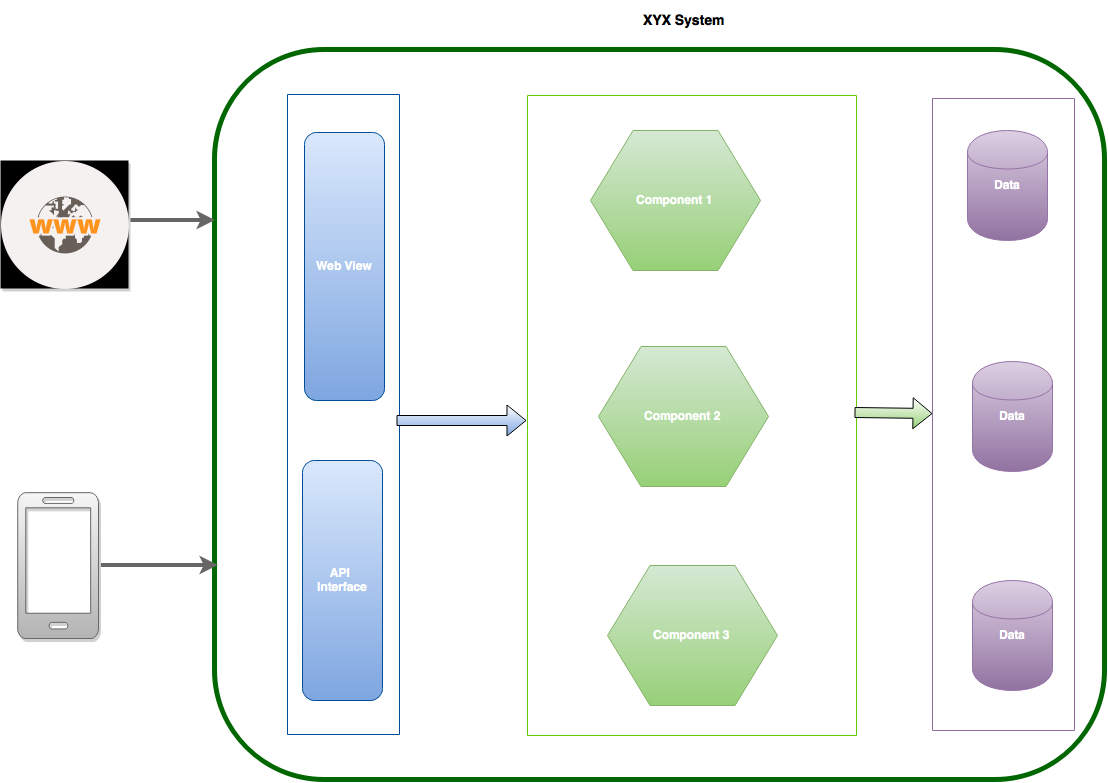
In monolithic architectural design we create a big cumbersome application with all modules tightly coupled inside a single executable, which is typically deployed on a web or application server.

A typical monolithic architecture application looks like:



There are some disadvantages to this architectural design. These disadvantages or drawbacks have become the strengths of microservices architecture:

1.No frequent and easy releases - As monolithic applications grow in size, due to tight coupling between components, it becomes difficult to do easy and frequent releases.

2.Problem in continuous delivery - We may not notice this problem if the application is small. In case of bigger monolithic applications, deployment times can be frustratingly long and slow. If a single change to the application would require the entire application to be redeployed, then this could become an obstacle to frequent deployments, and thus an impediment to continuous delivery. This could be a serious issue if you are serving a mobile application where users expect the latest cool new features all the time.

3.Difficult to manage team and project - Project management has its own challenges in monolithic application development. Even a modularized application has interdependency in terms of deployment and release. It takes a toll in terms of time and effort to plan the release and manage tightly coupled interdependent modular development.

4.Lack of technology diversity - When we choose a technology stack for a monolithic application, we consider a balanced stack which can serve well for all of our requirements. We can not employ specific technology for specialized needs.

