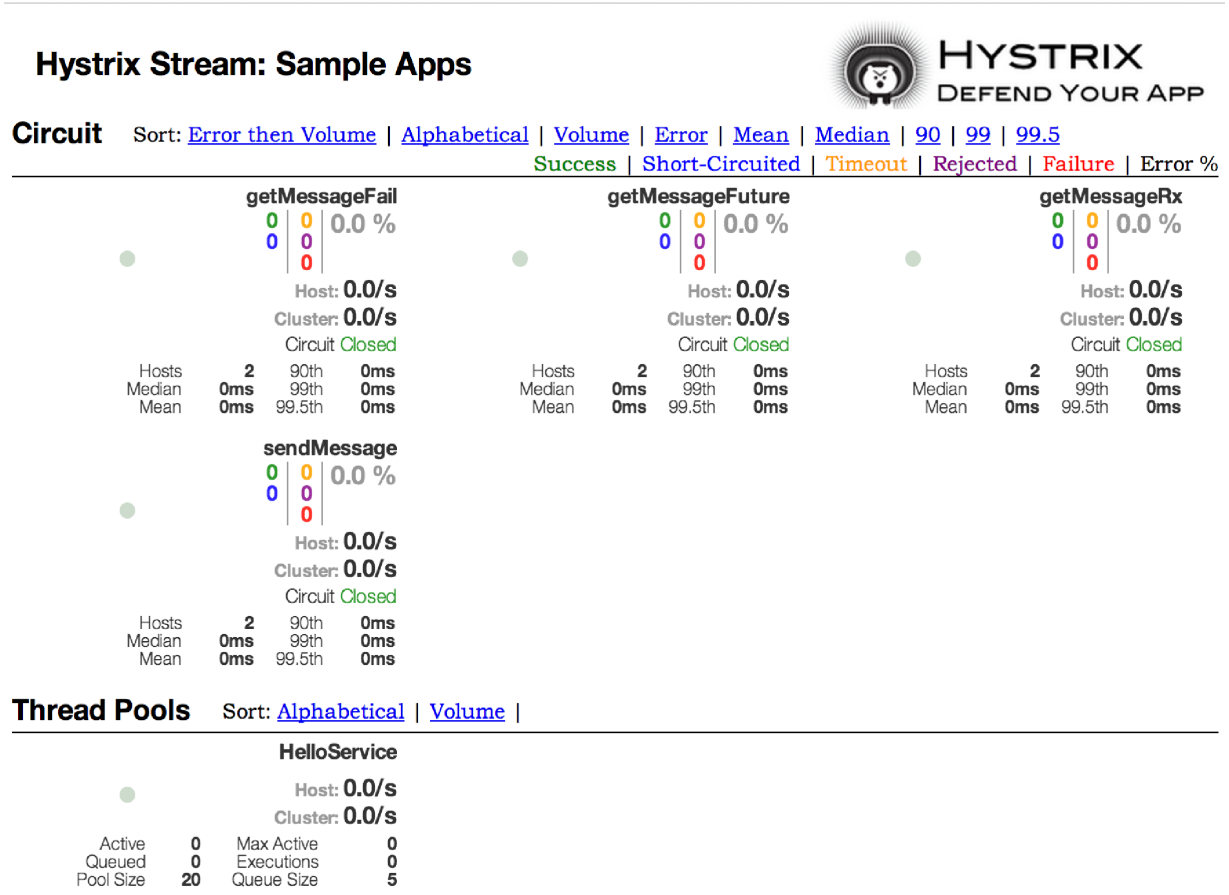
**Circuit Breaker: Hystrix Clients**

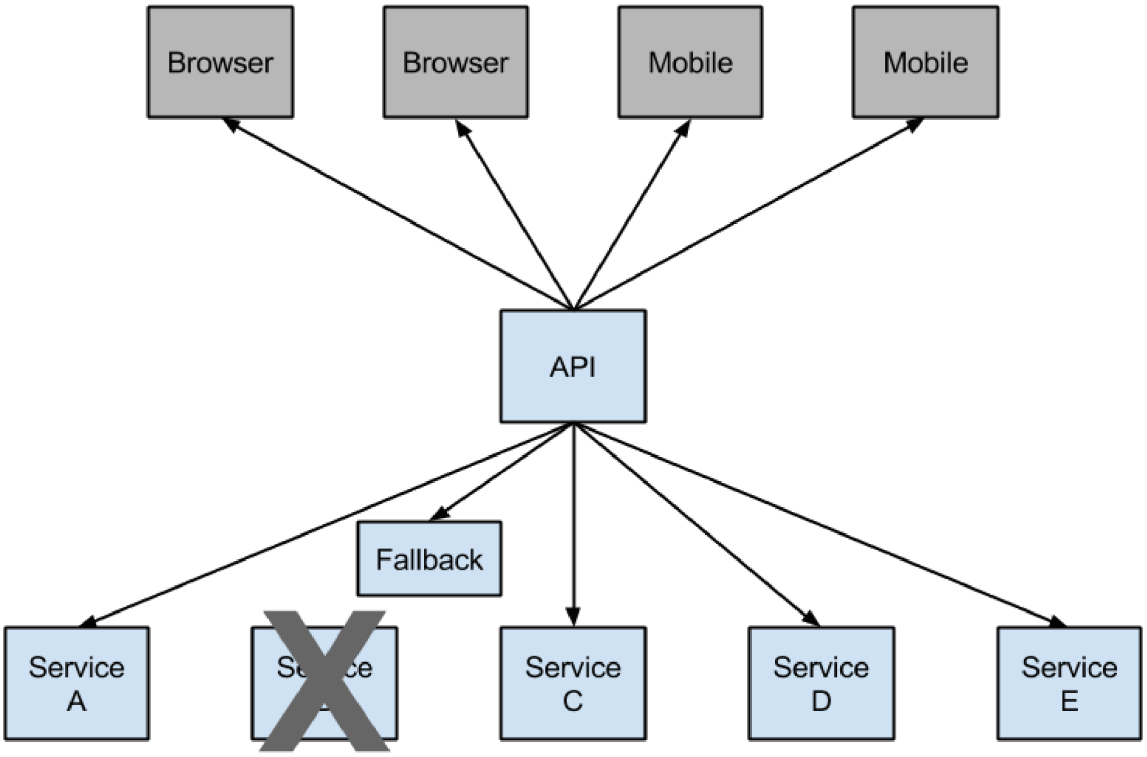
Netflix has created a library called Hystrix that implements the circuit breaker pattern. In a microservice architecture, it is common to have multiple layers of service calls, as shown in the following example:




