**Resource:**

# Microservice Architectures With Spring Cloud and Docker

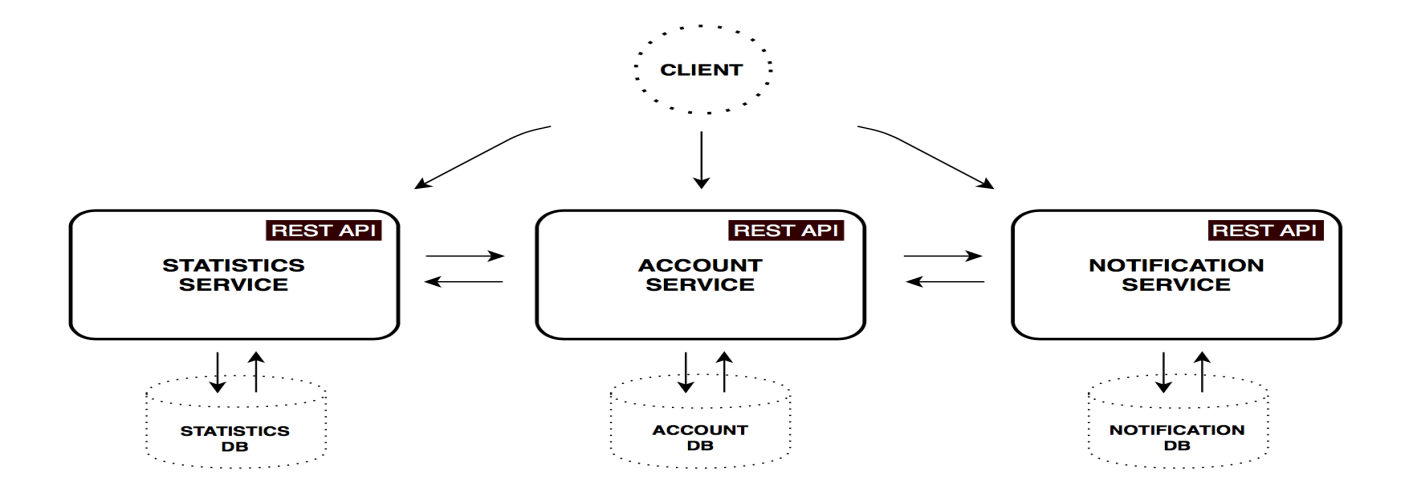
This article provides a starting point for understanding common Microservice architecture patterns by example of a proof-of-concept application built with Spring Boot, Spring Cloud, and Docker.

The code is available [on Github](https://github.com/sqshq/PiggyMetrics), and images are available on Docker Hub. You can start the whole system with just one command.

As a basis for this system I chose an old project, whose backend used to be a monolith. The application provides a way to deal with personal finances, organize incomes and expenses, manage savings, analyze statistics, and create simple forecasts.

## Functional Services

The monolith application was decomposed into three core microservices. All of them are independently deployable applications, organized around certain business capabilities.



#### Account Service

Contains general user input logic and validation: incomes/expenses items, savings, and account settings.

| **METHOD** | **PATH** | **DESCRIPTION** | **USER AUTHENTICATED** | **AVAILABLE FROM UI** |
| --- | --- | --- | --- | --- |
| GET | /accounts/{account} | Get specified account data |  |  |
| GET | /accounts/current | Get current account data | × | × |
| GET | /accounts/demo | Get demo account data (pre-filled incomes/expenses items, etc) |  | × |
| PUT | /accounts/current | Save current account data | × | × |
| POST | /accounts/ | Register new account |  | × |

#### Statistics Service

Performs calculations on major statistics parameters and captures time series for each account. A datapoint contains values normalized to base currency and time period. This data might be used to track cash flow dynamics in an account's lifetime.

| **METHOD** | **PATH** | **DESCRIPTION** | **USER AUTHENTICATED** | **AVAILABLE FROM UI** |
| --- | --- | --- | --- | --- |
| GET | /statistics/{account} | Get specified account statistics |  |  |
| GET | /statistics/current | Get current account statistics | × | × |
| GET | /statistics/demo | Get demo account statistics |  | × |
| PUT | /statistics/{account} | Create or update time series datapoint for specified account |  |  |

#### Notification Service

Stores a user's contact information and notification settings (like remind and backup frequency). Scheduled worker collects required information from other services and sends e-mail messages to subscribed customers.