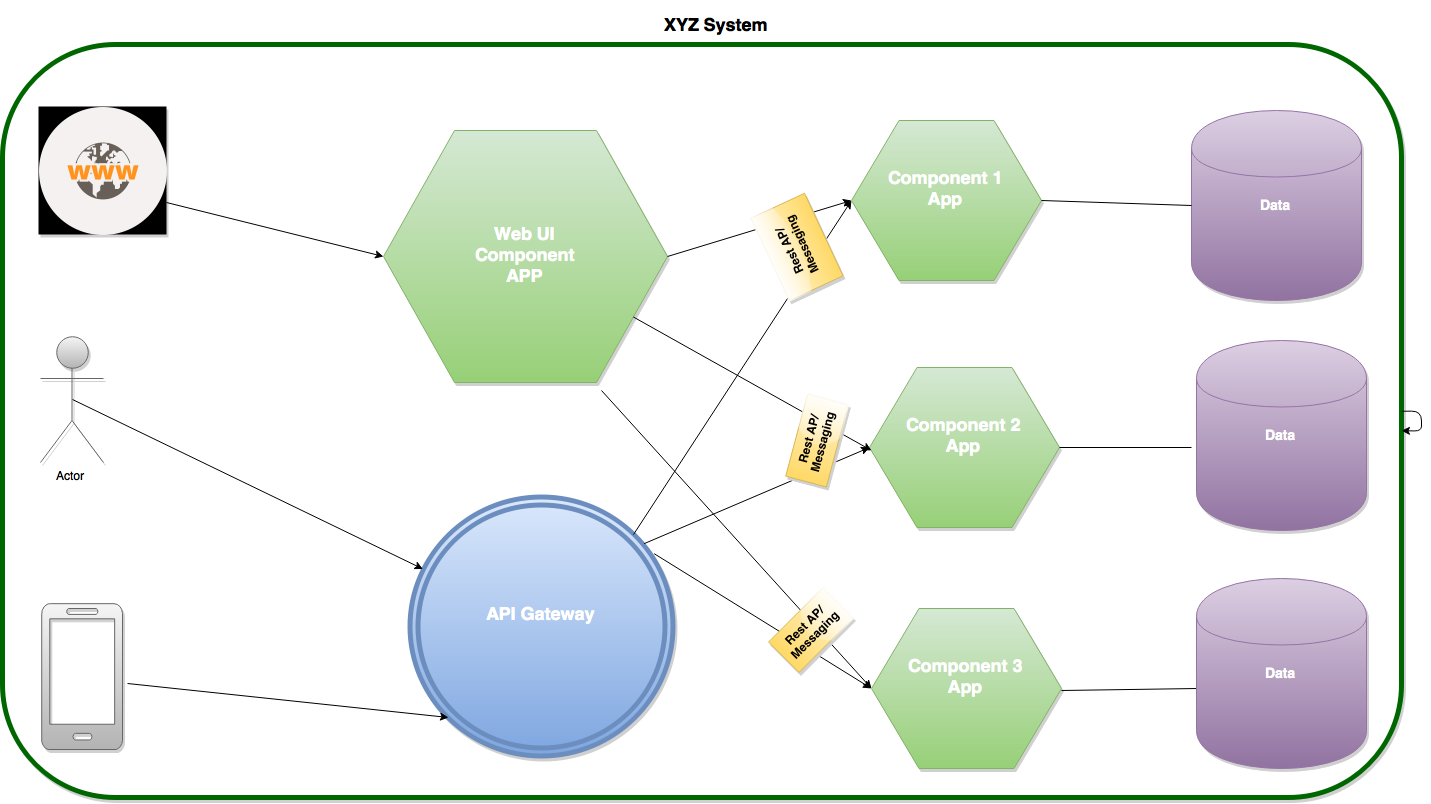
5.Not easy to replace components - It is very difficult to replace any component with a better design or performance without affecting the whole architecture.

