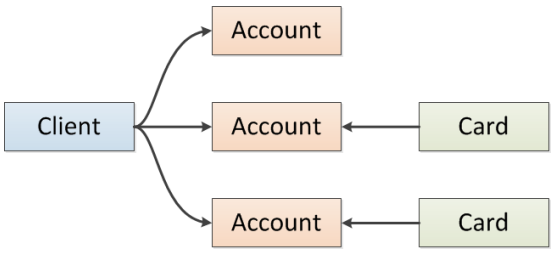
Introduction to Spring Cloud

Lab Guide & Solutions

Contents

Lab Guide

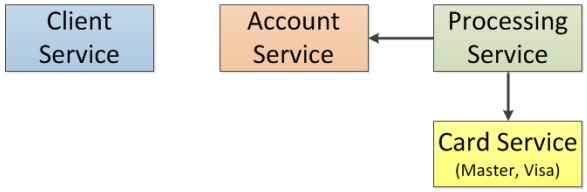
We are going to develop banking software with microservices approach. Out implementation will be oversimplified and is intended to demonstrate building blocks to build applications with microservices architecture. Out implementation covers a few basic aspects of banking domain. The story is about bank 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.
    a. Mark `ClientRest` class with `@RestController` annotation;
    b. Inject (autowire) `ClientDao` and `ClientRepository` instances;
    c. Implement "/create" endpoint with request parameter "name" `@RequestMapping` and `@RequestParam` annotations. To implement method logic use helper method `ClientDao::create`;
    d. 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;
    e. Implement "/delete" endpoint with path variable "id" calling `ClientRepository::delete`;
    f. Implement "/get" endpoint using `ClientRepository::findAll`;
    g. 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.
    a. Add `@RestController` annotation to class;
    b. Inject instances of `AccountDAO` and `AccountRepository` classes;
    c. Implement request mapping "/create" using `AccountDAO::create`;
    d. 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()`);
    e. 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()`);
    f. 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.
    a. Mark `ClientRest` class with `@RestController` annotation;
    b. Inject `CardNumberGenerator` instance;
    c. 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.
    a. Mark class with `@RestController` annotation;
    b. Inject instances of `ProcessingRepository`, `AccountServiceClient` and `CardServiceClient` interfaces;
    c. Implement "/issue/{accountId}" request mapping with following algorithm:
        i. Get generated card number using `CardServiceClient`;
        ii. Create a `ProcessingEntity` object with generated card number and provided account id;
        iii. Save `ProcessingEntity` using `ProcessingRepository::save`;
    d. Implement "/checkout/{card}" request mapping with Boolean return value having additional request parameter "sum" with following algorithm:
        i. Search for entities of type `ProcessingEntity` using helper repository method `ProcessingRepository::findByCard`;
        ii. Return `false` if entity was not found;
        iii. If entity found, return value of `AccountServiceClient::checkout`;
    e. 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:
        i. Read list of processing entities from database using helper repository method `ProcessingRepository::findByAccountIdIn`;
        ii. 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";
    a. configure each bootstrap file to listen to port 870* from file name;

    b. 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;
    a. 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;
    a. ClientService
    b. AccountService
    c. CardService
    d. ProcessingService
4. Build and run discovery servers and microservices;
5. Open both Eureka consoles at http://localhost:8701 and http://localhost:8702
    a. 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. after microservices are registered, stop any of them;
4. wait up to 4 minutes and check if dead microservices were unregistered by Eureka;
    a. this can be observed in logs or web-console

Task 3 – Feign client with Hystrix circuit breaker

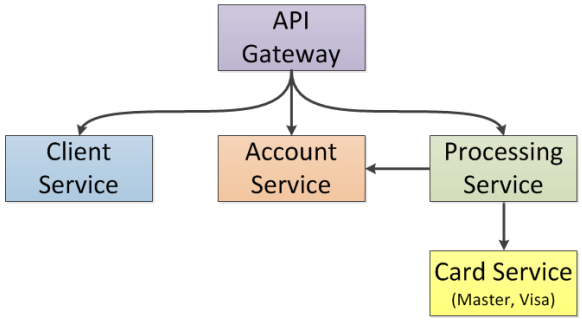
Now we are going to register fallback methods for cases when services are not available or it will take more time than required. Feign is already integrated with Hystrix circuit breaker. Default Hystrix timeout for Feign calls is 1 second. We should be able to returned predefined values in case of failed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