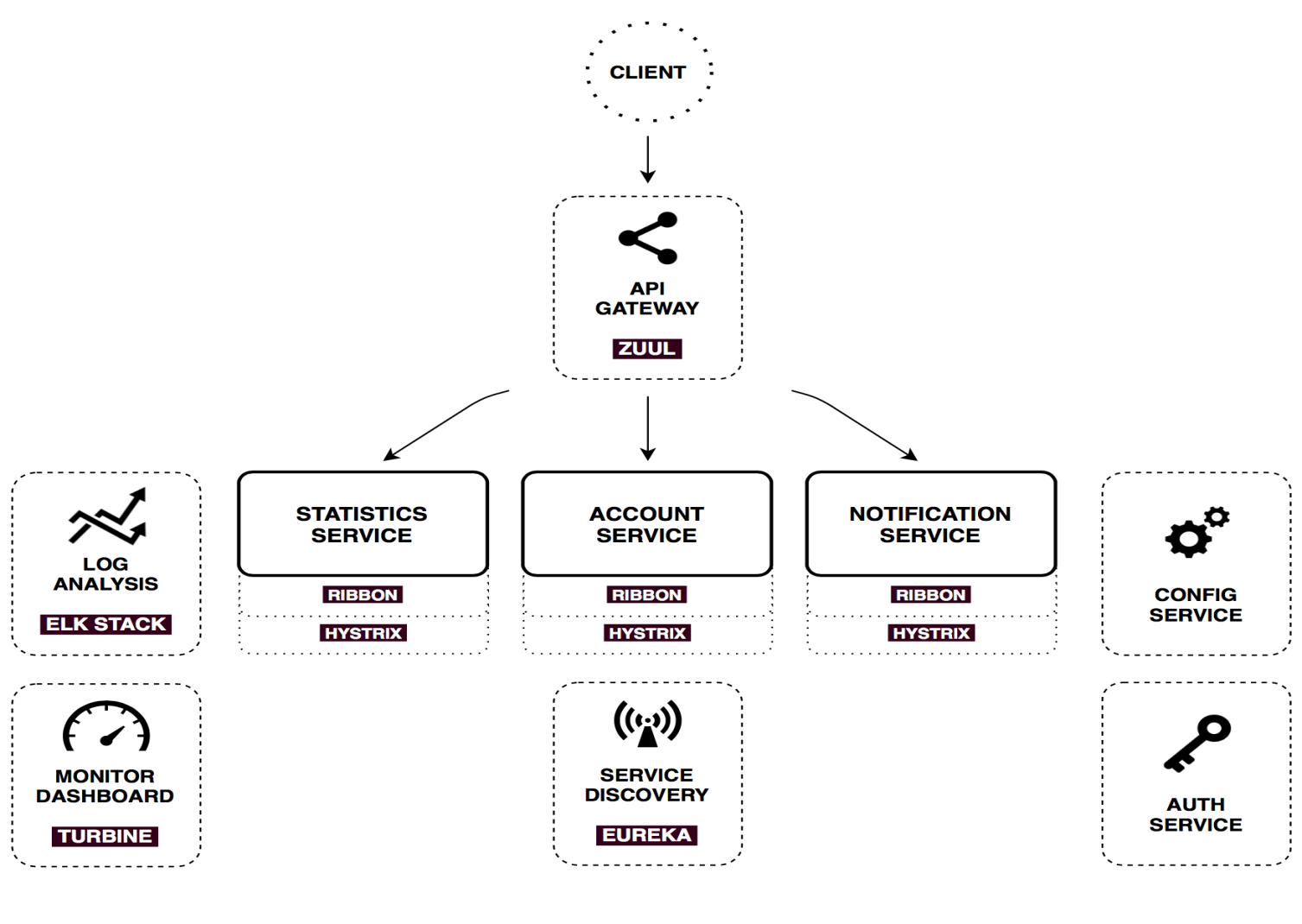
| **METHOD** | **PATH** | **DESCRIPTION** | **USER AUTHENTICATED** | **AVAILABLE FROM UI** |
| --- | --- | --- | --- | --- |
| GET | /notifications/settings/current | Get current account notification settings | × | × |
| PUT | /notifications/settings/current | Save current account notification settings | × | × |

