**\* [1.What are some kind of challenges that distributed systems introduces?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-1319)**

When you are implementing microservices architecture, there are some challenges that you need to deal with every single microservices. Moreover, when you think about the interaction with each other, it can create a lot of challenges. As well as if you pre-plan to overcome some of them and standardize them across all microservices, then it happens that it also becomes easy for developers to maintain services.

Some of the most challenging things are testing, debugging, security, version management, communication (sync or async), state maintenance etc. Some of the cross-cutting concerns which should be standardized are monitoring, logging, performance improvement, deployment, security etc.

**[2.On what basis should microservices be defined?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-1320)**

It should be based on the following criteria.

* Business functionalities that change together in bounded context
* Service should be testable independently.
* Changes can be done without affecting clients as well as dependent services.
* It should be small enough that can be maintained by 2-5 developers.
* Reusability of a service

**[3. How to tackle service failures when there are dependent services?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-1321)**

In real time, it happens that a particular service is causing a downtime, but the other services are functioning as per mandate. So, under such conditions, the service and its dependent services get affected due to the downtime.

To solve this issue, there is a concept in the microservices architecture pattern, called the circuit breaker. Any service calling remote service can call a proxy layer which acts as an electric circuit breaker. If the remote service is slow or down for ‘n’ attempts, then proxy layer should fail fast and keep checking the remote service for its availability again. As well as the calling services should handle the errors and provide retry logic. Once the remote service resumes then the services start working again and the circuit becomes complete.

This way, all other functionalities work as expected. Only one or the dependent services get affected.

**[4. How can one achieve automation in microservice based architecture?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-1322)**

This is related to the automation for cross-cutting concerns. We can standardize some of the concerns like monitoring strategy, deployment strategy, review and commit strategy, branching and merging strategy, testing strategy, code structure strategies etc.

For standards, we can follow the 12-factor application guidelines. If we follow them, we can definitely achieve great productivity from day one. We can also containerize our application to utilize the latest DevOps themes like dockerization. We can use mesos, marathon or kubernetes for orchestrating docker images. Once we have dockerized source code, we can use CI/CD pipeline to deploy our newly created codebase. Within that, we can add mechanisms to test the applications and make sure we measure the required metrics in order to deploy the code.

We can use strategies like blue-green deployment or canary deployment to deploy our code so that we know the impact of code which might go live on all of the servers at the same time. We can do AB testing and make sure that things are not broken when live. In order to reduce a burden on the IT team, we can use AWS / Google cloud to deploy our solutions and keep them on autoscale to make sure that we have enough resources available to serve the traffic we are receiving.

**[5. What should one do so that troubleshooting becomes easier in microservice based architecture?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-1323)**

In monolith where HTTP Request waits for a response, the processing happens in memory and it makes sure that the transaction from all such modules work at its best and ensures that everything is done according to expectation. But it becomes challenging in the case of microservices because all services are running independently, their datastores can be independent, REST APIs can be deployed on different endpoints. Each service is doing a bit without knowing the context of other microservices.

In this case, we can use the following measures to make sure we are able to trace the errors easily.

1. Services should log and **aggregators push logs to centralized logging servers**. For example, use ELK Stack to analyse.
2. Unique value per client request(correlation-id) which should be logged in all the microservices so that errors can be traced on a central logging server.
3. One should have good monitoring in place for each microservice in the ecosystem, which can record application metrics and health checks of the services, traffic pattern and service failures.

**[6. How should microservices communicate with each other?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-1325)**

It is an important design decision. The communication between services might or might not be necessary. It can happen synchronously or asynchronously. It can happen sequentially, or it can happen in parallel. So, once we have decided what should be our communication mechanism, we can decide the technology which suits the best.

Here are some of the examples which you can consider.

* Communication can be done by using some queuing service like rabbitmq, activemq and kafka. This is called asynchronous communication.
* Direct API calls can also be made to microservice. With this approach, interservice dependency increases. This is called synchronous communication.
* Webhooks to push data to connected clients/services.

**[7. How would you implement authentication in microservice architecture?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-1326)**

There are mainly two ways to achieve authentication in microservices architecture.

* **Centralized sessions**

All the microservices can use a central session store and user authentication can be achieved. This approach works but has many drawbacks as well. Also, the centralized session store should be protected, and services should connect securely. The application needs to manage the state of the user, so it is called stateful session.

* **Token-based authentication/authorization**

In this approach, unlike the traditional way, information in the form of token is held by the clients and the token is passed along with each request. A server can check the token and verify the validity of the token like expiry, etc. Once the token is validated, the identity of the user can be obtained from the token. However, encryption is required for security reasons. JWTis the new open standard for this, which is widely used. Mainly used in stateless applications. Or, you can use OAuth based authentication mechanisms as well.

