Introduction to Spring Cloud

Lab Guide & Solutions

Contents

Lab Guide 4

Lab 1 – Microservices 4

Task 1 – Discovery Server 4

Task 2 – Client Service 5

Task 3 – Account Service 5

Task 4 – Card Service 6

Task 5 – Processing Service 6

Lab 2 – Bottlenecks & Issues 7

Task 1 – Eureka peer to peer communication 7

Task 2 – Eureka self-preservation mode 8

Task 3 – Feign client with Hystrix circuit breaker 8

Lab 3 – API Gateway 9

Task 1 – Zuul Edge Proxy 9

Lab 4 – Centralized Configuration 10

Task 1 – Configuration Server 10

Task 2 – Configuration Client 10

Lab 5 – Hystrix 10

Task 1 – Hystrix Command 11

Task 2 – Hystrix Stream 11

Task 3 – Hystrix Dashboard 11

Lab 6 – Turbine 12

Task 1 – Turbine Stream 12

Task 2 – Processing Service 8805 12

Lab 7 – Turbine AMQP 13

Task 1 – AMQP Broker 13

Task 2 – Hystrix Stream 13

Task 3 – Turbine Stream 14

Lab 8 – Distributed Tracing with Zipkin 14

Task 1 – Instrumenting Applications 14

Task 2 – Zipkin Server 15

Lab 9 –Microservices Security 15

Task 1 – User Authentication & Authorization (UAA) Server 15

Task 2 – Microservices Security 16

Task 3 – Authorization behind Zuul 17

Lab 10 – Cloud Bus 17

Task 1 – Push Configurations 17

Task 2 – History Service 17

Lab 11 – Cloud Streams 19

Task 1 – Publisher 19

Task 2 – Subscriber 20

Lab 12 – Advanced Streams 20

Task 1 – Processor 20

Task 2 – Consumer Groups 21

Task 3 - Partitioning 21

Solutions 23

Lab 1 – Microservices 23

Task 1 – Discovery Server 23

Task 2 – Client Service 24

Task 3 – Account Service 25

Task 4 – Card Service 26

Task 5 – Processing Service 27

Lab 2 – Bottlenecks & Issues 29

Task 1 – Eureka peer to peer communication 29

Task 2 – Eureka self-preservation mode 30

Task 3 – Feign client with Hystrix circuit breaker 30

Lab 3 – API Gateway 31

Task 1 – Zuul Edge Proxy 31

Lab 4 – Centralized Configuration 32

Task 1 – Configuration Server 32

Task 2 – Configuration Client 33

Lab 5 – Hystrix 33

Task 1 – Hystrix Command 33

Task 2 – Hystrix Stream 34

Task 3 – Hystrix Dashboard 34

Lab 6 – Turbine 35

Task 1 – Turbine Stream 35

Task 2 – Processing Service 8805 35

Lab 7 – Turbine AMQP 36

Task 2 – Hystrix Stream 36

Task 3 – Turbine Stream 36

Lab 8 – Distributed Tracing with Zipkin 37

Task 1 – Instrumenting Applications 37

Task 2 – Zipkin Server 37

Lab 9 –Microservices Security 37

Task 1 – User Authentication & Authorization (UAA) Server 37

Task 2 – Microservices Security 39

Task 3 – Authorization behind Zuul 41

Lab 10 – Cloud Bus 42

Task 1 – Push Configurations 42

Task 2 – History Service 43

Lab 11 – Cloud Streams 45

Task 1 – Publisher 45

Task 2 – Subscriber 45

Lab 12 – Advanced Streams 46

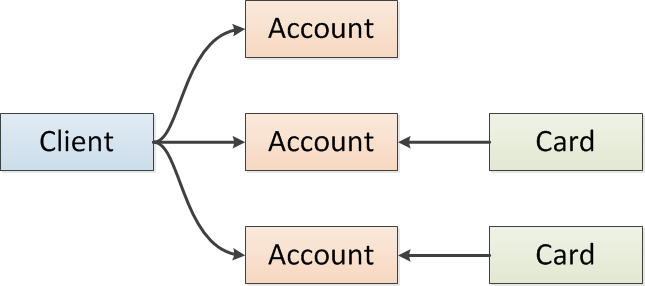
Task 1 – Processor 46

Task 2 – Consumer Groups 47

Task 3 – Partitioning 47

# Lab Guide

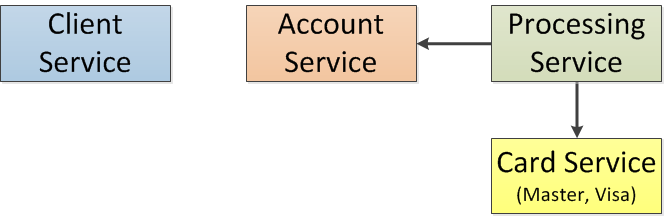
We are going to develop banking software with microservices approach. Out implementation will be oversimplified and is intended to demonstrate building blocks to build applications with microservices architecture. Out implementation covers a few basic aspects of banking domain. The story is about bank clients that may open accounts, order plastic cards and do checkout/withdrawal and fund/refund operations using different terminals/entry points.



Internet banking and other internal projects should also reuse existing infrastructure. This is very similar to SOA. We can overview microservices as special case of SOA, however in microservices approach there details that makes it to be considered a separate architectural pattern.

## Lab 1 – Microservices

In this lab we will create our first microservices with Spring Cloud. There will be many applications running at once. Figure below demonstrates topology of microservices to be built:



You can find source code for tasks in this lab under Labs directory in your training materials folder. It is suggested to add JVM command line argument *-Xmx128m* to all run configurations to limit memory usage.

### Task 1 – Discovery Server

Discovery servers could be considered as main components of microservices architecture. They help services to discover other services dynamically enabling horizontal scalability and many other features. In Labs folder, under Microservices directory you will find Eureka project. It is already configured to listen to port 8761. All our discovery servers will be running on ports 87\*\*. Perform the following steps using materials from your presentations:

1. Add SpringBoot parent and SpringCloud dependency management sections to parent pom.xml file.
2. Add eureka and eureka-server starter dependencies.
3. Implement SpringBoot main application class Eureka enabling Eureka server with annotations.
4. Finalize bootstrap.yml configuration file.
5. Build and run application
6. Open <http://localhost:8761> in web browser to load Eureka home page

### Task 2 – Client Service

First microservice to build is the client service. This service encapsulates CRUD operations with clients. For quick start, use the source code available in ClientService project. It is already preconfigured to listen to port 8801. All future services will listen to ports 88\*\*. Perform the following actions:

1. Implement SpringBoot main application class ClientService enabling Eureka client with annotations.
2. Finalize bootstrap.yml configuration file.
3. Implement ClientRest controller.
   1. Mark ClientRest class with @RestController annotation;
   2. Inject (autowire) ClientDao and ClientRepository instances;
   3. Implement “/create” endpoint with request parameter “name” @RequestMapping and @RequestParam annotations. To implement method logic use helper method ClientDao::create;
   4. Implement “/update/{id}” endpoint with path variable “id” using and request parameter “name”. Use @PathVariable annotation to achieve result. In this method, call ClientDao::update which returns a Boolean. In case of success (true) return HttpStatus.OK and HttpStatus.NOT\_FOUND in case of failure (false). Use ResponseBody class as method return type;
   5. Implement “/delete” endpoint with path variable “id” calling ClientRepository::delete;
   6. Implement “/get” endpoint using ClientRepository::findAll;
   7. Implement “/get” with path variable “id” using ClientRepository::findOne;
4. Build and run application
5. Open <http://localhost:8801> and test REST API using web browser;

Under resources directory, near bootstrap.yml file you will find application.yml SpringBoot configuration file. In this file there are commented lines related to Hibernate Auto-DDL management. After first run you will have an H2 database file created under user home directory of your OS. By default this file will be overwritten every time you run the project. Uncomment Auto-DDL related lines to leave the created database file untouched thus reusing your input data for next tests.

### Task 3 – Account Service

Another microservice to build is account service that encapsulates operations and actions related to accounts. In a real-world example microservices will not actually encapsulate CRUD operations; you would see more examples of business logic encapsulation behind methods that can be called over network instead of local calls. Take following actions in AccountService project from your lab source code which is configured with port 8802:

1. Create SpringBoot main application class AccountService enabling Eureka client with annotations.
2. Create bootstrap.yml configuration file.
3. Create AccountRest controller.
   1. Add @RestController annotation to class;
   2. Inject instances of AccountDAO and AccountRepository classes;
   3. Implement request mapping “/create” using AccountDAO::create;
   4. Implement request mapping “/fund” with Integer path variable “id” and BigDecimal request parameter “sum” using AccountDAO::addBalance with absolute value of sum provided (sum.abs());
   5. Implement request mapping “/checkout” with Integer path variable “id” and BigDecimal request parameter “sum” using AccountDAO::addBalance with negative value of sum provided (sum.abs().negate());
   6. Implement request mapping “/get” with Integer path variable “clientId” returning list of client accounts using AccountRepository::findByClientId;
4. Build and run application
5. Open <http://localhost:8802> and test REST API

Each service has its own H2 database. If you want to reuse data that you already entered, uncomment necessary lines in application.yml. This applies to all services that use a database. Review application.yml file to see if a service uses a database.