#### Notes

* Each microservice has it's own database, so there is no way to bypass the API and access persistance data directly.
* For this project, I used MongoDB as the primary database for each service. It also might make sense to have a polyglot persistence architecture (to сhoose the type of database that is best suited to the service requirements).
* Service-to-service communication is quite simplified: microservices talking using only synchronous REST API. Common practice in a real-world systems is to use combination of interaction styles. For example, perform synchronous GET request to retrieve data and use asynchronous approach via Message broker for create/update operations in order to decouple services and buffer messages. However, this brings us in [eventual consistency](http://martinfowler.com/articles/microservice-trade-offs.html#consistency) world.

## Infrastructure Services

There's a bunch of common patterns in distributed systems, which could help us to make described core services work. [Spring cloud](http://projects.spring.io/spring-cloud/) provides powerful tools that enhance Spring Boot applications behaviour to implement those patterns. I'll cover them briefly.



### Config Service

[Spring Cloud Config](http://cloud.spring.io/spring-cloud-config/spring-cloud-config.html) is horizontally scalable centralized configuration service for distributed systems. It uses a pluggable repository layer that currently supports local storage, Git, and Subversion.

In this project, I use native profile, which simply loads config files from the local classpath. You can see shareddirectory in [Config service resources](https://github.com/sqshq/PiggyMetrics/tree/master/config/src/main/resources). Now, when Notification-service requests it's configuration, Config service responses with shared/notification-service.yml and shared/application.yml (which is shared between all client applications).

#### Client-side Usage

Just build Spring Boot application with spring-cloud-starter-config dependency, autoconfiguration will do the rest.

Now you don't need any embedded properties in your application. Just provide bootstrap.yml with the application name and Config service url:

spring:

application:

name: notification-service

cloud:

config:

uri: http://config:8888

fail-fast: true

#### With Spring Cloud Config, You Can Change App Configuration Dynamically

For example, the [EmailService bean](https://github.com/sqshq/PiggyMetrics/blob/master/notification-service/src/main/java/com/piggymetrics/notification/service/EmailServiceImpl.java) was annotated with @RefreshScope. That means you can change e-mail text and subject lines without rebuilding and restarting the Notification service application.

First, change the required properties in the Config server. Then, perform the refresh request to the Notification service: curl -H "Authorization: Bearer #token#" -XPOST http://127.0.0.1:8000/notifications/refresh

You could also use [webhooks to automate this process](http://cloud.spring.io/spring-cloud-config/spring-cloud-config.html" \l "_push_notifications_and_spring_cloud_bus).

#### Notes

* There are some limitations for dynamic refreshes though. @RefreshScope doesn't work with @Configuration classes and can't affect @Scheduled methods.
* fail-fast property means that the Spring Boot application will fail startup immediately if it cannot connect to the Config Service. That's very useful when you're starting [all applications together](https://github.com/sqshq/PiggyMetrics#how-to-run-all-the-things).
* There are significant security notes below.

### Auth Service

Authorization responsibilities are completely extracted to separate server, which grants [OAuth2 tokens](https://tools.ietf.org/html/rfc6749) for backend resource services. Auth Server is used for user authorization as well as for secure machine-to-machine communication inside a perimeter.

In this project, I use [Password credentials](https://tools.ietf.org/html/rfc6749#section-4.3) as a grant type for user authorization (since it's only used by the native application UI) and [Client Credentials](https://tools.ietf.org/html/rfc6749#section-4.4) as a grant type for microservices authorization.

Spring Cloud Security provides convenient annotations and autoconfigurations to make this really easy to implement from both the server and client side. You can learn more about it in the [documentation](http://cloud.spring.io/spring-cloud-security/spring-cloud-security.html) and check configuration details in [Auth Server code](https://github.com/sqshq/PiggyMetrics/tree/master/auth-service/src/main/java/com/piggymetrics/auth).

From the client side, everything works exactly the same as with traditional session-based authorization. You can retrieve Principal objects from request, check user roles and other stuff with expression-based access control and @PreAuthorize annotation.

