Mercury

Release 0.1

Various

Mar 31, 2022

INTRODUCTION

The Mercury project is created with one primary objective - `to make software easy to write, read, test, deploy, scale and manage.`

If you want digital decoupling, this is the technology that you should invest 30 minutes of your time to get familiar with.

This project is created by architects and computer scientists who have spent years to perfect software decoupling, scalability and resilience, high performance and massively parallel processing, with a very high level of decoupling, you can focus in writing business logic without distraction.

It introduces the concept of platform abstraction and takes event driven programming to the next level of simplicity and sophistication.

Everything can be expressed as anonymous functions and they communicate with each other using events. This includes turning synchronous HTTP requests and responses into async events using the REST automation system. However, event driven and reactive programming can be challenging. The Mercury framework hides all the complexity of event driven and reactive patterns and the magic of inter-service communication.

Since everything can be expressed as anonymous functions, the framework itself is written using this approach, including the cloud connectors and language pack in the project. In this way, you can add new connectors, plugins and language packs as you like. The framework is extensible.

Mercury supports unlimited service route names on top of event stream and messaging systems such as Kafka, Hazelcast, Tibco EMS and ActiveMQ artemis. While we make the event stream system works as a service mesh, Mercury can be used in standalone mode for applications that use pub/sub directly.

In fact, you can encapsulate other event stream or even enterprise service bus (ESB) with Mercury. Just use the connectors as examples. It would make your ESB runs like an event stream system for RPC, async, callback, streaming, pipeline and pub/sub use cases.

The pre-requisites are minimal. The foundation technology requires only Java (OpenJDK 8 or higher) and the Maven build system ("mvn"). Docker/Kubernetes are optional. The application modules that you create using the Mercury framework will run in bare metal, VM and any cloud environments.

Best regards, the Mercury team, Accenture

January 2022

Check out the Developer Guide section for further information.

Note: This project is under active development.

CHAPTER

ONE

CONTENTS

1.1 Getting Started

1.1.1 Before You Start

If you haven't already, please start a terminal and clone the repository:

```
git clone https://github.com/Accenture/mercury.git
cd mercury
```

To get the system up and running, you should compile and build the foundation libraries from sources. This will install the libraries into your ".m2/repository/org/platformlambda" folder.

`Important` - please close all Java applications and web servers in your PC if any. The build process will invoke unit tests that simulate HTTP and websocket servers.

```
# start a terminal and go to the mercury sandbox folder
mvn clean install
```

The platform-core, rest-spring, hazelcast-connector and kafka-connector are libraries and you can rebuild each one individually using `mvn clean install`

The rest of the subprojects are executables that you can rebuild each one with `mvn clean package`.

1.1.2 Getting Started

You can compile the rest-example as a microservices executable like this:

```
cd mercury/examples
cd rest-example
mvn clean package
java -jar target/rest-example-2.3.2.jar
# this will run the rest-example without a cloud connector
```

Try http://127.0.0.1:8083/api/hello/world with a browser.

It will respond with some sample output like this:

```
{
"body" : {
    "body" : {
```

(continues on next page)

(continued from previous page)

```
    "accept" : "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/
↪apng,*/*;q=0.8",
    "accept-encoding" : "gzip, deflate, br",
    "accept-language" : "en-US,en;q=0.9",
    "cache-control" : "max-age=0",
    "connection" : "keep-alive",
    "host" : "127.0.0.1:8083",
    "time" : "2018-12-21T16:39:33.546Z",
    "upgrade-insecure-requests" : "1",
    "user-agent" : "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) ..."
    },
    "headers" : {
    "seq" : "4"
    },
    "instance" : 4,
    "origin" : "2018122170c4d0e80ef94b459763636da74d6b5f"
},
"execution_time" : 0.298,
"headers" : { },
"round_trip" : 0.862,
"status" : 200
}
```

The REST endpoint makes a request to the "hello.world" service that echoes back the headers of the HTTP request. The underlying Spring Boot HTTP server has been pre-configured with HTML, JSON and XML message serializers and exception handlers.

If you change the "accept" header to "application/xml", the output will be shown in XML.

To demonstrate concurrent processing, you may visit http://127.0.0.1:8083/api/hello/concurrent

Now try http://127.0.0.1:8083/api/hello/pojo/1

This will return a HTTP-500 "Route hello.pojo not found" error. This is expected behavior because the "hello.pojo" service is not available in the rest-example container. Let's walk thru distributed and scalable application in the next section.

1.1.3 Scalable application using Kafka

For development and testing, you can start a standalone Kafka server.

```
# start a terminal and go to the mercury sandbox folder, then go to the kafka-standalone␣
↪folder
cd mercury/connectors/kafka/kafka-standalone/
mvn clean package
java -jar target/kafka-standalone-2.3.2.jar
# this will run a standalone Kafka server in the foreground
```

The next step is to start the "presence-monitor" application.

```
# start another terminal and go to kafka-presence folder
cd mercury/connectors/kafka/kafka-presence/
mvn clean package
```

(continues on next page)

(continued from previous page)

```
java -jar target/kafka-presence-2.3.2.jar
# this will run the presence monitor at port 8080 in the foreground

# when an application instance is started, it connects to the presence monitor to get␣
↪topic.
# you will see log similar to the following:
Adding member 20210405aa0220674e404169a5ec158a99714da6
Monitor (me) 20210405209f8e20ed3f4c0a80b035a50273b922 begins RSVP
multiplex.0001-001 assigned to 20210405aa0220674e404169a5ec158a99714da6 lambda-example,␣
↪2.1.1
Monitor (me) 20210405209f8e20ed3f4c0a80b035a50273b922 finished RSVP
```

Optionally, if you want to test resilience of the presence monitor, you can start another instance like this:

```
# start another terminal and go to kafka-presence folder
cd mercury/connectors/kafka/kafka-presence/
mvn clean package
java -Dserver.port=8081 -jar target/kafka-presence-2.3.2.jar
# this will run the presence monitor at port 8081 in the foreground
```

You can then run the lambda-example and rest-example applications

```
# go to the lambda-example project folder in one terminal
java -Dcloud.connector=kafka -jar target/lambda-example-2.3.2.jar
# the lambda-example will connect to the "presence monitor", obtain a topic and connect␣
↪to Kafka

# go to the rest-example project folder in another terminal
java -Dcloud.connector=kafka -jar target/rest-example-2.3.2.jar
# the rest-example will also connect to the "presence monitor", obtain a topic and␣
↪connect to Kafka

# the lambda-example and rest-example apps will show the topic assignment like this
presence.monitor, partition 1 ready
multiplex.0001, partition 0 ready
```

You may visit http://127.0.0.1:8083/api/hello/pojo/1 with a browser, you will see output like this:

```
{
"id": 1,
"name": "Simple PoJo class",
"address": "100 World Blvd, Planet Earth",
"date": "2021-04-05T22:13:38.351Z",
"instance": 1,
"origin": "20210405aa0220674e404169a5ec158a99714da6"
}
```

This means the rest-example app accepts the HTTP request and sends the request event to the lambda-example app which in turns return the response event as shown above.

The "cloud.connector=kafka" parameter overrides the application.properties in the rest-example and lambda-example projects so they can select Kafka as the service mesh.

If you use Jetbrains Idea ("IntelliJ") to run your project, please edit the start up config, click 'Modify options' and tick

'add VM options'. This allows you to set default run-time parameters. e.g. "-Dcloud.connector=kafka". Eclipse and other IDE would have similar run-time parameter options.

Get additional info from the presence monitor You may visit http://127.0.0.1:8080 and select "info". It may look like this:


```
{
"app": {
    "name": "kafka-presence",
    "description": "Presence Monitor",
    "version": "2.1.1"
},
"memory": {
    "max": "8,524,922,880",
    "free": "470,420,400",
    "allocated": "534,773,760"
},
"personality": "RESOURCES",
"additional_info": {
    "total": {
    "virtual_topics": 3,
    "topics": 2,
    "connections": 3
    },
    "virtual_topics": [
    "multiplex.0001-000 -> 2021082260f0b955a9ad427db14a9db751340d61, rest-example v2.1.1
↪",
    "multiplex.0001-001 -> 20210822371949df090644428cd98ae0ba91a69b, lambda-example v2.1.
↪1",
    "multiplex.0001-002 -> 2021082295c030edc07a4a4eb5cb78b6421f5fb7, lambda-example v2.1.
↪1"
    ],
    "topics": [
    "multiplex.0001 (32)",
    "service.monitor (11)"
    ],
    "connections": [
    {
        "created": "2021-08-22T17:29:45Z",
        "origin": "20210822371949df090644428cd98ae0ba91a69b",
        "name": "lambda-example",
        "topic": "multiplex.0001-001",
        "monitor": "20210822c750f194403241c49ef8fae113beff75",
        "type": "APP",
        "updated": "2021-08-22T17:29:53Z",
        "version": "2.1.1",
        "seq": 2,
        "group": 1
    },
    {
        "created": "2021-08-22T17:29:55Z",
        "origin": "2021082295c030edc07a4a4eb5cb78b6421f5fb7",
        "name": "lambda-example",
        "topic": "multiplex.0001-002",
```


(continues on next page)

(continued from previous page)


```
        "monitor": "20210822c750f194403241c49ef8fae113beff75",
        "type": "APP",
        "updated": "2021-08-22T17:30:04Z",
        "version": "2.1.1",
        "seq": 2,
        "group": 1
    },
    {
        "created": "2021-08-22T17:29:30Z",
        "origin": "2021082260f0b955a9ad427db14a9db751340d61",
        "name": "rest-example",
        "topic": "multiplex.0001-000",
        "monitor": "20210822c750f194403241c49ef8fae113beff75",
        "type": "WEB",
        "updated": "2021-08-22T17:30:01Z",
        "version": "2.1.1",
        "seq": 3,
        "group": 1
    }
    ],
    "monitors": [
    "20210822c750f194403241c49ef8fae113beff75 - 2021-08-22T17:30:01Z"
    ]
},
"vm": {
    "java_vm_version": "11.0.10+9",
    "java_runtime_version": "11.0.10+9",
    "java_version": "11.0.10"
},
"streams": {
    "count": 0
},
"origin": "20210822c750f194403241c49ef8fae113beff75",
"time": {
    "current": "2021-08-22T17:30:06.712Z",
    "start": "2021-08-22T17:28:56.128Z"
}
}
```


1.2 Architecture

1.2.1 Rationale

The microservices movement is gaining a lot of momentum in recent years. Very much inspired with the need t o modernize service-oriented architecture and to transform monolithic applications as manageable and reusable pieces, it was first mentioned in 2011 for an architectural style that defines an application as a set of loosely coupled single purpose services.