### Task 4 – Card Service

Now we have services to clients and accounts. We can onboard clients and open accounts for them. Card service is a mock service that encapsulates MasterCard, Visa or other plastic card vendor service that our bank is working with. It is configured at port 8803. Implement following steps in CardService project from your labs:

1. Implement SpringBoot main application class CardService enabling Eureka client with annotations.
2. Finalize bootstrap.yml configuration file.
3. Implement CardRest controller.
   1. Mark ClientRest class with @RestController annotation;
   2. Inject CardNumberGenerator instance;
   3. Implement “/create” request mapping returning the String number of the new card generated with CardNumberGenerator::generate helper method;
4. Build and run application
5. Open <http://localhost:8803> and test REST API

### Task 5 – Processing Service

Now that we have client, account and card services implemented, it is time for our most complicated service, the processing service. Processing service encapsulates card-related complex logic that involves several services to do a single job. Using processing service a bank service can issue a card for an account, checkout/withdraw funds from account using card and etc. Perform following steps in ProcessingService project to see cascading calls between microservices in action:

1. Add feign dependency.
2. Implement SpringBoot main application class ProcessingService enabling Eureka and Feign clients with according annotations.
3. Finalize bootstrap.yml configuration file.
4. Declare AccountServiceClient feign client with a method mapping for “/checkout/{id}”
5. Declare CardServiceClient feign client with a method mapping for “/create”
6. Implement ProcessingRest controller.
   1. Mark class with @RestController annotation;
   2. Inject instances of ProcessingRepository, AccountServiceClient and CardServiceClient interfaces;
   3. Implement “/issue/{accountId}” request mapping with following algorithm:
      1. Get generated card number using CardServiceClient;
      2. Create a ProcessingEntity object with generated card number and provided account id;
      3. Save ProcessingEntity using ProcessingRepository::save;
   4. Implement “/checkout/{card}” request mapping with Boolean return value having additional request parameter “sum” with following algorithm:
      1. Search for entities of type ProcessingEntity using helper repository method ProcessingRepository::findByCard;
      2. Return false if entity was not found;
      3. If entity found, return value of AccountServiceClient::checkout;
   5. Implement “/get” request mapping that returns accounts mapped to cards (Map<Integer, String>). For this purpose, accept list of Integer account ids and follow steps below:
      1. Read list of processing entities from database using helper repository method ProcessingRepository::findByAccountIdIn;
      2. Loop through the entity list and collect account ids and related cards into a map;
7. Build and run application
8. Open <http://localhost:8804> and test REST API

## Lab 2 – Bottlenecks & Issues

Microservices architecture carries out concepts that help building self-healing and fault-tolerant distributed systems. However, there several bottlenecks and issues that prevents these properties. Good news is that known bottlenecks and issues can be addressed easily.

### Task 1 – Eureka peer to peer communication

To eliminate single discovery server bottleneck we will emulate Eureka mirrored-style cluster running two Eureka instances at once. These instances will be running on ports 8701 and 8702. For this purpose,

1. create two new bootstrap files, “bootstrap-8701.yml” and “boostrap-8702.yml”;
   1. configure each bootstrap file to listen to port 870\* from file name;
   2. configure Eureka client in each bootstrap to register at another instance and fetch it’s registry;
2. create two Eureka run configurations in your IDE;
   1. configure run configurations to set special environment property running with command line parameter *-Dspring.cloud.bootstrap.name=bootstrap-870\** where “bootstrap-870\*” is name of the exact bootstrap configuration file;
3. reconfigure all microservices to lookup for any of two available Eureka instances declaring both of them comma separated;
   1. ClientService
   2. AccountService
   3. CardService
   4. ProcessingService
4. Build and run discovery servers and microservices;
5. Open both Eureka consoles at <http://localhost:8701> and <http://localhost:8702>
   1. Review cluster state in console and logs of Eureka

### Task 2 – Eureka self-preservation mode

We have already discussed network partitioning related issues. By default Eureka handles these kind of issues not forgetting registered services. Let’s consider we have a virtual network for our banking infrastructure and physical network partitions are addressed by some other team. In our banking infrastructure we want dead services to be unregistered dynamically. For this purpose we will turn off the self-preservation mode. So, dead services should be unregistered in 8-10 heartbeats timeframe. Let’s turn off self-preservation mode on all Eureka instances.

1. turn self-preservation mode off in bootstrap configuration files;
2. restart discovery servers;
3. modify configuration of some service by setting eureka.client.should-unregister-on-shutdown to false
4. after microservice is registered, stop it;
5. wait up to 4 minutes and check if dead microservices were unregistered by Eureka;
   1. this can be observed in logs or web-console

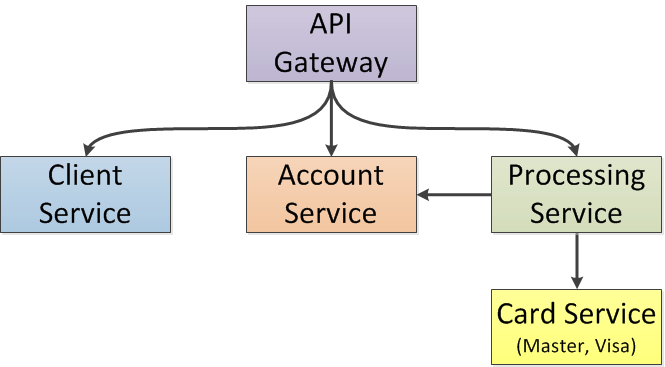
### Task 3 – Feign client with Hystrix circuit breaker

Now we are going to register fallback methods for cases when services are not available or it will take more time than required. Feign is already integrated with Hystrix circuit breaker. Default Hystrix timeout for Feign calls is 1 second. We should be able to returned predefined values in case of failed REST calls or timeouts. To arm Hystrix, take following steps in ProcessingService:

1. to implement CardServiceClient circuit breaker:
   1. create class CardServiceFallback implementing CardServiceClient interface;
   2. mark the class with @Component annotation;
   3. in the createCard method return null;
   4. in CardServiceClient interface add CardServiceFallback class as fallback parameter of @FeignClient annotation
   5. in implementation of ProcessingRest controller if CardServiceClient returns null, return “CARD\_SERVICE\_NOT\_AVAILABLE” message;
2. to implement AccountServiceClient circuit breaker:
   1. create AccountServiceFallback class implementing existing interface AccountServiceClient;
   2. mark the class with @Component annotation;
   3. in checkout method return false;
   4. in AccountServiceClient interface add AccountServiceFallback class as fallback parameter of @FeignClient annotation
3. Add dependency spring-cloud-starter-netflix-hystrix to ProcessingService pom.xml
4. Set feign.hystrix.enabled to true in ProcessingService configuration
5. Redeploy changes and run tests with all services up;
6. Kill card service and test how circuit breaker works;
7. Kill account service and test how circuit breaker works;

## Lab 3 – API Gateway

Usually microservices cloud is not exposed to outside world. Instead of that, an API gateway is exposed, so that client applications has a single entry point.



### Task 1 – Zuul Edge Proxy

To create an API gateway we will use Zuul, the embeddable edge proxy from Netflix. To build API gateway with Zuul, perform the following steps in Zuul project (preconfigured on port 8700) from your lab source code:

1. Add zuul starter dependency spring-cloud-starter-netflix-zuul;
2. Implement SpringBoot main application class Zuul enabling zuul-proxy with according annotation;
3. Finalize bootstrap.yml configuration file:
   1. fetch registry for Zuul but do not register with Eureka as it is not a microservice;
   2. ignore all services by default;
   3. register custom routes:
      1. ClientService: /client;
      2. AccountService: /account;
      3. ProcessingService: /processing;
4. Build and run application
5. Test API gateway using <http://localhost:8700>

## Lab 4 – Centralized Configuration

As you have already seen, it is too much of hassle to edit bootstrap and application configurations when something related to microservices changes.

### Task 1 – Configuration Server

A centralized configuration server helps to remove configuration overhead and keeps all necessary configuration files in one repository. Perform steps below using ConfigServer project from lab source codes (preconfigured on port 8600):

1. Add config-server dependency spring-cloud-config-server.
2. Implement SpringBoot main application class ConfigServer enabling configuration server with annotations.
3. Create a local Git repository on your HDD
4. Finalize bootstrap.yml configuration file with spring.cloud.config.server.git.uri property set to local Git repository using file: prefix using absolute path.
5. In Git repository created before
   1. create application.yml file with Eureka client configuration common to all microservices;
   2. create named application YAML configurations for all other applications
      1. for Eureka create files with port number as profile name (use spring.profiles.active command line argument with port number as profile name instead of spring.cloud.bootstrap.name with file name);
      2. for Zuul override only eureka.client.registerWithEureka property;
6. Build and run configuration server
7. Test using [http://localhost:8600/${AppName}/${profile}](http://localhost:8600/$%7bAppName%7d/$%7bprofile%7d). For instance:
   1. <http://localhost:8600/CardService/default>
   2. <http://localhost:8600/Eureka/8701>

### Task 2 – Configuration Client

Configure all applications leaving in bootstrap files only application name and configuration server URL in spring.cloud.config.uri parameter. All other configuration options including application.yml files should be moved to ConfigServer.

## Lab 5 – Hystrix

### Task 1 – Hystrix Command

To see an example of hystrix command works, we will create a test method that throws exception depending on passed parameters value. Go to ProcessingService project and follow steps below:

1. Add hystrix dependency spring-cloud-starter-netflix-hystrix
2. Add @EnableHystrix annotation to ProcessingService configuration class
3. Create a method with ‘/test’ request mapping
   1. Accept Boolean ‘fail’ parameter;
   2. If TRUE, throw a RuntimeException;
   3. If FALSE return “OK”;
   4. Annotate it with @HystrixCommand and provide a name for fallback method
4. Implement the fallback method
   1. Always return “FAILED”;
5. Run the application and test if HystrixCommand works when ‘/test?fail=1’ is called.