**[8. What would be your logging strategy in a microservice architecture?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-1327)**

Logging is a very important aspect of any application. If we have done proper logging in an application, it becomes easy to support other aspects of the application as well. Like to debug the issues / in order to understand what business logic might have been executed, it becomes very critical to log important details.

Ideally, you should follow the following practices for logging.

* In a microservice architecture, each request should have a unique value (correlationId) and this value should be passed to each microservice so the correlation-id can be logged across the services. Thus, the requests can be traced.
* Logs generated by all the services should be aggregated in a single location so that while searching becomes easier. Generally, people use ELK stack for the same. So that it becomes easy for support persons to debug the issue.

**[9.How does docker help in microservice architecture?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-1328)**

Docker helps in many ways for microservices architecture.

1. In a microservice architecture, there can be many different services written in different languages. So a developer might have to setup few services along with its dependency and platform requirements. This becomes difficult with the growing number of services in an ecosystem. However, this becomes very easy if these services run inside a Docker container.
2. Running services inside a container also give a similar setup across all the environments, i.e development, staging and production.
3. Docker also helps in scaling along with container orchestration.
4. Docker helps to upgrade the underlying language very easily. We can save many man-hours.
5. Docker helps to onboard the engineers fast.
6. Docker also helps to reduce the dependencies on IT Teams to set up and manage the different kind of environment required.

**[10. How would you manage application configuration in microservice running in a container?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-1329)**

As container-based deployment involves a single image per microservice, it is a bad idea to bundle the configuration along with the image.

This approach is not at all scalable because we might have multiple environments and also we might have to take care of geographically distributed deployments where we might have different configurations as well.

Also, when there are application and cron application as part of the same codebase, it might need to take additional care on production as it might have repercussions how the crons are architected.

To solve this, we can put all our configuration in a centralized config service which can be queried by the application for all its configurations at the runtime. Spring cloud is one of the example services which provides this facility.

It also helps to secure the information, as the configuration might have passwords or access to reports or database access controls. Only trusted parties should be allowed to access these details for security reasons.

**[11.What is container orchestration and how does it helps in a microservice architecture?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-1330)**

In a production environment, you don’t just deal with the application code/application server. You need to deal with API Gateway, Proxy Servers, SSL terminators, Application Servers, Database Servers, Caching Services, and other dependent services.

As in modern microservice architecture where each microservice runs in a separate container, deploying and managing these containers is very challenging and might be error-prone.

Container orchestration solves this problem by managing the life cycle of a container and allows us to automate the container deployments.

It also helps in scaling the application where it can easily bring up a few containers. Whenever there is a high load on the application and once the load goes down. it can scale down as well by bringing down the containers. It is helpful to adjust cost based on requirements.

Also in some cases, it takes care of internal networking between services so that you need not make any extra effort to do so. It also helps us to replicate or deploy the docker images at runtime without worrying about the resources. If you need more resources, you can configure that in orchestration services and it will be available/deployed on production servers within minutes.

**\* [12.Explain the API gateway and why one should use it?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-1333)**

An API Gateway is a service which sits in front of the exposed APIs and acts as an entry point for a group of microservices. Gateway also can hold the logic of routing calls to microservices and an aggregation of the response.

* A gateway can also authenticate requests by verifying the identity of a user by routing every request to authentication service before routing it to the microservice with authorization details in the token.
* Gateways are also responsible to load balance the requests.
* API Gateways are responsible to rate limit a certain type of request to save itself from blocking several kinds of attacks etc.
* API Gateways can whitelist or blacklist the source IP Addresses or given domains which can initiate the call.
* API Gateways can also provide plugins to cache certain type of API responses to boost the performance of the application.

**\* [13.How will you ensure data consistency in microservice based architecture?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-1335)**

One should avoid sharing database between microservices, instead APIs should be exposed to perform the change.

If there is any dependency between microservices then the service holding the data should publish messages for any change in the data for which other services can consume and update the local state.

If consistency is required, then microservices should not maintain local state and instead can pull the data whenever required from the source of truth by making an API call.

**[14.What is event sourcing in microservices architecture?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-1336)**

In the microservices architecture, it is possible that due to service boundaries, a lot of times you need to update one or more entities on the state change of one of the entities. In that case, one needs to publish a message and new event gets created and appended to already executed events. In case of failure, one can replay all events in the same sequence, and you will get the desired state as required. You can think of event sourcing as your bank account statement.

You will start your account with initial money. Then all the credit and debit events happen, and the latest state is generated by calculating all the events one by one. In a case where events are too many, the application can create a periodic snapshot of events so that there isn’t any need to replay all the events again and again.