Classical model of microservices architecture often focuses in the use of REST as interface and the self-containment of data and process. Oftentimes, there is a need for inter-service communication because one service may consume another service. Usually this is done with a service broker. This is an elegant architectural concept. However, many production

systems face operational challenges. In reality, it is quite difficult to decompose a solution down to functional level. This applies to both green field development or application modernization. As a result, many microservices modules are indeed smaller subsystems. Within a traditional microservice, business logic is tightly coupled with 3rd party and open sources libraries including cloud platform client components and drivers. This is suboptimal.

1.2.2 User friendly reactive programming

Summary

Completely and natively event driven Fully open source - Apache-2.0 license Simple configuration and simple API hides all the scaffolding code Code naturally, without worrying about asynchrony Naturally code in functional style, which makes code very portable Mix and match languages - currently Java and Python are supported, Node.js and dotnet are WIP Built-in distributed tracing Avoid going to the network when running in the same process, a huge performance boost for latency critical applications Doesn't require all services to adopt, will provide the same benefits to those that do adopt it Learn and start coding in less than one hour

Benefits

Developer productivity - both authoring and maintenance Improves latency of calls within the same process Enables fully reactive implementations Scalability Portability of code between containers and clouds Observability - know who is calling who and when, know where you are in workflows Improves future-proofing - as better eventing libraries/frameworks/platforms become available, they can be brought in without changing a line of code Features

Multiple languages supported - currently Java and Python are supported, Node.js and dotnet are WIP Plug in your favorite event stream. Popular ones are already supported. Multiple eventing patterns supported - pub/sub, broadcast, command, etc. REST automation for event-based services Built in distributed tracing

Gnarly Use Cases solved with Mercury

Workflows with event-based architecture, utilizing Saga pattern Support of "closed user groups" for added security

1.2.3 Architecture principles

For simplicity, we advocate 3 architecture principles to write microservices

minimalist event driven context bound Minimalist means we want user software to be as small as possible. The Mercury framework allows you to write business logic down to functional level using simple input-process-output pattern.

Event driven promotes loose coupling. All functions should run concurrently and independently of each other.

Lastly, context bound means high level of encapsulation so that a function only expose API contract and nothing else.

1.2.4 Platform abstraction

Mercury offers the highest level of decoupling where each piece of business logic can be expressed as an anonymous function. A microservices module is a collection of one or more functions. These functions connect to each other using events.

The framework hides the complexity of event-driven programming and cloud platform integration. For the latter, the service mesh interface is fully encapsulated so that user functions do not need to be aware of network connectivity and details of the cloud platform.

1.2.5 Simple Input-Process-Output function

This is a best practice in software development. Input is fed into an anonymous function. It will process the input according to some API contract and produce some output.

This simplifies software development and unit tests.

1.3 Libraries and Components

`platform core`

With Mercury, any software module can be expressed as an anonymous functions or a Java class that implements the LambdaFunction interface.

The platform core library enables the following:

1. High level of decoupling - You may write business logic as anonymous functions that run concurrently and independently. In addition to business logic, you may also encapsulate platform components and databases as anonymous functions.
2. Event driven programming - Each function is addressable with a unique "route name" and they communicate by sending events. You make request from one function to another by making a service call through some simple Mercury Post Office API.
3. One or more functions can be packaged together as a microservice executable, usually deployed as a Docker image or similar container technology.

`rest-spring`

The rest-spring library customizes and simplifies the use of Spring Boot as an application container to hold functions. This includes preconfigured message serializers and exception handlers.

`cloud-connector` and `service-monitor`

The cloud-connector and service-monitor are the core libraries to support implementations of various cloud connectors.

Mercury supports Kafka, Hazelcast, ActiveMQ-artemis and TIBCO-EMS out of the box.

`kafka-connector`

This is the kafka specific connector library is designed to work with Kafka version 2.7.0 out of the box that you add it into your application's pom.xml. A convenient standalone Kafka server application is available in the kafka-standalone project under the connector/kafka directory. The standalone Kafka server is a convenient tool for application development and tests in the developer's laptop.

`kafka-presence`

This "presence monitor" manages mapping of Kafka topics and detects when an application instance is online or offline. Your application instances will report to the presence monitors (2 monitor instances are enough for large installations). Since version 2.0, it uses a single partition of a topic to serve an application instance. To avoid deleting topics, the system can re-use topics and partitions automatically.

The kafka-presence system is fully scalable.

`language-connector`

The python language pack is available in https://github.com/Accenture/mercury-python

`rest-automation`

This extension package is a system that automates the creation of REST endpoints by configuration instead of code. Not only it eliminates the repetitive work of writing REST endpoints, it makes HTTP completely non-blocking.

`distributed-tracer`

This extension package is an example application that consolidates distributed trace metrics.

`lambda-example`

This is an example project to illustrate writing microservices without any HTTP application server.

`rest-example`

If your application module needs info and health admin endpoints, you may want to use this example project as a template.

It encapsulates Spring Boot, automates JSON/XML/PoJo serialization and simplfies application development. It also allows you to write custom REST endpoints using WebServlet and JAX-RS REST annotations.

For rapid application development, we do recommend you to use the REST automation system to define REST endpoints by configuration rather than code.

1.4 Compatability

1.4.1 JDK Compatability

The Mercury project, with the exception of the `kafka-standalone` subproject, has been tested with OpenJDK and AdaptOpenJDK version 8 to 16.

Please use Java version 11 or higher to run the kafka-standalone application which is provided as a convenient tool for development and testing. The kafka standalone server would fail due to a known memory mapping bug when running under Java version 1.8.0_292.

1.4.2 Kafka Compatability

As of December 2020, Mercury has been tested and deployed with Apache Kafka, Confluent Kafka, IBM Event Streams and Microsoft Azure Event Hubs.

1.4.3 Other Consideration

- Timestamp: the Mercury system uses UTC time and ISO-8601 string representation when doing serialization. https://en.wikipedia.org/wiki/ISO_8601
- UTF8 text encoding: we recommend the use of UTF8 for text strings.

1.5 Terminology

1. `Microservices` - Generally, we use this term to refer to an architectural pattern where business logic is encapsulated in a bounded context, the unit of service is minimalist and services are consumed in an event driven manner.
2. `Microservices function` or simply `function` - This refers to the smallest unit that encapsulates business logic or 3rd party library. As a best practice, we advocate clear separation of business logic from proprietary libraries. By wrapping 3rd party libaries as functions, we can keep the event API intact when switching to an alternative library.
3. `Microservices module` or `microservice` - This refers to a single unit of deployment that contains at least one public function and optionally some private functions. All functions communicate with each others through a memory based event stream system within the module. Each microservices module can be independently deployed and scaled.
4. `Microservices application` - It is a collection of microservices modules running together as a single application.
5. `User facing endpoints` - This refers to public API for REST, websocket or other communication protocols.
6. `Event API` - Microservices functions expose event APIs internally for inter-service communications. For example, a user facing endpoint may use REST protocol. The HTTP request is then converted to an event to a microservice.
7. `Real-time events` - Messages that are time sensitive. e.g. RPC requests and responses.
8. `Store-n-forward events` - Events that are not time sensitive. First, a calling function can send an event without waiting for a response. Second, the called function may not even available when the event is delivered. e.g. data pipeline applications.
9. `Streaming events` - This refers to continuous flow of events from one function to another. Note that streaming can be either real-time or store-n-forward.
10. `public function` - This means the function is visible to all application containers and instances in the system.
11. `private function` - This means the function is visible only to functions in the same memory space inside an application instance.
12. `route name` - Each service can be uniquely identified with a name.
13. `closed user groups` - Conceptually, we treat the underlying event stream system as a conduit where one group of application instances can be logically separated from another group.

1.6 How-Tos

1.6.1 Microservices and anonymous function

Mercury is all about microservices that are minimalist, event-driven and context bounded.

To this end, we use anonymous functions to encapsulate different domains of business logic and library dependencies.

Under the Mercury framework, business logic wrapped in anonymous functions are callable using a `route name`. Mercury resolves routing automatically so that you do not need to care whether the calling and called functions are in the same memory space or in different application instances. Mercury will route requests using a high performance memory event bus when the calling and called functions are in the same memory space and route requests through a network event stream system when the calling and called parties reside in different containers.

1.6.2 Writing your first microservices function

Your first function may look like this using Java 1.8 anonymous function syntax:

```
LambdaFunction f = (headers, body, instance) -> {
    // do some business logic
    return something
};
```

The easiest way to write your first microservices module is to use either the "lambda-example" or "rest-example" as a template.

Let's try with the "rest-example". You should update the application name in both the `application.properties` and the `pom.xml`. Then you can use your favorite IDE to import it as a "maven" project.

1.6.3 Application unit

The rest-example is a deployable application unit. Behind the curtain, the mercury framework is using Spring Boot to provide REST and websocket capabilities. For microservices modules that do not need REST endpoints, you can use the "lambda-example" as a template.

1.6.4 Main application

For each application unit, you will need a main application. This is the entry of your application unit.

The `MainApplication` annotation indicates that this is the main method for the application unit. Main application should also implements the `EntryPoint` interface which only has the "start" method. The "args" are optional command line arguments.

In the following example, when the application unit starts, it creates a microservices function and register "hello.world" as its route name. For concurrency, it also specifies 20 worker instances.

Application units are horizontally scalable. Within the application unit, you may specify concurrent "workers". This provides horizontal and verticial scalability respectively.

```
1 @MainApplication
2 public class MainApp implements EntryPoint {
3
4     @Override
```

(continues on next page)

(continued from previous page)

```
    public void start(String[] args) throws Exception {
        ServerPersonality.getInstance().setType(ServerPersonality.Type.WEB);
        Platform platform = Platform.getInstance();
        LambdaFunction echo = (headers, body, instance) -> {
            Map<String, Object> result = new HashMap<>();
            result.put("headers", headers);
            result.put("body", body);
            result.put("instance", instance);
            result.put("origin", platform.getOrigin());
            return result;
        };
        platform.register("hello.world", echo, 20);
    }
}
```

Alternatively, for typed body and response without casting you can use TypedLambdaFunction<I, O> where I and O stand for input and output class respectively.

```
TypedLambdaFunction<String, Map<String, String>> lambda = (headers, body, instance) -> {
    Map<String, String> result = new HashMap<>();
    result.put("body", body);
    return result;
};
```

1.6.5 Calling a function

Unlike traditional programming, you call a function by sending an event instead of calling its method. Mercury resolves routing automatically so events are delivered correctly no matter where the target function is, in the same memory space or another computer elsewhere in the network.

