service Registry and Discovery mechanism so that service-to-service communication does not depend on hard-coded hostnames and port numbers. Spring Cloud provides Netflix Eureka-based Service Registry and Discovery support with just minimal configuration. We can also use Consul or ZooKeeper for Service Registry and Discovery.**

**Redundancy-\*\*\*\*\***

**Redundancy issues in distributed systems.**

**Loadbalancing-**

**Load balancing improves the distribution of workloads across multiple computing resources, such as computers, a computer cluster, network links,**

**central processing units, or disk drives.**

**Circuit Breaker:**

**In microservices-based architecture, one service might depend on another service, and if one service goes down, then failures may cascade to other services as well. Spring Cloud provides a Netflix Hystrix-based Circuit Breaker to handle these kinds of issues.**

**Spring Cloud Data Streams: We may need to work with huge volumes of data streams using Kafka or Spark. Spring Cloud Data Streams provides higher-level abstractions to use those frameworks more easily.**

**Spring Cloud Security:**

**Some microservices need to be accessible to authenticated users only, and most likely, we'll want a Single Sign-On feature to propagate the authentication context across services. Spring Cloud Security provides authentication services using OAuth2.**

**Distributed Tracing:**

**One of the pain points with microservices is the ability to debug issues. One simple end-user action might trigger a chain of microservice calls; there should be a mechanism to trace the related call chains. We can use Spring Cloud Sleuth with Zipkin to trace cross-service invocations.**

**Spring Cloud Contract: ?????**

**There is a high chance that separate teams will work on different microservices. There should be a mechanism for teams to agree upon API endpoint contracts so that each team can develop their APIs independently. Spring Cloud Contract helps to create such contracts and validate them by both the service provider and consumer.**

**Performance issues-**

**Performance issues due to various operational overheads.**

**Deployment complexities-**

**2) Name three commonly used tools for Microservices**

**Wiremock, 2.) Docker and 3.) Hysrix are important Microservices tool.**

**3) microservice benefits**

**Here, are some significant advantages of using Microservices:**

**Technology diversity, e., Microservices can mix easily with other frameworks, libraries, and databases**

**Fault isolation, e., a process failure should not bring the whole system down.**

**Greater support for smaller and parallel team**

**Independent deployment**

**Deployment time reduce.**

**All microservices can be developed independently.**

**Once they are developed, they are deployed independently for a particular application.**

**Even if one of the services will not work, the system still continues to work.**

**Different technologies or languages in Microservices can be used to build multiple services of an application.**

**You can scale individual component one by one instead of scaling all components together.**

**Q4). When designing Microservices, what are best practices to follow?**

**Here are the best practices that every developer should follow when designing Microservices.**

**The data store should always be kept separated from each Microservice.**

**The code should be arranged at a similar level of maturity.**

**A separate build should be designed for each Microservice.**

**Each build should be deployed into containers.**

**The server should always be treated as stateless.**

**Q6). List down the pros and cons of a Microservice Architecture.**

**What are the pros?**

**PROS**

**It has the freedom to use different technologies.**

**Each Microservices will focus on single capability only.**

**It supports individual deployable units.**

**It allows frequent releases of a software application.**

**It gives maximum security to each of the services.**

**It is possible developing and deploying multiple services together.**

**What are the cons?**

**CONS**

**The troubleshooting challenges will increase.**

**It will increase delays due to remote calls.**

**The configuration options are not possible to manage.**

**It is difficult to maintain transaction safety.**

**Data cannot be tracked across different boundaries.**

**There is a tough coding between multiple microservices.**

**Q8). How SOA, monolithic, and Microservices Architectures are different from each other?**

**SOA (Service-Oriented Architecture): This is a wide collection of services that could communicate together.**

**Monolithic: This is highly similar to the big container where all software components of an application are assembled and packed together.**

**Microservices: Microservices can be defined as the architectural style that structures an application as a collection of small autonomous services that are modeled around a business domain.**

**Q19). What are different types of tests for Microservices?**

**These are – Technology-facing tests, Exploratory Testing, and Acceptable Testing.**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*DESIGN PATTERN\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**5) Design patterns of Microservices:**

**a. Decomposition**

**Decomposition by Business Capibility**

**Microservices is all about making services loosely coupled, applying the single responsibility principle. It decomposes by business capability. Define services corresponding to business capabilities. A business capability is a concept from business architecture mode**

**ling [2]. It is something that a business does in order to generate value. A business capability often corresponds to a business object, e.g.**

