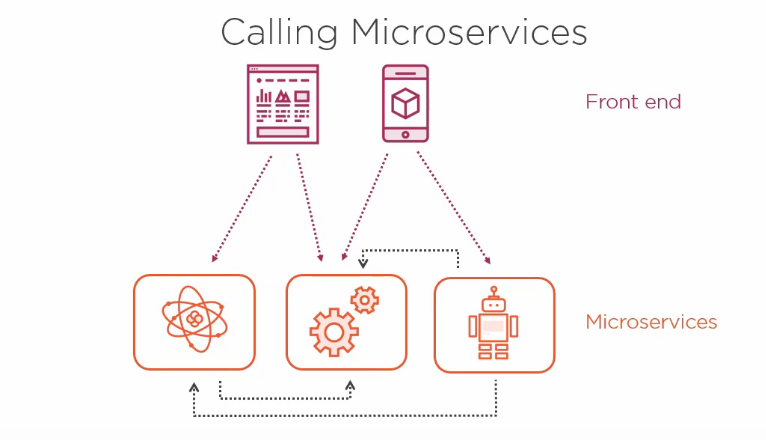
### Communication between Microservices

****

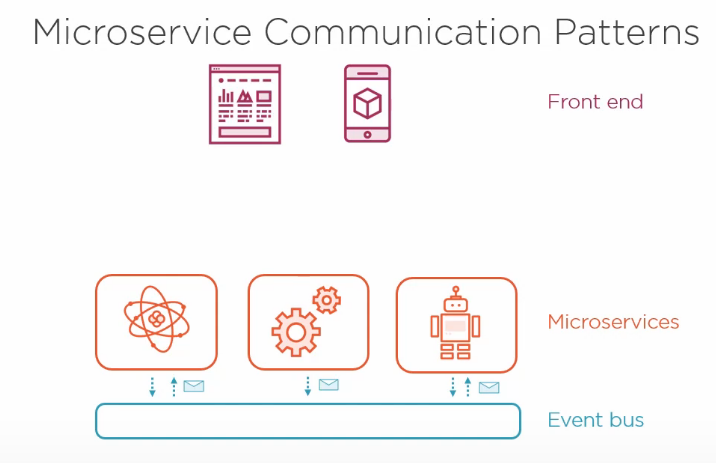
Are there any rules that govern which microservices should be allowed to talk to each other?

For example: If we have few micro services, Is it okay for them all to just call each other whenever it’s necessary ?

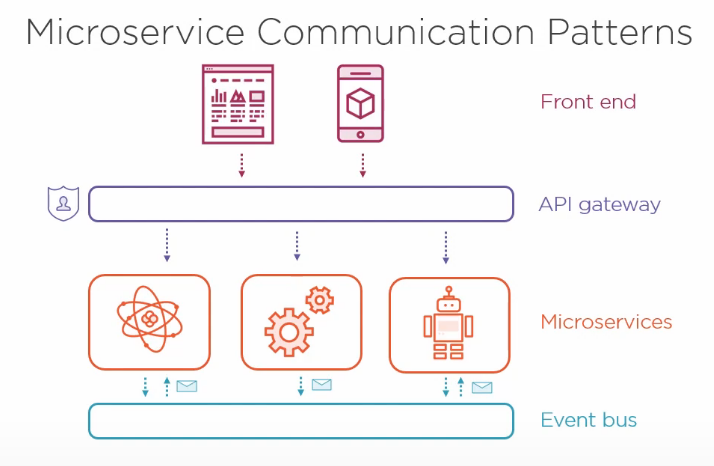
And If there’s a front end application, like a Website or a Mobile application, should that also be allowed to call directly to any of the microservices it wants to?

Well, there isn’t any hard or fast rules, but there are some difficulties with allowing a complete free-flow situation. You can end up creating a mess of tangled dependencies between services that can result in cascading failures where one microservice failing causes the others to fail as well. And you can also end up with poor performance, if making a call to one microservice ends up requiring multiple hops to call additional microservices.

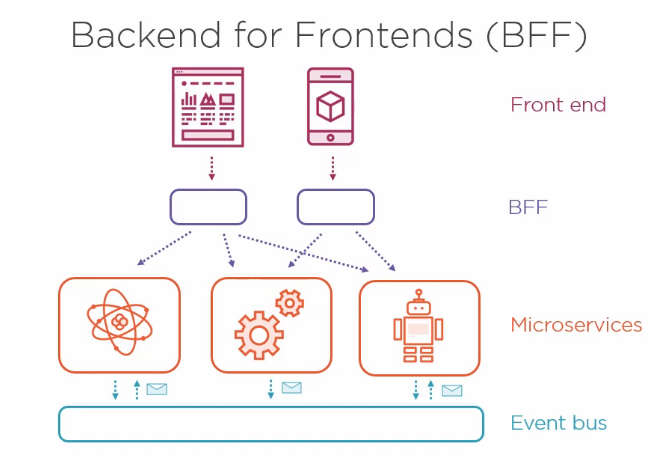
A better approach is to minimized the calls between micro services. Your microservice could publish messages to a shared event bus and also subscribe to messages on that bus. This promots a synchronous communication between microservices.