**[15.How will you implement service discovery in microservices architecture?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3605)**

Servers come and go in a cloud environment, and new instances of same services can be deployed to cater increasing load of requests. So, it becomes essential to have service registry & discovery that can be queried for finding address (host, port & protocol) of a given server. We may also need to locate servers for the purpose of client-side load balancing (Ribbon) and handling failover gracefully (Hystrix).

Spring Cloud solves this problem by providing a few ready-made solutions for this challenge. There are mainly two options available for the service discovery - Netflix Eureka Server and Consul. Let's discuss both briefly:

**Netflix Eureka Server**

Eureka is a REST based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers. The main features of Netflix Eureka are:

1. It provides service-registry.
2. zone aware service lookup is possible.
3. eureka-client (used by microservices) can cache the registry locally for faster lookup. The client also has a built-in load balancer that does basic round-robin load balancing.

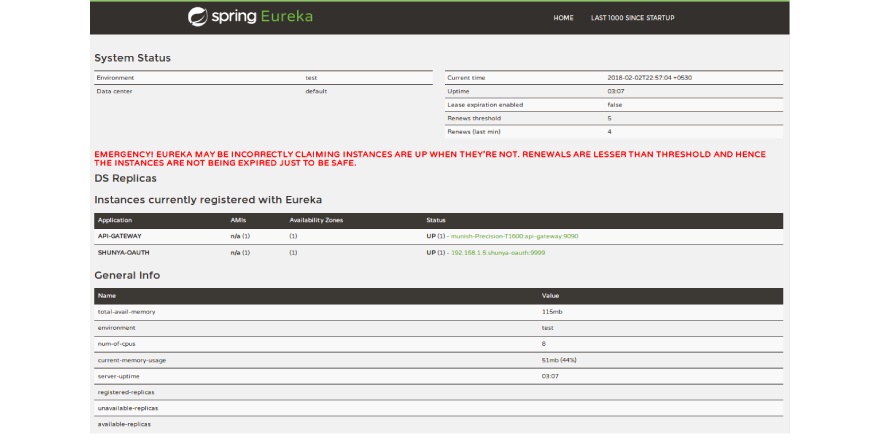
Spring Cloud provides two dependencies –“eureka-server” and “eureka-client”. Eureka server dependency is only required in eureka server’s build.gradle

* *build.gradle - Eureka Server*
* compile('org.springframework.cloud:spring-cloud-starter-netflix-eureka-server')

On the other hand, each microservice need to include the eureka-client dependencies to enables eureka discovery.

*build.gradle - Eureka Client (to be included in all microservices)*compile('org.springframework.cloud:spring-cloud-starter-netflix-eureka-client').

Eureka server provides a basic dashboard for monitoring various instances and their health in the service registry. The UI is written in freemarker and provided out of the box without any extra configuration. Screenshot for Eureka Server looks like the following.



It contains a list of all services that are registered with Eureka Server. Each server has information like zone, host, port, and protocol.

**Consul Server**

It is a REST-based tool for dynamic service registry. It can be used for registering a new service, locating a service and health checkup of a service.

You have the option to choose any one of the above in your spring cloud-based distributed application. In this book, we will focus more on the Netflix Eureka Serveroption.

**[16.How will you use config-server for your development, stage and production environment?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3607)**

If you have 3 different environments (develop/stage/production) in your project setup, then you need to create three different config storage projects. So, in total, you will have four projects:

* **config-server**

It is the config-server that can be deployed in each environment. It is the Java Code without configuration storage.

* **config-dev**

It is the git storage for your development configuration. All configuration related to each microservices in the development environment will fetch its config from this storage. This project has no Java code, and t is meant to be used with config-server.

* **config-qa**

Same as config-dev but its meant to be used only in qa environment.

* **Config-prod**
  + Same as config-dev, but meant for production environment.
  + So depending upon the environment, we will use config-server with either config-dev, config-qa or config-prod.

**[17.How does Eureka Server work?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3608)**

There are two main components in Eureka project: eureka-server and eureka-client.

* **Eureka Server**

The central server (one per zone) that acts as a service registry. All microservices register with this eureka server during app bootstrap.

* **Eureka Client**

Eureka also comes with a Java-based client component, the eureka-client, which makes interactions with the service much easier. The client also has a built-in load balancer that does basic round-robin load balancing. Each microservice in the distributed ecosystem much include this client to communicate and register with eureka-server.

* **Typical use case for Eureka**

There is usually one eureka server cluster per region (US, Asia, Europe, Australia) which knows only about instances in its region. Services register with Eureka and then send heartbeats to renew their leases every 30 seconds. If the service can not renew their lease for a few times, it is taken out of server registry in about 90 seconds. The registration information and the renewals are replicated to all the eureka nodes in the cluster. The clients from any zone can look up the registry information (happens every 30 seconds) to locate their services (which could be in any zone) and make remote calls.