To make a service call to a function, you may do the following:

```
PostOffice po = PostOffice.getInstance();
EventEnvelope response = po.request("hello.world", 1000, "a test message");
System.out.println("I got response here..."+response.getBody());

// the above is an RPC call. For async call, it would be something like this:
po.send("hello.world", "another message");
```

You can call the function from another function or a REST endpoint. The latter connects REST API with a microservices function.

The following example forwards a request from the REST endpoint (`GET /api/hello/world`) to the "hello.world" service. Note that there are basic performance metrics from the response object.

```
@Path("/hello")
public class MyRestEndpoint {

    private static AtomicInteger seq = new AtomicInteger(0);

    @GET
    @Path("/world")
```

(continues on next page)

(continued from previous page)

```
    @Produces({MediaType.TEXT_PLAIN, MediaType.APPLICATION_JSON, MediaType.APPLICATION_
↪XML, MediaType.TEXT_HTML})
    public Map<String, Object> hello(@Context HttpServletRequest request) throws␣
↪IOException, TimeoutException, AppException {

        PostOffice po = PostOffice.getInstance();

        Map<String, Object> forward = new HashMap<>();
        forward.put("time", new Date());

        Enumeration<String> headers = request.getHeaderNames();
        while (headers.hasMoreElements()) {
            String key = headers.nextElement();
            forward.put(key, request.getHeader(key));
        }
        // As a demo, just put the incoming HTTP headers as a payload and a parameter␣
↪showing the sequence counter.
        // The eco service will return both.
        int n = seq.incrementAndGet();
        EventEnvelope response = po.request("hello.world", 3000, forward, new Kv("seq",␣
↪n));

        Map<String, Object> result = new HashMap<>();
        result.put("status", response.getStatus());
        result.put("headers", response.getHeaders());
        result.put("body", response.getBody());
        result.put("execution_time", response.getExecutionTime());
        result.put("round_trip", response.getRoundTrip());
        return result;
    }

}
```

1.6.6 Massive parallel processing

A function is invoked when an event happens. Before the event arrives, the function is just an entry in a routing table, and it does not consume any additional resources like threads.

All functions are running in parallel without special coding. Behind the curtain, the system uses Java futures and asynchronous event loops for very efficient function execution.

1.6.7 Built-in service mesh

The above demonstrates distributed applications using Kafka as a service mesh.

1.6.8 Built-in pub/sub

You can also use Mercury with other service mesh of your choice. In this case, you can use the built-in pub/sub APIs of Mercury for your app to communicate with Kafka and other event stream systems.

To enable Kafka pub/sub without using it as a service mesh, use these parameters in application.properties

cloud.connector=none cloud.services=kafka.pubsub This means the system encapsulates the original pub/sub feature of the underlying event stream system. The built-in publishers and listeners will do the heavy lifting for you in a consistent manner. Note that Kafka supports rewinding read "offset" so that your application can read older messages. In Hazelcast, the older events are dropped after delivery.

Example:

```
// setup your subscriber function
LambdaFunction myFunction = (headers, body, instance) -> {
log.info("Got ---> {}", body);
return true;
};
```

```
PubSub ps = PubSub.getInstance();
/*
* Pub/sub service starts asynchronously.
* If your runs pub/sub code before the container is completely initialized,
* you may want to "waitForProvider" for a few seconds.
*/
ps.waitForProvider(10);
// this creates a topic with one partition
ps.createTopic("some.kafka.topic", 1);
// this subscribe the topic with your function
ps.subscribe("some.kafka.topic", myFunction, "client-101", "group-101");
// this publishes an event to the topic
ps.publish("some.kafka.topic", null, "my test message");
```

If you run this application for the first time and you do not see the test message, the kafka topic has just been created when your application starts. Due to racing condition, Kafka would skip the offset and you cannot see the first message. Just restart the application and you will see your test message.

However, if you create the topic administratively before running this test app, your app will always show the first test message. This is a normal Kafka behavior.

You may also notice that the Kafka client sets the read offset to the latest pointer. To read from the beginning, you may reset the read offset by adding a parameter "0" after the clientId and groupId in the subscribe statement above.

1.6.9 Work nicely with reactive frameworks

Mercury provides a stream abstraction that can be used with reactive frameworks.

For example, developers using Spring reactor with Mercury may setup a stream between two app modules within the same container or in different containers like this:

1.6.10 Write your own microservices

You may use the lambda-example and rest-example as templates to write your own applications.

Please update pom.xml and application.properties for application name accordingly.

1.6.11 Cloud Native

The Mercury framework is Cloud Native. While it uses the local file system for buffering, it expects local storage to be transient.

If your application needs to use the local file system, please consider it to be transient too, i.e., you cannot rely on it to persist when your application restarts.

If there is a need for data persistence, use external databases or cloud storage.

1.6.12 Dockerfile

Creating a docker image from the executable is easy. First you need to build your application as an executable with the command mvn clean package. The executable JAR is then available in the target directory.

The Dockerfile may look like this:

```
FROM adoptopenjdk/openjdk11:jre-11.0.11_9-alpine
EXPOSE 8083
WORKDIR /app
COPY target/your-app-name.jar .
ENTRYPOINT ["java","-jar","your-app-name.jar"]
```

To build a new docker image locally:

```
docker build -t your-app-name:latest .
```

To run the newly built docker image, try this:

```
docker run -p 8083:8083 -i -t your-app-name:latest
```

Change the exposed port numnber and application name accordingly. Then build the docker image and publish it to a docker registry so you can deploy from there using Kubernetes or alike.

For security reason, you may want to customize the docker file to use non-priliveged Unix user account. Please consult your company's enterprise architecture team for container management policy.

1.6.13 VM or bare metal deployment

If you are deploying the application executables in a VM or bare metal, we recommend using a cross-platform process manager. The system has been tested with "pm2" (https://www.npmjs.com/package/pm2).

A sample process.json file is shown below. Please edit the file accordingly. You may add "-D" or "-X" parameters before the "-jar" parameter. To start the application executable, please do pm2 start my-app.json.

You may create individual process.json for each executable and start them one-by-one. You can then monitor the processes with pm2 list or pm2 monit.

To deploy using pm2, please browse the pm2-example folder as a starting point.

1.6.14 Distributed tracing

Microservices are likely to be deployed in a multi-tier environment. As a result, a single transaction may pass through multiple services.

Distributed tracing allows us to visualize the complete service path for each transaction. This enables easy trouble shooting for large scale applications.

With the Mercury framework, distributed tracing does not require code at application level. To enable this feature, you can simply set "tracing=true" in the rest.yaml configuration of the rest-automation application.

1.7 Application Properties

Table 1: application.properties

Key	Value (example)	Required
application.name or spring.application.name	Application name	Yes
info.app.version	major.minor.build (e.g. 1.0.0)	Yes
info.app.description	Something about application	Yes
web.component.scan	Your own package path or parent path	Yes
server.port	e.g. 8083	Yes*
rest.server.port	e.g. 8083	Required*2
spring.mvc.static-path-pattern	/**	Yes*
spring.resources.static-locations	classpath:/public/	Yes*
jax.rs.application.path	/api	Optional*1
show.env.variables	comma separated list of variable names	Optional*1
show.application.properties	comma separated list of property names	Optional*1
cloud.connector	kafka, hazelcast, none, etc.	Optional
cloud.services	e.g. some.interesting.service	Optional

continues on next page

Table 1 – continued from previous page

Key	Value (example)	Required
snake.case.serialization	true (recommended)	Optional
env.variables	e.g. MY_ENV:my.env	Optional
safe.data.models	packages pointing to your PoJo classes	Optional
protect.info.endpoints	true or false (default is false)	Optional*1
trace.http.header	comma separated list traceId labels	*2
trace.log.header	default value is X-Trace-Id	Optional
index.redirection	comma separated list of URI paths	Optional*1
index.page	default value is index.html	Optional*1
application.feature.route.substitution	default value is false	Optional
route.substitution.file	comma separated file(s) or classpath(s)	Optional
application.feature.topic.substitution	default value is false	Optional
topic.substitution.file	comma separated file(s) or classpath(s)	Optional
cloud.client.properties	e.g. classpath:/kafka.properties	connectors
kafka.replication.factor	3	Kafka
default.app.group.id	kafka groupId for the app instance	Optional
default.monitor.group.id	kafka groupId for the presence-monitor	Optional
monitor.topic	kafka topic for the presence-monitor	Optional
app.topic.prefix	multiplex (default value, DO NOT change)	Optional
app.partitions.per.topic	Max Kafka partitions per topic	Optional
max.virtual.topics	Max virtual topics = partitions * topics	Optional
max.closed.user.groups	Number of closed user groups	Optional
closed.user.group	Closed user group (default 1)	Optional
transient.data.store	Default value: /tmp/reactive	Optional
running.in.cloud	Default value: false	Optional
multicast.yaml	This is used to define config file path	Optional
journal.yaml	This is used to define config file path	Optional
distributed.trace.aggregation	Default value: true	Optional
deferred.commit.log	Default value: false	Optional*3

*1 - when using the "rest-spring" library *2 - applies to the REST automation application only *3 - optional for unit

test purpose only. DO NOT set this parameter in "main" branch for production code.

1.7.1 transient data store

The system handles back-pressure automatically by overflowing memory to a transcient data store. As a cloud native best practice, the folder must be under "/tmp". The default is "/tmp/reactive". The "running.in.cloud" must be set to false when your apps are running in IDE or in your laptop. When running in kubernetes, it can be set to true.

1.7.2 safe.data.models

PoJo may execute Java code. As a result, it is possible to inject malicious code that does harm when deserializing a PoJo. This security risk applies to any JSON serialization engine.

For added security and peace of mind, you may want to white list your PoJo package paths. When the "safe.data.models" property is configured, the underlying serializers for JAX-RS, Spring RestController, Servlets will respect this setting and enforce PoJo white listing.

Usually you do not need to use the serializer in your code because it is much better to deal with PoJo in your IDE. However, if there is a genuine need to do low level coding, you may use the pre-configured serializer so that the serialization behavior is consistent.

You can get an instance of the serializer with `SimpleMapper.getInstance().getWhiteListMapper()`.

