**Microservices**

A microservice is an engineering approach focused on **decomposing** applications into **single-function** modules with **well-defined interfaces** which are **independent**ly deployed and operated by **small teams** who own the **entire lifecycle** of the service.

**Decomposing**

So, instead of having one large application, we decompose it into separate, different, mini-applications (services).

Each service handles a specific business domain (logging, auth, orders, customers) and provides the implementation for user interface, business logic, and connection to database.

class UserApp() {  
 User getUser() {  
 // 1. auth user  
 // 2. get user data  
 // 3. log user actions  
 }  
}

class UserApp() {  
 void authUser(User user) { ... }  
 User getUserData() { ... }  
 void logUserActions() { ... }  
}

**Single-function**

Each and every service has a specific function, or responsibility. And yes, a service can do many tasks, but all of them are nevertheless relevant to this single function.

**Well-defined interfaces**

Services must provide an interface that defines how can we communicate with it. This basically defines a list of methods, and their inputs and outputs.

**Independent**

Independent means services doesn’t know about each other implementation. They can get tested, deployed, and maintained independenly.

It might be the case where services are implemented using different language stacks, and communicate with different databases.

But that doesn’t mean they don’t work together. They do, in order to complete there required operation.

class UserApp() {  
 void authUser(User user) {  
 // log user login action (success or failure)   
 // using logUserActions  
 }  
 User getUserData() { ... }  
 void logUserActions() { ... }  
}

**Small Teams**

We split the work up and team across the services. Each team focuses on a specific service, they don’t need to know about internal workings of other teams.

Those teams are can work efficiently, communicate easily, and each service can be deployed rapidly as soon as it’s ready.

**Entire Lifecycle**

The team is resposible for the entire lifecycle of the service; from coding, testing, staging, deploying, debugging, maintaining.

In traditional application, we may have a team for coding, and another one for deployment. In microservices, that’s not the case.

**Minimizing Communication**

Minimizing communication doesn’t mean that the team members should ignore each other. It means the only essential cross-team communication should be through the interface that each service provides.

They all need to agree on the external interface, so that communication between services is clearly defined.

**The scope and risk of change**

Services should be changed without breaking other services. And so long as we don’t change the external interface there will be no problem for other services.

**Monolithic Vs Mircoservices**

The choice between the two approaches depends on the context, and complexity of the application.

Indeed microservices solves problems occurs in the large application from scaling, managing, but it’s not always the way to go.

It is important to remember that microservices might be used in contexts where they are not meant to be used, resulting in extra effort and cost, project failures.

Most of the problems in Microservices are inherited as a result of having a separate components.

For example, communication between methods in monolithic is much faster when compared to services asycnrousns communications, which slower, harder to debug, and must be secured.

And for sure, there will be extra effort for operations, deployment, scaling, configuration, monitoring and testing as each service is separate.

For that being said, we need to have a skilled DevOps team to handle the complexity involved in deployment and monitoring automation.

**Circuit Breaker or Fallback Method or Fault Tolerance**

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.

@EnableHystrixDashboard

@EnableCircuitBreaker

public class DemoschoolApplication {

}

@RequestMapping(value = "/getSchoolDetails/{schoolname}", method = RequestMethod.***GET***)

**public** String getStudents(@PathVariable String schoolname) {

System.***out***.println("Going to call student service to get data!");

**return** studentServiceDelegate.callStudentServiceAndGetData(schoolname);

}

@Service

**public** **class** StudentServiceDelegate {

@Autowired

RestTemplate restTemplate;

@HystrixCommand(fallbackMethod = "**callStudentServiceAndGetData\_Fallback**")

**public** String callStudentServiceAndGetData(String schoolname) {

System.***out***.println("Getting School details for " + schoolname);

String response = restTemplate.exchange("http://localhost:8098/getStudentDetailsForSchool/{schoolname}"

, HttpMethod.***GET***

,**null**

, **new** ParameterizedTypeReference<String>() {

}, schoolname).getBody();

System.***out***.println("Response Received as " + response + " -  " + **new** Date());

**return** "NORMAL FLOW !!! - School Name -  " + schoolname + " :::  " +

" Student Details " + response + " -  " + **new** Date();

}

@SuppressWarnings("unused")

**private** String **callStudentServiceAndGetData\_Fallback**(String schoolname) {

System.***out***.println("Student Service is down!!! fallback route enabled...");

**return** "CIRCUIT BREAKER ENABLED!!! No Response From Student Service at this moment. " +

" Service will be back shortly - " + **new** Date();

}

@Bean

**public** RestTemplate restTemplate() {

**return** **new** RestTemplate();

}

Ok. So, we’re done with the Intro to microservices. Next, we’ll start implementing our microservices.