Each client in PiggyMetrics (account-service, statistics-service, notification-service and browser) has a scope: serverfor backend services, and ui - for the browser. So we can also protect controllers from external access, for example:

@PreAuthorize("#oauth2.hasScope('server')")

@RequestMapping(value = "accounts/{name}", method = RequestMethod.GET)

public List<DataPoint> getStatisticsByAccountName(@PathVariable String name) {

return statisticsService.findByAccountName(name);

}

### API Gateway

As you can see, there are three core services, which expose external APIs to the client. In a real-world system, this number can grow very quickly as well as whole system complexity. Actuallyy, [hundreds of services](http://highscalability.com/amazon-architecture) might be involved in rendering one complex webpage.

In theory, a client could make requests to each of the microservices directly. But obviously there are challenges and limitations with this option, like necessity to know all endpoints addresses, perform http request for each peace of information separately, merge the result on a client side. Another problem is non-web-friendly protocols, which might be used on the backend.

Usually a much better approach is to use an API Gateway. It is a single entry point into the system, used to handle requests by routing them to the appropriate backend service or by invoking multiple backend services and [aggregating the results](http://techblog.netflix.com/2013/01/optimizing-netflix-api.html). Also, it can be used for authentication, insights, stress and canary testing, service migration, static response handling, active traffic management.

Netflix open sourced [such an edge service](http://techblog.netflix.com/2013/06/announcing-zuul-edge-service-in-cloud.html), and now with Spring Cloud we can enable it with one @EnableZuulProxyannotation. In this project, I use Zuul to store static content (UI application) and to route requests to the appropriate microservices. Here's a simple prefix-based routing configuration for the Notification service:

zuul:

routes:

notification-service:

path: /notifications/\*\*

serviceId: notification-service

stripPrefix: false

That means all requests starting with /notifications will be routed to Notification service. There is no hardcoded address, as you can see. Zuul uses [Service discovery](https://github.com/sqshq/PiggyMetrics/blob/master/README.md#service-discovery) mechanism to locate Notification service instances and also [Circuit Breaker and Load Balancer](https://github.com/sqshq/PiggyMetrics/blob/master/README.md#http-client-load-balancer-and-circuit-breaker), described below.

### Service Discovery

Another commonly known architecture pattern is service discovery. It allows automatic detection of network locations for service instances, which could have dynamically assigned addresses because of auto-scaling, failures, and upgrades.

The key part of service discovery is the registry. I used Netflix Eureka for this project. Eureka is a good example of the client-side discovery pattern, when the client is responsible for determining the locations of available service instances (using a registry server) and load balancing requests across them.

With Spring Boot, you can easily build Eureka Registry with a spring-cloud-starter-eureka-server dependency, @EnableEurekaServer annotation, and simple configuration properties.

Client support is enabled with @EnableDiscoveryClient annotation and bootstrap.yml with application name:

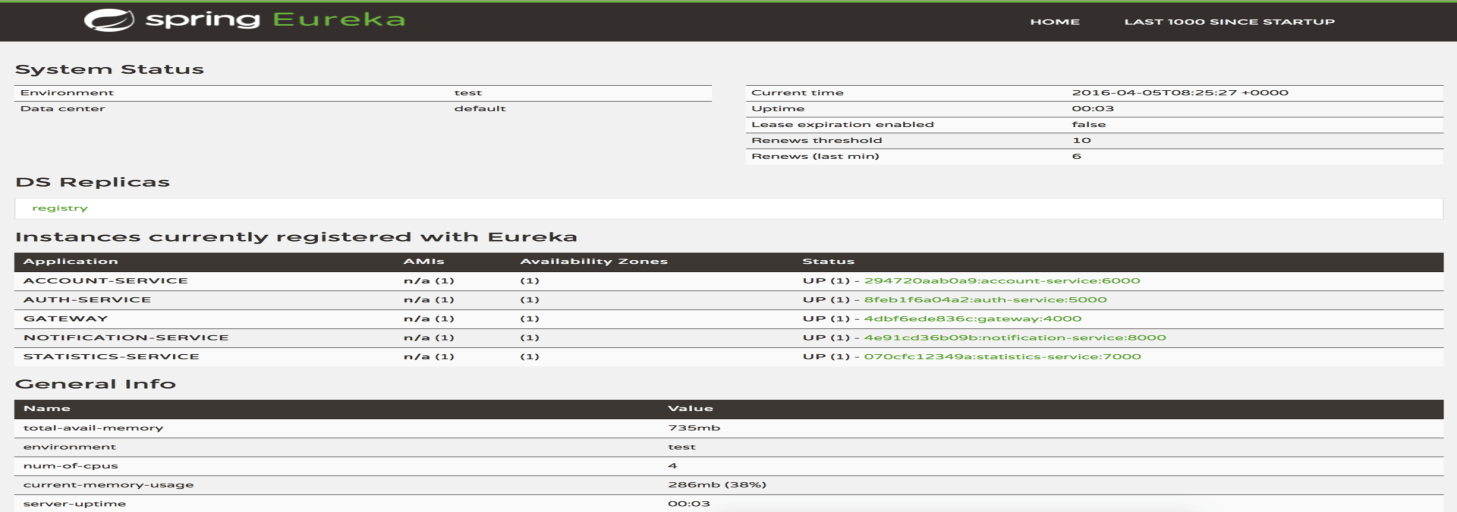
spring:

application:

name: notification-service

Now, on application startup, it will register with Eureka Server and provide meta-data, such as host and port, health indicator URL, home page, etc. Eureka receives heartbeat messages from each instance belonging to a service. If the heartbeat fails over a configurable timetable, the instance will be removed from the registry.

Also, Eureka provides a simple interface, where you can track running services and the number of available instances: http://localhost:8761



### Load Balancer, Circuit Breaker, and Http Client

Netflix OSS provides another great set of tools.

#### Ribbon

Ribbon is a client side load balancer which gives you a lot of control over the behavior of HTTP and TCP clients. Compared to a traditional load balancer, there is no need of an additional hop for every over-the-wire invocation — you can contact the desired service directly.

Out of the box, it natively integrates with Spring Cloud and Service Discovery. [Eureka Client](https://github.com/sqshq/PiggyMetrics#service-discovery) provides a dynamic list of available servers so Ribbon could balance between them.

#### Hystrix

Hystrix is the implementation of a [Circuit Breaker pattern](http://martinfowler.com/bliki/CircuitBreaker.html), which gives a control over latency and failure from dependencies accessed over the network. The main idea is to stop cascading failures in a distributed environment with a large number of microservices. That helps to fail fast and recover as soon as possible — important aspects of fault-tolerant systems that self-heal.

Besides circuit breaker control, with Hystrix you can add a fallback method that will be called to obtain a default value in case the main command fails.

Moreover, Hystrix generates metrics on execution outcomes and latency for each command, that we can use to [monitor system behavior](https://github.com/sqshq/PiggyMetrics#monitor-dashboard).

#### Feign

Feign is a declarative HTTP client, which seamlessly integrates with Ribbon and Hystrix. Actually, with one spring-cloud-starter-feign dependency and @EnableFeignClients annotation you have a full suite of a load balancer, circuit breaker, and HTTP client with a sensible ready-to-go default configuration.

Here is an example from Account Service:

@FeignClient(name = "statistics-service")

public interface StatisticsServiceClient {

@RequestMapping(method = RequestMethod.PUT, value = "/statistics/{accountName}", consumes = MediaType.APPLICATION\_JSON\_UTF8\_VALUE)

void updateStatistics(@PathVariable("accountName") String accountName, Account account);

}

* Everything you need is just an interface
* You can share @RequestMapping part between Spring MVC controller and Feign methods
* Above example specifies just desired service id - statistics-service, thanks to autodiscovery through Eureka (but obviously you can access any resource with a specific url)

### Monitor Dashboard

In this project configuration, each microservice with Hystrix on board pushes metrics to Turbine via Spring Cloud Bus (with AMQP broker). The Monitoring project is just a small Spring boot application with [Turbine](https://github.com/Netflix/Turbine) and [Hystrix Dashboard](https://github.com/Netflix/Hystrix/tree/master/hystrix-dashboard).

Let's see our system behavior under load: Account service calls Statistics service and it responses with a vary imitation delay. Response timeout threshold is set to 1 second.