1.7.3 trace.http.header

Identify the HTTP header for traceId. When configured with more than one label, the system will retrieve traceID from the corresponding HTTP header and propagate it through the transaction that may be served by multiple services.

If traceId is presented in a HTTP request, the system will use the same label to set HTTP response traceId header.

e.g. X-Trace-Id: a9a4e1ec-1663-4c52-b4c3-7b34b3e33697 or X-Correlation-Id: a9a4e1ec-1663-4c52-b4c3-7b34b3e33697

1.7.4 trace.log.header

If tracing is enabled for a transaction, this will insert the trace-ID into the logger's ThreadContext using the trace.log.header.

Note that trace.log.header is case sensitive and you must put the corresponding value in log4j2.xml. The default value is "X-Trace-Id" if this parameter is not provided in application.properties. e.g.

```
<PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level %logger:%line
[%X{X-Trace-Id}] - %msg%n" />
```

1.7.5 Kafka specific configuration

If you use the kafka-connector (cloud connector) and kafka-presence (presence monitor), you may want to externalize kafka.properties. It is recommended to set `kafka.client.properties=file:/tmp/config/kafka.properties`

Note that "classpath" refers to embedded config file in the "resources" folder in your source code and "file" refers to an external config file.

You want also use the embedded config file as a backup like this:

```
# this is the configuration used by the kafka.presence monitor
kafka.client.properties=file:/tmp/config/kafka.properties,classpath:/kafka.properties
```

`kafka.replication.factor` is usually determined by DevOps. Contact your administrator for the correct value. If the available number of kafka brokers are more than the replication factor, it will use the given replication factor. Otherwise, the system will fall back to a smaller value and this optimization may not work in production. Therefore, please discuss with your DevOps administrator for the optimal replication factor value.

`Number of replicated copies = replication factor - 1`

1.7.6 presence monitor

You will deploy a "presence monitor" to assign topics to any connected user application instances.

Your user application instances will connect to the presence monitor using websocket. When an application instance fails, the presence monitor can detect it immediately so that its peer application instances can update the routing table.

1.7.7 distributed trace logging, aggregation and transaction journaling

To enable distributed trace logging, please set this in log4j2.xml. For cloud native apps, you should redirect logging from standard out to a centralized logging system such as Elastic Search or Splunk.

`<logger name="org.platformlambda.core.services.DistributedTrace" level="INFO" />`

Distributed trace aggregation will be available when you deploy your custom aggregation service with the route name `distributed.trace.processor`.

If you want to disable aggregation for an application, set "distributed.trace.aggregation" to false in application.properties.

Journaling service will be available with you deploy your custom aggregation service with the route name `distributed.trace.processor` and configure the "journal.yaml" parameter in application.properties pointing to a YAML file containing the following:

```
journal:
- "my.service.1"
- ...
# my.service.1 is an example. You should enter the correct route names for your services␣
↪to be journaled.
```

The system will send transaction input/output payloads to the journaling service when trace is turned on and journal.yaml contains the service route name.

For security and privacy, transaction journal should be saved to a database with access control because the transaction payload may contain sensitive information.

1.7.8 Spring Boot

The foundation code uses Spring Boot in the "rest-spring" library. For loose coupling, we use the `@MainApplication` as a replacement for the `SpringApplication`. Please refer to the MainApp class in the "rest-example" project. This allows us to use any REST application server when technology evolves.

If your code uses Spring Boot or Spring Framework directly, you can set the corresponding key-values in the application.properties file in the resources folder. e.g. changing the "auto-configuration" parameters.

1.8 Reserved Route Names

The Mercury foundation functions are also written using the same event-driven platform-core. The following route names are reserved for the Mercury system functions.

Please do not use them in your application functions as it would disrupt the normal operation of the event-driven system and your application may not work as expected.

Table 2: Reserved Route Names

Route Name	Purpose	Modules
actuator.services	Reserved for actuator admin endpoint	platform-core
elastic.queue.cleanup	Elastic queue clean up task	platform-core
distributed.tracing	Distributed trace logger	platform-core
distributed.trace.processor	Distributed trace aggregator	User defined trace handler
system.service.registry	Distributed routing registry	cloud connectors
system.service.query	Distributed routing query	cloud connectors
cloud.connector.health	Cloud connector health service	cloud connectors
additional.info	Additional info service	cloud connectors
cloud.manager	Cloud manager service	Cloud connectors
presence.service	Presence reporter service	cloud connectors
presence.housekeeper	Presence housekeeper service	cloud connectors
cloud.connector	Cloud event emitter	Cloud connectors
async.http.request	HTTP request event handler	REST automation system
async.http.response	HTTP response event handler	REST automation system
notification.manager	Simple notification service	REST automation system
ws.token.issuer	Websocket access token issuer	REST automation system
ws.notification	Websocket notification handler	REST automation system
language.pack.inbox	RPC inbox handler	Language Connector
language.pack.registry	Routing registry	Language Connector
pub.sub.controller	Pub/sub handler	Language Connector
system.deferred.delivery	Deferred delivery handler	Language Connector
object.streams.io	Object stream manager	Language Connector
cron.scheduler	Cron job scheduler	Scheduler helper application
init.service.monitor.*	reserved for event stream startup	service monitor core
completion.service.monitor.*	reserved for event stream clean up	service monitor core
init.multiplex.*	reserved for event stream startup	cloud connector core
completion.multiplex.*	reserved for event stream clean up	cloud connector core

1.8.1 Distributed trace processor

The route name "distributed.trace.processor" is reserved for user defined trace handler. If you implement a function with this route name, it will receive trace metrics in a real-time basis. Your custom application can then decide how to persist the metrics. e.g. Elastic Search or a database.

1.9 Platform API

Mercury has a small set of API for declaring microservices functions and sending events to functions.

1.9.1 Basic platform classes and utilities

1. `Platform` - the platform class is a singleton object for managing life cycle of functions.
2. `PostOffice` - a singleton object for sending events to functions.
3. `ServerPersonality` - a singleton object for setting the personality or type of an application unit. e.g. REST, WEB, APP, RESOURCES and DEVOPS
4. `AppConfigReader` - a singleton object for reading application.properties that can be overriden by run-time parameters or environment variables.
5. `CryptoApi` - a convenient crypto library for common cryptographic tasks like hashing, AES and RSA encryption and digital signing.
6. `ManagedCache` - a useful in-memory cache store.
7. `MultiLevelMap` - a HashMap wrapper that allows you to retrieve item using the dot-bracket syntax. e.g. map.getElement("hello.world[3]")
8. `Utility` - a convenient singleton object for commonly methods such as UTC time string, UTF, stream, file, etc.
9. `SimpleMapper` - a preconfigured JSON serializer.
10. `SimpleXmlParser` and `SimpleXmlWriter` - efficient XML serializer.

1.9.2 EventEnvelope

EventEnvelope is a vehicle for storing and transporting an event that contains headers and body. `headers` can be used to carry parameters and `body` is the message payload. Each event should have either or both of headers and body. You can set Java primitive, Map or PoJo into the body.

Mercury automatically performs serialization and deserialization using the EventEnvelope's `toBytes()` and `load(bytes)` methods. For performance and network efficiency, it is using Gson and MsgPack for serialization into Map and byte array respectively.

EventEnvelope is used for both input and output. For simple use cases in asynchronous operation, you do not need to use the EventEnvelope. For RPC call, the response object is an EventEnvelope. The service response is usually stored in the "body" in the envelope. A service may also return key-values in the "headers" field.

Mercury is truly schemaless. It does not care if you are sending a Java primitive, Map or PoJo. The calling function and the called function must understand each other's API interface contract to communicate properly.

1.9.3 Platform API

Obtain an instance of the platform object

```
Platform platform = Platform.getInstance();
```

1.9.4 Register a public function

To register a function, you can assign a route name to a function instance. You can also set the maximum number of concurrent workers in an application instance. This provides vertical scalability in addition to horizontal scaling by Docker/Kubernetes.

To create a singleton function, set instances to 1.

```
platform.register(String route, LambdaFunction lambda, int instances) throws IOException;
```

For example:

```
platform.register("hello.world", echo, 20);
```

1.9.5 Register a private function

Public functions are advertised to the whole system while private functions are encapsulated within an application instance.

You may define your function as private if it is used internally by other functions in the same application instance.

```
platform.registerPrivate(String route, LambdaFunction lambda, int instances) throws␣
↪IOException;
```

1.9.6 Release a function

A function can be long term or transient. When a function is no longer required, you can cancel the function using the "release" method.

```
void release(String route) throws IOException;
```

1.9.7 Connect to the cloud

You can write truly event-driven microservices as a standalone application. However, it would be more interesting to connect the services together through a network event stream system.

To do this, you can ask the platform to connect to the cloud.

```
platform.connectToCloud();
```

1.10 Post Office API

Post Office is a platform abstraction layer that routes events among functions. It maintains a distributed routing table to ensure that service discovery is instantaneous,

1.10.1 Obtain an instance of the post office object

```
PostOffice po = PostOffice.getInstance();
```

1.10.2 Communication patterns

- RPC "Request-response", best for interactivity
- Asynchronous e.g. Drop-n-forget and long queries
- Call-back e.g. Progressive rendering
- Pipeline e.g. Work-flow application
- Streaming e.g. Data ingest
- Broadcast e.g. Concurrent processing of the same dataset with different outcomes

1.10.3 RPC (Request-response)

The Mercury framework is 100% event-driven and all communications are asynchronous. To emulate a synchronous RPC, it uses temporary Inbox as a callback function. The target service will send reply to the callback function which in turns delivers the response.

To make an RPC call, you can use the `request` methods. These methods would throw exception if the response status is not 200.

```
EventEnvelope request(String to, long timeout, Object body) throws IOException,␣
↪TimeoutException, AppException;
EventEnvelope request(String to, long timeout, Object body, Kv... parameters) throws␣
↪IOException, TimeoutException, AppException;
EventEnvelope request(EventEnvelope event, long timeout) throws IOException,␣
↪TimeoutException, AppException;

// example
EventEnvelope response = po.request("hello.world", 1000, somePayload);
System.out.println("I got response..."+response.getBody());
```

A non-blocking version of RPC is available with the `asyncRequest` method. Only timeout exception will be sent to the onFailure method. All other cases will be delivered to the onSuccess method. You should check event.getStatus() to handle exception.

```
Future<EventEnvelope> asyncRequest(final EventEnvelope event, long timeout) throws␣
↪IOException;

// example
Future<EventEnvelope> future = po.asyncRequest(new EventEnvelope().setTo(SERVICE).
↪setBody(TEXT), 2000);
```

(continues on next page)

(continued from previous page)

```
future.onSuccess(event -> {
    // handle the response event
}).onFailure(ex -> {
    // handle exception
});
```

Note that Mercury supports Java primitive, Map and PoJo in the message body. If you put other object, it may throw serialization exception or the object may become empty.