**Microservices Architecture :-**

Microservices architectural style defines a setup, where application components are standalone applications of their own. These independent application components talk to each other either using RMI (Remote Method Invocation), Restful Web Services or Push Messaging.

Microservices architecture - is an architectural style that structures an application as a collection of loosely coupled services, which implement business capabilities. The microservice architecture enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack.

A typical Microservices based application setup is given here:



While designing systems in microservices architecture, we should be identifying independent components/modules appropriately. These components will be mini applications, which will be developed separately. They will follow their own development and deployment lifecycle.

Consider we are developing one school management system. In a school management system we have various important components like student registration, attendance, fees, assessments, etc.

**Service Discovery: Eureka Clients**

Eureka is the Netflix Service Discovery Server and Client. The server can be configured and deployed to be highly available, with each server replicating state about the registered services to the others.

To include Eureka Client in your project use the starter with group org.springframework.cloud and artifact id spring-cloud-starter-netflix-eureka-client.

When a client registers with Eureka, it provides meta-data about itself 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 is normally removed from the registry.

By having spring-cloud-starter-netflix-eureka-client on the classpath your application will automatically register with the Eureka Server. Configuration is required to locate the Eureka server. Example:

**application.yml.**

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

where "defaultZone" is a magic string fallback value that provides the service URL for any client that doesn’t express a preference (i.e. it’s a useful default).

The default application name (service ID), virtual host and non-secure port, taken from the Environment, are ${spring.application.name},${spring.application.name} and ${server.port} respectively.

To disable the Eureka Discovery Client you can set eureka.client.enabled to false.

**1.3 Authenticating with the Eureka Server**

HTTP basic authentication will be automatically added to your eureka client if one of the eureka.client.serviceUrl.defaultZone URLs has

credentials embedded in it (curl style, like http://user:password@localhost:8761/eureka ).

## 1.5 Registering a Secure Application

If your app wants to be contacted over HTTPS you can set two flags in the EurekaInstanceConfig, vizeureka.instance.[nonSecurePortEnabled,securePortEnabled]=[false,true] respectively. This will make Eureka publish instance information showing an explicit preference for secure communication. The Spring Cloud DiscoveryClient will always return a URI starting with https for a service configured this way, and the Eureka (native) instance information will have a secure health check URL.

Because of the way Eureka works internally, it will still publish a non-secure URL for status and home page unless you also override those explicitly. You can use placeholders to configure the eureka instance urls, e.g.

**application.yml.**

eureka:

instance:

statusPageUrl: https://${eureka.hostname}/info

healthCheckUrl: https://${eureka.hostname}/health

homePageUrl: https://${eureka.hostname}/

(Note that ${eureka.hostname} is a native placeholder only available in later versions of Eureka. You could achieve the same thing with Spring placeholders as well, e.g. using ${eureka.instance.hostName}.)

## 1.6 Eureka’s Health Checks

By default, Eureka uses the client heartbeat to determine if a client is up. Unless specified otherwise the Discovery Client will not propagate the current health check status of the application per the Spring Boot Actuator.

**application.yml.**

eureka:

client:

healthcheck:

enabled: true

|  |
| --- |
| [Warning] |
| eureka.client.healthcheck.enabled=true should only be set in application.yml. Setting the value in bootstrap.yml will cause undesirable side effects like registering in eureka with an UNKNOWN status. |

If you require more control over the health checks, you may consider implementing your own com.netflix.appinfo.HealthCheckHandler.

### EurekaClient without Jersey

By default, EurekaClient uses Jersey for HTTP communication. If you wish to avoid dependencies from Jersey, you can exclude it from your dependencies. Spring Cloud will auto configure a transport client based on Spring RestTemplate.