1. to implement `CardServiceClient` circuit breaker:
    a. create class `CardServiceFallback` implementing `CardServiceClient` interface;
    b. mark the class with `@Component` annotation;
    c. in the `createCard` method return null;
    d. in `CardServiceClient` interface add `CardServiceFallback` class as `fallback` parameter of `@FeignClient` annotation
    e. in implementation of `ProcessingRest` controller if `CardServiceClient` returns null, return "CARD_SERVICE_NOT_AVAILABLE" message;

2. to implement `AccountServiceClient` circuit breaker:
    a. create `AccountServiceFallback` class implementing existing interface `AccountServiceClient`;
    b. mark the class with `@Component` annotation;
    c. in `checkout` method return false;
    d. in `AccountServiceClient` interface add `AccountServiceFallback` class as `fallback` parameter of `@FeignClient` annotation
3. Redeploy changes and run tests with all services up;
4. Kill card service and test how circuit breaker works;
5. 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;
2. Implement SpringBoot main application class `Zuul` enabling zuul-proxy with according annotation;
3. Finalize bootstrap.yml configuration file:
    a. fetch registry for Zuul but do not register with Eureka as it is not a microservice;
    b. ignore all services by default;
    c. register custom routes:
        i. ClientService: /client;
        ii. AccountService: /account;
        iii. 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.
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
    a. create application.yml file with Eureka client configuration common to all microservices;
    b. create named application YAML configurations for all other applications
        i. 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);
        ii. for Zuul override only `eureka.client.registerWithEureka` property;
6. Build and run configuration server
7. Test using http://localhost:8600/${AppName}/${profile}. For instance:
    a. http://localhost:8600/CardService/default
    b. 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
2. Add @EnableHystrix annotation to ProcessingService configuration class
3. Create a method with '/test' request mapping
    a. Accept Boolean 'fail' parameter;

   b. If TRUE, throw a `RuntimeException`;
   c. If FALSE return "OK";
   d. Annotate it with `@HystrixCommand` and provide a name for fallback method
4. Implement the fallback method
   a. 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;
2. Test if "/hystrix.stream" is available;
   a. Note, hystrix stream may not produce any output until the first hystrix command execution;
   b. If a hystrix command is not executed, you may not see it appear in the hystrix stream;
3. To see an ouput in hystrix stream, run "/test" from previous task;
   a. 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;
   a. Set application name 'Dashboard';
   b. 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.
   a. Point it to hystrix stream of ProcessingService at http://localhost:8804/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;
    a. Add port 8900;
    b. Turn off registration at Eureka as we are only a client, not a service;
    c. 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";
    a. 8805 is the new profile name;
2. Create a new Run configuration in your IDE for ProcessingService
    a. You can copy existing configuration with different name, e.g. "ProcessingService-8805"
    b. 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
    a. For instance, run /test
4. Watch changes in turbine stream

a. 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
    a. Download it from http://www.erlang.org/downloads
2. Install RabbitMQ
    a. Download it from https://www.rabbitmq.com/download.html
3. Enable RabbitMQ management plugin
    a. Run "*rabbitmq-plugins enable rabbitmq_management*" from command line
    b. "*rabbitmq-plugins*" is located in "*sbin*" direcroty under RabbitMQ installation folder
4. Open http://localhost:15672/ to see RabbitMQ management web-page in action
    a. 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;
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
    a. Set application name “TurbineServer”
    b. Set port 8901
4. Implement main class enabling the Turbine Stream with corresponding annotation;
5. Run the server and point hystrix dashboard to http://localhost:8901

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 sleuth-stream and RabbitMQ stream starter to projects.
2. Run the projects.

For configuration server it may be necessary to add the **`org.springframework.boot:spring-boot-starter-aop`** dependency. Add it if you will get an error concerning **`NoClassDefFoundError: ReflectionWorld$ReflectionWorldException`** at startup.