### Task 2 – Hystrix Stream

To monitor activities over hystrix commands in an application, add spring boot actuator dependency to the application. When hystrix and spring boot actuator dependencies are on the classpath, spring cloud applications will have “/hystrix.stream” endpoint enabled

1. Add spring boot actuator dependency spring-boot-starter-actuator;
2. Add to ProcessingService configuration: management.endpoints.web.exposure.include: hystrix.stream
3. Test if “/actuator/hystrix.stream” is available;
   1. Note, hystrix stream may not produce any output until the first hystrix command execution;
   2. If a hystrix command is not executed, you may not see it appear in the hystrix stream;
4. To see an ouput in hystrix stream, run “/test” from previous task;
   1. or any of endpoints that utilize a feign client;

### Task 3 – Hystrix Dashboard

Hystrix stream itself is not human readable. To visualize a hystrix stream, we will run a hystrix dashboard application. For this purpose, take following steps in Dashboard application from your labs.

1. Create the bootstrap configuration;
   1. Set application name ‘Dashboard’;
   2. Set port 8900;
2. Add hystrix dashboard dependency;
3. Enable hystrix dashboard adding correct annotation to Dashboard configuration class;
4. Now you can run the Dashboard application and navigate to <http://localhost:8900/hystrix> to see the hystrix dashboard.
   1. Point it to hystrix stream of ProcessingService at <http://localhost:8804/actuator/hystrix.stream>

Usually it makes sense to monitor API gateway too, as it is the entry point for end-users. It will be enough to simply navigate “/hystrix.stream” endpoint of Zuul application. Zuul by default should include hystrix and actuator dependencies. If for any reason “/hystrix.stream” is not available for you, simply throw in spring-boot-actuator dependency.

## Lab 6 – Turbine

### Task 1 – Turbine Stream

Turbine streams are aggregation of hystrix streams from available instances of same service. Aggregation is done by turbine, which is an external component from service point of view. So, turbine could be integrated to any external application to provide a turbine stream. Let’s use Dashboard we developed in previous lab as our turbine aggregator. This means that Dashboard should access Eureka. If you remember, previously we prepared ConfigServer with default Eureka access for all applications and Dashboard was not using ConfigServer. However, this time we will add Dashboard configuration to ConfigServer too. To create the turbine stream do the following actions in Dashboard project:

1. Add starter dependencies for configuration client and turbine;
2. Enable turbine adding appropriate annotation to configuration class;
3. Change bootstrap configuration to contain application name and configuration server URL;
4. Create the Dashboard.yml configuration in configuration server Git repository;
   1. Add port 8900;
   2. Turn off registration at Eureka as we are only a client, not a service;
   3. Add turbine configuration for Processing Service cluster
5. Run the necessary applications and navigate hystrix dashboard to <http://localhost:8900/turbine.stream?cluster=PROCESSINGSERVICE>

Please, note that, as turbine stream aggregates output of hystrix streams, there may be no data until a hystrix command will be executed. Because, as we have already seen before, hystrix streams provide data after the first hystrix command is executed.

/hystrix.stream endpoint is provided by Spring Boot Actuator and is available under management context. Thus, if you set a custom management port, /hystrix.stream won’t be available to turbine. Turbine uses Eureka to discover hosts and ports of microservices. Management port is not populated to Eureka. You should disable management port in order to use vanilla turbine with HTTP streams. Turbine AMQP should be used, in order to enable management port independently from turbine.

### Task 2 – Processing Service 8805

There is only one instance of Processing Service. So far, we have simply mirrored the hystrix stream at turbine. To see results of the real aggregation, let’s start another instance of Processing Server. To do so,

1. Create a copy of ProcessingService.yml in configuration server repository with file name “ProcessingService-8805.yml”;
   1. 8805 is the new profile name;
2. Create a new Run configuration in your IDE for ProcessingService
   1. You can copy existing configuration with different name, e.g. “ProcessingService-8805”
   2. Add “-Dspring.profiles.active=8805” parameter to JVM options

**Note:** H2 database is used in file mode. This way second instance won’t be able to startup as first instance locks the file. Avoiding lock, can damage the database file. To eliminate the issue:

1. Add “;AUTO\_SERVER=TRUE” to the end of JDBC URL for both configurations of Processing Service;
2. Run new configuration and check if ProcessingService:8805 appears in Eureka
3. Execute any hystrix command at 8805
   1. For instance, run /test
4. Watch changes in turbine stream
   1. If turbine stream does not see the second instance, restart it (the turbine stream).

Notice that if you failed and successful /test commands at different Processing Service instances (“:8804/test?fail=0” and “:8805/test?fail=1”) turbine stream will show 50% of system is working. Watch thread pools and other options in action while triggering different actions with different results on separate instance to see behavior of the system monitored in real-time.

## Lab 7 – Turbine AMQP

In this lab we will aggregate all available stream to a single turbine stream through an AMQP broker. We’ll then use hystrix dashboard to see analyze the aggregated stream.

### Task 1 – AMQP Broker

We will use RabbitMQ as our AMQP broker. First of all, it should be installed locally. Please execute following steps, to install RabbitMQ to your computer:

1. Install Erlang
   1. Download it from <http://www.erlang.org/downloads>
2. Install RabbitMQ
   1. Download it from <https://www.rabbitmq.com/download.html>
3. Enable RabbitMQ management plugin
   1. Run “*rabbitmq-plugins enable rabbitmq\_management*” from command line
   2. “*rabbitmq-plugins*” is located in “*sbin*” direcroty under RabbitMQ installation folder
4. Open <http://localhost:15672/> to see RabbitMQ management web-page in action
   1. Default administrator login/password for RabbitMQ on fresh install is guest/guest

### Task 2 – Hystrix Stream

Now, it is time to aggregate all available hystrix streams to single turbine stream. To do so, let’s add hystrix AMQP feed stream to Processing Service and Zuul.

1. Add spring cloud hystrix stream dependency spring-cloud-netflix-hystrix-stream;
2. Add spring cloud rabbitmq stream starter dependency;
3. Re-run applications;

### Task 3 – Turbine Stream

Hystrix streams from applications are now feed to RabbitMQ exchange. However, we still need a single entry point, which consumes data from stream and pushes aggregated data to turbine stream. This is a server functionality that does not belong to any of existing application. Instead, this server should be the new standalone application, which is always available and any hystrix dashboard can be pointed to it. For this purpose, we will create a Turbine Server. Follow steps below using TurbineServer project folder from your labs.

1. Add spring cloud turbine stream starter dependency;
2. Add spring cloud rabbitmq stream starter dependency;
3. In bootstrap.yml
   1. Set application name “TurbineServer”
   2. Set port 8901
4. Implement main class enabling the Turbine Stream with corresponding annotation;
5. Since the Rabbit MQ queue is hystrixStreamOutput, and TurbineServer is receiving data from springCloudHystrixStream, we need either to redirect stream or ask TurbineServer to read from hystrixStreamOutput. To configure TurbineServer to read data from hystrixStreamOutput, add this to configuration: turbine.stream.destination: hystrixStreamOutput.
6. Run the server and point hystrix dashboard (<http://localhost:8900/hystrix>) to <http://localhost:8901>
7. If you are running multiple instances of the service on the same host (and different port), its instanceId will be the same (i.e. for ProcessingService it is "instanceId":"ProcessingService-1" for both instances, which doesn’t allow to aggregate instance information). To fix it, you should configure ProcessingService with hystrix.stream.queue.send-id: false. It will disable sending context id as the metric id, Turbine AMQP then defaults to serviceId:host:port and aggregation works properly.

Now you can run some hystrix commands and see aggregated results at hystrix dashboard that is monitoring turbine stream that aggregates data from message queue.

## Lab 8 – Distributed Tracing with Zipkin

In this lab we are going to instrument our applications with Spring Cloud Sleuth to provide tracing information to Zipkin server. We will then analyze this information with Zipkin UI to see some meaningful information about what is happening when cascading service calls take place and how much of latency there is between them.

### Task 1 – Instrumenting Applications

We will stream Sleuth tracing information to RabbitMQ we already did set up in previous lab. The default Sleuth stream (exchange) name is “sleuth” and it will be auto-configured automatically. Please follow the steps below for all services, the configuration service and discovery servers.

1. Add dependencies for spring-cloud-starter-zipkin and RabbitMQ stream (groupId org.springframework.amqp, artifactId spring-rabbit) to projects.
2. Run the projects.

### Task 2 – Zipkin Server

1. Download and run Zipkin server.
2. Add logging to CardRest:

**private static Logger *log* = LoggerFactory.*getLogger*(CardRest.class);**

Use it in createNewCard() method:

***log*.info("generating a card number");**

1. Add logging to ProcessingRest issueNewCard
2. Execute new card generation via Processing service, find traceId in log
3. Use traceId to analyze a trace (enter it into right top corner in Zipkin), look at dependencies
4. Do the same for AccountRest

## Lab 9 –Microservices Security

In this lab we will create a security layer between microservices, enable token exchange and relay and will provide the single sign on feature through API gateway. Practical tasks in this section are here for demonstration purposes only and are not covering all aspects of security as there way more nuances to be reviewed and worked on. In production environment you will have to go through more details to make sure that you cover all security concerns of your organization.

### Task 1 – User Authentication & Authorization (UAA) Server

UAA will be available to all applications in our network and will provide authentication and authorization services. It will be a simple Spring Boot application. UAA servers usually are used by many applications, not necessarily microservices. For this reason, UAA servers are scaled using different approaches and not necessarily Eureka or service discovery at all. Please proceed to following steps to create the UAA using AuthService project in your labs directory:

1. Generate a new key store using command line:
   * keytool -genkeypair -alias jwt -keyalg RSA -keypass mySecretKey -keystore jwt.jks -storepass mySecretKey
2. Put the generated jwt.jks key store into the /main/resource folder
3. Add dependencies
   * **org.springframework.boot:spring-boot-starter-security**
   * **org.springframework.cloud:** **spring-cloud-starter-oauth2**
4. Check the application configuration (application.yml) to have
   * AuthService as application name
   * Port 8500
   * Servlet context path “/uaa”
5. Implement the OAuth2Configuration class as shows in slides
6. Implement the WebSecurityConfig class as shows in slides
7. Run the application to start the UAA server.