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 exceed **circuitBreaker.requestVolumeThreshold (default: 20 requests)** and the failure 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.

Having an open circuit stops cascading failures and allows overwhelmed or failing services time to recover. The fallback can be another Hystrix protected call, static data, or a sensible empty value. Fallbacks may be chained so that the first fallback makes some other business call, which in turn falls back to static data.

Figure 3.2. Hystrix fallback prevents cascading failures




**How to Include Hystrix**
To include Hystrix in your project, use the starter with a group ID of **org.springframework.cloud** and a artifact ID of **spring-cloud-starter-netflix-hystrix**.

The following example shows a minimal Eureka server with a Hystrix circuit breaker:

```
@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"**. 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.

To configure the **@HystrixCommand** you can use the **commandProperties** attribute with a list of **@HystrixProperty** annotations. See here **(https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica#configuration)** for more details. See the Hystrix wiki **(https://github.com/Netflix/Hystrix/wiki/Configuration)** for details on the properties available.

**Propagating the Security Context or Using Spring Scopes**

If you want some thread local context to propagate into a @HystrixCommand, the default declaration does 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 through configuration or directly in the annotation, by asking it to use a different "Isolation Strategy". The following example demonstrates setting the thread in the annotation:

```
@HystrixCommand(fallbackMethod = "stubMyService",
    commandProperties = {
      @HystrixProperty(name="execution.isolation.strategy", value="SEMAPHORE")
    }
)
...
```

The same thing applies if you are using **@SessionScope** or **@RequestScope**. If you encounter a runtime exception that says it cannot find the scoped context, you need to use the same thread.

You also have the option to set the **hystrix.shareSecurityContext** property to true. Doing so auto-configures a Hystrix concurrency strategy plugin hook to transfer the SecurityContext from your main thread to the one used by the Hystrix command. Hystrix does not let multiple Hystrix concurrency strategy be registered so an extension mechanism is available by declaring your own HystrixConcurrencyStrategy as a Spring bean. Spring Cloud looks for your implementation within the Spring context and wrap it inside its own plugin.

**Health Indicator**

The state of the connected circuit breakers are also exposed in the /health endpoint of the calling application, as shown in the following example:

```
{
    "hystrix": {
        "openCircuitBreakers": [
            "StoreIntegration::getStoresByLocationLink"
        ],
```