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

<exclusions>

<exclusion>

<groupId>com.sun.jersey</groupId>

<artifactId>jersey-client</artifactId>

</exclusion>

<exclusion>

<groupId>com.sun.jersey</groupId>

<artifactId>jersey-core</artifactId>

</exclusion>

<exclusion>

<groupId>com.sun.jersey.contribs</groupId>

<artifactId>jersey-apache-client4</artifactId>

</exclusion>

</exclusions>

</dependency>

## Zones

If you have deployed Eureka clients to multiple zones than you may prefer that those clients leverage services within the same zone before trying services in another zone. To do this you need to configure your Eureka clients correctly.

First, you need to make sure you have Eureka servers deployed to each zone and that they are peers of each other. See the section on [zones and regions](https://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html#spring-cloud-eureka-server-zones-and-regions) for more information.

Next you need to tell Eureka which zone your service is in. You can do this using the metadataMap property. For example if service 1 is deployed to both zone 1and zone 2 you would need to set the following Eureka properties in service 1

**Service 1 in Zone 1**

eureka.instance.metadataMap.zone = zone1

eureka.client.preferSameZoneEureka = true

**Service 1 in Zone 2**

eureka.instance.metadataMap.zone = zone2

eureka.client.preferSameZoneEureka = true

# Eureka Server

## 2.1 How to Include Eureka Server

To include Eureka Server in your project use the starter with group org.springframework.cloud and artifact id spring-cloud-starter-netflix-eureka-server.

## How to Run a Eureka Server

Example eureka server;

*@SpringBootApplication*

*@EnableEurekaServer*

**public** **class** Application {

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(Application.**class**).web(true).run(args);

}

}

## 2.3 High Availability, Zones and Regions

The Eureka server does not have a backend store, but the service instances in the registry all have to send heartbeats to keep their registrations up to date (so this can be done in memory). Clients also have an in-memory cache of eureka registrations (so they don’t have to go to the registry for every single request to a service).

By default every Eureka server is also a Eureka client and requires (at least one) service URL to locate a peer. If you don’t provide it the service will run and work, but it will shower your logs with a lot of noise about not being able to register with the peer.

# 3. Circuit Breaker: Hystrix Clients

Netflix has created a library called [Hystrix](https://github.com/Netflix/Hystrix" \t "_top) that implements the [circuit breaker pattern](http://martinfowler.com/bliki/CircuitBreaker.html).

**Figure 3.1. Microservice Graph**

HystrixGraph

A service failure in the lower level of services can cause cascading failure all the way up to the user. When calls to a particular service is greater than circuitBreaker.requestVolumeThreshold (default: 20 requests) and failue percentage is greater than circuitBreaker.errorThresholdPercentage (default: >50%) in a rolling window defined

by metrics.rollingStats.timeInMilliseconds (default: 10 seconds), the circuit opens and the call is not made. In cases of error and an open circuit a fallback can be provided by the developer.

**Figure 3.2. Hystrix fallback prevents cascading failures**

HystrixFallback

The fallback can be another Hystrix protected call, static data or a sane empty value. Fallbacks may be chained so the first fallback makes some other business call which in turn falls back to static data.

## **How to Include Hystrix**

To include Hystrix in your project use the starter with group org.springframework.cloud and artifact id spring-cloud-starter-netflix-hystrix.

Example boot app:

@SpringBootApplication

@EnableCircuitBreaker

public class Application {

public static void main(String[] args) {

new SpringApplicationBuilder(Application.class).web(true).run(args);

}

}

@Component

public class StoreIntegration {

@HystrixCommand(fallbackMethod = "defaultStores")

public Object getStores(Map<String, Object> parameters) {

//do stuff that might fail

}

public Object defaultStores(Map<String, Object> parameters) {

return /\* something useful \*/;

}

}

The @HystrixCommand is provided by a Netflix contrib library called ["javanica"](https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica). Spring Cloud automatically wraps Spring beans with that annotation in a proxy that is connected to the Hystrix circuit breaker. The circuit breaker calculates when to open and close the circuit, and what to do in case of a failure.