When we’re dealing with front-end application, particularly with mobile apps or Single Page Application where call are coming into our microservices directly from a client side application, then we might prefer to avoid the client application needing to know how to connect to all our different microservices. Instead, a pattern can be used called an API gateway, where all the calls from the front-end client application go through the gateway and are routed through to the correct microservice. This offers several benefits. It means we can implement authentication and generally it makes security easier to reason about if all the external traffic enters the system at a single point. This approach also decouples the client application from the specifics of our backend APIs, giving us more freedom and flexibility while maintaining a consistent Public API.

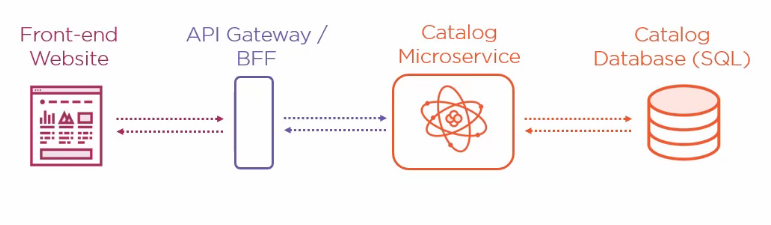


One more pattern called Back-End for Front-End builds on this idea. You can create an API gateway for each of your front-end applications, and this might even turn one incoming HTTP request into two back-end calls whose responses are aggregated or transformed into just the right format to meet the needs of the front-end application.



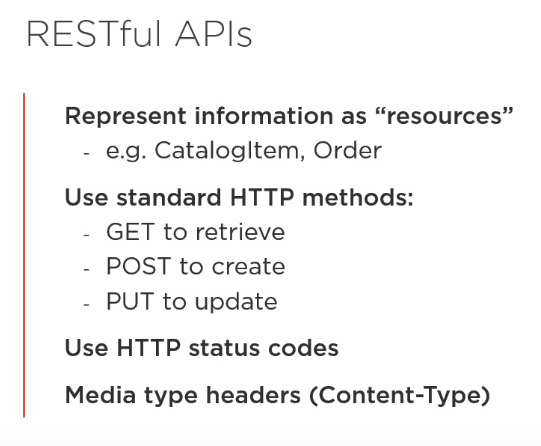
**Synchronous Communication**

Synchronous communication means making a direct call to a microservice and waiting for it to respond to us.



The above diagram is an example of Synchronous Communication. In such situation the performance is very important. HTTP is by far the most popular protocols for Synchronous Communication in microservices.

The payload for the microservices is typically going be in JSON.



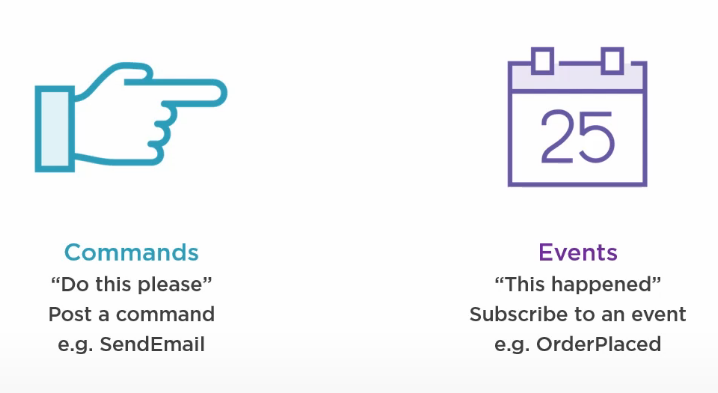
**Asynchronous Communication**

A very common pattern for asynchronous communication is for microservices to publish messages to an event bus. Rather that directly calling the other microservices, they simply create a message and send it to a message broker, which serves as an intermediary. Other Microservices then subscribe to those messages. And this pattern has a number of advantages:

1- It completely decouples the microservices from each other.

2- So, Instead of one service directly calling the next service, instead it simply talks to the message broker. This means if the second service is temporarily unavailable, then the first service is still able to function and second service can just process any queued-up messages once it’s online again. This approach is also beneficial for scaling patterns.