Task 2 – Zipkin Server

Now, we should implement the Zipkin server to access the Zipkin UI and watch traces. Go to ZipkinServer directory in your labs folder please and proceed with following steps:

1. Add dependencies for Sleuth Zipkin stream and RabbitMQ stream starter.
2. Add dependency to **`io.zipkin.java:zipkin-autoconfigure-ui`** for Zipkin UI.
3. Implement main class declaring corresponding annotation to enable Zipkin stream server.
4. Make sure your application configuration contains application name ZipkinServer and port 8902.
5. Run application and navigate to http://localhost:8902
    a. Run several methods at ProcessingService and other services
    b. Examine Zipkin UI for trace information

Note: Zipkin UI may show some unnecessary trace information related to hystrix streams and other annoying entries. Usually, it’s the DevOps team who configures all of this and not developers. Your task as a developer is to instrument the application and know the technology. The DevOps are the team who analyze available metrics to examine what is happening in production.

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.security.oauth:spring-security-oauth2`**
    - **`org.springframework.security:spring-security-jwt`**
    - **`org.springframework.boot:spring-boot-starter-web`**
4. Check the application configuration (application.yml) to have
    - AuthService as application name
    - Port 8500
    - 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 Security OAuth2
    - Spring Security JWT

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:

6. Declare default load balanced (@Primary) OAuth2RestTemplate
    - Name it "OAuth2RestTemplate"
7. 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
8. 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 – SSO with Zuul

It is time to enable Single Sign On (SSO) feature throw out API gateway. Please take the following steps, to switch on the SSO at Zuul:

1. Add dependencies for
    - Spring Cloud Security
    - Spring Security OAuth2
    - Spring Security JWT
2. Add @EnableOAuth2Sso annotation to configuration class
3. Configure 30s timeout for hystrix
    - default 1s will not be enough for whole authorization flow).

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 dependency 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).

Now start the services and test the following:

- try to change configuration files, watching logs and checking if application context reloads;
- trigger /bus/refresh endpoint manually using destination filters;
    - Test if all application reload the context;
    - Test if an exact service reloads;
    - Test if a single port reloads separately, e.g. Processing Service 8804 (or 8805);
- trigger /monitor manually using path filter to trigger manual change notifications;

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:
    a. Add cloud-bus dependency;
    b. Implement AbstractFinancialEvent class extending RemoteApplicationEvent;
        i. With default constructor;
        ii. With constructor that accept arguments:
            - String originService;
            - String destinationService;
            - BigDecimal sum;

iii. With getSum() method that returns the sum;

c. Implement FundEvent and WithdrawEvent classes extending AbstractFinancialEvent;
   i. With both constructors declared;

d. Implement RemoteEventPublisher class;
   i. With @Component annotation;
   ii. With SpringCloudBusClient auto-wired;
   iii. With publishEvent(RemoteApplicationEvent event) method using:
      - **`busClient.springCloudBusOutput().send(MessageBuilder.withPayload(event).build());`**

2. In Account Service project:
   a. Add dependency to Common project;
   b. Add @RemoteApplicationEventScan to configuration class;
   c. Autowire RemoteEventPublisher in REST controller and publish corresponding events to bus in “checkout” and “fund” methods;
3. In History Service project:
   a. Add dependency to “Common” project;
   b. Add starter bus-amqp dependency;
   c. Add starter config dependency;
   d. Check if bootstrap.yml contains correct application name and config server URI;
   e. Implement HistoryService class as follows:
      i. Implement ApplicationListener<AbstractFinancialEvent> or use @EventListener whichever you like most;
      ii. with Get mapping returning Map<Date, BigDecimal>;
      iii. with singleton synchronized LinkedHashMap of corresponding type;
      iv. with override to onApplicationEvent(AbstractFinancialEvent event) method;
         - use date value of event timestamp as key;
         - use event sum as value;
      v. Check the class to contain SpringBootApplication, RemoteApplicationEventScan and RestController annotations, as well as main method;
4. 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;
    a. Use @Output annotation;
    b. Use "testChannel" as channel name;
    c. 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
    a. Receive "word" parameter as String;
    b. Auto-wire the output channel;
    c. 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;
    a. Use @Input annotation;
    b. Use "testChannel" as channel name;
    c. 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;
    a. Use "words" channel for input;
    b. Use "upperWords" channel name for output channel;
4. Implement the Processor Spring Boot application main class;
    a. Create a process(String word) method that returns word.toUpperCase();
    b. Annotation the process() method with @StreamListener and @SendTo annotations;
    c. 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:
    a. Declare the group name “Processors”;
    b. Set consumer instance count 2;
    c. Set consumer instance index:
        i. 0 for 8003;
        ii. 1 for 8004;
3. Run the applications
    a. 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:
    a. Bind the “input” channel destination to “wordNumSplit”;
        i. With group “Splitters”;
        ii. With partitioned consumer;
        iii. With instance count 2;
        iv. With instance index 0;
    b. Bind the “output” channel destination to “words”;
    c. 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;
    a. Lister to Processor.INPUT;
    b. Send to Processor.OUTPUT;
5. Configure the Publisher as follows:

    a. Bind the “words” channel destination to “wordNumSplit”;
    b. With producer partition key “payload”
    c. With partition count 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.

Task 4 – Stream Aggregator

Sometimes it is necessary to unload the broker. For that purpose, you can use Aggregator applications that will aggregate several stream applications into one application. Aggregated components will not send/receive messages to/from broker, but instead they will form a pipeline.

In our case, we do not have enough processors, but we can use our single processor to have an example aggregator. Using Aggregator application from Labs directory,

1. Add dependency to Processor and WordNumSplitter;
2. Create the main method with AggregateApplicationBuilder using WordNumSplitter in from() and Processor in via() blocks;
3. Add the channel configuration from both application to aggregator bootstrap config and bind it to port 8007;
    a. You should add the channel destinations and groups configurations from WordNumSplitter and Processor configurations;
4. Run the aggregator and test if it works correctly;

It will actually do same job that WordNumSplitter and Processor did. This time it will listen to both partitions of “wordNumSplit”.

If you finished your lab earlier than others, you may try to partition the aggregator as two separate applications listening to each partition separately.

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>1.5.2.RELEASE</version>
</parent>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Camden.SR6</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

1. Add eureka and eureka-server starter dependencies

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

2. 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);
    }
}
```

3. 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);
    }
}
```

2. Create bootstrap.yml configuration file

```
spring:
  application:
    name: ClientService

server:
  port: 8801

eureka:
  client:
    fetchRegistry: true
    registerWithEureka: true
    serviceUrl:
      defaultZone: http://localhost:8761/eureka
```

3. 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.delete(id);
    }

    @RequestMapping("/get")
    public List<? extends Client> get() {
        return repo.findAll();
    }

    @RequestMapping("/get/{id}")
    public Client get(@PathVariable Integer id) {
        return repo.findOne(id);
    }
}
```

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);
    }
}
```

2. Create bootstrap.yml configuration file.

```
spring:
  application:
    name: AccountService

server:
  port: 8802

eureka:
  client:
    fetchRegistry: true
    registerWithEureka: true
    serviceUrl:
      defaultZone: http://localhost:8761/eureka
```

3. 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);
    }
}
```

2. Create bootstrap.yml configuration file

```
spring:
  application:
    name: CardService

server:
  port: 8803

eureka:
  client:
    fetchRegistry: true
    registerWithEureka: true
    serviceUrl:
      defaultZone: http://localhost:8761/eureka
```

3. 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-feign</artifactId>
</dependency>
```

2. 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.netflix.feign.EnableFeignClients;

@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients
public class ProcessingService {
    public static void main(String[] args) {
        SpringApplication.run(ProcessingService.class, args);
    }
}
```

3. Create bootstrap.yml configuration file.

```
spring:
  application:
    name: ProcessingService

server:
  port: 8804

eureka:
  client:
    fetchRegistry: true
    registerWithEureka: true
    serviceUrl:
      defaultZone: http://localhost:8761/eureka
```

4. Create AccountServiceClient feign client with a method mapping for "/checkout/{id}"

```
import org.springframework.cloud.netflix.feign.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);
}
```

5. Create CardServiceClient feign client with a method mapping for "/create"

```
import org.springframework.cloud.netflix.feign.FeignClient;
import org.springframework.web.bind.annotation.RequestMapping;

@FeignClient("CardService")
public interface CardServiceClient {
    @RequestMapping("create")
    String createCard();
}
```

6. 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.netflix.feign.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.netflix.feign.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-zuul</artifactId>
</dependency>
```

2. 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);
    }
}
```

3. 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>
```

2. 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);
    }
}
```

3. 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
```

4. 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>
```

2. 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-hystrix</artifactId>
</dependency>
```

2. 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.netflix.feign.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);
    }
}
```

3. 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";
}
```

4. 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
```

2. Add hystrix dashboard dependency;

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

3. 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-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-turbine</artifactId>
    <exclusions>
        <exclusion>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

2. 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);
    }
}
```

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

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

4. 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>
```

2. 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-turbine-stream</artifactId>
</dependency>
```

2. Add spring cloud rabbitmq stream starter dependency;

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

3. bootstrap.yml

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

4. 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-sleuth-stream</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

- For configuration server if required.