```
    "status": "CIRCUIT_OPEN"
  },
  "status": "UP"
}
```

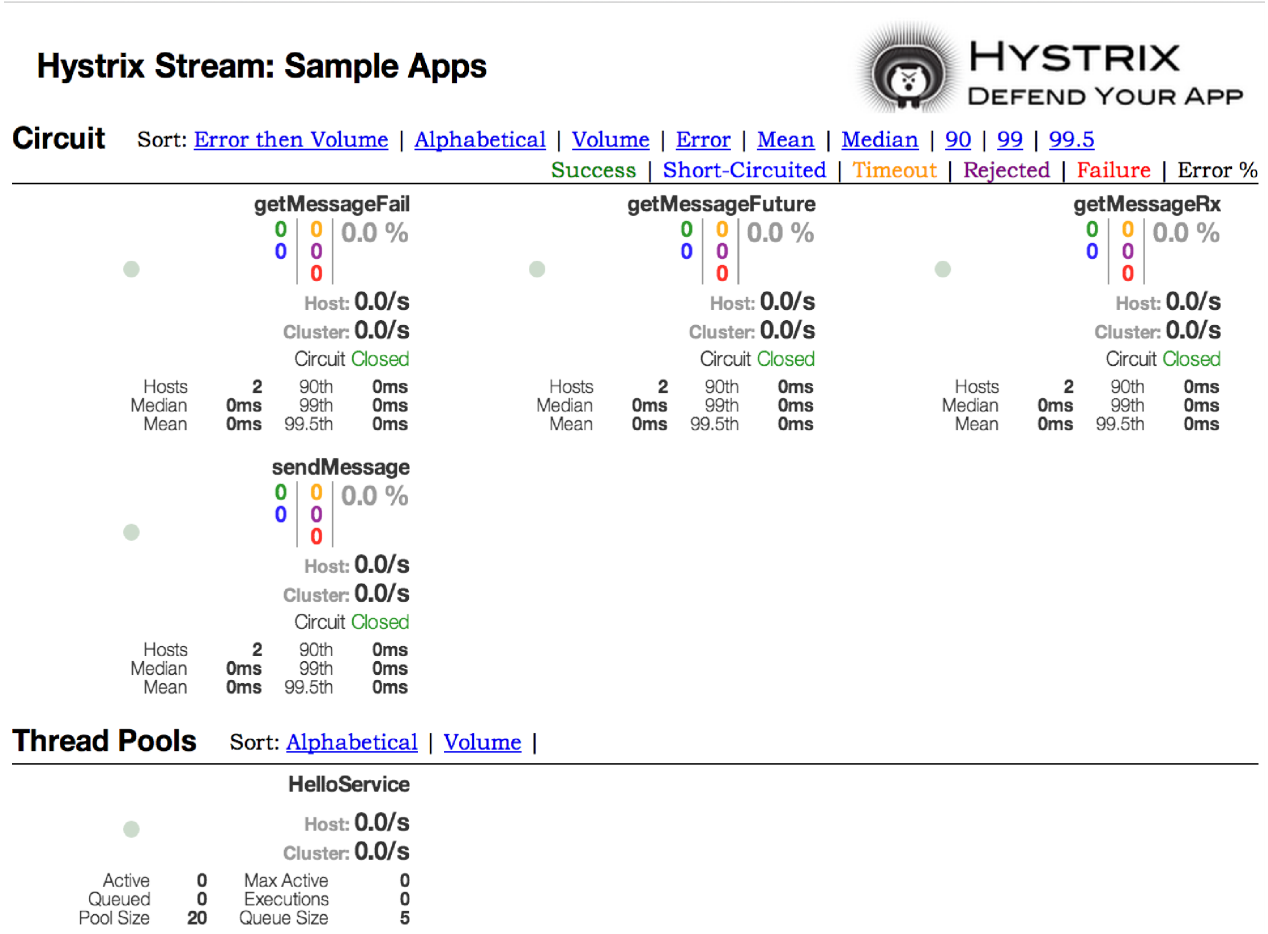
**Hystrix Metrics Stream**

To enable the Hystrix metrics stream, include a dependency on spring-boot-starter-actuator and set management.endpoints.web.exposure.include: hystrix.stream. Doing so exposes the /actuator/hystrix.stream as a management endpoint, as shown in the following example:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

**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.



**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.

**How to Include the Hystrix Dashboard**
To include the Hystrix Dashboard in your project, use the starter with a group ID of org.springframework.cloud and an artifact ID of spring-cloud-starter-netflix-hystrix-dashboard

To run the Hystrix Dashboard, annotate your Spring Boot main class with @EnableHystrixDashboard. Then visit /hystrix and point the dashboard to an individual instance's /hystrix.stream endpoint in a Hystrix client application.

**[Note]**
When connecting to a /hystrix.stream endpoint that uses HTTPS, the certificate used by the server must be trusted by the JVM. If the certificate is not trusted, you must import the certificate into the JVM in order for the Hystrix Dashboard to make a successful connection to the stream endpoint.

**Turbine**
Looking at an individual instance's Hystrix data is not very useful in terms of the overall health of the system. Turbine is an application that aggregates all of the relevant /hystrix.stream endpoints into a combined /turbine.stream for use in the Hystrix Dashboard. Individual instances are located through Eureka. Running Turbine requires annotating your main class with the @EnableTurbine annotation (for example, by using spring-cloud-starter-netflix-turbine to set up the classpath). All of the documented configuration properties from the Turbine 1 wiki apply. The only difference is that the turbine.instanceUrlSuffix does not need the port prepended, as this is handled automatically unless turbine.instanceInsertPort=false.

**[Note]**
By default, Turbine looks for the /hystrix.stream endpoint on a registered instance by looking up its hostName and port entries in Eureka and then appending /hystrix.stream to it. If the instance's metadata contains management.port, it is used instead of the port value for the /hystrix.stream endpoint. By default, the metadata entry called management.port is equal to the management.port configuration property. It can be overridden though with following configuration:

```
eureka:
  instance:
    metadata-map:
      management.port: ${management.port:8081}
```

The turbine.appConfig configuration key is a list of Eureka serviceIds that turbine uses to lookup instances. The turbine stream is then used in the Hystrix dashboard with a URL similar to the following:

https://my.turbine.server:8080/turbine.stream?cluster=CLUSTERNAME

The cluster parameter can be omitted if the name is default. The cluster parameter must match an entry in turbine.aggregator.clusterConfig. Values returned from Eureka are upper-case. Consequently, the following example works if there is an application called customers registered with Eureka:

```
turbine:
  aggregator:
    clusterConfig: CUSTOMERS
  appConfig: customers
```

If you need to customize which cluster names should be used by Turbine (because you do not want to store cluster names in turbine.aggregator.clusterConfig configuration), provide a bean of type Turbine-ClustersProvider.

The clusterName can be customized by a SPEL expression in turbine.clusterNameExpression with root as an instance of InstanceInfo. The default value is appName, which means that the Eureka serviceId becomes the cluster key (that is, the InstanceInfo for customers has an appName of CUSTOMERS). A different example is turbine.clusterNameExpression=aSGName, which gets the cluster name from the AWS ASG name. The following listing shows another example:

```
turbine:
  aggregator:
    clusterConfig: SYSTEM,USER
  appConfig: customers,stores,ui,admin
  clusterNameExpression: metadata['cluster']
```

In the preceding example, the cluster name from four services is pulled from their metadata map and is expected to have values that include SYSTEM and USER.

To use the “default” cluster for all apps, you need a string literal expression (with single quotes and escaped with double quotes if it is in YAML as well):

```
turbine:
  appConfig: customers,stores
  clusterNameExpression: "'default'"
```

Spring Cloud provides a spring-cloud-starter-netflix-turbine that has all the dependencies you need to get a Turbine server running. To add Turbine, create a Spring Boot application and annotate it with @EnableTurbine.

**[Note]**
By default, Spring Cloud lets Turbine use the host and port to allow multiple processes per host, per cluster. If you want the native Netflix behavior built into Turbine to not allow multiple processes per host, per cluster (the key to the instance ID is the hostname), set turbine.combineHostPort=false.

**Clusters Endpoint**
In some situations it might be useful for other applications to know what custers have been configured in Turbine. To support this you can use the /clusters endpoint which will return a JSON array of all the configured clusters.

GET /clusters.

```
[
```

```
  {
    "name": "RACES",
    "link": "http://localhost:8383/turbine.stream?cluster=RACES"
  },
  {
    "name": "WEB",
    "link": "http://localhost:8383/turbine.stream?cluster=WEB"
  }
]
```

This endpoint can be disabled by setting turbine.endpoints.clusters.enabled to false.

**Turbine Stream**
In some environments (such as in a PaaS setting), the classic Turbine model of pulling metrics from all the distributed Hystrix commands does not work. In that case, you might want to have your Hystrix commands push metrics to Turbine. Spring Cloud enables that with messaging. To do so on the client, add a dependency to spring-cloud-netflix-hystrix-stream and the spring-cloud-starter-stream-* of your choice. See the Spring Cloud Stream documentation for details on the brokers and how to configure the client credentials. It should work out of the box for a local broker.

On the server side, create a Spring Boot application and annotate it with @EnableTurbineStream. The Turbine Stream server requires the use of Spring Webflux, therefore spring-boot-starter-webflux needs to be included in your project. By default spring-boot-starter-webflux is included when adding spring-cloud-starter-netflix-turbine-stream to your application.

You can then point the Hystrix Dashboard to the Turbine Stream Server instead of individual Hystrix streams. If Turbine Stream is running on port 8989 on myhost, then put http://myhost:8989 in the stream input field in the Hystrix Dashboard. Circuits are prefixed by their respective serviceId, followed by a dot (.), and then the circuit name.

Spring Cloud provides a spring-cloud-starter-netflix-turbine-stream that has all the dependencies you need to get a Turbine Stream server running. You can then add the Stream binder of your choice — such as spring-cloud-starter-stream-rabbit.

Turbine Stream server also supports the cluster parameter. Unlike Turbine server, Turbine Stream uses eureka serviceIds as cluster names and these are not configurable.

If Turbine Stream server is running on port 8989 on my.turbine.server and you have two eureka serviceIds customers and products in your environment, the following URLs will be available on your Turbine Stream server. default and empty cluster name will provide all metrics that Turbine Stream server receives.

https://my.turbine.sever:8989/turbine.stream?cluster=customers
https://my.turbine.sever:8989/turbine.stream?cluster=products
https://my.turbine.sever:8989/turbine.stream?cluster=default
https://my.turbine.sever:8989/turbine.stream

So, you can use eureka serviceIds as cluster names for your Turbine dashboard (or any compatible dashboard). You don’t need to configure any properties like turbine.appConfig, turbine.clusterNameEx-

pression and turbine.aggregator.clusterConfig for your Turbine Stream server.

**[Note]**
Turbine Stream server gathers all metrics from the configured input channel with Spring Cloud Stream. It means that it doesn't gather Hystrix metrics actively from each instance. It just can provide metrics that were already gathered into the input channel by each instance.

**hystrix-javanica**
Java language has a great advantages over other languages such as reflection and annotations. All modern frameworks such as Spring, Hibernate, myBatis and etc. seek to use this advantages to the maximum. The idea of introduction annotations in Hystrix is obvious solution for improvement. Currently using Hystrix involves writing a lot of code that is a barrier to rapid development. You likely be spending a lot of time on writing a Hystrix commands. Idea of the Javanica project is make easier using of Hystrix by the introduction of support annotations.

First of all in order to use hystrix-javanica you need to add hystrix-javanica dependency in your project.

Example for Maven:
```
<dependency>
   <groupId>com.netflix.hystrix</groupId>
   <artifactId>hystrix-javanica</artifactId>
   <version>x.y.z</version>
</dependency>
```

To implement AOP functionality in the project was used AspectJ library. If in your project already used AspectJ then you need to add hystrix aspect in aop.xml as below:
```
<aspects>
      ...
      <aspect name="com.netflix.hystrix.contrib.javanica.aop.aspectj.HystrixCommandAspect"/>
      ...
</aspects>
```

More about AspectJ configuration read [here] (http://www.eclipse.org/aspectj/doc/next/devguide/ltw-configuration.html)
If you use Spring AOP in your project then you need to add specific configuration using Spring AOP namespace in order to make Spring capable to manage aspects which were written using AspectJ and declare HystrixCommandAspect as Spring bean like below:

```
   <aop:aspectj-autoproxy/>
     <bean id="hystrixAspect" class="com.netflix.hystrix.contrib.javanica.aop.aspectj.HystrixCommandAspect"></bean>
```

Or if you are using Spring code configuration:

```
@Configuration
public class HystrixConfiguration {

 @Bean
 public HystrixCommandAspect hystrixAspect() {
  return new HystrixCommandAspect();
 }

}
```

It doesn't matter which approach you use to create proxies in Spring, javanica works fine with JDK and

CGLIB proxies. If you use another framework for aop which supports AspectJ and uses other libs (Javassist for instance) to create proxies then let us know what lib you use to create proxies and we'll try to add support for this library in near future.

More about Spring AOP + AspectJ read [here] (http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html)

**Aspect weaving**
Javanica supports two weaving modes: compile and runtime. Load time weaving hasn't been tested but it should work. If you tried LTW mode and got any problems then raise javanica issue or create pull request with fix.

- CTW. To use CTW mode you need to use specific jar version: hystrix-javanica-ctw-X.Y.Z . This jar is assembled with aspects compiled with using AJC compiler. If you will try to use regular hystrix-javanica-X.Y.Z with CTW then you get NoSuchMethodError aspectOf() at runtime from building with iajc. Also, you need to start your app with using java property: -DWeavingMode=compile. NOTE: Javanica depends on aspectj library and uses internal features of aspectj and these features aren't provided as a part of open API thus it can change from version to version. Javanica tested with latest aspectj version 1.8.7. If you updated aspectj version and noticed any issues then please don't hestitate to create new issue or contribute.
- RTW works, you can use regular hystrix-javanica-X.Y.Z
- LTM hasn't been tested but it should work fine.

## Hystrix command

**Synchronous Execution**
To run method as Hystrix command synchronously you need to annotate method with @HystrixCommand annotation, for example

```
public class UserService {
...
   @HystrixCommand
   public User getUserById(String id) {
      return userResource.getUserById(id);
   }
}
...
```

In example above the getUserById method will be processed synchronously within new Hystrix command. By default the name of command key is command method name: **getUserById**, default group key name is class name of annotated method: **UserService**. You can change it using necessary **@HystrixCommand** properties:

```
@HystrixCommand(groupKey="UserGroup", commandKey = "GetUserByIdCommand")
public User getUserById(String id) {
   return userResource.getUserById(id);
}
```

To set threadPoolKey use @HystrixCommand#threadPoolKey()

**Asynchronous Execution**
To process Hystrix command asynchronously you should return an instance of AsyncResult in your command method as in the example below:

```
@HystrixCommand
public Future<User> getUserByIdAsync(final String id) {
   return new AsyncResult<User>() {
      @Override
      public User invoke() {
         return userResource.getUserById(id);
      }
   };
}
```

The return type of command method should be Future that indicates that a command should be executed [asynchronously] **(https://github.com/Netflix/Hystrix/wiki/How-To-Use#wiki-Asynchronous-Execution).**

**Reactive Execution**
To perform "Reactive Execution" you should return an instance of Observable in your command method as in the example below:

```
@HystrixCommand
public Observable<User> getUserById(final String id) {
   return Observable.create(new Observable.OnSubscribe<User>() {
         @Override
         public void call(Subscriber<? super User> observer) {
            try {
               if (!observer.isUnsubscribed()) {
                  observer.onNext(new User(id, name + id));
                  observer.onCompleted();
               }
            } catch (Exception e) {
               observer.onError(e);
            }
         }
      });
}
```

In addition to Observable Javanica supports the following RX types: Single and Completable. Hystrix core supports only one RX type which is Observable, HystrixObservableCommand requires to return Observable therefore javanica transforms Single or Completable to Observable using toObservable() method for appropriate type and before returning the result to caller it translates Observable to either Single or Completable using toSingle() or toCompletable() correspondingly.

HystrixObservable interface provides two methods: observe() - eagerly starts execution of the command the same as HystrixCommand#queue() and HystrixCommand#execute(); toObservable() - lazily starts execution of the command only once the Observable is subscribed to. To control this behaviour and swith between two modes @HystrixCommand provides specific parameter called observableExe-

cutionMode. @HystrixCommand(observableExecutionMode = EAGER) indicates that observe() method should be used to execute observable command @HystrixCommand(observableExecutionMode = LAZY) indicates that toObservable() should be used to execute observable command

**NOTE: EAGER mode is used by default**

**Fallback**
Graceful degradation can be achieved by declaring name of fallback method in @HystrixCommand like below:

```
@HystrixCommand(fallbackMethod = "defaultUser")
public User getUserById(String id) {
    return userResource.getUserById(id);
}

private User defaultUser(String id) {
    return new User("def", "def");
}
```

Its important to remember that Hystrix command and fallback should be placed in the same class and have same method signature (optional parameter for failed execution exception).

Fallback method can have any access modifier. Method defaultUser will be used to process fallback logic in a case of any errors. If you need to run fallback method defaultUser as separate Hystrix command then you need to annotate it with HystrixCommand annotation as below:

```
@HystrixCommand(fallbackMethod = "defaultUser")
public User getUserById(String id) {
    return userResource.getUserById(id);
}

@HystrixCommand
private User defaultUser(String id) {
    return new User();
}
```

If fallback method was marked with @HystrixCommand then this fallback method (defaultUser) also can has own fallback method, as in the example below:

```
@HystrixCommand(fallbackMethod = "defaultUser")
public User getUserById(String id) {
    return userResource.getUserById(id);
}

@HystrixCommand(fallbackMethod = "defaultUserSecond")
private User defaultUser(String id) {
    return new User();
}
```

```
@HystrixCommand
private User defaultUserSecond(String id) {
   return new User("def", "def");
}
```

Javanica provides an ability to get execution exception (exception thrown that caused the failure of a command) within a fallback is being executed. A fallback method signature can be extended with an additional parameter in order to get an exception thrown by a command. Javanica exposes execution exception through additional parameter of fallback method. Execution exception is derived by calling method getExecutionException() as in vanilla hystrix.

**Example**:

```
@HystrixCommand(fallbackMethod = "fallback1")
User getUserById(String id) {
   throw new RuntimeException("getUserById command failed");
}

@HystrixCommand(fallbackMethod = "fallback2")
User fallback1(String id, Throwable e) {
   assert "getUserById command failed".equals(e.getMessage());
   throw new RuntimeException("fallback1 failed");
}

@HystrixCommand(fallbackMethod = "fallback3")
User fallback2(String id) {
   throw new RuntimeException("fallback2 failed");
}

@HystrixCommand(fallbackMethod = "staticFallback")
User fallback3(String id, Throwable e) {
   assert "fallback2 failed".equals(e.getMessage());
   throw new RuntimeException("fallback3 failed");
}

User staticFallback(String id, Throwable e) {
   assert "fallback3 failed".equals(e.getMessage());
   return new User("def", "def");
}

// test
@Test
public void test() {
assertEquals("def", getUserById("1").getName());
}
```

As you can see, the additional Throwable parameter is not mandatory and can be omitted or specified. A fallback gets an exception thrown that caused a failure of parent, thus the fallback3 gets exception thrown by fallback2, no by getUserById command.

**Async/Sync fallback.**
A fallback can be async or sync, at certain cases it depends on command execution type, below listed all possible uses :

**Supported**
**case 1: sync command, sync fallback**

```
@HystrixCommand(fallbackMethod = "fallback")
User getUserById(String id) {
   throw new RuntimeException("getUserById command failed");
}

@HystrixCommand
User fallback(String id) {
   return new User("def", "def");
}
```

**case 2: async command, sync fallback**

```
@HystrixCommand(fallbackMethod = "fallback")
Future<User> getUserById(String id) {
   throw new RuntimeException("getUserById command failed");
}

@HystrixCommand
User fallback(String id) {
   return new User("def", "def");
}
```

**case 3: async command, async fallback**

```
@HystrixCommand(fallbackMethod = "fallbackAsync")
Future<User> getUserById(String id) {
   throw new RuntimeException("getUserById command failed");
}

@HystrixCommand
Future<User> fallbackAsync(String id) {
   return new AsyncResult<User>() {
      @Override
      public User invoke() {
         return new User("def", "def");
      }
   };
}
```

**Unsupported(prohibited)**
case 1: sync command, async fallback command. This case isn't supported because in the essence a caller does not get a future buy calling getUserById and future is provided by fallback isn't available

for a caller anyway, thus execution of a command forces to complete fallbackAsync before a caller gets a result, having said it turns out there is no benefits of async fallback execution. But it can be convenient if a fallback is used for both sync and async commands, if you see this case is very helpful and will be nice to have then create issue to add support for this case.

```
@HystrixCommand(fallbackMethod = "fallbackAsync")
User getUserById(String id) {
  throw new RuntimeException("getUserById command failed");
}

@HystrixCommand
Future<User> fallbackAsync(String id) {
  return new AsyncResult<User>() {
    @Override
    public User invoke() {
      return new User("def", "def");
    }
  };
}
```

case 2: sync command, async fallback. This case isn't supported for the same reason as for the case 1.

```
@HystrixCommand(fallbackMethod = "fallbackAsync")
User getUserById(String id) {
  throw new RuntimeException("getUserById command failed");
}

Future<User> fallbackAsync(String id) {
  return new AsyncResult<User>() {
    @Override
    public User invoke() {
      return new User("def", "def");
    }
  };
}
```

Same restrictions are imposed on using observable feature in javanica.

**Default fallback for class or concrete command**

This feature allows to define default fallback for the whole class or concrete command. If you have a batch of commands with exactly the same fallback logic you still have to define a fallback method for every command because fallback method should have exactly the same signature as command does, consider the following code:

```
public class Service {
  @RequestMapping(value = "/test1")
  @HystrixCommand(fallbackMethod = "fallback")
  public APIResponse test1(String param1) {
    // some codes here
```

```
        return APIResponse.success("success");
    }

    @RequestMapping(value = "/test2")
    @HystrixCommand(fallbackMethod = "fallback")
    public APIResponse test2() {
        // some codes here
        return APIResponse.success("success");
    }

    @RequestMapping(value = "/test3")
    @HystrixCommand(fallbackMethod = "fallback")
    public APIResponse test3(ObjectRequest obj) {
        // some codes here
        return APIResponse.success("success");
    }

    private APIResponse fallback(String param1) {
        return APIResponse.failed("Server is busy");
    }

    private APIResponse fallback() {
        return APIResponse.failed("Server is busy");
    }

    private APIResponse fallback(ObjectRequest obj) {
        return APIResponse.failed("Server is busy");
    }
}
```

**Default fallback feature allows to engage DRY principle and get rid of redundancy:**

