**Use-Case: Shopping Cart Application**

Let’s take a classic use case of a shopping cart application.

When you open a shopping cart application, all you see is just a website. But, behind the scenes, the shopping cart application has a service for accepting payments, a service for customer services and so on.

Assume that developers of this application have created it in a monolithic framework.

Refer to the diagram below:

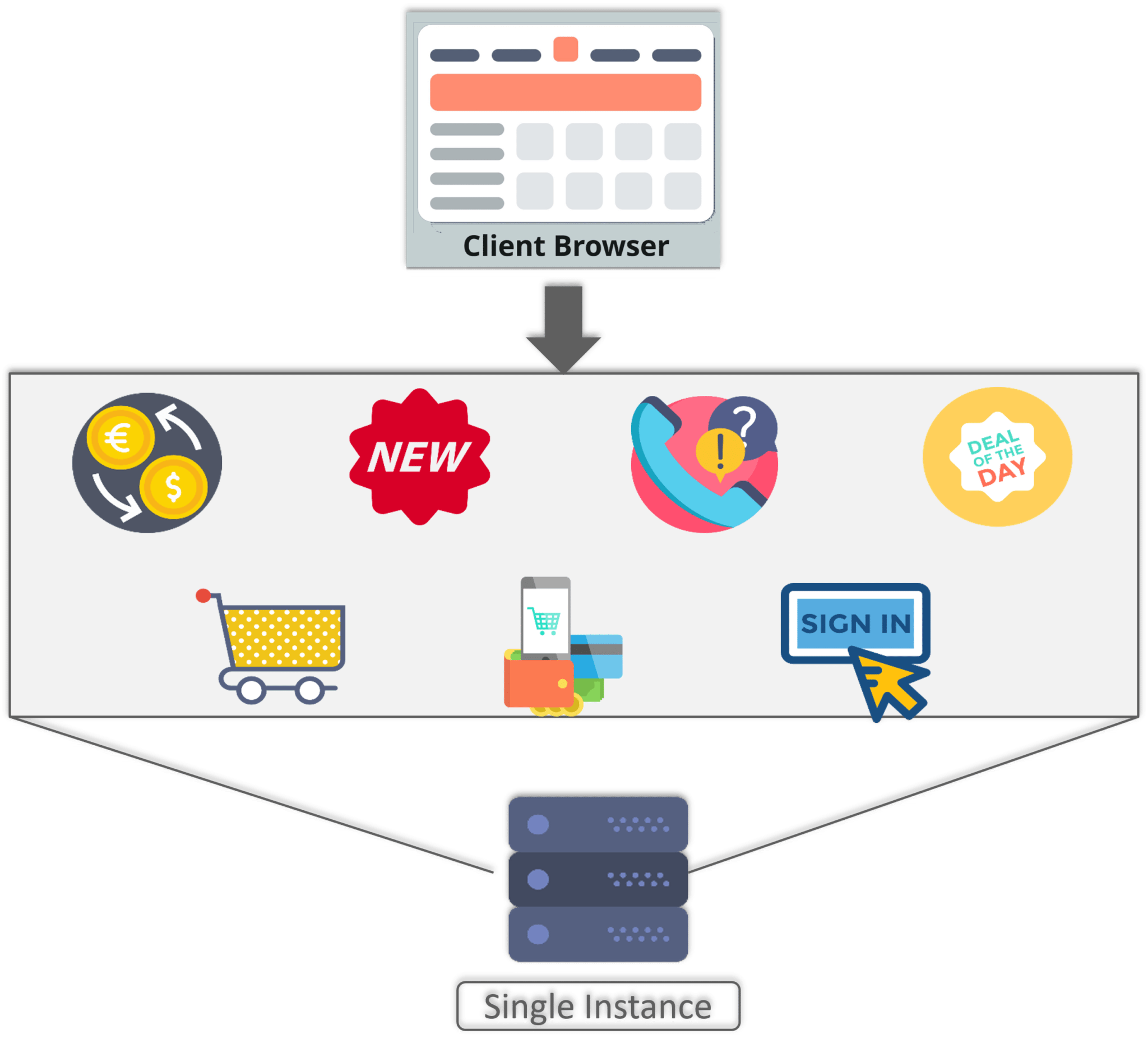


Figure 1: What Is Microservices – Monolithic Framework Of Shopping Cart Application

So, all the features are put together in a single code base and are under a single underlying database.