1.10.4 Asynchronous / Drop-n-forget

To make an asynchronous call, use the send method.

```
void send(String to, Kv... parameters) throws IOException;
void send(String to, Object body) throws IOException;
void send(String to, Object body, Kv... parameters) throws IOException;
void send(final EventEnvelope event) throws IOException;
```

Kv is a key-value pair for holding one parameter.

1.10.5 Deferred delivery

```
String sendLater(final EventEnvelope event, Date future) throws IOException;
```

A scheduled ID will be returned. You may cancel the scheduled delivery with `cancelFutureEvent(id)`.

1.10.6 Call-back

You can register a call back function and uses its route name as the "reply-to" address in the send method. To set a reply-to address, you need to use the EventEnvelope directly.

```
void send(final EventEnvelope event) throws IOException;

// example
EventEnvelope event = new EventEnvelope();
event.setTo("hello.world").setBody(somePayload);
po.send(event);
```

To handle exception from a target service, you may implement ServiceExceptionHandler. For example:

```
private static class SimpleCallback implements TypedLambdaFunction<PoJo, Void>,␣
↪ServiceExceptionHandler {

    @Override
    public void onError(AppException e, EventEnvelope event) {
        // handle exception here
    }

    @Override
    public Void handleEvent(Map<String, String> headers, PoJo body, int instance) throws␣
↪Exception {
```

(continues on next page)

(continued from previous page)

```
        // handle input. In this example, it is a PoJo
        return null;
    }
}
```

1.10.7 Pipeline

In a pipeline operation, there is stepwise event propagation. e.g. Function A sends to B and set the "reply-to" as C. Function B sends to C and set the "reply-to" as D, etc.

To pass a list of stepwise targets, you may send the list as a parameter. Each function of the pipeline should forward the pipeline list to the next function.

```
EventEnvelope event = new EventEnvelope();
event.setTo("function.b").setBody(somePayload).setReplyTo("function.c")
    .setHeader("pipeline",  "function.a->function.b->function.c->function.d");
po.send(event);
```

1.10.8 Streaming

You can use streams for functional programming. There are two ways to do streaming.

1. Singleton functions

To create a singleton, you can set `instances` of the calling and called functions to 1. When you send events from the calling function to the called function, the platform guarantees that the event sequencing of the data stream.

To guarantee that there is only one instance of the calling and called function, you should register them with a globally unique route name. e.g. using UUID like "producer-b351e7df-827f-449c-904f-a80f9f3ecafe" and "consumer-d15b639a-44d9-4bc2-bb54-79db4f866fe3".

Note that you can programmatically `register` and `release` a function at run-time.

If you create the functions at run-time, please remember to release the functions when processing is completed to avoid wasting system resources.

2. Object stream

To do object streaming, you can use the ObjectStreamIO to create a stream or open an existing stream. Then, you can use the `ObjectStreamWriter` and the `ObjectStreamReader` classes to write to and read from the stream.

For the producer, if you close the output stream, the system will send a `EOF` to signal that there are no more events to the stream.

For the consumer, When you detect the end of stream, you can close the input stream to release the stream and all resources associated with it.

I/O stream consumes resources and thus you must close the input stream at the end of stream processing. The system will automatically close the stream upon an expiry timer that you provide when a new stream is created.

The following unit test demonstrates this use case.

```
String messageOne = "hello world";
String messageTwo = "it is great";
/*
* Producer creates a new stream with 60 seconds inactivity expiry
```

(continues on next page)

(continued from previous page)

```
*/
ObjectStreamIO stream = new ObjectStreamIO(60);
ObjectStreamWriter out = new ObjectStreamWriter(stream.getOutputStreamId());
out.write(messageOne);
out.write(messageTwo);
/*
* If output stream is closed, it will send an EOF signal so that the input stream reader␣
↪will detect it.
* Otherwise, input stream reader will see a RuntimeException of timeout.
*
* For this test, we do not close the output stream to demonstrate the timeout.
*/
//  out.close();

/*
* Producer should send the inputStreamId to the consumer using "stream.getInputStreamId()
↪" after the stream is created.
* The consumer can then open the stream with the streamId.
*/
ObjectStreamReader in = new ObjectStreamReader(inputStreamId, 8000);
int i = 0;
try {
    for (Object data : in) {
        if (data == null) break; // EOF
        i++;
        if (i == 1) {
            assertEquals(messageOne, data);
        }
        if (i == 2) {
            assertEquals(messageTwo, data);
        }
    }
} catch (RuntimeException e) {
    // iterator will timeout since the stream was not closed
    assertTrue(e.getMessage().contains("timeout"));
}

// ensure that it has read the two messages
assertEquals(2, i);
// must close input stream to release resources
in.close();
```

1.10.9 Broadcast

Broadcast is the easiest way to do "pub/sub". To broadcast an event to multiple application instances, use the `broadcast` method.

```
void broadcast(String to, Kv... parameters) throws IOException;
void broadcast(String to, Object body) throws IOException;
void broadcast(String to, Object body, Kv... parameters) throws IOException;

// example
po.broadcast("hello.world", "hey, this is a broadcast message to all hello.world␣
↪providers");
```

1.10.10 Join-n-fork

You can perform join-n-fork RPC calls using a parallel version of the `request` method.

```
List<EventEnvelope> request(List<EventEnvelope> events, long timeout) throws IOException;

// example
List<EventEnvelope> parallelEvents = new ArrayList<>();

EventEnvelope event1 = new EventEnvelope();
event1.setTo("hello.world.1");
event1.setBody(payload1);
event1.setHeader("request", "#1");
parallelEvents.add(event1);

EventEnvelope event2 = new EventEnvelope();
event2.setTo("hello.world.2");
event2.setBody(payload2);
event2.setHeader("request", "#2");
parallelEvents.add(event2);

List<EventEnvelope> responses = po.request(parallelEvents, 3000);
```

A non-blocking version of fork-n-join is available with the `asyncRequest` method. Only timeout exception will be sent to the onFailure method. All other cases will be delivered to the onSuccess method. You should check event.getStatus() to handle exception.

```
Future<List<EventEnvelope>> asyncRequest(final List<EventEnvelope> event, long timeout)␣
↪throws IOException;

// example
List<EventEnvelope> requests = new ArrayList<>();
requests.add(new EventEnvelope().setTo(SERVICE1).setBody(TEXT1));
requests.add(new EventEnvelope().setTo(SERVICE2).setBody(TEXT2));
Future<List<EventEnvelope>> future = po.asyncRequest(requests, 2000);
future.onSuccess(events -> {
    // handle the response events
}).onFailure(ex -> {
    // handle timeout exception
});
```

1.10.11 Inspecting event's metadata

If you want to inspect the incoming event's metadata to make some decisions such as checking correlation-ID and sender's route address, you can use the TypedLambdaFunction to specify input as EventEnvelope.

Another way to inspect event's metadata is the use of the EventInterceptor annotation in your lambda function. Note that event interceptor service does not return result, it intercepts incoming event for forwarding to one or more target services. If the incoming request is a RPC call and the interceptor does not forward the event to the target service, the call will time out.

1.10.12 Default PoJo mapping

PoJo mapping is determined at the source. When the caller function sets the PoJo, the object is restored as the original PoJo in the target service provided that the common data model is available in both source and target services.

```
public Object getBody(); // <-- default mapping
```

1.10.13 Retrieve raw data as a Map

```
public Object getRawBody();
```

1.10.14 Custom PoJo mapping

In case you want to do custom mapping, the platform will carry out best effort mapping from the source PoJo to the target PoJo. You must ensure the target object is compatible with the source PoJo. Otherwise, there will be data lost or casting error.

```
public <T> T getBody(Class<T> toValueType); // <-- custom mapping
public <T> T getBody(Class<T> toValueType, Class<?>... parameterClass); // custom␣
↪generics
```

1.10.15 Check if a target service is available

To check if a target service is available, you can use the `exists` method.

```
boolean po.exists(String... route);

if (po.exists("hello.world")) {
    // do something
}

if (po.exists("hello.math", "v1.diff.equation")) {
    // do other things
}
```

This service discovery process is instantaneous using distributed routing table.

1.10.16 Get a list of application instances that provide a certain service

The search method is usually used for leader election for a certain service if more than one app instance is available.

```
List<String> originIDs = po.search(String route);
```

1.10.17 Pub/Sub for store-n-forward event streaming

Mercury provides real-time inter-service event streaming and you do not need to deal with low-level messaging.

However, if you want to do store-n-forward pub/sub for certain use cases, you may use the PubSub class.

In event streaming, pub/sub refers to the long term storage of events for event playback or "time rewind". For example, this "commit log" architecture is available in Apache Kafka.

To test if the underlying event system supports pub/sub, use the "isStreamingPubSub" API.

Following are some useful pub/sub API:

```
public boolean featureEnabled();
public boolean createTopic(String topic) throws IOException;
public boolean createTopic(String topic, int partitions) throws IOException;
public void deleteTopic(String topic) throws IOException;
public void publish(String topic, Map<String, String> headers, Object body) throws␣
↪IOException;
public void publish(String topic, int partition, Map<String, String> headers, Object␣
↪body) throws IOException;
public void subscribe(String topic, LambdaFunction listener, String... parameters)␣
↪throws IOException;
public void subscribe(String topic, int partition, LambdaFunction listener, String...␣
↪parameters) throws IOException;
public void unsubscribe(String topic) throws IOException;
public void unsubscribe(String topic, int partition) throws IOException;
public boolean exists(String topic) throws IOException;
public int partitionCount(String topic) throws IOException;
public List<String> list() throws IOException;
public boolean isStreamingPubSub();
```

Some pub/sub engine would require additional parameters when subscribing a topic. For Kafka, you must provide the following parameters

1. clientId
2. groupId
3. optional read offset pointer

If the offset pointer is not given, Kafka will position the read pointer to the latest when the clientId and groupId are first seen. Thereafter, Kafka will remember the read pointer for the groupId and resume read from the last read pointer.

As a result, for proper subscription, you must create the topic first and then create a lambda function to subscribe to the topic before publishing anything to the topic.