```
@DefaultProperties(defaultFallback = "fallback")
public class Service {
    @RequestMapping(value = "/test1")
    @HystrixCommand
    public APIResponse test1(String param1) {
        // some codes here
        return APIResponse.success("success");
    }

    @RequestMapping(value = "/test2")
    @HystrixCommand
    public APIResponse test2() {
        // some codes here
        return APIResponse.success("success");
    }

    @RequestMapping(value = "/test3")
```

```
    @HystrixCommand
    public APIResponse test3(ObjectRequest obj) {
        // some codes here
        return APIResponse.success("success");
    }

    private APIResponse fallback() {
        return APIResponse.failed("Server is busy");
    }
}
```

**Default fallback method should not have any parameters except extra one to get execution exception and shouldn't throw any exceptions.** Below fallbacks listed in descending order of priority:

- command **fallback** defined using **fallbackMethod** property of **@HystrixCommand**
- command **default fallback** defined using **defaultFallback** property of **@HystrixCommand**
- class **default fallback** defined using **defaultFallback** property of **@DefaultProperties**

**Error Propagation**

Based on this description, @HystrixCommand has an ability to specify exceptions types which should be ignored.

```
@HystrixCommand(ignoreExceptions = {BadRequestException.class})
public User getUserById(String id) {
    return userResource.getUserById(id);
}
```

If **userResource.getUserById(id);** throws an exception that type is **BadRequestException** then this exception will be wrapped in **HystrixBadRequestException** and re-thrown without triggering fallback logic. You don't need to do it manually, javanica will do it for you under the hood.

It is worth noting that by default a caller will always get the root cause exception e.g. **BadRequestException**, never **HystrixBadRequestException** or **HystrixRuntimeException** (except the case when executed code explicitly throws those exceptions).

Optionally this exception un-wrapping can be disabled for **HystrixRuntimeException** by using **raiseHystrixExceptions** i.e. all exceptions that are not ignored are raised as the cause of a **HystrixRuntimeException**:

```
@HystrixCommand(
    ignoreExceptions = {BadRequestException.class},
    raiseHystrixExceptions = {HystrixException.RUNTIME_EXCEPTION})
public User getUserById(String id) {
    return userResource.getUserById(id);
}
```

Note: If command has a fallback then only first exception that triggers fallback logic will be propagated to caller. Example:

```
class Service {
    @HystrixCommand(fallbackMethod = "fallback")
    Object command(Object o) throws CommandException {
        throw new CommandException();
    }

    @HystrixCommand
    Object fallback(Object o) throws FallbackException {
        throw new FallbackException();
    }
}

// in client code
{
    try {
        service.command(null);
    } catch (Exception e) {
      assert CommandException.class.equals(e.getClass())
    }
}
```

# Configuration

**Command Properties**

Command properties can be set using **@HystrixCommand's 'commandProperties'** like below:

```
@HystrixCommand(commandProperties = {
@HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "500")})
   public User getUserById(String id) {
      return userResource.getUserById(id);
   }
```

Javanica dynamically sets properties using Hystrix ConfigurationManager. For the example above Javanica behind the scenes performs next action:

```
ConfigurationManager.getConfigInstance().setProperty("hystrix.command.getUserById.execution.isolation.thread.timeoutInMilliseconds", "500");
```

More about Hystrix command(**https://github.com/Netflix/Hystrix/wiki/Configuration#CommandExecution**) properties command and fallback(**https://github.com/Netflix/Hystrix/wiki/Configuration#CommandFallback**)

ThreadPoolProperties can be set using @HystrixCommand's 'threadPoolProperties' like below:

```
@HystrixCommand(commandProperties = {
        @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "500")},
        threadPoolProperties = {
                @HystrixProperty(name = "coreSize", value = "30"),
                @HystrixProperty(name = "maxQueueSize", value = "101"),
                @HystrixProperty(name = "keepAliveTimeMinutes", value = "2"),
                @HystrixProperty(name = "queueSizeRejectionThreshold", value = "15"),
                @HystrixProperty(name = "metrics.rollingStats.numBuckets", value = "12"),
                @HystrixProperty(name = "metrics.rollingStats.timeInMilliseconds", value = "1440")
})
public User getUserById(String id) {
    return userResource.getUserById(id);
}
```

**DefaultProperties**

**@DefaultProperties** is class (type) level annotation that allows to default commands properties such as groupKey, threadPoolKey, commandProperties, threadPoolProperties, ignoreExceptions and raiseHystrixExceptions. Properties specified using this annotation will be used by default for each hystrix command defined within annotated class unless a command specifies those properties explicitly using corresponding **@HystrixCommand** parameters. Example:

```
@DefaultProperties(groupKey = "DefaultGroupKey")
class Service {

        @HystrixCommand // hystrix command group key is 'DefaultGroupKey'
        public Object commandInheritsDefaultProperties() {
                return null;
        }
```

```
        @HystrixCommand(groupKey = "SpecificGroupKey") // command overrides default group
        key
        public Object commandOverridesGroupKey() {
                return null;
        }
}
```

**Hystrix collapser**

Suppose you have some command which calls should be collapsed in one backend call. For this goal you can use **@HystrixCollapser** annotation.

Example:

```
   /** Asynchronous Execution */
   @HystrixCollapser(batchMethod = "getUserByIds")
   public Future<User> getUserByIdAsync(String id) {
      return null;
   }