Spring Boot main class is already implemented; it is a simple application main class that will automatically load the configuration files you created.

Using a REST client (e.g. DHC for Google Chrome) you can make a POST call to http://localhost:8500/uaa/oauth/token?grant\_type=password&username=account&password=account to get the token. Please note, that UAA requests will require HTTP Basic authentication with clientId and clientSecret.

### Task 2 – Microservices Security

Now we have the UAA server in place, we can turn our microservices into resource servers. For this purpose,

1. Add dependencies for
   * spring-boot-starter-security
   * spring-cloud-starter-oauth2
2. Add the resource server configuration class from slides to account, card, client and processing services.
3. Add oauth2 security configuration to application.yml (in config-repo).
4. Add annotation to configuration class
   * **@EnableGlobalMethodSecurity(prePostEnabled = true)**
5. Add method or class security to RestController classes using
   * **@PreAuthorize("hasAuthority('…')")**

In resource server configuration of ProcessingService additionally:

1. Declare default load balanced (@Primary) OAuth2RestTemplate
   * Name it “OAuth2RestTemplate”
2. Declare secondary load balanced OAuth2RestTemplate with password grant type and CardService credentials
   * Name it “OAuth2CardService”
   * Provide a new DefaultOAuth2ClientContext with DefaultAccessTokenRequest based on ResourceOwnerPasswordResourceDetails with corresponding accessTokenUri (from existing configuration), clientId, clientSecret, username and password
3. Use OAuth2RestTemplate instead of Feign clients
   * For AccountService you will have to provide token value from security context SecurityContextHolder.getContext().getAuthentication().getDetails()).getTokenValue() casting it to OAuth2AuthenticationDetails

Now you can run the services together with UAA and test method security using access tokens that you can fetch manually from UAA.

### Task 3 – Authorization behind Zuul

To be able to use Zuul with authorization, you need to forward authorization headers. For this, you need to remove Authorization header from sensitiveHeaders in Zuul configuration.

## Lab 10 – Cloud Bus

Now, let’s add the Cloud Bus to our microservices and organize a live event-bus scaled to microservices cloud. This cloud event bus allows us to receive Spring native Application Events remotely, but still very close to Spring standards.

Before you proceed to tasks, for simplicity of testing, configure applications to use a different management port and turn off management security. To achieve objectives, use following example for configuration server and use a 9\*\*\* port for all other services and applications you will be attaching to Spring Cloud Bus.

**management:  
 port:** 9600  
 **security:  
 enabled:** false

### Task 1 – Push Configurations

We have introduced Config Server previously in Externalized Configuration chapter. With Spring Cloud Bus and Spring Cloud Config Monitor we will enable automatic context reload of services upon a configuration change related to a single or group of services. We already have RabbitMQ installed, thus we will use Cloud Bus AMQP binding. For this purpose,

1. Add starter bus-amqp and config-monitor dependencies to Config Server
2. Add starter bus-amqp and actuator dependencies to all projects you want to listen for push notifications and automatic context reload with new configuration settings (e.g. Client Service, Account Service, Processing Service, Card Service, Zuul, Dahsboard projects).
3. Add @RefreshScope to the configurable bean and @Value for config value to be changed.

Now start the services and test the following (run inside 2 instances of ProcessingService):

* Try to change configuration files, watching logs and checking if application context reloads;
* Add management.endpoints.web.exposure.include: '\*' to the service parameters to allow various management endpoints;
* POST to /actuator/bus-env with updated parameter value;
* POST to /actuator/bus-refresh to refresh all refreshable values for all instances;
* Check if the parameter has changed for all the instances.

### Task 2 – History Service

History Service is an oversimplified service that is intended to demonstrate the basic usage of features available. In a bigger project, a more complex implementation could take place (in such architectures like CQRS and Event Sourcing).

Basically, we will trigger financial events on successful execution of “/fund” and “/checkout” endpoints of Account Services. History Service should receive these events and register financial operation in history registry, so that any other consumer can request history with no load to Account Service itself. Moreover, as Processing Service is dependent on Account Service, card operations on accounts will also be tracked by History Service. Please take following steps to implement the cloud event bus:

1. In “Common” project under Labs directory:
   1. Add cloud-bus dependency;
   2. Implement AbstractFinancialEvent class extending RemoteApplicationEvent;
      1. With constructor with no parameters;
      2. With constructor that accept arguments:
         * String originService;
         * String destinationService;
         * BigDecimal sum;
      3. With getSum() method that returns the sum;
   3. Implement FundEvent and WithdrawEvent classes extending AbstractFinancialEvent;
      1. With both constructors declared;
   4. Implement RemoteEventPublisher class;
      1. With @Component annotation;
      2. With SpringCloudBusClient auto-wired;
      3. With publishEvent(RemoteApplicationEvent event) method using:
         * **busClient.springCloudBusOutput()**

**.send(**

**MessageBuilder**

**.withPayload(event)**

**.build()**

**);**

1. In Account Service project:
   1. Add dependency to Common project;
   2. Add @RemoteApplicationEventScan to configuration class;
   3. Autowire RemoteEventPublisher in REST controller and publish corresponding events to bus in “checkout” and “fund” methods;
2. In History Service project:
   1. Add dependency to “Common” project;
   2. Add starter bus-amqp dependency;
   3. Add starter config dependency;
   4. Check if bootstrap.yml contains correct application name and config server URI;
   5. Implement HistoryService class as follows:
      1. Implement ApplicationListener<AbstractFinancialEvent> or use @EventListener whichever you like most;
      2. with Get mapping returning Map<Date, BigDecimal>;
      3. with singleton synchronized LinkedHashMap of corresponding type;
      4. with override to onApplicationEvent(AbstractFinancialEvent event) method;
         * use date value of event timestamp as key;
         * use event sum as value;
      5. Check the class to contain SpringBootApplication, RemoteApplicationEventScan and RestController annotations, as well as main method;
3. Now run the application, execute several checkout and fund operations and check test if History Service works correctly.

What is not covered in this lab and optionally could be covered if you finish earlier than others is:

* Financial events should carry information about the client and the account;
  + Client who execute the operation;
  + Account on which the operation was executed;
* Secure the History Service with method security to provide operation history only to Clients;
  + getHistory() method should check for client/client authorization
    - Think about how you could authorize an exact Client by his/her ID;
    - Expose this endpoint at API gateway;
    - Think about SSO;
* Use database instead of in-memory data structure;

## Lab 11 – Cloud Streams

We have already seen reactive properties such as Responsiveness, Resilience and Elasticity supported by Spring Cloud applications. The only reactive property left is Message-Driven. In this lab, we will cover Spring Cloud Stream, thus the left reactive property. This is where Spring Cloud helps to achieve full list of reactivity properties described by reactive manifesto and regarded by big systems developed.

In this lab we will develop a simple word-count application that counts how many times a word was repeatedly entered. Words are entered to an HTTP endpoint of Publisher application. Publisher then sends these words to “testChannel”. On the other side the Subscriber application is listening to “testChannel”. Whenever a word comes out from the channel, Subscriber counts the unique words and exposes an HTTP endpoint to view result.

### Task 1 – Publisher

The Publisher is the application that receives words to “/publish” endpoint and sends them to “testChannel” stream. Proceed with following steps using stub in Publisher directory under Labs folder:

1. Add the starter-web, starter-config and starter-stream-rabbit dependencies;
2. Create the TestChannel interface using corresponding annotation and channel type;
   1. Use @Output annotation;
   2. Use “testChannel” as channel name;
   3. Use output() method name with MessageChannel return value;
3. Create the Spring Boot application class with corresponding annotation to activate TestChannel
4. Create the REST controller with “/publish” endpoint
   1. Receive “word” parameter as String;
   2. Auto-wire the output channel;
   3. Send the word to channel using MessageBuilder as used with Spring Cloud Bus;
5. Add the bootstrap configuration to configuration repository with port 8001
6. Run the application

To test if Publisher works, hit the “/publish” endpoint with a word and watch the RabbitMQ management console to see if message is received by broker.

### Task 2 – Subscriber

The Subscriber application listens for incoming words. If the was already sent, it increments the counter. If the word is sent for the first time, then add it with count 1. Please, follow steps below:

1. Add the starter-web, starter-config and starter-stream-rabbit dependencies;
2. Create the TestChannel interface using corresponding annotation and channel type;
   1. Use @Input annotation;
   2. Use “testChannel” as channel name;
   3. Use input() method name with SubscribableChannel return value;
3. Create the Spring Boot application class with corresponding annotation to activate TestChannel
4. Create the StreamListener with word-count logic
5. Create the REST controller with GET mapping at root path (“/”) that return the map;
6. Add the bootstrap configuration to configuration repository with port 8002
7. Run the application

Now run the application and test:

* Publish word to Publisher HTTP endpoint;
* Check Subscriber HTTP endpoint for word counts;

Note that we have used custom interface for channel definition. You may use standard Source and Sink interfaces for Publisher and Subscriber respectively. However, in that case you are required to define the channel name in properties as follows:

* spring.cloud.stream.bindings.**output**.destination: testChannel
* spring.cloud.stream.bindings.**input**.destination: testChannel

## Lab 12 – Advanced Streams

In this lab we will modify our word-count application. We will add an intermediate processor between Publisher and Subscriber that will process all words to upper case to count unique words only.