Eureka clients are built to handle the failure of one or more Eureka servers. Since Eureka clients have the registry cache information in them, they can operate reasonably well, even when all the eureka servers go down.

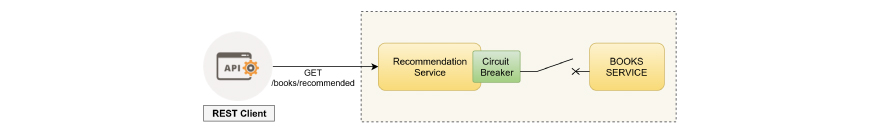
**[18.What is Circuit Breaker Pattern?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3609)**

Microservices often need to make remote network calls to another microservices running in a different process. Network calls can fail due to many reasons, including-

1. Brittle nature of the network itself
2. Remote process is hung or
3. Too much traffic on the target microservices than it can handle

This can lead to cascading failures in the calling service due to threads being blocked in the hung remote calls. A circuit breaker is a piece of software that is used to solve this problem. The basic idea is very simple - wrap a potentially failing remote call in a circuit breaker object that will monitor for failures/timeouts. Once the failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error, without the protected call being made at all. This mechanism can protect the cascading effects of a single component failure in the system and provide the option to gracefully downgrade the functionality.

A typical use of circuit breaker in microservices architecture looks like the following diagram-



**Typical Circuit Breaker Implementation**

Here a REST client calls the Recommendation Service which further communicates with Books Service using a circuit breaker call wrapper. As soon as the books-service API calls starts to fail, circuit breaker will trip (open) the circuit and will not make any further call to book-service until the circuit is closed again.

Martin Fowler has beautifully explained this phenomenon in detail on his blog.

Martin Fowler on Circuit Breaker Pattern : <https://martinfowler.com/bliki/CircuitBreaker.html>

**[19.What are Open, Closed and Half-Open states of Circuit Breaker?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3610)**

Circuit Breaker wraps the original remote calls inside it and if any of these calls fails, the failure is counted. When the service dependency is healthy and no issues are detected, the circuit breaker is in Closed state. All invocations are passed through to the remote service.

If the failure count exceeds a specified threshold within a specified period, the circuit trips into the Open State. In the Open State, calls always fail immediately without even invoking the actual remote call. The following factors are considered for tripping the circuit to Open State -

* An Exception thrown (HTTP 500 error, can’t connect)
* Call takes longer than the configured timeout (default 1 second)
* The internal thread pool (or semaphore depending on configuration) used by hystrix for the command execution rejects the execution due to exhausted resource pool.

After a predetermined period (by default 5 seconds), the circuit transitions into a half-open state. In this state, calls are again attempted to the remote dependency. Thereafter the successful calls transition the circuit breaker back into the closed state, while the failed calls return the circuit breaker into the open state.

**[20.What are use-cases for Circuit Breaker Pattern and benefits of using Circuit Breaker Pattern?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3611)**

1. Synchronous communication over the network that is likely to fail is a potential candidate for circuit breaker.
2. A circuit breaker is a valuable place for monitoring, any change in the breaker state should be logged to enable deep monitoring of microservices. It can easily troubleshoot the root cause of failure.
3. All places where a degraded functionality can be acceptable to the caller if the actual server is struggling/down.

**Benefits:-**

1. The circuit breaker can prevent a single service from failing the entire system by tripping off the circuit to the faulty microservice.
2. The circuit breaker can help to offload requests from a struggling server by tripping the circuit, thereby giving it a time to recover.
3. In providing a fallback mechanism where a stale data can be provided if real service is down.

**[21.What is the difference between config first bootstrap and discovery first bootstrap in context of Spring Cloud Config client?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3612)**

*Config first bootstrap*and *discovery first bootstrap*are two different approaches for using Spring Cloud Config client in Spring Cloud-powered microservices. Let’s discuss both:

**Config First Bootstrap**

This is the default behaviour for any spring boot application where Spring Cloud Config client is on the class-path. When a config client starts up it binds to the Config Server using the bootstrap configuration property and initializes Spring Environment with remote property sources.

* Config-first approach

The only configuration that each microservice (except config-server) needs to provide is the following:

* File:-  /src/main/resources/bootstrap.yml
* spring.cloud.config.uri: http://localhost:8888
* In config-first approach, even the eureka-server can fetch its own configuration from config-server. Point worth noting down here is that config-server must be the first service to boot up in the entire ecosystem, because each service will fetch its configuration from config-server.