   /** Reactive Execution */
   @HystrixCollapser(batchMethod = "getUserByIds")
   public Observable<User> getUserByIdReact(String id) {
      return null;
   }

   @HystrixCommand
   public List<User> getUserByIds(List<String> ids) {
      List<User> users = new ArrayList<User>();
      for (String id : ids) {
         users.add(new User(id, "name: " + id));
      }
      return users;
   }

   // Async
   Future<User> f1 = userService.getUserByIdAsync("1");
   Future<User> f2 = userService.getUserByIdAsync("2");
   Future<User> f3 = userService.getUserByIdAsync("3");
   Future<User> f4 = userService.getUserByIdAsync("4");
   Future<User> f5 = userService.getUserByIdAsync("5");

   // Reactive
   Observable<User> u1 = getUserByIdReact("1");
   Observable<User> u2 = getUserByIdReact("2");
   Observable<User> u3 = getUserByIdReact("3");
   Observable<User> u4 = getUserByIdReact("4");
   Observable<User> u5 = getUserByIdReact("5");
```

```
// Materialize reactive commands
Iterable<User> users = Observables.merge(u1, u2, u3, u4, u5).toBlocking().toIterable();
```

A method annotated with @HystrixCollapser annotation can return any value with compatible type, it does not affect the result of collapser execution, collapser method can even return null or another stub. There are several rules applied for methods signatures.

- Collapser method must have one argument of any type, desired a wrapper of a primitive type like Integer, Long, String and etc.
- A batch method must have one argument with type java.util.List parameterized with corresponding type, that's if a type of collapser argument is Integer then type of batch method argument must be List<Integer>.
- Return type of batch method must be java.util.List parameterized with corresponding type, that's if a return type of collapser method is User then a return type of batch command must be List<User>.

**Convention for batch method behavior**
The size of response collection must be equal to the size of request collection.

```
@HystrixCommand
public List<User> getUserByIds(List<String> ids); // batch method

List<String> ids = List("1", "2", "3");
getUserByIds(ids).size() == ids.size();
```

Order of elements in response collection must be same as in request collection.

```
@HystrixCommand
public List<User> getUserByIds(List<String> ids); // batch method

List<String> ids = List("1", "2", "3");
List<User> users = getUserByIds(ids);
System.out.println(users);
// output
User: id=1
User: id=2
User: id=3
```

**Why order of elements of request and response collections is important?**
The reason of this is in reducing logic, basically request elements are mapped one-to-one to response elements. Thus if order of elements of request collection is different then the result of execution can be unpredictable.

**Deduplication batch command request parameters.**
In some cases your batch method can depend on behavior of third-party service or library that skips duplicates in a request. It can be a rest service that expects unique values and ignores duplicates. In this case the size of elements in request collection can be different from size of elements in response collection. It violates one of the behavior principle. To fix it you need manually map request to response, for example:

```
// hava 8
@HystrixCommand
List<User> batchMethod(List<String> ids){
// ids = [1, 2, 2, 3]
List<User> users = restClient.getUsersByIds(ids);
// users = [User{id='1', name='user1'}, User{id='2', name='user2'}, User{id='3', name='user3'}]
List<User> response = ids.stream().map(it -> users.stream()
            .filter(u -> u.getId().equals(it)).findFirst().get())
            .collect(Collectors.toList());
// response = [User{id='1', name='user1'}, User{id='2', name='user2'}, User{id='2', name='user2'}, User{id='3', name='user3'}]
return response;
```

Same case if you want to remove duplicate elements from request collection before a service call. Example:

```
// hava 8
@HystrixCommand
List<User> batchMethod(List<String> ids){
// ids = [1, 2, 2, 3]
List<String> uniqueIds = ids.stream().distinct().collect(Collectors.toList());
// uniqueIds = [1, 2, 3]
List<User> users = restClient.getUsersByIds(uniqueIds);
// users = [User{id='1', name='user1'}, User{id='2', name='user2'}, User{id='3', name='user3'}]
List<User> response = ids.stream().map(it -> users.stream()
            .filter(u -> u.getId().equals(it)).findFirst().get())
            .collect(Collectors.toList());
// response = [User{id='1', name='user1'}, User{id='2', name='user2'}, User{id='2', name='user2'}, User{id='3', name='user3'}]
return response;
```

To set collapser properties use @HystrixCollapser#collapserProperties

Read more about Hystrix request collapsing [here] (https://github.com/Netflix/Hystrix/wiki/How-it-Works#wiki-RequestCollapsing)

Collapser error processing Batch command can have a fallback method. Example:

```
   @HystrixCollapser(batchMethod = "getUserByIdsWithFallback")
   public Future<User> getUserByIdWithFallback(String id) {
      return null;
   }

   @HystrixCommand(fallbackMethod = "getUserByIdsFallback")
   public List<User> getUserByIdsWithFallback(List<String> ids) {
      throw new RuntimeException("not found");
   }
```

```
@HystrixCommand
private List<User> getUserByIdsFallback(List<String> ids) {
    List<User> users = new ArrayList<User>();
    for (String id : ids) {
        users.add(new User(id, "name: " + id));
    }
    return users;
}
```

#Development Status and Future Please create an issue if you need a feature or you detected some bugs. Thanks

Note: Javanica 1.4.+ is updated more frequently than 1.3.+ hence 1.4+ is more stable.

It's recommended to use Javanica 1.4.+