**Decompose by locality**

**Sometimes you will find that there are some services that communicate more with one another compared to other services.**

**Find the services that communicate intensely with one another and reason if the can be merged as one service.**

**Keep services that communicate less in separate components**

**N.B: you need to understand the weight of the links between each of your sub-components. If the weight is too high, then consider having them as one service component.**

**Decompose by Greed**

**Some components are more resource greedy than others, in fact, it would make a lot of sense to isolate the greedy components in their own services.**

**Doing so, allows you to monitor the resource consumption more effectively and avoid overprovisioning resources.**

**N.B: you need to study the resource consumption of each sub-component before attempting the isolation**

**b. Strangler Pattern**

**The first three design patterns that you went through were decomposing applications for Greenfield, but 80% of the work you do is with brownfield applications, which are big, monolithic applications (legacy codebase). The Strangler pattern comes to the rescue or solution. This creates two separate applications that live side by side in the same URI space. Over time, the newly refactored application “strangles” or replaces the original application until finally, you can shut off the monolithic application. The Strangler Application steps are transformed, coexist, and eliminate [4]:**

**Transform:- Create a parallel new site with modern approaches.**

**Coexist:- Leave the existing site where it is for a time. Redirect from the existing site to the new one so the functionality is**

**implemented incrementally.**

**Eliminate:- Remove the old functionality from the existing site.**

**Bulkhead Pattern**

**This pattern isolates elements of an application into pools so that if one fails, the others will continue to function. This pattern is named Bulkhead because it resembles the sectioned partitions of a ship’s hull. Partition service instances into different groups, based on consumer load and availability requirements. This design helps to isolate failures, and allows you to sustain service functionality for some consumers, even during a failure.**

**Sidecar Pattern**

**This deploys components of an application into a separate processor container to provide isolation and encapsulation. This pattern can also enable applications to be composed of heterogeneous components and technologies. This pattern is named Sidecar because it resembles a sidecar attached to a motorcycle. In the pattern, the sidecar is attached to a parent application and provides supporting features for the application. The sidecar also shares the same lifecycle as the parent application, is created and retired alongside the parent. The sidecar pattern is sometimes referred to as the sidekick pattern and is the last decomposition pattern that we show in the post.**

**Integration Patterns**

**API Gateway Pattern When an application is broken down to smaller microservices, there are a few concerns that need to be addressed**

**There are multiple calls for multiple microservices by different channels.**

**There is a need for handling different type of Protocols.**

**Different consumers might need a different format of the responses.**

**An API Gateway helps to address many of the concerns raised by the microservice implementation, not limited to the ones above.**

**An API Gateway is the single point of entry for any microservice calls.**

**It can work as a proxy service to route a request to the concerned microservice.**

**It can aggregate the results to send back to the consumer.**

**This solution can create a fine-grained API for each specific type of client.**

**It can also convert the protocol request and respond.**

**It can also offload the authentication/authorization responsibility of the microservice.**

**Aggregator Pattern**

**When breaking the business functionality into several smaller logical pieces of code, it becomes necessary to think about how to collaborate the data returned by each service. This responsibility cannot be left with the consumer.**

**The Aggregator pattern helps to address this. It talks about how we can aggregate the data from different services and then send the final response to the consumer. This can be done in two ways [6]:**

**A composite microservice will make calls to all the required microservices, consolidate the data, and transform the data before sending back.**

**An API Gateway can also partition the request to multiple microservices and aggregate the data before sending it to the consumer.**

**It is recommended if any business logic is to be applied, then choose a composite microservice. Otherwise, the API Gateway is the established solution. Proxy Pattern API gateway just exposes Microservices over API gateway. In this example, the API gateway has three API modules:**

**Mobile API, which implements the API for the FTGO mobile client.**

**Browser API, which implements the API to the JavaScript application running in the browser.**

**Public API, which implements the API for third-party developers.**

**\*\*\*INTERVIEWS\*\*\*\*\***

**1) Spring vs Spring boot.**

**You can create standalone web application using springboot with embbed web server. No need to deploly war file in server to run. you**

**can start application using public static void main method. Maven file(pom file) creation is easy, fast, error free your time did not waste in creation of error free project setup.**

**Spring provides starter artifacts that provide bunch of required jar file for particular module. For example if we want to create**

**Restful web service then spring-boot-starter-web arctifact is enough because it has all jar files needed to create rest project for example Jackson dependency, servlet dependency and etc. You do not need to provide arctifact version, springboot itself choose appropriate versions so no need to worry about version conflict error.**