[https://cloud.githubusercontent.com/assets/6069066/14194375/d9a2dd80-f7be-11e5-8bcc-9a2fce753cfe.png](https://cloud.githubusercontent.com/assets/6069066/14194375/d9a2dd80-f7be-11e5-8bcc-9a2fce753cfe.png)

| **<https://cloud.githubusercontent.com/assets/6069066/14127349/21e90026-f628-11e5-83f1-60108cb33490.gif>** | **<https://cloud.githubusercontent.com/assets/6069066/14127348/21e6ed40-f628-11e5-9fa4-ed527bf35129.gif>** | **<https://cloud.githubusercontent.com/assets/6069066/14127346/21b9aaa6-f628-11e5-9bba-aaccab60fd69.gif>** | **<https://cloud.githubusercontent.com/assets/6069066/14127350/21eafe1c-f628-11e5-8ccd-a6b6873c046a.gif>** |
| --- | --- | --- | --- |
| 0 ms delay | 500 ms delay | 800 ms delay | 1100 ms delay |
| Well behaving system. The throughput is about 22 requests/second. Small number of active threads in Statistics service. The median service time is about 50 ms. | The number of active threads is growing. We can see purple number of thread-pool rejections and therefore about 30-40% of errors, but circuit is still closed. | Half-open state: the ratio of failed commands is more than 50%, the circuit breaker kicks in. After sleep window amount of time, the next request is let through. | 100 percent of the requests fail. The circuit is now permanently open. Retry after sleep time won't close circuit again, because the single request is too slow. |

### Log Analysis

Centralized logging can be very useful when attempting to identify problems in a distributed environment. Elasticsearch, Logstash, and Kibana stack lets you search and analyze your logs, utilization and network activity data with ease. Ready-to-go Docker configuration is described [in my other project](http://github.com/sqshq/ELK-docker).

## Security

An advanced security configuration is beyond the scope of this proof-of-concept project. For a more realistic simulation of a real system, consider using https and JCE keystore to encrypt microservices passwords and Config server properties content (see [documentation](http://cloud.spring.io/spring-cloud-config/spring-cloud-config.html#_security) for details).

**Sample Interview Questions:**

1. What is Architecture (Software)?

Architecture is the fundamental organization of a system embodied in its components (i.e. Web Server, Application Server, Databases, Storage, Communication layer, etc...), their relationships to each other, and to the environment (i.e. deployment environment shared server, dedicated server, cloud deployment, etc..), and the principles guiding its design and evolution.

2. What is microservice architecture?

Microservice means developing a single, small, meaningful functional feature as single service, each service has its own process and communicates with lightweight mechanism, deployed in single or multiple servers.

3. Advantages of microservice architecture?

* Each micro service is small and focused on a specific feature / business requirement.
* Microservice can be developed independently by small team of developers (normally 2 to 5 developers).
* Microservice is loosely coupled, means services are independent, in terms of development and deployment both.
* Microservice can be developed using different programming language (Personally I don't suggest doing it).
* Microservice allows easy and flexible way to integrate automatic deployment with Continuous Integration tools (for e.g.: Jenkins, Hudson, bamboo etc...).
* The productivity of a new team member will be quick enough.
* Microservice is easy to understand, modify and maintain for a developer because separation of code, small team and focused work.
* Microservice allows you to take advantage of emerging and latest technologies (framework, programming language, programming practice, etc.).
* Microservice has code for business logic only, No mix-up with HTML, CSS or other UI component.
* Microservice is easy to scale based on demand.
* Microservice can deploy on commodity hardware or low / medium configuration servers.
* Easy to integrate 3rd party service.
* Every microservice has it's own storage capability but it depends on the project’s requirement, you can have common database like MySQL or Oracle for all services.

4. Disadvantages of microservice architecture?

* Microservice architecture brings a lot of operations overhead.
* DevOps Skill required (http://en.wikipedia.org/wiki/DevOps).
* Duplication of Effort.
* Distributed System is complicated to manage.
* Default to trace problem because of distributed deployment.
* Complicated to manage whole products when number of services increases.

5. In which case / requirement microservice architecture best fit ?

When you need to support Desktop, web , mobile, Smart TVs, Wearable, etc... or you don't know in future which kind of devices you need to support.

6. Which products / companies are using Microservice architecture?

Most large scale web sites including Twitter, Netflix, Amazon and eBay have evolved from a monolithic architecture to microservices architecture.

7. How independent micro services communicate with each other?

It's depend upon requirement, normally developers use HTTP/REST with JSON or Protobuf(Binary protocol) but are free to use any communication protocol.