**Discovery First Bootstrap**

If you are using Spring Cloud Netflix and Eureka Service Discovery then you can have Config Server register with the Discovery Service and let all clients get access to config server via discovery service.

* Discovery-first approach

This is not the default configuration in Spring Cloud applications, so we need to manually enable it using the below property in bootstrap.yml

Listing 17. /src/main/resources/bootstrap.yml

  spring:

      cloud:

          config:

              discovery:

enabled: true

This property should be provided by all microservices so that they can take advantage of discovery first approach.

The benefit of this approach is that now config-server can change its host/port without other microservices knowing about it since each microservice can get the configuration via eureka service now. The downside of this approach is that an extra network round trip is required to locate the service registration at app startup.

**[22.What is Strangulation Pattern in microservices architecture?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3613)**

Strangulation is used to slowly decommission an older system and migrate the functionality to a newer version of microservices.

Normally one endpoint is Strangled at a time, slowly replacing all of them with the newer implementation. Zuul Proxy (API Gateway) is a useful tool for this because we can use it to handle all traffic from clients of the old endpoints but redirect only selected requests to the new ones.

Let’s take an example use-case:

*/src/main/resources/application.yml*

zuul:

    routes:

first:

path: /first/\*\*

url: http://first.example.com --1

legacy:

path: /\*\*

url: http://legacy.example.com  -- 2

1. Paths in /first/\*\* have been extracted into a new service with an external URL http://first.example.com
2. legacy app is mapped to handle all request that do not match any other patterns (/first/\*\*).

This configuration is for API Gateway (zuul reverse proxy), and we are strangling selected endpoints /first/ from the *legacy*app hosted at http://legacy.example.com slowly to newly created microservice with external URL http://first.example.com

**[23.What is Hystrix?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3614)**

Hystrix is Netflix implementation for circuit breaker pattern, that also employs bulkhead design patternby operating each circuit breaker within its own thread pool. It also collects many useful metrics about the circuit breaker’s internal state, including -

1. Traffic volume.
2. Request volume.
3. Error percentage.
4. Hosts reporting
5. Latency percentiles.
6. Successes, failures, and rejections.

All these metrics can be aggregated using another Netflix OSS project called Turbine. Hystrix dashboard can be used to visualize these aggregated metrics, providing excellent visibility into the overall health of the distributed system.  
Hystrix can be used to specify the fallback method for execution in case the actual method call fails. This can be useful for graceful degradation of functionality in case of failure in remote invocation.

*Add hystrix library to*build.gradle dependencies {

compile('org.springframework.cloud:spring-cloud-starter-hystrix')

*1) Enable Circuit Breaker in main application*

@EnableCircuitBreaker @RestController @SpringBootApplication

public class ReadingApplication {

... }

*2) Using HystrixCommand fallback method execution*

@HystrixCommand(fallbackMethod = "reliable")

public String readingList() {

URI uri = URI.create("http://localhost:8090/recommended"); **return this**.restTemplate.getForObject(uri, String.class);

}

public String reliable() { 2

return "Cached recommended response";

}

1. Using @HystrixCommand annotation, we specify the fallback method to execute in case of exception.
2. fallback method should have the same signature (return type) as that of the original method. This method provides a graceful fallback behavior while the circuit is in the open or half-open state.

**[24.What are the main features of the Hystrix library?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3615)**

Hystrix library makes our distributed system **resilient**(adaptable & quick to recover) to failures. It

provides three main features:

* **Latency and fault-tolerance**

It helps stop cascading failures, provide decent fallbacks and graceful degradation of service functionality to confine failures. It works on the idea of fail-fast and rapid recovery. Two different options namely Thread isolation and Semaphore isolation are available for use to confine failures.

* **Real-time operations**

Using real-time metrics, you can remain alert, make decisions, affect changes and see results.

* **Concurrency**

Parallel execution, concurrent aware request caching and finally automated batching through request collapsing improves the concurrency performance of your application.

More information on Netflix hystrix library:

* <https://github.com/Netflix/Hystrix/>
* <https://github.com/Netflix/Hystrix/wiki#principles>
* <https://github.com/Netflix/Hystrix/wiki/How-it-Works>

**[25.What is the difference between using a Circuit Breaker and a naive approach where we try/catch a remote method call and protect for failures?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3616)**

Let's say we want to handle service to service failure gracefully without using the Circuit Breaker pattern. The naive approach would be to wrap the   REST call in a try-catch clause. But Circuit Breaker does a lot more than try-catch can’t accomplish -

1. Circuit Breaker does not even try calls once the failure threshold is reached, doing so reduces the number of network calls. Also, several threads consumed in making faulty calls are freed up.
2. Circuit breaker provides fallback method execution for gracefully degrading the behaviour. Try catch approach will not do this out of the box without additional boiler plate code.
3. Circuit Breaker can be configured to use a limited number of threads for a particular host/API, doing so brings all the benefits of bulkhead design pattern.