### Task 1 – Processor

To introduce Processor, we need to make some necessary changes to existing Publisher and Subscriber application. Processor uses two channels, one for input and second for output. The Publisher should send its output to Processor’s input, and Processor will send its output to Subscribers input channel. Proceed with following steps please:

1. Create the PublisherChannel interface with channel name “words” in Publisher;
2. Attach PublisherChannel to Publisher project instead of TestChannel;
3. Create ProcessorChannel interface in Processor project;
   1. Use “words” channel for input;
   2. Use “upperWords” channel name for output channel;
4. Implement the Processor Spring Boot application main class;
   1. Create a process(String word) method that returns word.toUpperCase();
   2. Annotation the process() method with @StreamListener and @SendTo annotations;
   3. Bind Processor application to port 8003;
5. Create the SubscriberChannel interface with channel name “upperWords” in Subscriber;
6. Attach SubscriberChannel to Subscriber project instead of TestChannel;
7. Run the applications and test the unique word-count with case-insensitive words;

You might ask why we bind Processor to a port if it does not serve any HTTP request. In production environments this is required for management purposes. Any Spring Boot/Cloud application may benefit of Actuator and health-checks. Spring Cloud Stream exposes its health indicator to actuator namespace. Then you can manage it with REST, JMX or such applications like Spring Boot Admin or Actuator UI.

### Task 2 – Consumer Groups

Usually processor applications may take load and become slower. To unload the Processor we can create a consumer group of Processors that will load balance the incoming messages. For this purpose:

1. Create a separate profile for second Processor on port 8004;
2. In configuration for “words” channel:
   1. Declare the group name “Processors”;
   2. Set consumer instance count 2;
   3. Set consumer instance index:
      1. 0 for 8003;
      2. 1 for 8004;
3. Run the applications
   1. You may log values received to channel for each processor to see if it correctly works.

Notice that if you remove the group definition (“Processors”) you will see that subscription becomes anonymous, thus each processor will receive the word sent which is not right. We need a consumer group to be declared to be able to load balance between instances of this group.

### Task 3 - Partitioning

Now it is time to think about the traffic that is coming in from Publisher. First of all, we are counting words and not numbers. It makes sense to throw in another processor which will filter numbers and pass in only words. Let’s call it Splitter.

On the other hand, we may have big loads of words coming in to Splitter. It makes sense to partition splitter into several filtering processes on same channel but different queues. And then, the consumer group of processors will receive only words into a single queue load balanced between several consumers.

Please follow the steps below to implement the word-number splitter application in a partitioned scenario:

1. Implement WordNumSplitter in your Labs directory using Processor interface of SpringCloudStream;
2. Configure WordNumSplitter bootstrap as follows:
   1. Bind the “input” channel destination to “wordNumSplit”;
      1. With group “Splitters”;
      2. With partitioned consumer;
      3. With instance count 2;
      4. With instance index 0;
   2. Bind the “output” channel destination to “words”;
   3. Bind to port 8005;
3. Create a second profile “8006” bound to port 8006 and with instance index 1 for input channel;
4. Implement the method to filter words and number;
   1. Lister to Processor.INPUT;
   2. Send to Processor.OUTPUT;
5. Configure the Publisher as follows:
   1. Bind the “words” channel destination to “wordNumSplit”;
   2. Use producer partition key “payload.startsWith("h")?0:1” – this way we will forward messages started with “h” to WordNumSplitter with instance index 0, and all others to WordNumSplitter with instance index 1
   3. Set partition count to 2
6. Now restart the applications and check if the flow works.

If you check the RabbitMQ management console, you will notice two separate queues with partition index appended.

# Solutions

## Lab 1 – Microservices

### Task 1 – Discovery Server

1. Add SpringBoot parent and SpringCloud dependency management sections to parent pom.xml file.

**<parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.2.7.RELEASE</version>  
 </parent>  
 <dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-dependencies</artifactId>  
 <version>Hoxton.SR4</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 </dependencies>  
 </dependencyManagement>**

1. Add eureka-server dependency to Eureka

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  
 </dependency>**

1. Create SpringBoot main application class Eureka enabling Eureka server with annotations

**import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;  
  
@SpringBootApplication  
@EnableEurekaServer  
public class Eureka {  
 public static void main(String[] args) {  
 SpringApplication.*run*(Eureka.class, args);  
 }  
}**

1. Create bootstrap.yml configuration file

**spring:  
 application:  
 name: Eureka  
  
server:  
 port: 8761  
  
eureka:  
 client:  
 fetchRegistry: false  
 registerWithEureka: false**

### Task 2 – Client Service

1. Implement SpringBoot main application class ClientService enabling Eureka client with annotations

**import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  
  
@SpringBootApplication  
@EnableEurekaClient  
public class ClientService {  
 public static void main(String[] args) {  
 SpringApplication.*run*(ClientService.class, args);  
 }  
}**

1. Add eureka-client dependency

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
 </dependency>**

1. Create bootstrap.yml configuration file

**spring:  
 application:  
 name: ClientService  
  
server:  
 port: 8801  
  
eureka:  
 client:  
 fetchRegistry: true  
 registerWithEureka: true  
 serviceUrl:  
 defaultZone: http://localhost:8761/eureka**

1. Create Client REST controller

**import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.http.HttpStatus;  
import org.springframework.http.ResponseEntity;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestParam;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.util.List;  
  
@RestController  
public class ClientRest {  
 @Autowired  
 private ClientDAO dao;  
  
 @Autowired  
 private ClientRepository repo;  
  
 @RequestMapping("/create")  
 public Client create(@RequestParam String name) {  
 return dao.create(name);  
 }  
  
 @RequestMapping("/update/{id}")  
 public ResponseEntity update(@PathVariable Integer id, @RequestParam String name) {  
 if (dao.update(id, name)) {  
 return new ResponseEntity(HttpStatus.*OK*);  
 } else {  
 return new ResponseEntity(HttpStatus.*NOT\_FOUND*);  
 }  
 }  
  
 @RequestMapping("/delete/{id}")  
 public void delete(@PathVariable Integer id) {  
 repo.deleteById(id);  
 }  
  
 @RequestMapping("/get")  
 public List<? extends Client> get() {  
 return repo.findAll();  
 }  
  
 @RequestMapping("/get/{id}")  
 public Client get(@PathVariable Integer id) {  
 return repo.findById(id).orElse(null);  
 }  
}**

### Task 3 – Account Service

1. Create SpringBoot main application class AccountService enabling Eureka client with annotations.

**import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  
  
@SpringBootApplication  
@EnableEurekaClient  
public class AccountService {  
 public static void main(String[] args) {  
 SpringApplication.*run*(AccountService.class, args);  
 }  
}**

1. Add eureka-client dependency

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
 </dependency>**

1. Create bootstrap.yml configuration file.

**spring:  
 application:  
 name: AccountService  
  
server:  
 port: 8802  
  
eureka:  
 client:  
 fetchRegistry: true  
 registerWithEureka: true  
 serviceUrl:  
 defaultZone: http://localhost:8761/eureka**

1. Create AccountRest controller.

**import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestParam;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.math.BigDecimal;  
import java.util.List;  
  
@RestController  
public class AccountRest {  
 @Autowired  
 private AccountDAO dao;  
  
 @Autowired  
 private AccountRepository repo;  
  
 @RequestMapping("/create")  
 public void create(@RequestParam("client\_id") Integer clientId) {  
 dao.create(clientId);  
 }  
  
 @RequestMapping("/fund/{id}")  
 public boolean fund(@PathVariable Integer id, @RequestParam BigDecimal sum) {  
 return dao.addBalance(id, sum.abs());  
 }  
  
 @RequestMapping("/checkout/{id}")  
 public boolean checkout(@PathVariable Integer id, @RequestParam BigDecimal sum) {  
 return dao.addBalance(id, sum.abs().negate());  
 }  
  
 @RequestMapping("/get/{clientId}")  
 public List<? extends Account> getByClient(@PathVariable Integer clientId) {  
 return repo.findByClientId(clientId);  
 }  
}**

### Task 4 – Card Service

1. Create SpringBoot main application class CardService enabling Eureka client with annotations

**import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  
  
@SpringBootApplication  
@EnableEurekaClient  
public class CardService {  
 public static void main(String[] args) {  
 SpringApplication.*run*(CardService.class, args);  
 }  
}**

1. Create bootstrap.yml configuration file

**spring:  
 application:  
 name: CardService  
  
server:  
 port: 8803  
  
eureka:  
 client:  
 fetchRegistry: true  
 registerWithEureka: true  
 serviceUrl:  
 defaultZone: http://localhost:8761/eureka**

1. Create Client REST controller

**import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class CardRest {  
 @Autowired  
 private CardNumberGenerator generator;  
  
 @RequestMapping("create")  
 public String createNewCard() {  
 return generator.generate();  
 }  
}**

### Task 5 – Processing Service

1. Add feign dependency.

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-openfeign</artifactId>  
</dependency>**

1. Add eureka-client dependency

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
 </dependency>**

1. Create SpringBoot main application class ProcessingService enabling Eureka and Feign clients with according annotations.