**Problems:**

Now, let’s suppose that there is a new brand coming up in the market and developers want to put all the details of the upcoming brand in this application.

Then, they not only have to rework on the service for new labels, but they also have to reframe the complete system and deploy it accordingly.

**Solution:**

To avoid such challenges developers of this application decided to shift their application from a monolithic architecture to micro services.

Refer to the diagram below to understand the micro services architecture of shopping cart application

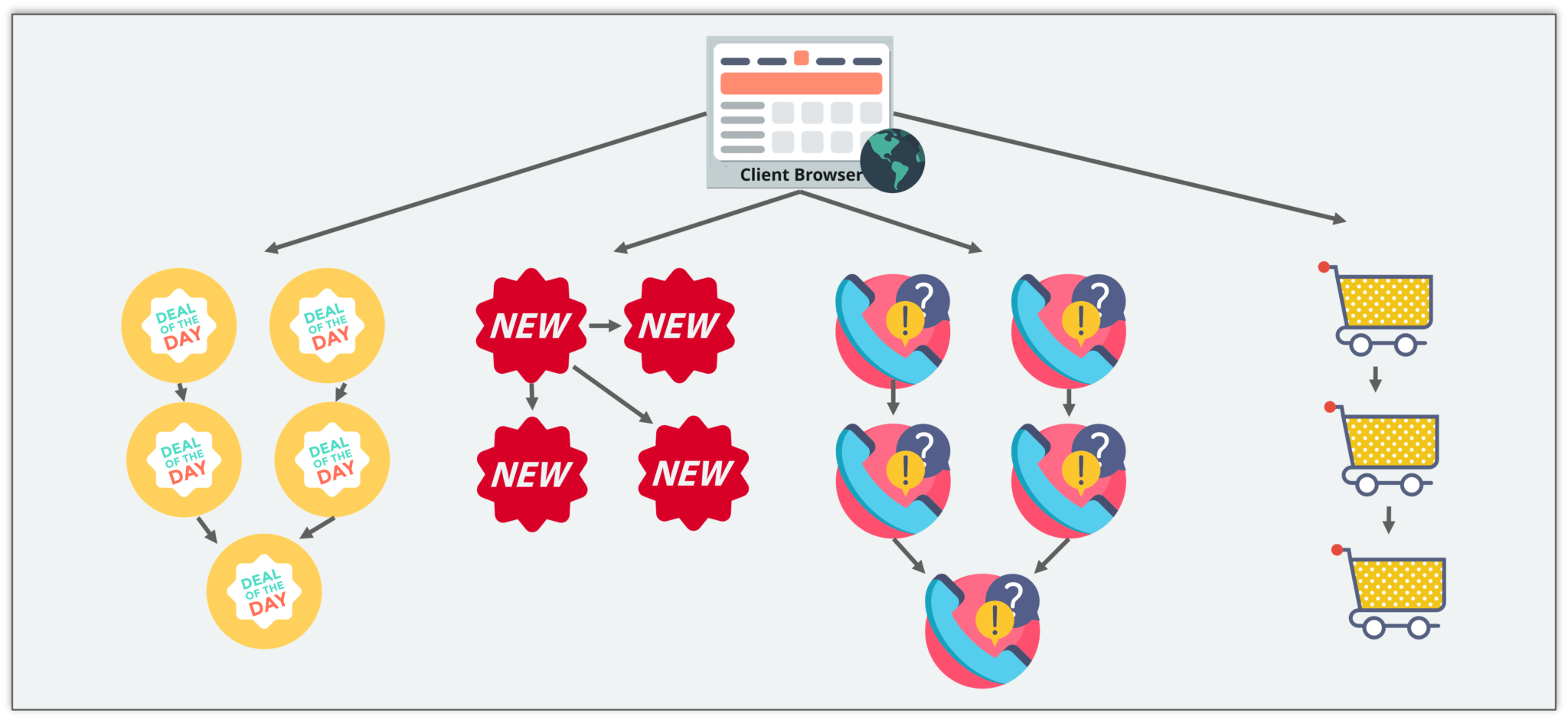


Figure 2: What Is Microservices – Microservice Architecture Of Shopping Cart Application

This means that developers don’t create a web micro service, a logic micro service, or a database micro service. Instead, they create separate micro services for search, recommendations, and customer services and so on.