So instead of wrapping service to service calls with try/catch clause, we must use the circuit breaker pattern to make our system resilient to failures.

**[26.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3617)**

**[How does Hystrix implement Bulkhead Design Pattern?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3617)**

The bulkhead implementation in Hystrix limits the number of concurrent calls to a component/service. This way, the number of resources (typically threads) that are waiting for a reply from the component/service is limited.

Let's assume we have a fictitious web e-commerce application as shown in the figure below. The WebFront communicates with 3 different components using remote network calls (REST over HTTP).

* Product catalogue Service
* Product Reviews Service
* Order Service

Now let's say due to some problem in Product Review Service, all requests to this service start to hang (or timeout), eventually causing all request handling threads in WebFront Application to hang on waiting for an answer from Reviews Service. This would make the entire WebFront Application non-responsive. The resulting behavior of the WebFront Application would be same if request volume is high and Reviews Service is taking time to respond to each request.

**The Hystrix Solution**

Hystrix’s implementation for bulkhead pattern would limit the number of concurrent calls to components and would have saved the application in this case by gracefully degrading the functionality. Assume we have 30 total request handling threads and there is a limit of 10 concurrent calls to Reviews Service. Then at most 10 request handling threads can hang when calling Reviews Service, the other 20 threads can still handle requests and use components Products and Orders Service. This will approach will keep our WebFront responsive even if there is a failure in Reviews Service.

**[27.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3618)**

**[In a microservices architecture, what are smart endpoints and dumb pipes?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3618)**

Martin Fowler introduced the concept of "smart endpoints & dumb pipes" while describing microservices architecture.

To give context, one of the main characteristic of a   based system is to build small utilities and connect them using pipes. For example, a very popular way of finding all java processes in Linux system is Command pipeline in Unix shell ps elf | grep java

Here two commands are separated by a pipe, the pipe’s job is to forward the output of the first command as an input to the second command, nothing more.   like a dumb pipe which has no business logic except the routing of data from one utility to another.

In his article Martin Fowler compares Enterprise Service Bus (ESB) to ZeroMQ/RabbitMQ, ESB is a pipe but has a lot of logic inside it while ZeroMQ has no logic except the persistence/routing of messages. ESB is a fat layer that does a lot of things like - security checks, routing, business flow & validations, data transformations, etc. So ESB is a kind of smart pipe that does a lot of things before passing data to next endpoint (service). Smart endpoints & dumb pipes advocate an exactly opposite idea where the communication channel should be stripped of any business-specific logic and should only distribute messages between components. The components (endpoints/services) should do all the data validations, business processing, security checks, etc on those incoming messages.

Microservices team should follow the principles and protocols that worldwide web & Unix is built on.

**[28.How to handle versioning of microservices?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3619)**

There are different ways to handle the versioning of your REST api to allow older consumers to still consume the older endpoints. The ideal practice is that any non-backward compatible change in each REST endpoint shall lead to a new versioned endpoint.

Different mechanisms of versioning are:

* Add version in the URL itself
* Add version in API request header

Most common approach in versioning is the URL versioning itself. A versioned URL looks like the following:

Versioned URL

* https://<host>:<port>/api/v1/...
* https://<host>:<port>/api/v2/...

As an API developer you must ensure that only backward-compatible changes are accommodated in a single version of URL. Consumer-Driven-Tests can help identify potential issues with API upgrades at an early stage.

**[29.How to refresh configuration changes on the fly in Spring Cloud environment?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3620)**

Using config-server, it's possible to refresh the configuration on the fly. The configuration changes will only be picked by Beans that are declared with @RefreshScope annotation.

The following code illustrates the same. The property message is defined in the config-server and changes to this property can be made at runtime without restarting the microservices.

package hello;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.context.config.annotation.RefreshScope;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication

public class ConfigClientApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigClientApplication.class, args);

}

}

@RefreshScope 1 @RestController

class MessageRestController {

@Value("${message:Hello World}") private String message;

@RequestMapping("/message") String getMessage() {

return this.message; }}

1 @RefreshScope makes it possible to dynamically reload the configuration for this bean.

**[30.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3621)**

**[How will you ignore certain exceptions in Hystrix fallback execution?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3621)**

@HystrixCommand annotation provides attribute ignoreExceptions that can be used to provide a list of ignored exceptions.