## Propagating the Security Context or using Spring Scopes

If you want some thread local context to propagate into a @HystrixCommand the default declaration will not work because it executes the command in a thread pool (in case of timeouts). You can switch Hystrix to use the same thread as the caller using some configuration, or directly in the annotation, by asking it to use a different "Isolation Strategy". For example:

*@HystrixCommand(fallbackMethod = "stubMyService",*

*commandProperties = {*

*@HystrixProperty(name="execution.isolation.strategy", value="SEMAPHORE")*

*}*

*)*

...

The same thing applies if you are using @SessionScope or @RequestScope. You will know when you need to do this because of a runtime exception that says it can’t find the scoped context.

You also have the option to set the hystrix.shareSecurityContext property to true. Doing so will auto configure an Hystrix concurrency strategy plugin hook who will transfer the SecurityContext from your main thread to the one used by the Hystrix command. Hystrix does not allow multiple hystrix concurrency strategy to be registered so an extension mechanism is available by declaring your own HystrixConcurrencyStrategy as a Spring bean.

## 3.3 Health Indicator

The state of the connected circuit breakers are also exposed in the /health endpoint of the calling application.

**{**

"hystrix": **{**

"openCircuitBreakers": **[**

"StoreIntegration::getStoresByLocationLink"

]**,**

"status": "CIRCUIT\_OPEN"

**},**

"status": "UP"

**}**

## 3.4 Hystrix Metrics Stream

To enable the Hystrix metrics stream include a dependency on spring-boot-starter-actuator. This will expose the /hystrix.stream as a management endpoint.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

# 4. Circuit Breaker: Hystrix Dashboard

One of the main benefits of Hystrix is the set of metrics it gathers about each HystrixCommand. The Hystrix Dashboard displays the health of each circuit breaker in an efficient manner.

# 5. Hystrix Timeouts And Ribbon Clients

When using Hystrix commands that wrap Ribbon clients you want to make sure your Hystrix timeout is configured to be longer than the configured Ribbon timeout, including any potential retries that might be made. For example, if your Ribbon connection timeout is one second and the Ribbon client might retry the request three times, than your Hystrix timeout should be slightly more than three seconds.

## 5.1 **How to Include Hystrix Dashboard**

To include the Hystrix Dashboard in your project use the starter with group org.springframework.cloud and artifact id spring-cloud-starter-hystrix-netflix-dashboard.

To run the Hystrix Dashboard annotate your Spring Boot main class with @EnableHystrixDashboard. You then visit /hystrix and point the dashboard to an individual instances /hystrix.stream endpoint in a Hystrix client application.

**Step 1:** Add the Spring-Cloud-Starter-hystrix:

[view sourc](http://www.java-allandsundry.com/2016/01/spring-cloud-with-turbine.html#viewSource)

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

**Step 2:** Enable Hystrix support for the Application, this will expose the hystrix stream at a "/hystrix.stream" uri:

@SpringBootApplication

@EnableHystrix

public class SpringCloudApp {

public static void main(String[] args) {

SpringApplication.run(SpringCloudApp.class, args);

}

}

Now for the Hystrix Dashboard application to graphically view the Hystrix stream, the following annotation will enable that and the application should be available at "/hystrix" uri:

@SpringBootApplication

@EnableHystrixDashboard

public class AggregateApp {

public static void main(String[] args) {

SpringApplication.run(AggregateApp.class, args);

}

}

## **Spring Cloud With Turbine**

Hystrix stream provides information on a single application, Turbine provides a way to aggregate this information across all installations of an application in a cluster. Integrating turbine into a Spring-Cloud based application is straightforward, all it requires is information on which clusters to expose information on and how to aggregate information about the specific clusters. As before to pull in the dependencies of Turbine:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-turbine</artifactId>

<exclusions>

<exclusion>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

</exclusion>

</exclusions>

</dependency>

And to enable Turbine support in a Spring Boot based application:

@SpringBootApplication

@EnableHystrixDashboard

@EnableTurbine

public class MonitorApplication {

public static void main(String[] args) {

SpringApplication.run(MonitorApplication.class, args);