To read the event stream of a topic from the beginning, you can set offset to "0".

The system encapsulates the headers and body (aka payload) in an event envelope so that you do not need to do serialization yourself. The payload can be PoJo, Map or Java primitives.

1.10.18 Thread safety

The "handleEvent" is the event handler for each LamdbaFunction. When you register more than one worker instance for your function, please ensure that you use "functional scope" variables instead of global scope variables.

If you must use global scope variables, please use Java Concurrent collections.

1.10.19 Exception handling

When your lambda function throws exception, the exception will be encapsulated in the result event envelope to the calling function.

If your calling function uses RPC, the exception will be caught as an AppException. You can then use the exception's getCause() method to retrieve the original exception chain from the called function.

If your calling function uses CALLBACK pattern, your callback function can inspect the incoming event envelope and use the getException() method to obtain the original exception chain from the called function.

1.11 Rest Automation

REST automation turns HTTP requests and responses into events for truly non-blocking operation.

1.11.1 Library and Application

The REST automation system consists of a library and an application. This sub-project is the library.

1.11.2 REST endpoint creation

It allows us to create REST and websocket endpoints by configuration instead of code.

Let us examine the idea with a sample configuration.

1.11.3 REST endpoint routing

A routing entry is a simple mapping of URL to a microservice.

1.11.4 YAML syntax

```
rest:
# service should be a target service name or a list of service names
# If more than one service name is provided, the first one is the primary target
# and the rest are secondary target(s). The system will copy the HTTP request event to␣
↪the secondary targets.
#
# This feature is used for seamless legacy system migration where we can send
# the same request to the old and the new services at the same time for A/B comparison.
- service: ["hello.world", "hello.world.2"]
    methods: ['GET', 'PUT', 'POST', 'HEAD', 'PATCH', 'DELETE']
    url: "/api/hello/world"
```

(continues on next page)

(continued from previous page)

```
    timeout: 10s
    # Optional authentication service which should return a true or false result
    # The authentication service can also add session info in headers using␣
↪EventEnvelope as a response
    # and annotate trace with key-values that you want to persist into distributed trace␣
↪logging.
    #
    # You can also route the authentication request to different functions based on HTTP␣
↪request header
    # i.e. you can provide a single authentication function or a list of functions␣
↪selected by header routing.
    #
    # If you want to route based on header/value, use the "key: value : service" format.
    # For routing using header only, use "key: service" format
    # For default authentication service, use "default: service" format
    #
    #    authentication: "v1.api.auth"
    authentication:
    - "x-app-name: demo : v1.demo.auth"
    - "authorization: v1.basic.auth"
    - "default: v1.api.auth"
    cors: cors_1
    headers: header_1
    # for HTTP request body that is not JSON/XML, it will be turned into a stream if it␣
↪is undefined
    # or larger than threshold. Otherwise, it will be delivered as a byte array in the␣
↪message body.
    # Default is 50000 bytes, min is 5000, max is 500000
    threshold: 30000
    # optionally, you can turn on Distributed Tracing
    tracing: true

- service: "hello.world"
    methods: ['GET', 'PUT', 'POST']
    url: "/api/test/ok*"
    # optional "upload" key if it is a multi-part file upload
    upload: "file"
    timeout: 15s
    headers: header_1
    cors: cors_1

- service: "hello.world"
    methods: ['GET', 'PUT', 'POST']
    url: "/api/nice/{task}/*"
    timeout: 12
    headers: header_1
    cors: cors_1

#
# When service is a URL, it will relay HTTP or HTTPS requests.
# "trust_all_cert" and "url_rewrite" are optional.
#
```

(continues on next page)

(continued from previous page)

```
# For target host with self-signed certificate, you may set "trust_all_cert" to true.
# trust_all_cert: true
#
# "url_rewrite", when present as a list of 2 strings, is used to rewrite the url.
# e.g. url_rewrite: ['/api/v1', '/v1/api']
# In this example, "/api/v1" will be replaced with "/v1/api"
#
- service: "http://127.0.0.1:8100"
    trust_all_cert: true
    methods: ['GET', 'PUT', 'POST']
    url: "/api/v1/*"
    url_rewrite: ['/api/v1', '/api']
    timeout: 20
    cors: cors_1
    headers: header_1
    tracing: true

#
# CORS HEADERS for pre-flight (HTTP OPTIONS) and normal cases
#
cors:

- id: cors_1
    options:
    - "Access-Control-Allow-Origin: *"
    - "Access-Control-Allow-Methods: GET, DELETE, PUT, POST, OPTIONS"
    - "Access-Control-Allow-Headers: Origin, Authorization, X-Session-Id, Accept,␣
↪Content-Type, X-Requested-With"
    - "Access-Control-Max-Age: 86400"
    headers:
    - "Access-Control-Allow-Origin: *"
    - "Access-Control-Allow-Methods: GET, DELETE, PUT, POST, OPTIONS"
    - "Access-Control-Allow-Headers: Origin, Authorization, X-Session-Id, Accept,␣
↪Content-Type, X-Requested-With"
    - "Access-Control-Allow-Credentials: true"

#
# add/drop/keep header parameters
#
headers:

- id: header_1
    # headers to be inserted
    add: ["hello-world: nice"]
#    keep: ['x-session-id', 'user-agent']
    drop: ['Upgrade-Insecure-Requests', 'cache-control', 'accept-encoding']
```

The `service` must be a service that is registered as public function.

The system will find the service at run-time. If the service is not available, the user will see HTTP-503 "Service not reachable".

The `keep` and `drop` entries are mutually exclusive where keep has precedent over drop. When `keep` is empty and `drop` is not, it will drop only the headers in the drop list. The `add` entry allows the developer to insert additional header

key-values before it reaches the target service that serves the REST endpoint.

For content types of XML and JSON, the system will try to convert the input stream into a map.

For binary content type, the system will check if the input stream is larger than a threshold. `threshold` - the default threshold buffer is 50,000 bytes, min is 5,000 and max is 500,000

If the input stream is small, it will convert the input stream into a byte array and store it in the "body" parameter.

if the input stream is large, it will return a stream ID so that your service can read the data as a stream.

Cors header processing is optional. If present, it will return the configured key-values for pre-flight and regular responses.

1.11.5 URL matching

For performance and readability reason, the system is not using regular expression for URL matching. Instead it is using wildcard and path parameters.

wildcard

The wildcard * character may be used as a URL path segment or as a suffix in a segment. Characters after the wildcard character in a segment will be ignored.

e.g. the following are valid wildcard URLs

`/api/hello/world/*` `/api/hello*/world*/*`

path parameters

The {} bracket can be used to indicate a path parameter. The value inside the bracket is the path parameter ID.

For example, the following URL will set path parameter "first_name" and "last_name".

`/api/hello/{first_name}/{last_name}`

1.11.6 HTTP request dataset

The HTTP request (headers, query, body) will be encoded as a Map and the target service will receive it as the message body.

```
headers: key-values of HTTP headers
cookies: optional key-values
method: HTTP request method
ip: caller's remote IP address
parameters:
path: optional path parameters
query: optional query or URL encoded request body parameters
url: URL of the incoming request
timeout: timeout value (in seconds) of the REST endpoint

# optional (body or stream)
body: byte array of input data if it is smaller than a given threshold
stream: stream-ID of incoming data stream (e.g. file)
```

(continues on next page)

(continued from previous page)

```
filename: filename if it is a file upload
content-length: number of bytes of the incoming stream
```

For Java, you can use AsyncHttpRequest as a convenient wrapper to read this dataset.

```
AsyncHttpRequest request = new AsyncHttpRequest(body);
```

Http response body

For simple use case, you can just send text, bytes or Map as the return value in your service.

For advanced use cases, you can send status code, set HTTP headers if your service returns an EventEnvelope object.

HTTP status code

If your service set status code directly, you should use the standard 3-digit HTTP status code because it will be sent to the browser directly. i.e. 200 means OK and 404 is NOT_FOUND, etc.

Setting cookies

You can ask a browser to set a cookie by setting the "Set-Cookie" key-value in an EventEnvelope's header which will be converted to a HTTP response header. For details, please see https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie

There is a convenient "getHtmlDate()" method in the Utility class if you want to set the "Expires" directive in a cookie.

Exception handling

If your service throws exception, it will automatically translate into a regular HTTP exception with status code and message.

To control the HTTP error code, your service should use the AppException class. It allows you to set HTTP status code and error message directly.

Input stream

If you want to support file upload, you can specify the upload parameter in the rest.yaml configuration file. The default value is "file". If your user uses different tag, you must change the upload parameter to match the upload tag.

If incoming request is a file, your service will see the "stream", "filename" and "content-length" parameters. If incoming request is a byte stream, your service will find the "stream" and "content-length" parameters.

Output stream

If your service wants to send the output to the browser as a stream of text or bytes, you can create an ObjectStreamIO with a timeout value. You can then set the streamId in the HTTP header "stream". You should also set the HTTP header "timeout" to tell the REST endpoint to use it as IO stream read timeout value. The "stream" and "timeout" headers are used by the REST automation framework. They will not be set as HTTP response headers.

You timeout value should be short and yet good enough for your service to send one block of data. The timer will be reset when there is I/O activity. One use case of output stream is file download.

Location of the YAML configuration file By default, the system will search for `file:/tmp/config/rest.yaml` and then `classpath:/rest.yaml`.

If needed, you can change this in the `rest.automation.yaml` property in the application.properties file.

The `/tmp/config/rest.yaml` allows you to externalize the REST automation configuration without rebuilding the REST automation helper application.

Running the REST automation helper application

Before you build this application, you should follow the README in the top level project to build the Mercury libraries (platform-core, rest-core, rest-spring, kafka-connector, hazelcast-connector).

Then you can build this helper app and run with:

```
java -Dcloud.connector=event.node target/rest-automation...jar
```

If you plan to use Hazelcast or Kafka as the network event stream system, you should also build the corresponding presence monitor (hazelcast-presence and kafka-presence).

Once the Mercury libraries and the presence monitors are built. You can start either Hazelcast or Kafka and build this helper app.