**import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  
import org.springframework.cloud.openfeign.EnableFeignClients;  
  
@SpringBootApplication  
@EnableEurekaClient  
@EnableFeignClients  
public class ProcessingService {  
 public static void main(String[] args) {  
 SpringApplication.*run*(ProcessingService.class, args);  
 }  
}**

1. Create bootstrap.yml configuration file.

**spring:  
 application:  
 name: ProcessingService  
  
server:  
 port: 8804  
  
eureka:  
 client:  
 fetchRegistry: true  
 registerWithEureka: true  
 serviceUrl:  
 defaultZone: http://localhost:8761/eureka**

1. Create AccountServiceClient feign client with a method mapping for “/checkout/{id}”

**import org.springframework.cloud.openfeign.FeignClient;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestParam;  
  
import java.math.BigDecimal;  
  
@FeignClient("AccountService")  
public interface AccountServiceClient {  
 @RequestMapping("/checkout/{id}")  
 boolean checkout(@PathVariable("id") Integer accountId, @RequestParam("sum") BigDecimal sum);  
}**

1. Create CardServiceClient feign client with a method mapping for “/create”

**import org.springframework.cloud.openfeign.FeignClient;  
import org.springframework.web.bind.annotation.RequestMapping;  
  
@FeignClient("CardService")  
public interface CardServiceClient {  
 @RequestMapping("create")  
 String createCard();  
}**

1. Create ProcessingRest controller.

**import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestParam;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.math.BigDecimal;  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
  
@RestController  
public class ProcessingRest {  
 @Autowired  
 private ProcessingRepository repo;  
  
 @Autowired  
 private AccountServiceClient accountServiceClient;  
  
 @Autowired  
 private CardServiceClient cardServiceClient;  
  
 @RequestMapping("/issue/{accountId}")  
 public String issueNewCard(@PathVariable Integer accountId) {  
 final String card = cardServiceClient.createCard();  
 ProcessingEntity pe = new ProcessingEntity();  
 pe.setCard(card);  
 pe.setAccountId(accountId);  
 repo.save(pe);  
 return card;  
 }  
  
 @RequestMapping("/checkout/{card}")  
 public boolean checkout(@PathVariable String card, @RequestParam BigDecimal sum) {  
 ProcessingEntity pe = repo.findByCard(card);  
 if (pe == null) {  
 return false;  
 }  
 return accountServiceClient.checkout(pe.getAccountId(), sum);  
 }  
  
 @RequestMapping("/get")  
 public Map<Integer, String> getByAccount(@RequestParam("account\_id") List<Integer> accountIdList) {  
 List<ProcessingEntity> list = repo.findByAccountIdIn(accountIdList);  
 Map<Integer, String> map = new HashMap<Integer, String>();  
 for (ProcessingEntity pe: list) {  
 map.put(pe.getAccountId(), pe.getCard());  
 }  
 return map;  
 }  
}**

## Lab 2 – Bottlenecks & Issues

### Task 1 – Eureka peer to peer communication

* bootstrap-8701.yml

**spring:  
 application:  
 name: Eureka  
  
server:  
 port: 8701  
  
eureka:  
 client:  
 fetchRegistry: true  
 registerWithEureka: true  
 serviceUrl:  
 defaultZone: http://localhost:8702/eureka**

* bootstrap-8702.yml

**spring:  
 application:  
 name: Eureka  
  
server:  
 port: 8702  
  
eureka:  
 client:  
 fetchRegistry: true  
 registerWithEureka: true  
 serviceUrl:  
 defaultZone: http://localhost:8701/eureka**

* microservices

**eureka:  
 client:  
 fetchRegistry: true  
 registerWithEureka: true  
 serviceUrl:  
 defaultZone: http://localhost:8701/eureka,http://localhost:8702/eureka**

### Task 2 – Eureka self-preservation mode

* bootstrap-8701.yml

**spring:  
 application:  
 name: Eureka  
  
server:  
 port: 8701  
  
eureka:  
 server:  
 enable-self-preservation: false  
 client:  
 fetchRegistry: true  
 registerWithEureka: true  
 serviceUrl:  
 defaultZone: http://localhost:8702/eureka**

* bootstrap-8702.yml

**spring:  
 application:  
 name: Eureka  
  
server:  
 port: 8702  
  
eureka:  
 server:  
 enable-self-preservation: false  
 client:  
 fetchRegistry: true  
 registerWithEureka: true  
 serviceUrl:  
 defaultZone: http://localhost:8701/eureka**

### Task 3 – Feign client with Hystrix circuit breaker

* CardServiceFallback

**import** org.springframework.stereotype.Component;

@Component

**public class CardServiceFallback implements CardServiceClient {  
 @Override  
 public String createCard() {  
 return null;  
 }  
}**

* CardServiceClient

**import** org.springframework.stereotype.Component;

**import org.springframework.cloud.openfeign.FeignClient;  
import org.springframework.web.bind.annotation.RequestMapping;  
  
@FeignClient(name = "CardService", fallback = CardServiceFallback.class)  
public interface CardServiceClient {  
 @RequestMapping("create")  
 String createCard();  
}**

* AccountServiFallback

**import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RequestParam;  
import java.math.BigDecimal;**@Component

**public class AccountServiceFallback implements AccountServiceClient {  
 @Override  
 public boolean checkout(@PathVariable("id") Integer accountId, @RequestParam("sum") BigDecimal sum) {  
 return false;  
 }  
}**

* AccountServiceClient

**import org.springframework.cloud.openfeign.FeignClient;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestParam;  
import java.math.BigDecimal;  
  
@FeignClient(name = "AccountService", fallback = AccountServiceFallback.class)  
public interface AccountServiceClient {  
 @RequestMapping("/checkout/{id}")  
 boolean checkout(@PathVariable("id") Integer accountId, @RequestParam("sum") BigDecimal sum);  
}**

## Lab 3 – API Gateway

### Task 1 – Zuul Edge Proxy

1. Add zuul starter dependency;

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-zuul</artifactId>  
</dependency>**

1. Add eureka-client dependency:

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
</dependency>**

1. Create SpringBoot main application class Zuul enabling Zuul proxy with annotation;

**import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;  
  
@SpringBootApplication  
@EnableZuulProxy  
public class Zuul {  
 public static void main(String[] args) {  
 SpringApplication.*run*(Zuul.class, args);  
 }  
}**

1. Create bootstrap.yml configuration file

**spring:  
 application:  
 name: Zuul  
  
server:  
 port: 8700  
  
eureka:  
 client:  
 fetchRegistry: true  
 registerWithEureka: false  
 serviceUrl:  
 defaultZone: http://localhost:8701/eureka,http://localhost:8702/eureka  
  
zuul:  
 ignoredServices: '\*'  
 routes:  
 clients:  
 path: /client/\*\*  
 serviceId: ClientService  
 stripPrefix: true  
 accounts:  
 path: /account/\*\*  
 serviceId: AccountService  
 stripPrefix: true  
 card-processing:  
 path: /processing/\*\*  
 serviceId: ProcessingService  
 stripPrefix: true**

## Lab 4 – Centralized Configuration

### Task 1 – Configuration Server

1. Add config-server dependency.

<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-config-server</**artifactId**>  
</**dependency**>

1. Create SpringBoot main application class ConfigServer enabling configuration server with annotations.

**import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.config.server.EnableConfigServer;  
  
@SpringBootApplication  
@EnableConfigServer  
public class ConfigServer {  
 public static void main(String[] args) {  
 SpringApplication.*run*(ConfigServer.class, args);  
 }  
}**

1. Create bootstrap.yml configuration file with spring.cloud.config.server.git.uri property set to local Git repository using file: prefix using absolute path.

**spring:  
 application:  
 name: ConfigServer  
  
server:  
 port: 8600  
  
spring.cloud.config.server.git.uri: file:/C:/ExamplePath/config-repo**

1. In Git repository files should be created using your existing configuration files

### Task 2 – Configuration Client

1. Dependency

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-config</artifactId>  
</dependency>**

1. bootstrap.yml

**spring.application.name: AppName  
spring.cloud.config.uri: http://localhost:8600**

## Lab 5 – Hystrix

### Task 1 – Hystrix Command

1. Add hystrix dependency

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>  
</dependency>**

1. Add @EnableHystrix annotation to ProcessingService configuration class

**import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  
import org.springframework.cloud.openfeign.EnableFeignClients;  
import org.springframework.cloud.netflix.hystrix.EnableHystrix;  
  
@SpringBootApplication  
@EnableEurekaClient  
@EnableFeignClients  
@EnableHystrix  
public class ProcessingService {  
 public static void main(String[] args) {  
 SpringApplication.*run*(ProcessingService.class, args);  
 }  
}**

1. Create a method with ‘/test’ request mapping

**@HystrixCommand(fallbackMethod = "testFallback")  
@RequestMapping("/test")  
public String testHystrix(Boolean fail) {  
 if (Boolean.*TRUE*.equals(fail)) {  
 throw new RuntimeException();  
 }  
 return "OK";  
}**

1. Implement the fallback method

**private String testFallback(Boolean fail) {  
 return "FAILED";  
}**

### Task 2 – Hystrix Stream

1. Add spring boot actuator dependency

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

### Task 3 – Hystrix Dashboard

1. Create the bootstrap configuration;

**spring.application.name: Dashboard  
server.port: 8900**

1. Add hystrix dashboard dependency;

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>  
</dependency>**

1. Enable hystrix dashboard adding correct annotation to Dashboard configuration class;

**import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;  
  
@SpringBootApplication  
@EnableHystrixDashboard  
public class Dashboard {  
 public static void main(String[] args) {  
 SpringApplication.*run*(Dashboard.class, args);  
 }  
}**

## Lab 6 – Turbine

### Task 1 – Turbine Stream

1. Add starter dependencies for configuration client and turbine;

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-turbine</artifactId>  
 <exclusions>  
 <exclusion>  
 <groupId>javax.servlet</groupId>  
 <artifactId>servlet-api</artifactId>  
 </exclusion>  
 </exclusions>  
</dependency>**

1. Enable turbine adding appropriate annotation to configuration class;

**import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;  
import org.springframework.cloud.netflix.turbine.EnableTurbine;  
  
@SpringBootApplication  
@EnableHystrixDashboard  
@EnableTurbine  
public class Dashboard {  
 public static void main(String[] args) {  
 SpringApplication.*run*(Dashboard.class, args);  
 }  
}**

1. Change bootstrap configuration to contain application name and configuration server URL;

**spring.application.name: Dashboard  
spring.cloud.config.uri: http://localhost:8600**

1. Create the Dashboard.yml configuration in configuration server Git repository;

**server.port: 8900  
  
eureka.client.registerWithEureka: false  
  
turbine:  
 aggregator:  
 clusterConfig: PROCESSINGSERVICE,ZUUL  
 appConfig: ProcessingService,zuul**