This type of architecture for the application not only helps the developers to overcome all the challenges faced with the previous architecture but also helps the shopping cart application to be built, deployed, and scale up easily.

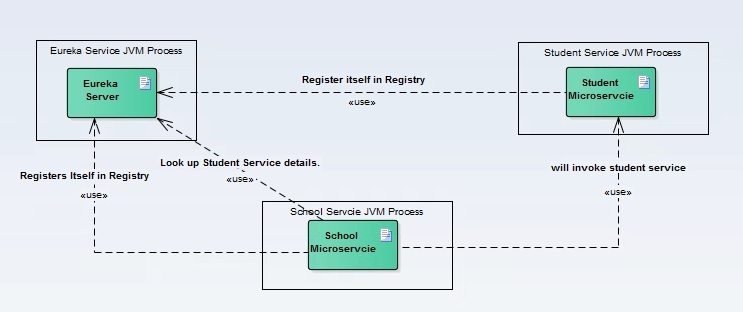
**Spring Cloud Service Discovery with Netflix Eureka**

**What is Netflix Eureka Server and Clients?**

As we know these days, there is a lot of momentum around Microservices. The transition from Monolithic to Microservice based architecture gives many benefits for future in terms of maintainability, scalability, high availability etc. However at the same time, there are many challenges also while doing this migration. One of them is to maintain individual Microservices addresses. This task can be hugely complex – depending on number of services and their dynamic nature. If whole infrastructure is distributed and there is some replication as well, then maintaining this service addresses becomes harder.

To solve this, in the distributed computing are there is a concept called ‘Service registration and discovery’ where one dedicated server is responsible to maintain the registry of all the Microservice that has been deployed and removed. This will act like a phone book of all other applications/microservices.

Think of it as a lookup service where microservices (clients) can register themselves and discover other registered microservices. When a client microservice registers with Eureka it provides metadata such as host, port, and health indicator thus allowing for other microservices to discover it. The discovery server expects a regular heartbeat message from each microservice instance. If an instance begins to consistently fail to send a heartbeat, the discovery server will remove the instance from his registry. This way we will have a very stable ecosystem of Microservices collaborating among each other, and on top of it we don’t have to manually maintain address of other Microservice, which is a next to impossible task if the scale up/down is very frequent, on demand and we use virtual host to host the services specially in the cloud environment.



# **Hystrix Circuit Breaker Pattern – Spring Cloud**

It is generally required to enable fault tolerance in the application where some underlying service is down/throwing error permanently, we need to fall back to different path of program execution automatically. This is related to distributed computing style of Eco system using lots of underlying Microservices. This is where circuit breaker pattern helps and Hystrix is an tool to build this circuit breaker.

## Why is Circuit Breaker Pattern?

If we design our systems on microservice based architecture, we will generally develop many Microservices and those will interact with each other heavily in achieving certain business goals. Now, all of us can assume that this will give expected result if all the services are up and running and response time of each service is satisfactory.

Now what will happen if any service, of the current Eco system, has some issue and stopped servicing the requests. It will result in timeouts/exception and the whole Eco system will get unstable due to this single point of failure.

Here circuit breaker pattern comes handy and it redirects traffic to a fall back path once it sees any such scenario. Also it monitors the defective service closely and restore the traffic once the service came back to normalcy.

So circuit breaker is a kind of a wrapper of the method which is doing the service call and it monitors the service health and once it gets some issue, the circuit breaker trips and all further calls goto the circuit breaker fall back and finally restores automatically once the service came back !! That’s cool right?

**Hystrix Circuit Breaker Example**

To demo circuit breaker, we will create following two microservices where first is dependent on another.

* **Student Microservice** – Which will give some basic functionality on Student entity. It will be a REST based service. We will call this service from School Service to understand Circuit Breaker. It will run on port 8098 in localhost.
* **School Microservice** – Again a simple REST based microservice where we will implement circuit breaker using Hystrix. Student Service will be invoked from here and we will test the fall back path once student service will be unavailable. It will run on port 9098 in localhost.