Code

* import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
* import org.springframework.beans.factory.annotation.Autowired;
* import org.springframework.cloud.client.ServiceInstance;
* import org.springframework.cloud.client.loadbalancer.LoadBalancerClient;
* import org.springframework.stereotype.Service;
* import org.springframework.web.client.RestTemplate;
* import java.net.URI;

@Service

public class HystrixService {

@Autowired

private LoadBalancerClientloadBalancer;

@Autowired

private RestTemplaterestTemplate;

@HystrixCommand(fallbackMethod = "reliable", ignoreExceptions = IllegalStateException.class, MissingServletRequestParameterException.class, TypeMismatchException.class)

public String readingList() {

ServiceInstance instance = loadBalancer.choose("product-service"); URI uri = URI.create("http://product-service/product/recommended"); return this.restTemplate.getForObject(uri, String.class);}

public String reliable(Throwable e) { return "Cloud Native Java (O'Reilly)";

In the above example, if the actual method call throws IllegalStateException, MissingServletRequestParameterException or TypeMismatchException then hystrix will not trigger the fallback logic (reliable method), instead the actual exception will be wrapped inside HystrixBadRequestException and re-thrown to the caller. It is taken care by javanica library under the hood.

**[31.Is it a good idea to share a common database across multiple microservices?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3622)**

In a microservices architecture, each microservice shall own its private data which can only be accessed by the outside world through owning service. If we start sharing microservice’s private datastore with other services, then we will violate the principle of Bounded Context.

Practically we have three approaches -

1. **Database server per microservice -** Each microservice will have its own database server instance. This approach has the overhead of maintaining database instance and its replication/backup, hence its rarely used in a practical environment.
2. **Schema per microservice -**Each microservice owns a private database schema which is not accessible to other services. Its most preferred approach for RDMS database (MySql, Postgres, etc.)
3. **Private Table per microservice -** Each microservice owns a set of tables that must only be accessed by that service. It’s a logical separation of data. This approach is mostly used for the hosted database as a service solution (Amazon RDS).

**32.What are best practices for microservices architecture?**

Microservices Architecture can become cumbersome & unmanageable if not done properly. There are best practices that help design a resilient & highly scalable system. The most important ones are

Partition correctly –

Get to know the domain of your business, that's very very important. Only then you will be able to define the bounded context and partition your microservice correctly based on business capabilities.

DevOps culture –

Typically, everything from continuous integration all the way to continuous delivery and deployment should be automated. Otherwise, a big pain to manage a large fleet of microservices.

Design for stateless operations

We never know where a new instance of a particular microservice will be spun up for scaling out or for handling failure, so maintaining a state inside service instance is a very bad idea.

Design for failures

Failures are inevitable in distributed systems, so we must design our system for handling failures gracefully. failures can be of different types and must be dealt with accordingly, for example -

Failure could be transient due to inherent brittle nature of the network, and the next retry may succeed. Such failures must be protected using retry operations.

Failure may be due to a hung service which can have cascading effects on the calling service. Such failures must be protected using Circuit Breaker Patterns. A fallback mechanism can be used to provide degraded functionality in this case.

A single component may fail and affect the health of the entire system, bulkhead pattern must be used to prevent the entire system from failing.

Design for versioning

We should try to make our services backward compatible, explicit versioning must be used to cater different versions of the RESt endpoints.

Design for asynchronous communication b/w services

Asynchronous communication should be preferred over synchronous communication in inter microservice communication. One of the biggest advantages of using asynchronous messaging is that the service does not block while waiting for a response from another service.

Design for eventual consistency

Eventual consistency is a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

Design for idempotent operations

Since networks are brittle, we should always design our services to accept repeated calls without any side effects. We can add some unique identifier to each request so that service can ignore the duplicate request sent over the network due to network failure/retry logic.

Share as little as possible

In monolithic applications, sharing is considered to be a best practice but that's not the case with Microservices. Sharing results in a violation of Bounded Context Principle, so we shall refrain from creating any single unified shared model that works across microservices. For example, if different services need a common Customer model, then we should create one for each microservice with just the required fields for a given bounded context rather than creating a big model class that is shared in all services.

The more dependencies we have between services, the harder it is to isolate the service changes, making it difficult to make a change in a single service without affecting other services. Also, creating a unified model that works in all services brings complexity and ambiguity to the model itself, making it hard for anyone to understand the model.

In a way are want to violate the DRY principle in microservices architecture when it comes to domain models.