# [Difference between PUT and POST in REST WebService in Java](https://javarevisited.blogspot.com/2016/10/difference-between-put-and-post-in-restful-web-service.html)

If you remember REST WebServices uses HTTP methods to map CRUD (create, retrieve, update, delete) operations to HTTP requests. Even though both PUT and POST methods can be used to perform create and update operation in REST WebServices, Idempotency is the main difference between PUT and POST. Similar to the [GET request](http://www.java67.com/2014/08/difference-between-post-and-get-request.html), PUT request is also idempotent in HTTP, which means it will produce the same results if executed once more multiple times. Another practical difference PUT and POST method in the context of REST WebService are that **POST is often used to create a new entity**, and PUT is often used to update an existing entity. If you replace an existing entity using PUT than you should be aware that if only a subset of data elements is passed then the rest will be replaced by empty or null.  
  
There is also another theory which says that for creating new things, you should use PUT if the unique identifier is provided by client i.e. client is responsible for creating entity e.g. client can create resource /user/joe/ by providing username joe and that would be unique URI. Similar, use POST if the server is responsible for creating new resources e.g. if ID is part of URI and automatically created by the server.  
  
Let's see a couple of more differences between PUT and POST in REST WebServices.

## PUT vs POST in REST WebService

As I said, even though both PUT and POST can be used to create and update an entity, POST is usually preferred for creating and PUT is preferred for updating an existing entity.  
  
For example, to create a new Order you should use:  
  
POST /orders  
  
and to update an existing order, you should use  
  
PUT /orders/13892  
  
which means modify the order with OrderId 13892  
  
If you execute POST request multiple times, it will end up create that many orders, but when you execute PUT it will always produce the same result because of its [idempotent](http://javarevisited.blogspot.com/2016/05/what-are-idempotent-and-safe-methods-of-HTTP-and-REST.html). You should also remember that both PUT and POST are **unsafe methods**. Safe methods in HTTP do not modify the resource in the server e..g GET or HEAD, while Idempotent HTTP methods return same result irrespective of how many times you call them.  
  
  
Idempotency is an important thing while building a fault-tolerant RESTful API. Idempotency is also the reason of why you should use PUT over POST to update a resource in REST. For example, suppose a client wants to update a resource through POST. Since POST is not an idempotent method, calling it multiple times may result in incorrect updates.  
  
In the real world its quite likely that your POST request may timeout, what will happen to the resource that. Is the resource actually updated? Does the timeout happened during sending the request to the server, or the response to the client? Can we safely retry again, or do we need to figure out first what has happened with the resource? By using idempotent methods like PUT, you don't have to answer this question, but we can safely resend the request until we actually get a response back from the server.  
  
See [HTTP: The Definitive Guide](http://www.amazon.com/HTTP-Definitive-Guide-Guides/dp/1565925092?tag=javamysqlanta-20)  by David Gourley to learn more about idempotent and safe methods. Remember, strong knowledge of HTTP is key to success in REST.  
When to use PUT and POST methods in REST?

Now' it's time for some practical knowledge about when to use the PUT and POST methods to call RESTful WebServices.  
  
1) You should use POST to create new resources and PUT to update existing resources.  
  
2) Use PUT when you know the "id" of the object e.g. Order, Book, Employee  
  
3) Use POST when you need the server to be in control of URL generation of your resources.  
  
4) Examples  
PUT /items/1 update  
POST /items create

[Difference between PST and PUT in REST](http://www.amazon.com/REST-Practice-Hypermedia-Systems-Architecture/dp/0596805829?tag=javamysqlanta-20)

Btw, there is also another theory, which states that **if the client is responsible for creating the ID's** of the resource, use PUT to create new things e.g.  
  
PUT /users/john  
  
Here John is unique and given by the client, this is a new URI.  
  
Similarly, if server is responsible for creating the ID's of the new resources then use POST, for example  
  
POST /users/  
  
Now, POST will carry a key and value which client uses to send username=john and ID will be automatically generated by Server.

[Difference between PUT and POST in REST WebService in Java](https://2.bp.blogspot.com/-60VdsNLAhsk/WA7B-tIOZgI/AAAAAAAAHZo/_Ggqm1I4NtgSz2txFndVkkQBwO_9N5FuQCLcB/s1600/REST+API+design+PUT+vs+POST.jpg)