In Asynchronous communication the messages are of two types:

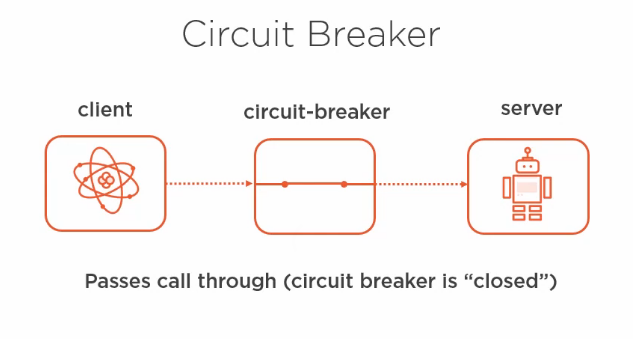


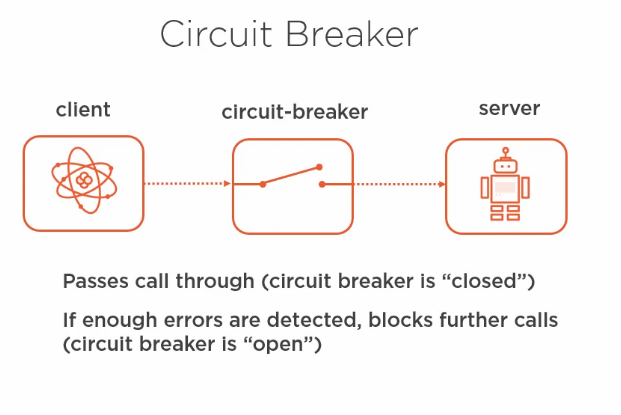
**Resilient Communication Pattern**

We can’t assume in a distributed system that all the microservices will be up and running the whole time. We can’t even assume that the network is going to be reliable. So we must expect from time to that communication from service to service will fail due to issues. And it’s very important to handle these failures well. There are several techniques and patterns that we can apply to increase the resilience of our service-to-service communication.

If you’re making any kind of network call, such as a regular HTTP Request to another service or maybe making a query to a database, then it’s a great idea to build in retires with back-off. This means that if your first attempts fails, then you wait a few seconds and try again, and if that fails, then you wait a bit longer and make another attempt. And this simple technique can often be all that’s necessary to handle various transient service outages. However if too may retries are attempted, you can end up making the situation worse, and potentially you end effectively issuing a DOS (Denial of Service) attack on your downstream services, because you are calling them so frequently.

Another pattern that can help is called as Circuit Breaker. A Circuit Breaker sits in between the client and the Server and initially it allows all calls though it. And we call this being in the closed state.

However, if it detects any errors, maybe the server is returning error codes or the server is not responding at all, then the circuit breaker opens, which means now whenever the client makes a call, it’s going to fail fast rather than passing on that request to the server.



But after a configured timeout has elapsed, the circuit allows a few calls through again to see if the downstream service has recovered. If so, then the circuit breaker closes and it allows all calls through again. If the downstream service hasn’t recovered, then it remains in the open state a bit longer, again quickly rejecting any incoming requests. And this is a very simple but powerful technique.

Message Brokers have a inherent support for resilience, We can post messages to the message broker and it doesn’t matter if our downstream services are not currently online. When they start up again, they can catch up on the backlog of messages that are waiting for them. Message Broker also often support the ability to retry sending a message. If a message handler fails for any reason, the message can be returned to the message broker to be redelivered later.

**Service Discovery**

The first one is to make use of what is called as Service Registry. This is a central Service that knows where all the other microservices are located. Sometimes it works by each service reporting its location to the service registry when it starts up and keeping in touch to allow the service registry to know that it’s still available at that address. That means whenever you want to communicate with another microservice, you can ask the service registry to tell you where that microservice can be found. And the service registry itself is typically distributed across all the machines in your cluster.

But you don’t need to build your own service registry, many microservices hosting platform solve this tricky problem for us by using DNS. For example, if you are using a PaaS platform to host your microservices, then often each microservice you deploy will automatically be allocated a DNS name that points to a load balancer sitting in front of the actual instances of your microservice. And that means you can just use the DNS name to communicate with your microservice and not have to worry about the actual IP addresses of the individual Virtual Machines that are running the service and might change over time. Or if you’re using a container orchestration platform, like Kubernetes, then you can simply refer to other services just by their service name. Kubernetes got its own built-in DNS, and that means you don’t need to know the IP address of each container. You just need to know the name of the service you want to talk to, and it takes care of routing traffic to the container that’s running your service and load balancing if necessary. So the challenge of service discovery tends to be much easier if you’re planning to host your microservice application on a modern microservice friendly platform.