You can run the app with:

```
java -Dcloud.connector=event.node target/rest-automation...jar
or
java -Dcloud.connector=hazelcast -Dcloud.services=hazelcast.reporter target/rest-
↪automation...jar
or
java -Dcloud.connector=kafka -Dcloud.services=kafka.reporter target/rest-automation...jar
```

1.11.7 WebSocket automation

You can define a websocket endpoint for 2 purposes:

- a websocket server service that you write
- a websocket notification service for UI applications

Websocket server service

To deploy a websocket endpoint for your own application, add this "websocket" section to the rest.yaml config file. You also need to update the "rest" section to expose a websocket access token issurance endpoint. You should implement your websocket authentication API. In this example, it is "v1.ws.api.auth". For testing, you can comment out the authentication portion.

```
websocket:
- application: "notification"
    recipient: "my.ws.handler"

rest:
# The REST automation system includes a built-in websocket notification system.
# If you want to enable this feature, you must add the ws.token.issuer service as
# shown in this example. For added security, please enable authentication and point
# it to a user defined authentication service that may authenticate the user credentials
# and validate that the user is allowed to use a specific websocket application.
#
# the "ws.token.issuer" and "ws.notification" are reserved by the system for
# websocket connection token issuance and notification topic query services
#
- service: "ws.token.issuer"
    methods: ['GET']
    url: "/api/ws/token/{application}"
    #    authentication: "v1.ws.api.auth"
    tracing: true
```

This asks the REST automation to route incoming websocket connection, message and close events to your function. In the above example, it is "my.ws.handler" that points to your custom function.

```
LambdaFunction myWsHandler = (headers, body, instance) -> {
// your custom websocket server logic here to handle open, close and message events
//
// nothing to return because this is asynchronous
return null;
};
platform.register("my.ws.handler", myWsHander, 1);
```

Websocket open event

An open event contains a header of "type" = "open". The event body contains the following:

```
{
"query": {"key": "value"},
"application": "notification",
"ip": "192.168.1.100",
"origin": "ddd3bca7e7744a67a1b938dc67a76cd7",
"tx_path": "ws.12345@ddd3bca7e7744a67a1b938dc67a76cd7"
}
```

To connect to your websocket server function, the UI may issue a websocket connection request to the following URL:

wss://hostname/ws/api/notification:{access_token}?optional_query_string The application name "notification" is an example only. You can define any application name in the rest.yaml config file.

Note that your websocket server handler function will receive all websocket connection to the specific websocket application. Please implement logic to handle individual user which is identified by the "tx_path".

UI keep-alive

Websocket connection is persistent. To release unused resources, REST automation will disconnect any idle websocket connection in 60 seconds. Please implement keep-alive by sending a "hello" message from the UI like this:

```
{
"type": "hello",
"message": "keep-alive",
"time": "ISO-8601 time-stamp"
}
```

The REST automation will echo this "hello" message to the UI where it can be ignored.

Websocket message event

For simplicity, the REST automation system supports TEXT message only.

The event's body contains the incoming text message and the headers contains the following:

```
{
"type": "message",
"application": "notification",
"origin": "ddd3bca7e7744a67a1b938dc67a76cd7",
"tx_path": "ws.12345@ddd3bca7e7744a67a1b938dc67a76cd7"
}
```

The "tx_path" is the outgoing route name for your application to send text messages. e.g.

Payload can be Map or String where Map payload will be converted to a JSON string for delivery

```
po.send("ws.12345@ddd3bca7e7744a67a1b938dc67a76cd7", payload);
```

Websocket close event

A close event contains a header of "type" = "close". The event body contains the following:

```
{
"application": "notification",
"origin": "ddd3bca7e7744a67a1b938dc67a76cd7",
"tx_path": "ws.12345@ddd3bca7e7744a67a1b938dc67a76cd7"
}
```

If your websocket server function creates temporary resource, you may release the resource using the "tx_path" as a reference.

WebSocket is usually employed as a notification channel to the browser so that your service can detect "presence" of the user and asynchronously send notification events to the browser.

The REST automation helper application supports this websocket notification use case. The sample rest.yaml configuration file contains a websocket routing entry to the sample.ws.auth and ws.notification services.

Using websocket for simple notification to the browser

You can remove the "recipient" and add the publish/subscibe features in the rest.yaml config file like this:

```
websocket:
- application: "notification"
    publish: true
    subscribe: true

rest:
- service: "ws.token.issuer"
    methods: ['GET']
    url: "/api/ws/token/{application}"
    #    authentication: "v1.ws.api.auth"
    tracing: true
- service: "ws.notification"
    methods: ['GET']
    url: "/api/notification"
    #    authentication: "v1.api.auth"
    tracing: true
- service: "ws.notification"
    methods: ['GET']
    url: "/api/notification/{topic}"
    #    authentication: "v1.api.auth"
    tracing: true
```

The "subscribe" feature must be set to `true` for the browser to subscribe to one or more notification topics. The "publish" feature, if turn on, allows peer-to-peer messaging. For security, we recommend to set it to false. You can expose a REST endpoint for a user to send events through a backend service.

The `/api/notification` endpoints are for admin purpose if you want to expose them to DevOps. The two admin endpoints show a list of all topics or a list of websocket connections under a specific topic respectively.

Subscribe to a notification topic

The browser can send a subscription request like this:

Unsubscribe from a notification topic

The browser can unsubscribe from a topic like this:

```
{
"type": "unsubscribe",
"topic": "my.simple.topic"
}
```

A browser will also automatically unsubscribe from all subscribed topics when the browser closes. When the connected websocket backend service application instance fails, the websocket connection to the browser will be closed and current subscriptions will be dropped. The browser application should acquire a websocket access token and reconnect to an available backend service instance. Then subscribe to the topic(s) again.

Note that a browser can subscribe to more than one notification topics. e.g. system.alerts, user.120, workflow.100, etc.

Notification topic vs service route name

While both notification topics and service route names use the same convention of lower case and "dots", they are maintained in different registries and thus there is no conflict between the two types of names.

Publish from a browser

If "publish" feature is turned on, the browser can send text events to a topic like this:

```
{
"type": "publish",
"topic": "my.simple.topic",
"message": "text message here"
}
```

Publish from a backend service

For security, we recommend disabling the publish feature in rest.yaml and perform publishing from a backend service.

There is a convenient class "UserNotification" in the platform core library that provides the following APIs:

```
public void publish(String topic, String message);
public List<String> listTopics() throws TimeoutException, AppException, IOException;
public Map<String, List<String>> getTopic(String topic) throws TimeoutException,␣
↪AppException, IOException;
```

Sample browser application for connecting to a notification channel

The ws.html sample app is available under the resources folder and the browser app can be tested by visiting http://127.0.0.1:8100/ws.html

Custom HTML error page

You may customize the standardized errorPage.html in the resources folder for your organization.

Static HTML folder

You can tell the rest-automation application to use a static HTML folder in the local file system with one of these methods:

application.properties

```
spring.resources.static-locations=file:/tmp/html
```

or startup parameters

```
java -jar rest-automation.jar -html file:/tmp/html
```

1.11.8 API definition file (rest.yaml)

Please design and deploy your own rest.yaml file in /tmp/config/rest.yaml

The rest.yaml in "/tmp/config" will override the sample rest.yaml in the resources folder.

If your application does not support websocket notification channel, you can remove the websocket section in rest.yaml

1.12 JAX-RS and WebServlets

The recommended way to create REST endpoints is the REST automation system described in chapter 4.

However, if you want to write REST and websocket endpoints in traditional ways, Mercury supports the following for backward compatibility.

1. JAX-RS annotation
2. Spring REST controller
3. Java Servlet

1.12.1 JAX-RS annotated endpoints

You class should include the `@Path(String pathPrefix)` annotation. Set pathPrefix accordingly.

All JAX-RS REST endpoints are prefixed with "/api". If your pathPrefix is "/hello". The path is "/api/hello".

In your REST endpoint method, you should use another `@Path` to indicate any additional path information. Note that you can use `{pathParam}` in the URL path. If you have pathParam, please add `@PathParam` parameter annotation to the argument in your method.

You can set the HTTP method in the method with `@GET`, `@POST`, `@PUT`, `@DELETE`, etc. and specify content types in @Consumes and @Produces method annotation.

If you want your REST endpoint to be optional when a property in the application.properties exists, annotate your class with OptionalService.

Please refer to example code in `rest-example` and the JAX-RS reference site for details.

Java Servlet

For low level control, you can use the `@WebServlet(String urlPath)` to annotate your Servlet class that must extend the `HttpServlet` class.

If you want your Java Servlet to be optional when a property in the application.properties exists, annotate your class with `OptionalService`.

Spring REST controller

If you are more familar with the Spring framework, you may use Spring REST controller. Note that this will create tight coupling and make your code less portable.

Please refer to Spring documentation for details. Note that Spring REST controllers are installed in the URL root path.

1.12.2 Websocket service

You can use `@WebSocketService(handlerName)` to annotate a websocket service that implements the LambdaFunction interface. Your websocket service function becomes the incoming event handler for websocket. The system will automatically create an outgoing message handler that works with your websocket service.

Please refer to the sample code `WsEchoDemo` in the `rest-example`.

Mercury is 100% event-driven. This includes websocket service. The sample code includes example for OPEN, CLOSE, BYTES and TEXT events. In the OPEN event, you can detect the query parameter and token.

For standardization, websocket service uses a URL path as follows:

```
ws://host:port/ws/{handlerName}/{token}
```

The websocket client application or browser that connects to the websocket service must provide the "token" which is usually used for authentication.

IMPORTANT: websocket does not support custom HTTP headers for authentication. As a result, we usually use an authentication token in the URL or query parameter. For the Mercury framework, we recommend the use of a token as a URL suffix. Typically, a user logs on to your application with a REST endpoint like "/login" and then a session cookie is created. The browser may then use the sessionId as a token in the websocket connection.

`Websocket idle timeout` - there is a one-minute idle timeout for all websocket connection. To keep the websocket connection alive, you may send a message to the websocket service. For example, sending a BYTES or TEXT message to the service with some agreed protocol.

To disconnect a websocket connection, you may use the Utility class as follows:

```
void closeConnection(String txPath, CloseReason.CloseCodes status, String message) throws
IOException;
```

The txPath is available to the websocket service mentioned earlier.

1.12.3 Calling other microservices functions

You may call other microservices functions from your REST and websocket endpoints using the send or request methods of the Post Office.

1.12.4 Static contents for HTML, CSS and Javascript

Under the hood, we are using Spring Boot. Therefore you may put static contents under the "/public" folder in the project's "resources". The static contents will be bundled with your executable application JAR when you do mvn clean package.

1.12.5 Loose coupling with Spring Boot

The Mercury framework is loosely coupled with Spring Boot to support REST and websocket endpoints. The light-weight application server for Spring Boot can be Tomcat, Jetty or Undertow.