}

}

This application is playing the role of both showing the Hystrix Dashboard and exposing turbine stream. Finally the configuration for turbine:

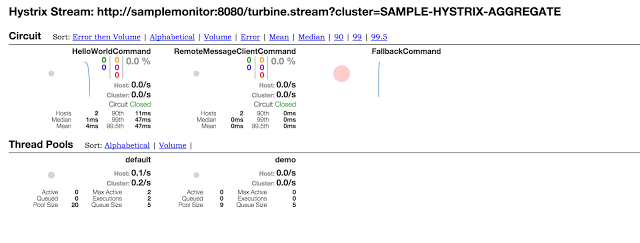
turbine:

aggregator:

clusterConfig: SAMPLE-HYSTRIX-AGGREGATE

appConfig: SAMPLE-HYSTRIX-AGGREGATE

Given this configuration a turbine stream for SAMPLE-HYSTRIX-AGGREGATE cluster is available at "/turbine.stream?cluster=SAMPLE-HYSTRIX-AGGREGATE" uri, it would figure out the instances of the cluster using Eureka, source the Hystrix stream from each instance and aggregate it into the Turbine stream. If we were to view the Hystrix dashboard against this stream:

[](http://2.bp.blogspot.com/-DtsvBUgWdgQ/Vpribug_HdI/AAAAAAAAWV4/cMKfUt0rMjY/s1600/HystrixDashboard2.png)

If you look at the counts against the host now, it indicates the two hosts for which the stream is being aggregated.

# 6. Client Side Load Balancer: Ribbon

Netflix [Ribbon](https://github.com/Netflix/ribbon) is an Inter Process Communication (IPC) cloud library. Ribbon primarily provides client-side load balancing algorithms.

Apart from the client-side load balancing algorithms, Ribbon provides also other features:

* **Service Discovery Integration** – Ribbon load balancers provide service discovery in dynamic environments like a cloud. Integration with Eureka and Netflix service discovery component is included in the ribbon library
* **Fault Tolerance** – the Ribbon API can dynamically determine whether the servers are up and running in a live environment and can detect those servers that are down
* **Configurable load-balancing rules** – Ribbon supports *RoundRobinRule*, *AvailabilityFilteringRule*, *WeightedResponseTimeRule* out of the box and also supports defining custom rules

### **Client-Side Load Balancing**

An approach to load balancing is to deliver a list of server IPs to the client, and then to have client randomly select the IP from the list on each connection. It has been claimed that client-side random load balancing tends to provide better load distribution than **round-robin DNS.** With this approach, the method of delivery of lists of IPs to the client can vary, and may be implemented as a DNS list (delivered to all the clients without any round-robin), or via hardcoding it to the list. If a "smart client" is used, detecting that randomly selected server is down and connecting randomly again, it also provides fault tolerance.

Ribbon provides the following features:

* Load balancing
* Fault tolerance
* Multiple protocols (HTTP, TCP, UDP) support in an asynchronous and reactive model
* Caching and batching
* **Using Ribbon - Code:**
* First Service is **hello-service**
* This service provides some aleatory greeting when accessing <http://localhost:8090/greeting>
* @RestController
* @SpringBootApplication
* public class HelloApplication {
* private static Logger log = LoggerFactory.getLogger(HelloApplication.class);
* @RequestMapping(value = "/greeting")
* public String greet() {
* log.info("Access /greeting");
* List<String> greetings = Arrays.asList("Hi there", "Greetings", "Salutations");
* Random rand = new Random();
* int randomNum = rand.nextInt(greetings.size());
* return greetings.get(randomNum);
* }
* @RequestMapping(value = "/")
* public String home() {
* log.info("Access /");
* return "Hi!";
* }
* public static void main(String[] args) {
* SpringApplication.run(HelloApplication.class, args);
* }
* }
* **File: application.yml**
* spring:
* application:
* name: hello-service
* server:
* port: 8090
* OBS: should create two more hello-service in Port **9092**, **9999**
* Second Service is **user-service**
* This service call for **hello-service,**operation **greeting,** with ribbon configured, this service call hello-service based on algorithmround-robin.
* @SpringBootApplication
* @RestController
* @RibbonClient(name = "hello-service", configuration = HelloServiceConfiguration.class)
* public class UserApplication {
* @LoadBalanced
* @Bean
* RestTemplate restTemplate() {
* return new RestTemplate();
* }
* @Autowired
* RestTemplate restTemplate;
* @RequestMapping("/hi")
* public String hi(@RequestParam(value = "name", defaultValue = "Rafael") String name) {
* String greeting = this.restTemplate.getForObject("http://hello-service/greeting", String.class);
* return String.format("%s, %s!", greeting, name);
* }
* public static void main(String[] args) {
* SpringApplication.run(UserApplication.class, args);
* }
* }
* public class HelloServiceConfiguration {
* @Autowired
* IClientConfig ribbonClientConfig;
* @Bean
* public IPing ribbonPing(IClientConfig config) {
* return new PingUrl();
* }
* @Bean
* public IRule ribbonRule(IClientConfig config) {
* return new AvailabilityFilteringRule();
* }
* }
* **File: application.yml**
* In Ribbon, configuration lists all services available.
* spring:
* application:
* name: user-service
* server:
* port: 8888
* hello-service:
* ribbon:
* eureka:
* enabled: false
* listOfServers: localhost:8090,localhost:9092,localhost:9999
* ServerListRefreshInterval: 15000
* When we call the service on the URL <http://localhost:8090/greeting> on console, it is possible to see each service it is called, for example, in the fist call any host configured on the list of servers that could be used.

## Customizing default for all Ribbon Clients

A default configuration can be provided for all Ribbon Clients using the @RibbonClients annotation and registering a default configuration as shown in the following example:

*@RibbonClients(defaultConfiguration = DefaultRibbonConfig.class)*

**public** **class** RibbonClientDefaultConfigurationTestsConfig {

**public** **static** **class** BazServiceList **extends** ConfigurationBasedServerList {

**public** BazServiceList(IClientConfig config) {

**super**.initWithNiwsConfig(config);

}

}

}

*@Configuration*

**class** DefaultRibbonConfig {

*@Bean*

**public** IRule ribbonRule() {

**return** **new** BestAvailableRule();

}

*@Bean*

**public** IPing ribbonPing() {

**return** **new** PingUrl();

}

*@Bean*

**public** ServerList<Server> ribbonServerList(IClientConfig config) {

**return** **new** RibbonClientDefaultConfigurationTestsConfig.BazServiceList(config);

}

*@Bean*

**public** ServerListSubsetFilter serverListFilter() {

ServerListSubsetFilter filter = **new** ServerListSubsetFilter();

**return** filter;

}

}

## 6.4 Customizing the Ribbon Client using properties

Starting with version 1.2.0, Spring Cloud Netflix now supports customizing Ribbon clients using properties to be compatible with the [Ribbon documentation](https://github.com/Netflix/ribbon/wiki/Working-with-load-balancers#components-of-load-balancer).

This allows you to change behavior at start up time in different environments.

The supported properties are listed below and should be prefixed by <clientName>.ribbon.:

* NFLoadBalancerClassName: should implement ILoadBalancer
* NFLoadBalancerRuleClassName: should implement IRule
* NFLoadBalancerPingClassName: should implement IPing
* NIWSServerListClassName: should implement ServerList
* NIWSServerListFilterClassName should implement ServerListFilter

|  |
| --- |
| [Note] |
| Classes defined in these properties have precedence over beans defined using @RibbonClient(configuration=MyRibbonConfig.class) and the defaults provided by Spring Cloud Netflix. |

To set the IRule for a service name users you could set the following:

**application.yml.**

users:

ribbon:

NIWSServerListClassName: com.netflix.loadbalancer.ConfigurationBasedServerList

NFLoadBalancerRuleClassName: com.netflix.loadbalancer.WeightedResponseTimeRule