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

Task 2 – Zipkin Server

1. Add dependencies for Sleuth Zipkin stream and RabbitMQ stream starter.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin-stream</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

2. Add dependency to **`io.zipkin.java:zipkin-autoconfigure-ui`** for Zipkin UI.

```
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>
```

3. Implement main class declaring corresponding annotation to enable Zipkin stream server.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.sleuth.zipkin.stream.EnableZipkinStreamServer;

@SpringBootApplication
@EnableZipkinStreamServer
public class ZipkinServer {
    public static void main(String[] args) {
        SpringApplication.run(ZipkinServer.class, args);
    }
}
```

4. Make sure your application configuration contains application name ZipkinServer and port 8902.

```
spring.application.name: ZipkinServer
server.port: 8902
```

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.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-jwt</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

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

```
spring.application.name: AuthService
server.port: 8500
server.contextPath: /uaa
```

5. 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.ClientDetails
ServiceConfigurer;
import
org.springframework.security.oauth2.config.annotation.web.configuration.Authori
zationServerConfigurerAdapter;
import
org.springframework.security.oauth2.config.annotation.web.configuration.EnableA
uthorizationServer;
import
org.springframework.security.oauth2.config.annotation.web.configurers.Authoriza
tionServerEndpointsConfigurer;
import
org.springframework.security.oauth2.config.annotation.web.configurers.Authoriza
tionServerSecurityConfigurer;
import org.springframework.security.oauth2.provider.token.TokenStore;
import
org.springframework.security.oauth2.provider.token.store.JwtAccessTokenConverte
r;
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("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()).authentica
tionManager(authenticationManager);
    }

    @Override
    public void configure(AuthorizationServerSecurityConfigurer oauthServer)
throws Exception {

oauthServer.tokenKeyAccess("permitAll()").checkTokenAccess("permitAll()");
    }
}
```

6. 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.Authenti
cationManagerBuilder;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityCon
figurerAdapter;

@Configuration
class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    @Bean
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .csrf().disable()
            .authorizeRequests()
                .antMatchers("/**").authenticated()
            .and().httpBasic()
            .and().formLogin().permitAll()
            .and().logout();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
        auth.inMemoryAuthentication()
                .withUser("account")
                .password("account")
                .authorities("ACCOUNT_READ", "ACCOUNT_WRITE",
"ACCOUNT_PROCESS")
                .and()
                .withUser("card")
                .password("card")
                .authorities("CARD_WRITE", "ACCOUNT_READ")
                .and()
                .withUser("client")
                .password("client")
                .authorities("CLIENT_READ", "CLIENT_WRITE", "ACCOUNT_READ",
"CARD_READ")
                .and()
```

```
                .withUser("processing")
                .password("processing")
                .authorities("PROCESSING", "ACCOUNT_PROCESS");
    }
}
```

Task 2 – Microservices Security

1. Add dependencies

```
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-jwt</artifactId>
</dependency>
```

2. 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.EnableR
esourceServer;
import
org.springframework.security.oauth2.config.annotation.web.configuration.Resourc
eServerConfigurerAdapter;

@Configuration
@EnableResourceServer
public class ResourceServerConfig extends ResourceServerConfigurerAdapter {
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
            .csrf().disable()
            .authorizeRequests()
            .antMatchers("/**").authenticated();
    }
}
```

3. 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
```

4. 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.EnableGloba
lMethodSecurity;
```

```
@SpringBootApplication
@EnableEurekaClient
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ClientService {
    public static void main(String[] args) {
        SpringApplication.run(ClientService.class, args);
    }
}
```

5. 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:

6. Declare default load balanced (@Primary) OAuth2RestTemplate

```
@Primary
@Bean(name = "OAuth2RestTemplate")
@LoadBalanced
public OAuth2RestTemplate oAuth2RestTemplate(OAuth2ProtectedResourceDetails
details) {
    return new OAuth2RestTemplate(details);
}
```

7. Declare secondary load balanced OAuth2RestTemplate with password grant type and CardService credentials

```
@Value("${security.oauth2.client.accessTokenUri}")
private String accessTokenUri;

@Bean(name = "OAuth2CardService")
@LoadBalanced
public OAuth2RestTemplate oAuth2CardService() {
    ResourceOwnerPasswordResourceDetails resource = new
ResourceOwnerPasswordResourceDetails();
    resource.setAccessTokenUri(accessTokenUri);
    resource.setClientId("client");
    resource.setClientSecret("secret");
    resource.setUsername("card");
    resource.setPassword("card");
    DefaultAccessTokenRequest accessTokenRequest = new
DefaultAccessTokenRequest();
    OAuth2ClientContext context = new
DefaultOAuth2ClientContext(accessTokenRequest);
    return new OAuth2RestTemplate(resource, context);
}
```

8. Use OAuth2RestTemplate instead of Feign clients

AccountServiceClient.java

```
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.oauth2.client.OAuth2RestTemplate;
import org.springframework.security.oauth2.common.DefaultOAuth2AccessToken;
import
org.springframework.security.oauth2.provider.authentication.OAuth2Authenticatio
nDetails;
import org.springframework.stereotype.Component;

import java.math.BigDecimal;

@Component
public class AccountServiceClient {
    @Autowired
    private OAuth2RestTemplate oAuth2RestTemplate;

    public boolean checkout(Integer accountId, BigDecimal sum) {
        String token = ((OAuth2AuthenticationDetails)
SecurityContextHolder.getContext().getAuthentication().getDetails()).getTokenVa
lue();
        return checkout(token, accountId, sum);
    }

    @HystrixCommand(fallbackMethod = "checkoutFallback")
    private boolean checkout(String token, Integer accountId, BigDecimal sum) {
        oAuth2RestTemplate.getOAuth2ClientContext().setAccessToken(new
DefaultOAuth2AccessToken(token));
        String url = "http://AccountService/checkout/" + accountId.toString() +
"?sum=" + sum.toPlainString();
        return oAuth2RestTemplate.getForObject(url, Boolean.class);
    }

    private boolean checkoutFallback(String token, Integer accountId,
BigDecimal sum) {
        return false;
    }
}
```

CardServiceClient.java

```
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.security.oauth2.client.OAuth2RestTemplate;
import org.springframework.stereotype.Component;

@Component
public class CardServiceClient {
    @Autowired
    @Qualifier("OAuth2CardService")
    private OAuth2RestTemplate oAuth2CardService;

    @HystrixCommand(fallbackMethod = "createCardFallback")
    public String createCard() {
        return oAuth2CardService.getForObject("http://CardService/create",
String.class);
    }

    private String createCardFallback() {
        return null;
    }
}
```

Task 3 – SSO with Zuul

1. Add dependencies

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-jwt</artifactId>
</dependency>
```

2. Add @EnableOAuth2Sso annotation to configuration class

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.boot.autoconfigure.security.oauth2.client.EnableOAuth2Sso;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@SpringBootApplication
@EnableZuulProxy
@EnableOAuth2Sso
public class Zuul {
    public static void main(String[] args) {
        SpringApplication.run(Zuul.class, args);
    }
}
```

3. Configure 30s timeout for hystrix

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 30000
```

Lab 10 – Cloud Bus

Task 1 – Push Configurations

```
<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>
```

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 output();
}
```

- 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

- WordNumProcessor.java

```
@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
    partitionCount: 2
```

Task 4 – Stream Aggregator

- Aggregator.java

```
@SpringBootApplication
public class Aggregator {
    public static void main(String[] args) {
        new AggregateApplicationBuilder()
                .from(WordNumSplitter.class)
                .via(Processor.class)
                .run(args);
    }
}
```

- Aggregator.yml

```
spring:
  application:
    name: Aggregator

server:
  port: 8007

spring.cloud.stream.bindings.input:
  destination: wordNumSplit
  group: Splitters

spring.cloud.stream.bindings.output.destination: words

spring.cloud.stream.bindings.words.group: Processors
```