### Task 2 – Processing Service 8805

* ProcessingService.yml

**spring.datasource:  
 url: jdbc:h2:file:~/ProcessingService;DB\_CLOSE\_ON\_EXIT=FALSE;AUTO\_SERVER=TRUE  
  
server:  
 port: 8804**

* ProcessingService-8805.yml

**spring.datasource:  
 url: jdbc:h2:file:~/ProcessingService;DB\_CLOSE\_ON\_EXIT=FALSE;AUTO\_SERVER=TRUE  
  
server:  
 port: 8805**

## Lab 7 – Turbine AMQP

### Task 2 – Hystrix Stream

1. Add spring cloud hystrix stream dependency

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-netflix-hystrix-stream</artifactId>  
</dependency>**

1. Add spring cloud rabbitmq stream starter dependency

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-stream-rabbit</artifactId>  
</dependency>**

### Task 3 – Turbine Stream

1. Add spring cloud turbine stream starter dependency;

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-turbine-stream</artifactId>  
</dependency>**

1. Add spring cloud rabbitmq stream starter dependency;

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-stream-rabbit</artifactId>  
</dependency>**

1. bootstrap.yml

**spring.application.name: TurbineServer  
server.port: 8901**

1. Implement main class enabling the Turbine Stream with corresponding annotation

**import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.turbine.stream.EnableTurbineStream;  
  
@SpringBootApplication  
@EnableTurbineStream  
public class TurbineServer {  
 public static void main(String[] args) {  
 SpringApplication.*run*(TurbineServer.class, args);  
 }  
}**

## Lab 8 – Distributed Tracing with Zipkin

### Task 1 – Instrumenting Applications

1. Add dependencies for sleuth-stream and RabbitMQ stream starter to projects.

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-zipkin</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.amqp</groupId>  
 <artifactId>spring-rabbit</artifactId>  
</dependency>**

### Task 2 – Zipkin Server

1. Download zipkin server

curl -sSL https://zipkin.io/quickstart.sh | bash -s

1. Run zipkin with

java –jar zipkin.jar

## Lab 9 –Microservices Security

### Task 1 – User Authentication & Authorization (UAA) Server

1. Generate a new key store using command line:
   * keytool -genkeypair -alias jwt -keyalg RSA -keypass mySecretKey -keystore jwt.jks -storepass mySecretKey
2. Put the generated jwt.jks key store into the /main/resource folder
3. Add dependencies

**<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-oauth2</artifactId>  
</dependency>  
<dependency>**

1. Check the application configuration (application.yml) to have

**spring.application.name: AuthService  
server.port: 8500  
server.servlet.context-path: /uaa**

1. Implement the OAuth2Configuration class

**import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.core.io.ClassPathResource;  
import org.springframework.security.authentication.AuthenticationManager;  
import org.springframework.security.oauth2.config.annotation.configurers.ClientDetailsServiceConfigurer;  
import org.springframework.security.oauth2.config.annotation.web.configuration.AuthorizationServerConfigurerAdapter;  
import org.springframework.security.oauth2.config.annotation.web.configuration.EnableAuthorizationServer;  
import org.springframework.security.oauth2.config.annotation.web.configurers.AuthorizationServerEndpointsConfigurer;  
import org.springframework.security.oauth2.config.annotation.web.configurers.AuthorizationServerSecurityConfigurer;  
import org.springframework.security.oauth2.provider.token.TokenStore;  
import org.springframework.security.oauth2.provider.token.store.JwtAccessTokenConverter;  
import org.springframework.security.oauth2.provider.token.store.JwtTokenStore;  
import org.springframework.security.oauth2.provider.token.store.KeyStoreKeyFactory;  
  
@Configuration  
@EnableAuthorizationServer  
public class OAuth2Configuration extends AuthorizationServerConfigurerAdapter {  
 @Autowired  
 @Qualifier("authenticationManagerBean")  
 private AuthenticationManager authenticationManager;  
  
 @Bean  
 public TokenStore tokenStore() {  
 return new JwtTokenStore(jwtTokenEnhancer());  
 }  
  
 @Bean  
 protected JwtAccessTokenConverter jwtTokenEnhancer() {  
 KeyStoreKeyFactory keyStoreKeyFactory = new KeyStoreKeyFactory(new ClassPathResource("jwt.jks"), "mySecretKey".toCharArray());  
 JwtAccessTokenConverter converter = new JwtAccessTokenConverter();  
 converter.setKeyPair(keyStoreKeyFactory.getKeyPair("jwt"));  
 return converter;  
 }  
  
 @Override  
 public void configure(ClientDetailsServiceConfigurer clients) throws Exception {  
 clients.inMemory()  
 .withClient("client")  
 .secret("{noop}secret")  
 .scopes("openid")  
 .autoApprove(true)  
 .authorizedGrantTypes("implicit","refresh\_token",**

**"password", "authorization\_code");  
 }  
  
 @Override  
 public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {  
 endpoints**

**.tokenStore(tokenStore())**

**.tokenEnhancer(jwtTokenEnhancer())**

**.authenticationManager(authenticationManager);  
 }  
  
 @Override  
 public void configure(AuthorizationServerSecurityConfigurer oauthServer) throws Exception {  
 oauthServer**

**.tokenKeyAccess("permitAll()")**

**.checkTokenAccess("permitAll()");  
 }  
}**

1. Implement the WebSecurityConfig class

**import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.security.authentication.AuthenticationManager;  
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;  
import org.springframework.security.config.annotation.web.builders.HttpSecurity;  
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  
  
@Configuration  
class WebSecurityConfig extends WebSecurityConfigurerAdapter {  
 @Override  
 @Bean  
 public AuthenticationManager authenticationManagerBean() throws Exception {  
 return super.authenticationManagerBean();  
 }  
  
 @Override  
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.inMemoryAuthentication()  
 .withUser("account")  
 .password("{noop}account")  
 .authorities("ACCOUNT\_READ",**

**"ACCOUNT\_WRITE", "ACCOUNT\_PROCESS")  
 .and()  
 .withUser("card")  
 .password("{noop}card")  
 .authorities("CARD\_WRITE", "ACCOUNT\_READ")  
 .and()  
 .withUser("client")  
 .password("{noop}client")  
 .authorities("CLIENT\_READ",**

**"CLIENT\_WRITE", "ACCOUNT\_READ", "CARD\_READ")  
 .and()  
 .withUser("processing")  
 .password("{noop}processing")  
 .authorities("PROCESSING", "ACCOUNT\_PROCESS");  
 }  
}**

### Task 2 – Microservices Security

1. Add dependencies

**<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-oauth2</artifactId>  
</dependency>**

1. Add the resource server configuration class from slides to account, card, client and processing services.

**import org.springframework.context.annotation.Configuration;  
import org.springframework.security.config.annotation.web.builders.HttpSecurity;  
import org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceServer;  
import org.springframework.security.oauth2.config.annotation.web.configuration.ResourceServerConfigurerAdapter;  
  
@Configuration  
@EnableResourceServer  
public class ResourceServerConfig extends ResourceServerConfigurerAdapter {  
 @Override  
 public void configure(HttpSecurity http) throws Exception {  
 http  
 .csrf().disable()  
 .authorizeRequests()  
 .antMatchers("/\*\*").authenticated();  
 }  
}**

1. Add oauth2 security configuration to application.yml (in config-repo).

**security:  
 oauth2:  
 client:  
 clientId: client  
 clientSecret: secret  
 scope: openid  
 accessTokenUri: http://localhost:8500/uaa/oauth/token  
 userAuthorizationUri: http://localhost:8500/uaa/oauth/authorize  
 resource:  
 jwt:  
 key-uri: http://localhost:8500/uaa/oauth/token\_key**

1. Add annotation to configuration class

**import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  
import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;  
  
@SpringBootApplication  
@EnableEurekaClient  
@EnableGlobalMethodSecurity(prePostEnabled = true)  
public class ClientService {  
 public static void main(String[] args) {  
 SpringApplication.*run*(ClientService.class, args);  
 }  
}**

1. Add method or class security to RestController classes using

**import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.security.access.prepost.PreAuthorize;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class CardRest {  
 @Autowired  
 private CardNumberGenerator generator;  
  
 @PreAuthorize("hasAuthority('CARD\_WRITE')")  
 @RequestMapping("create")  
 public String createNewCard() {  
 return generator.generate();  
 }  
}**

In resource server configuration of ProcessingService additionally:

1. Change configuration of **AccountServiceClient** by adding configuration:

**@FeignClient(value = "AccountService",  
 fallback = AccountServiceFallback.class,  
 configuration = AccountServiceClientConfiguration.class)**

1. Add class **AccountServiceClientConfiguration:**

**public class AccountServiceClientConfiguration {  
 @Value("${security.oauth2.client.accessTokenUri}")  
 private String accessTokenUri;  
  
 @Bean  
 public RequestInterceptor requestInterceptor() {  
 return new OAuth2FeignRequestInterceptor(  
 new DefaultOAuth2ClientContext(),  
 resource());  
 }  
  
 private OAuth2ProtectedResourceDetails resource() {  
 ResourceOwnerPasswordResourceDetails resource =**

**new ResourceOwnerPasswordResourceDetails();  
 resource.setAccessTokenUri(accessTokenUri);  
 resource.setClientId("client");  
 resource.setClientSecret("secret");  
 resource.setUsername("account");  
 resource.setPassword("account");  
 return resource;  
 }  
}**

1. Do the same for **CardServiceClient** (change username/password to card/card for class **CardServiceClientConfiguration**).

### Task 3 – Authorization behind Zuul

1. Add configuration

zuul.sensitiveHeaders: Cookie,Set-Cookie

This way we will exclude Authorization from ensitive headers (which are not forwarded to service behind Zuul) and allow to pass authorization header to the service.