**Springboot automatically generate configuration file based on the dependecies in pom files, Hence no need to write these configuration files**

**manually. this is how we do not waste time and effort on writing boilerplate codes for configuration files.**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-web</artifactId>**

**</dependency>**

**<properties>**

**<spring.version>3.2.2.RELEASE</spring.version>**

**<jackson.version>1.9.10</jackson.version>**

**<jdk.version>1.6</jdk.version>**

**</properties>**

**<!-- Spring 3 dependencies -->**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-core</artifactId>**

**<version>${spring.version}</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-web</artifactId>**

**<version>${spring.version}</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-webmvc</artifactId>**

**<version>${spring.version}</version>**

**</dependency>**

**<!-- Jackson JSON Mapper -->**

**<dependency>**

**<groupId>org.codehaus.jackson</groupId>**

**<artifactId>jackson-mapper-asl</artifactId>**

**<version>${jackson.version}</version>**

**</dependency>**

**Configuration Boiler plate codes files**

**a.**

**file 1**

**public class SpringWebAppInitializer implements WebApplicationInitializer {**

**@Override**

**public void onStartup(ServletContext servletContext) throws ServletException {**

**AnnotationConfigWebApplicationContext appContext = new AnnotationConfigWebApplicationContext();**

**appContext.register(ApplicationContextConfig.class);**

**ServletRegistration.Dynamic dispatcher = servletContext.addServlet("SpringDispatcher",**

**new DispatcherServlet(appContext));**

**dispatcher.setLoadOnStartup(1);**

**dispatcher.addMapping("/");**

**// UTF8 Charactor Filter.**

**FilterRegistration.Dynamic fr = servletContext.addFilter("encodingFilter", CharacterEncodingFilter.class);**

**fr.setInitParameter("encoding", "UTF-8");**

**fr.setInitParameter("forceEncoding", "true");**

**fr.addMappingForUrlPatterns(null, true, "/\*");**

**}**

**}**

**file 2**

**@Configuration**

**@EnableWebMvc**

**public class WebMvcConfig extends WebMvcConfigurerAdapter {**

**@Override**

**public void addResourceHandlers(ResourceHandlerRegistry registry) {**

**}**

**@Override**

**public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {**

**configurer.enable();**

**}**

**}**

**Design Patterns in Microservices.**

1. Aggregator Microservice Design Pattern
2. 2- API Gateway Design Pattern
3. Chain Microservice Design Pattern
4. Branch Microservice Design Pattern
5. Circuit Breaker Design Pattern
6. Asynchronous Messaging Design Pattern

**Explanation:**

**1- Aggregator Microservice Design Pattern** The user makes a single call to the Aggregator, and the aggregator then calls each relevant   
 microservice and collects the data, apply business logic to it, and further publish as a REST   
 Endpoint.

**2- API Gateway Design Pattern** An API gateway is the entry point into the application from the outside world.  
 It’s responsible for request routing, API composition, and other functions, such as   
 authentication.

**3- Chained microservice design pattern:** Chained microservice design pattern produce a single consolidated response to the request. In this case, the request from the client is received by Service A, which is then communicating   
 with Service B, which in turn may be communicating with Service C. All the services are likely   
 using a synchronous HTTP request/response messaging.

Note:- One more important aspect which you need to understand, is that the request from   
 Service A to Service B may look different from Service B to Service C.  
 Similarly the response from Service C to Service B may look completely different from   
 Service B to Service A.

**4- Branch Microservice Design Pattern** A microservice may need to get the data from multiple sources including other microservices.   
 Branchmicroservice pattern is a mix of Aggregator & Chain design patterns and allows   
 Simultaneousrequest/response processing from two or more microservices. The invoked   
 microservice can be chainsof microservices.

**5- Circuit Breaker Design Pattern** Circuit Breaker Design Pattern is used for fault tolarance. If any microservice is down or getting  
 delay in sending response then Circuit breaker stop flow of that request to effected   
 microservice and handle the request and send the response. This Fault tolarance design   
 pattern is implemented with Hystrix or Resilience4J.

**6. Asynchronous Messaging Design Pattern** While REST design pattern is quite prevalent, and well understood, but it has the limitation of  
 being synchronous, and Thus blocking. Asynchrony can be achieved but that is done in an   
 application specific way. Some microservice architectures may elect to use message queues   
 instead of REST request/response because of that.