**1. What is Microservices Architecture?**

Microservices architecture allows avoiding monolith application for the large system. It provides loose coupling between collaborating processes which running independently in different environments with tight cohesion.

* ***Loose Coupling***– Effect of changes isolated
* ***Tight Cohesion***– Code perform a single well-defined task
* ***Loose Coupling***– Application build from collaboration services or processes, so any process change without affecting another process.
* ***Tight Cohesion***-An individual service or process that deals with a single view of data.

## **Microservices Benefits**

* The smaller code base is easy to maintain.
* Easy to scale as an individual component.
* Technology diversity i.e. we can mix libraries, databases, frameworks etc.
* Fault isolation i.e. a process failure should not bring the whole system down.
* Better support for smaller and parallel team.
* Independent deployment
* Deployment time reduce

## **Principles of Microservices**

There are the following principles of Microservices:

* Single Responsibility principle
* Modelled around business domain
* Isolate Failure
* Infrastructure automation
* Deploy independently

## **Disadvantages of Microservices**

* Microservices has all the associated complexities of the distributed system.
* There is a higher chance of failure during communication between different services.
* Difficult to manage a large number of services.
* The developer needs to solve the problem, such as network latency and load balancing.
* Complex testing over a distributed environment.

### Write main features of Microservices.

Some of the main features of Microservices include:

* **Decoupling**: Within a system, services are largely decoupled. The application as a whole can therefore be easily constructed, altered, and scalable
* **Componentization**: Microservices are viewed as independent components that can easily be exchanged or upgraded
* **Business Capabilities**: Microservices are relatively simple and only focus on one service
* **Team autonomy**: Each developer works independently of each other, allowing for a faster project timeline
* **Continuous Delivery**: Enables frequent software releases through systematic automation of software development, testing, and approval
* **Responsibility:** Microservices are not focused on applications as projects. Rather, they see applications as products they are responsible for
* **Decentralized Governance:** Choosing the right tool according to the job is the goal. Developers can choose the best tools to solve their problems
* **Agility:** Microservices facilitate agile development. It is possible to create new features quickly and discard them again at any time.

**What are main differences between Microservices and Monolithic Architecture?**

| **Microservices** | **Monolithic Architecture** |
| --- | --- |
| Service Startup is fast | Service startup takes time |
| Microservices are loosely coupled architecture. | Monolithic architecture is mostly tightly coupled. |
| Changes done in a single data model does not affect other Microservices. | Any changes in the data model affect the entire database |
| Microservices  focuses  on products, not projects | Monolithic put emphasize over the whole project |

https://www.edureka.co/blog/microservices-design-patterns

### ****Circuit Breaker Pattern****

As the name suggests, the Circuit Breaker design pattern is used to stop the process of request and response if a service is not working. So, for example, let’s say a client is sending a request to retrieve data from multiple services. But, due to some issues, one of the services is down. Now, there are mainly two problems you will face: first, **since the client will not have any knowledge about a particular service being down, the request will be continuously sent to that service**. **The second problem is that the network resources will be exhausted with low performance and bad user experience.**

So, to avoid such problems, you can use the Circuit Breaker Design Pattern. With the help of this pattern, the client will invoke a remote service via a proxy. This proxy will basically behave as a circuit barrier. So, when the number of failures crosses the threshold number, the circuit breaker trips for a particular time period. Then, all the attempts to invoke the remote service will fail in this timeout period. Once that time period is finished, the circuit breaker will allow a limited number of tests to pass through and if those requests succeed, the circuit breaker resumes back to the normal operation. Else, if there is a failure, then the time out period begins again.

[How to handle microservice Interaction when one of the microservice is down](https://stackoverflow.com/questions/50562495/how-to-handle-microservice-interaction-when-one-of-the-microservice-is-down)

if we have 3 microservices M1,M2,M3 . M1 is interacting with M2 and M2 is interacting with M3. In case M2 microservice cluster is down how should we handle this situation?

When any one of the microservice is down, Interaction between services becomes very critical as **isolation of failure, resilience and fault tolerance** are some of key characteristics for any microservice based architecture.

Totally agreed what @jayant had answered, **in your case Implementing proper fallback mechanism makes more sense and you can implement required logic you wanna write based on use case and dependencies between M1, M2 and M3. you can also raise events in your fallback if needed.**

Since you are new to microservice, you need to know below **common techniques and architecture patterns** for resilience and fault tolerance against the situation which you have raised in your question. And here you are using Spring-Boot, you can easily add Netflix-OSS in your microservices.

Netflix has released [Hystrix](https://medium.com/netflix-techblog/introducing-hystrix-for-resilience-engineering-13531c1ab362), a library designed to control points of access to remote systems, services and 3rd party libraries, providing greater tolerance of latency and failure.

It include below important characteristics:

* **Importance of Circuit breaker and Fallback Mechanism:**

Hystrix implements the [circuit breaker pattern](https://martinfowler.com/bliki/CircuitBreaker.html) which is useful when a service failure can cause cascading failure all the way up to the user. When calls to a particular service exceed circuitBreaker.requestVolumeThreshold (default: 20 requests) and the failure percentage is greater than circuitBreaker.errorThresholdPercentage (default: >50%) in a rolling window defined by metrics.rollingStats.timeInMilliseconds (default: 10 seconds), the circuit opens and further calls are not made.

In cases of error and an open circuit, a fallback can be provided by the developer. Fallbacks may be chained so that the first fallback makes some other business call. check out [Fallback Implementation of Hystrix](https://github.com/Netflix/Hystrix/wiki/How-To-Use#Fallback)

As mentioned in the comment, there are many ways you can go about it,

Case 1: all are independent services, trivial case, no need to do anything, call all the services in blocking or non-blocking way, calling service 2 will in both case result in timeout

Case 2: services are dependent M2 depends on M1 and M3 depends on M2

option a) M1 can wait for service M2 to come back up, doing periodic pings or fetching details from registry or naming server if M2 is up or not

option b) use hystrix as a circuit breaker implementation and handle fallback gracefully in M3 or your orchestrator(guy who is calling these services i.e M1,M2,M3 in order)