1. Use this code (put to zuul-auth.rest) to test authorization

***### Get token for account*POST http://localhost:8500/uaa/oauth/token?grant\_type=password&username=account&password=account  
*Authorization*: Basic client secret  
  
> {%  
 client.global.set("account\_token", response.body.access\_token);  
 %}  
  
  
*### Create account for the client*GET http://localhost:8700/account/create?client\_id=7  
*Authorization*: Bearer {{account\_token}}**

## Lab 10 – Cloud Bus

### Task 1 – Push Configurations

* Configuration for Spring Config server:

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-bus-amqp</artifactId>  
</dependency>**

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-config-monitor</artifactId>  
</dependency>**

* Configuration for Spring Config clients:

**<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-bus-amqp</artifactId>  
</dependency>**

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

Add rest service with configurable parameter (for example to ProcessingService):

**@RefreshScope**

**@RestController  
public class ConfigurableRest {**

**@Value("${tparam}")  
String param;  
  
@RequestMapping("param")  
public String param() {  
 return param;  
}**

**}**

Set its initial value in configuration (ProcessingService.yml):

**tparam: 1**

Try to change the value and see logs, also check if endpoint result is updated.

Now lets perform change of the parameter with /actuator/bus-env.

For this, set value to application.yml to allow all management endpoints:

**management.endpoints.web.exposure.include: '\*'**

Also remove **management.endpoints.web.exposure.include** configuration from ProcessingService.yml (it had value hystrix.stream) to prevent overriding of the property.

Then perfrorm requests (put in **refresh.rest)**:

**POST localhost:8804/actuator/bus-env  
*Authorization*: Bearer {{processing\_token}}  
*Content-Type*: application/json  
  
{"name":"tparam", "value":"777"}**

To refresh values for all instances of the service, use this endpoint:

**POST localhost:8804/actuator/bus-refresh**

(Single instance can be refreshed with **POST localhost:8804/actuator/refresh**)

Then check both endpoints

**GET localhost:8804/param**

**GET localhost:8805/param**

### Task 2 – History Service

* AbstractFinancialEvent.java

**import org.springframework.cloud.bus.event.RemoteApplicationEvent;  
import java.math.BigDecimal;  
  
public abstract class AbstractFinancialEvent extends RemoteApplicationEvent {  
 private static Object *source* = new Object();  
 private BigDecimal sum;  
  
 public AbstractFinancialEvent() {  
 super();  
 }  
  
 public AbstractFinancialEvent(String originService, String destinationService, BigDecimal sum) {  
 super(*source*, originService, destinationService);  
 this.sum = sum;  
 }  
  
 public BigDecimal getSum() {  
 return sum;  
 }  
}**

* RemoteEventPublisher.java

**import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.cloud.bus.SpringCloudBusClient;  
import org.springframework.cloud.bus.event.RemoteApplicationEvent;  
import org.springframework.messaging.support.MessageBuilder;  
import org.springframework.stereotype.Component;  
  
@Component  
public class RemoteEventPublisher {  
 @Autowired  
 private SpringCloudBusClient busClient;  
  
 public void publishEvent(RemoteApplicationEvent event) {  
 busClient.springCloudBusOutput().send(**

**MessageBuilder.*withPayload*(event).build());  
 }  
}**

* AccountRest.java

**@RequestMapping("/fund/{id}")  
public boolean fund(@PathVariable Integer id, @RequestParam BigDecimal sum) {  
 try {  
 return dao.addBalance(id, sum.abs());  
 } finally {  
 eventPublisher.publishEvent(new FundEvent("AccountService", "HistoryService", sum));  
 }  
}  
  
@RequestMapping("/checkout/{id}")  
public boolean checkout(@PathVariable Integer id, @RequestParam BigDecimal sum) {  
 BigDecimal negativeSum = sum.abs().negate();  
 try {  
 return dao.addBalance(id, negativeSum);  
 } finally {  
 eventPublisher.publishEvent(new WithdrawEvent("AccountService", "HistoryService", negativeSum));  
 }  
}**

* HistoryService.java

**import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.bus.jackson.RemoteApplicationEventScan;  
import org.springframework.context.ApplicationListener;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.math.BigDecimal;  
import java.util.Collections;  
import java.util.Date;  
import java.util.LinkedHashMap;  
import java.util.Map;  
  
@SpringBootApplication  
@RemoteApplicationEventScan  
@RestController  
public class HistoryService implements ApplicationListener<AbstractFinancialEvent> {  
 private Map<Date, BigDecimal> historyMap = Collections.*synchronizedMap*(new LinkedHashMap<Date, BigDecimal>());  
  
 @GetMapping  
 public Map<Date, BigDecimal> getHistory() {  
 return historyMap;  
 }  
  
 @Override  
 public void onApplicationEvent(AbstractFinancialEvent event) {  
 historyMap.put(new Date(event.getTimestamp()), event.getSum());  
 }  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(HistoryService.class, args);  
 }  
}**

## Lab 11 – Cloud Streams

### Task 1 – Publisher

* Dependencies

**<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-stream-rabbit</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-config</artifactId>  
</dependency>**

* TestChannel.java

**public interface TestChannel {  
 String *CHANNEL\_NAME* = "testChannel";  
  
 @Output(*CHANNEL\_NAME*)  
 MessageChannel testChannel ();  
}**

* Publisher.java

**@SpringBootApplication  
@EnableBinding(TestChannel.class)  
@RestController  
public class Publisher {  
 @Autowired  
 @Qualifier(TestChannel.CHANNEL\_NAME)  
 public MessageChannel output;  
  
 @GetMapping("/publish")  
 private void publish(@RequestParam String word) {  
 output.send(MessageBuilder.*withPayload*(word).build());  
 }  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(Publisher.class, args);  
 }  
}**

### Task 2 – Subscriber

* TestChannel.java

**public interface TestChannel {  
 String *CHANNEL\_NAME* = "testChannel";  
  
 @Input(*CHANNEL\_NAME*)  
 SubscribableChannel input();  
}**

* Subscriber.java

**@SpringBootApplication  
@EnableBinding(TestChannel.class)  
@RestController  
public class Subscriber {  
 private Map<String, AtomicInteger> wordMap = new ConcurrentHashMap<>();  
  
 @GetMapping  
 public Map<String, AtomicInteger> getWords() {  
 return wordMap;  
 }  
  
 @StreamListener(TestChannel.*CHANNEL\_NAME*)  
 public void countWords(String s) {  
 if (wordMap.containsKey(s)) {  
 wordMap.get(s).incrementAndGet();  
 } else {  
 wordMap.put(s, new AtomicInteger(1));  
 }  
 }  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(Subscriber.class, args);  
 }  
}**

## Lab 12 – Advanced Streams

### Task 1 – Processor

* PublisherChannel.java

**public interface PublisherChannel {  
 String *WORDS* = "words";  
  
 @Output(*WORDS*)  
 MessageChannel words();  
}**

* ProcessorChannel.java

**public interface ProcessorChannel {  
 String *WORDS* = "words";  
 String *UPPER\_WORDS* = "upperWords";  
  
 @Input(*WORDS*)  
 SubscribableChannel words();  
  
 @Output(*UPPER\_WORDS*)  
 MessageChannel upperWords();  
}**

* Processor.java

**@SpringBootApplication  
@EnableBinding(ProcessorChannel.class)  
public class Processor {  
 @StreamListener(ProcessorChannel.*WORDS*)  
 @SendTo(ProcessorChannel.*UPPER\_WORDS*)  
 private String process(String word) {  
 return word.toUpperCase();  
 }  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(Processor.class, args);  
 }  
}**

* SubscriberChannel.java

**public interface SubscriberChannel {  
 String *UPPER\_WORDS* = "upperWords";  
  
  
 @Input(*UPPER\_WORDS*)  
 SubscribableChannel upperWords();  
}**

### Task 2 – Consumer Groups

* Processor.yml

**spring:  
 application:  
 name: Processor  
  
server:  
 port: 8003  
  
spring.cloud.stream.bindings.words:  
 group: Processors  
 consumer:  
 instanceCount: 2  
 instanceIndex: 0**

* Processor-8004.yml

**server:  
 port: 8004  
  
spring.cloud.stream.bindings.words:  
 consumer:  
 instanceIndex: 1**

### Task 3 – Partitioning

* WordNumSplitter

**@SpringBootApplication  
@EnableBinding(Processor.class)  
public class WordNumSplitter {  
 @StreamListener(Processor.*INPUT*)  
 @SendTo(Processor.*OUTPUT*)  
 private String split(String word) {  
 try {  
 BigDecimal num = new BigDecimal(word);  
 System.*out*.println(word + " - is a number");  
 return null;  
 } catch (Exception ex) {  
 System.*out*.println(word + " - sent");  
 return word;  
 }  
 }  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(WordNumSplitter.class, args);  
 }  
}**

* WordNumProcessor.yml

**spring:  
 application:  
 name: WordNumSplitter  
  
server:  
 port: 8005  
  
spring.cloud.stream.bindings.input:  
 destination: wordNumSplit  
 group: Splitters  
 consumer:  
 partitioned: true  
 instanceCount: 2  
 instanceIndex: 0  
  
spring.cloud.stream.bindings.output.destination: words**

* WordNumProcessor-8006.yml

**server:  
 port: 8006  
  
spring.cloud.stream.bindings.input:  
 consumer:  
 instanceIndex: 1**

* Publisher.yml

**spring:  
 application:  
 name: Publisher  
  
server:  
 port: 8001  
  
spring.cloud.stream.bindings.words:  
 destination: wordNumSplit  
 producer:  
 partitionKeyExpression: payload.startsWith("h")?0:1  
 partitionCount: 2**