**[33.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3624)**

**[How will you implement caching for microservices?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3624)**

Caching is a technique of performance improvement for getting query results from a service. It helps minimize the calls to network, database, etc. We can use caching at multiple levels in microservices architecture -

1. **Server-Side Caching** - Distributed caching software like Redis/MemCache/etc are used to cache the results of business operations. The cache is distributed so all instances of a microservice can see the values from the shared cache. This type of caching is opaque to clients.
2. **Gateway Cache** - central API gateway can cache the query results as per business needs and provide improved performance. This way we can achieve caching for multiple services at one place. Distributed caching software like Redis or Memcache can be used in this case.
3. **Client-Side Caching** - We can set cache-headers in http response and allow clients to cache the results for a pre-defined time. This will drastically reduce the load on servers since the client will not make repeated calls to the same resource. Servers can inform the clients when information is changed, thereby any changes in the query result can also be handled. E-Tags can be used for client-side load balancing. If the end client is a microservice itself, then Spring Cache support can be used to cache the results locally.

**[34.What is a good tool for documenting Microservices?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3625)**

Swagger is a very good open-source tool for documentingAPIs provided by microservices. It provides very easy to use interactive documentation.

By the use of swagger annotation on REST endpoint, API documentation can be auto-generated and exposed over the web interface. An internal and external team can use web interface, to see the list of APIs and their inputs & error codes. They can even invoke the endpoints directly from web interface to get the results.

Swagger UI is a very powerful tool for your microservices consumers to help them understand the set of endpoints provided by a given microservice.

**[35.Why Basic Authentication is not suitable in the Microservices Context?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3626)**

Basic Authentication is natively supported by almost all servers and clients, even Spring security has very good support for it and its configured out of the box. But it is not a good fit for Microservices due to many reasons, including -

1. We need credentials (username and password) every time we authenticate. This may be fine where all the participants can share the secrets securely, but Users may not be willing to share their credentials with all the applications.
2. There is no distinction between Users and Client Apps (an application that is making a request). In a realistic environment, we often need to know if a real user is making a request or a client app is making a request (for inter-service communication).
3. It only covers authentication. what about scopes, Authorizations? Basic Auth does not support adding additional attributes in the authentication headers. There is no concept of Tokens in basic auth.
4. Performance reasons for BCrypt Matching. Passwords are often stored in the database using one-way hash i.e. Bcrypt, it takes a lot of cpu cycles depending upon the strength (a.k.a. log rounds in BCrypt) to compare the user’s plain password with db saved bcrypt password, so it may not be efficient to match password on every request. The larger the strength parameter the more work will have to be done (exponentially) to hash the passwords. If you set the strength to 12, then in total 212 iterations will be done in Bcrypt Logic. Usually, 4-8 passwords can be matched per second on a T2.Micro instance on Amazon AWS instance. See BCryptPasswordEncoder for more info.
5. If we use Basic Auth for a mobile application client, then we might have to store user’s credentials on the device to allow remember me feature. This is quite risky as anyone getting access to the device may steal the plain credentials.

**[36.How does JWT look like?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3627)**

There are 3 parts in every JWT claim - Header, Claim and Signature. These 3 parts are separated by a dot. The entire JWT is encoded in Base64 format.

JWT = {header}.{payload}.{signature}

A typical JWT is shown here for reference.

Encoded JSON Web Token

Entire JWT is encoded in Base64 format to make it compatible with HTTP protocol. Encoded JWT looks like the following:



Decoded JSON Web Token

**Header**

Header contains algorithm information e.g. HS256 and type e.g. JWT

{

"alg": "HS256", "typ": "JWT"

}

**Claim**

claim part has an expiry, issuer, user\_id, scope, roles, client\_id etc. It is encoded as a JSON object. You can add custom attributes to the claim. This is the information that you want to exchange with the third party.

{

"uid": "2ce35360-ef8e-4f69-a8d7-b5d1aec78759", "user\_name": "user@mail.com",

"scope": ["read"],

"exp": 1520017228,

"authorities": ["ROLE\_USER","ROLE\_ADMIN"], "jti": "5b42ca29-8b61-4a3a-8502-53c21e85a117", "client\_id": "acme-app"

}

**Signature**

Signature is typically a one way hash of (header + payload), is calculated using HMAC SHA256 algorithm. The secret used for signing the claim should be kept private. Pubic/private key can also be used to encrypt the claim instead of using symmetric cryptography.

HMACSHA256(base64(header) + "." + base64(payload), "secret")

**[37.How OAuth2 Works?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3628)**

OAuth2.0 is a delegation protocol where the Client (Mobile App or web app) does not need to know about the credentials of Resource Owner (end-user).

Oauth2 defines four roles.

1. **Resource Owner** - The person or the application that owns the data to be shared. When a resource owner is a person, it is called as an end-user.
2. **Resource Server** - The application that holds the protected resources. It is usually a microservice.
3. **Authorization Server** - the application that verifies the identity of the resource owner (users/clients). These server issues access tokens after obtaining the authorization.
4. **Client** - the application that makes a request to Resource Server on behalf of Resource Owner. It could be a mobile app or a web app (like stackoverflow).