For consistency, we have optimized Spring Boot with custom serializers and exception handlers in the `rest-spring` module.

If you know what you are doing, you can use Spring Boot feature directly with the exception of the `@SpringApplication` annotation because we use the `@MainApplication` to enable additional automation. You can change the behavior of Spring Boot including auto-config classes using the `application.properties` file in the resources folder in the maven project.

We are currently using Tomcat. If your organization prefers Jetty or Undertow, you may adjust the pom.xml file in the `rest-spring` and `platform-core` projects.

1.12.6 application.properties

In additon to the parameters defined by Spring Boot, the Mercury framework uses the following parameters.

1. web.component.scan - you should add your organizaton packages as a comma separated list to tell Mercury to scan for your packages.
2. snake.case.serialization - we recommend the use of snake case for modern API
3. safe.data.models - Optional. For higher security, you may specify a list of safe data models to be accepted by the serialization engine. This is to protect against hidden "evil" Java classes in certain open sources that you have not vetted directly.
4. protected.info.endpoints - Optional. You may protect certain "known" REST endpoints such as "/info" or "/env" from unauthorized access. It uses a simple API key set in the environment.
5. env.variables - parameters in the application.properties are automatically overriden by Java properties. To allow some environment variables to override your run-time parameters, you may define them in this parameter.
6. spring.application.name/application.name, info.app.version and info.app.description - please update application name and information before you start your project. spring.application.name and application.name can be used interchangeably.

1.13 Cloud Connectors

Mercury has been integrated and tested with both event stream systems and enterprise service bus messaging systems.

1. Event stream system - Apache Kafka
2. Messaging system - Hazelcast
3. Enterprise Service Bus - ActiveMQ artemis and Tibco EMS

1.13.1 Reusable topics

The default topics include the following:

1. service.monitor.n
2. multiplex.x.y

where `service.monitor.0` is used by the presence monitor to communicate with its peers for fault tolerant operation. service.monitor.1 and higher topics are used for closed user groups. i.e. service.monitor.1 for closed user group 1 and service.monitor.2 for closed user group 2, etc.

Usually all user application instances should use the same closed user group unless you want to logically segregate different application domains into their own closed user groups. Application modules in one group are invisible to another group.

multiplex.x.y topics are used by user application instances. The presence monitor will assign one topic to one application instance dynamically and release the topic when the application instance leaves the system. This allows topics to be reused automatically.

There is a RSVP reservation protocol that the presence monitors will coordinate with each other to assign a unique topic to each application instance.

The value `x` must be a 4-digit number starting from 0001 and `y` is the partition number for the topic. If the underlying messaging system supports pub/sub, the partition number maps to the physical partition of a topic. For enterprise service bus, the partition number is a logical identifier that corresponds to a physical topic.

1.13.2 Topic Substitution

Some enterprises do not allow automatic creation of messaging topics by user applications.

In this case, you can turn on topic substitution with the following parameters in application.properties

```
application.feature.topic.substitution=true
topic.substitution.file=file:/tmp/config/topic-substitution.yaml,classpath:/topic-
↪substitution.yaml
```

You can point the `topic.substitution.file` to a file.

Supported cloud connectors

Kafka, ActiveMQ and Tibco support mapping of system topics to pre-allocated topics. Hazelcast topics are generated dynamically and thus topic pre-allocation is not required.

Let's examine the configuration concept with two sample topic substitution files.

ActiveMQ and Tibco

```
#
# topic substitution example
#
# The default service monitor and multiplex topics are prefixed as service.monitor and␣
↪multiplex.n
# where n starts from 0001
#
```

(continues on next page)

(continued from previous page)

```
service:
monitor:
    0: some.topic.one
    1: some.topic.two
    2: some.topic.three
    3: some.topic.four

#
# the 4-digit segment ID must be quoted to preserve the number of digits when the system␣
↪parses this config file
#
multiplex:
"0001":
    0: user.topic.one
    1: user.topic.two
    2: user.topic.three
    3: user.topic.four
    4: user.topic.five
    5: user.topic.six
```

Kafka

Since Kafka is a pub/sub event streaming system with partitioning support, you can use the "#n" suffix to specify the partition number for each replacement topic to map to the system topics (service.monitor.n and multiple.x.y). (Note that the "#n" syntax is not applicable to ActiveMQ and Tibco connectors above)

If you do not provide the partition number, it is assumed to be the first partition. i.e. partition-0.

```
#
# topic substitution example
#
# The default service monitor and multiplex topics are prefixed as service.monitor and␣
↪multiplex.n
# where n starts from 0001
#
# PARTITION DEFINITION
# --------------------
# For the replacement topic, an optional partition number can be defined using the "#n"␣
↪suffix.
# Partition number must be a positive number from 0 to the max partition of the specific␣
↪topic.
#
# If partition number if not given, the entire topic will be used to send/receive events.
# e.g.
# SERVICE.APP.ONE is a topic of at least one partition
# user.app#0 is the topic "user.app" and partition "0"
#
service:
monitor:
    0: SERVICE.APP.ONE#0
    1: SERVICE.APP.TWO#0
```

(continues on next page)

(continued from previous page)

```
    2: SERVICE.APP.THREE#0
    3: SERVICE.APP.FOUR#0

#
# the 4-digit segment ID must be quoted to preserve the number of digits when the system␣
↪parses this config file
#
multiplex:
"0001":
    0: user.app#0
    1: user.app#1
    2: user.app#2
    3: user.app#3
    4: user.app#4
    5: user.app#5
```

1.14 Multicast

Multicast is a convenient feature that relays an event from a route to multiple targets.

It is configured using a multicast.yaml file like this:

```
multicast:
- source: "v1.hello.world"
    targets:
    - "v1.hello.service.1"
    - "v1.hello.service.2"
    - "v1.hello.service.3"
```

In the above example, the route "v1.hello.world" will be automatically multicasted to the target services with the configured routes.

1.14.1 application.properties

To enable multicast, please add the following to application.properties

`multicast.yaml=classpath:/multicast.yaml`

If you want to externalize the multicast.yaml configuration file, you can change the parameter like this:

`multicast.yaml=file:/tmp/config/multicast.yaml`

1.14.2 Routing behavior

Multicast is designed for best effort delivery. If a target route is not reachable, it will display a warning in the application log like this:

```
Unable to relay v1.hello.world -> v1.hello.service.3 - target not reachable
```

1.14.3 Real-time vs store-n-forward

The PostOffice is a real-time event system. For store-n-forward use case, please refer to streaming Pub/Sub APIs in Post Office API

1.15 Additonal Features

1.15.1 Admin endpoints to stop, suspend or resume

You can stop, suspend or resume an application instance from a `presence monitor`.

1. `Shutdown` - stop an application so that the container management system will restart it
2. `Suspend` - tell the application instance not to accept incoming requests
3. `Resume` - tell the application instance to accept incoming requests

Suspend and resume commands are usually used to simulate error cases for development and regression test purposes.

For example, to simulate a stalled application instance, you can use the "POST /suspend/later" command.

If you do not want your application instance to receive any service request, you can isolate it with the "POST /suspend/now" command.

```
POST /shutdown
POST /suspend/{now | later}
POST /resume/{now | later}

HTTP request header
X-App-Instance=origin_id_here
```

1.15.2 Actuator endpoints

The following admin endpoints are available.

```
GET /info
GET /info/routes
GET /info/lib
GET /env
GET /health
GET /livenessprobe

Optional HTTP request header
X-App-Instance=origin_id_here
```

If you provide the optional X-App-Instance HTTP header, you can execute the admin endpoint from `any application instance` using the event stream system.

1.15.3 Custom health services

You can extend the "/health" endpoint by implementing and registering lambda functions to be added to the health.dependencies.

```
mandatory.health.dependencies=cloud.cache.health,cloud.connector.health
#optional.health.dependencies=other.service.health
```

Your custom health service must respond to the following requests:

1. Info request (type=info) - it should return a map that includes service name and href (protocol, hostname and port)
2. Health check (type=health) - it should return a text string of the health check. e.g. read/write test result. It can throw AppException with status code and error message if health check fails.

The health service can retrieve the "type" of the request from the "headers".

1.15.4 Application instance life-cycle events

Any application can subscribe to life-cycle events of other application instances. Sample code is available in extensions/rest-automation-lib/src/main/java/org/platformlambda/automation/services/NotificationManager.java

To listen to life cycle events, you can do something like this:

```
String AUTOMATION_NOTIFICATION = "member.life.cycle.listener";
String appName = platform.getName();
LambdaFunction f = (headers, body, instance) -> {
    if (CONNECTED.equals(type)) {
        // handle connection event - this event is fired when your app is connected to␣
↪the event stream system
        log.info("connected");
    }
    if (DISCONNECTED.equals(type)) {
        // handle disconnection event - this event is fired when your app is␣
↪disconnected from the event stream system
        log.info("disconnected");
    }
    if (JOIN.equals(type) && headers.containsKey(ORIGIN) && appName.equals(headers.
↪get(NAME))) {
        // handle member join event
    }
    if (LEAVE.equals(type) && headers.containsKey(ORIGIN)) {
        // handle member leave event
    }
}
platform.registerPrivate(AUTOMATION_NOTIFICATION, f, 1);

...
EventEnvelope event = new EventEnvelope()
                        .setTo(AUTOMATION_NOTIFICATION).setHeader(TYPE, SUBSCRIBE_LIFE_
↪CYCLE);
po.sendLater(event, new Date(System.currentTimeMillis() + 2000));
```

1.15.5 HttpClient as a service

Starting from version 1.12.30, the rest-automation system, when deployed, will provide the "async.http.request" service.

This means you can make a HTTP request without using a HttpClient.

For example, a simple HTTP GET request may look like this:

```
// the target URL is constructed from the relay
PostOffice po = PostOffice.getInstance();
AsyncHttpRequest req = new AsyncHttpRequest();
req.setMethod("GET");
req.setHeader("accept", "application/json");
req.setUrl("/api/search?keywords="+body);
req.setTargetHost("https://service_provider_host");
try {
    EventEnvelope res = po.request("async.http.request", 5000, req.toMap());
    log.info("GOT {} {}", res.getHeaders(), res.getBody());
    /*
    * res.getHeaders() contains HTTP response headers
    * res.getBody() is the HTTP response body
    *
    * Note that the HTTP body will be provided as be set a HashMap
    * if the input content-type is application/json or application/xml.
    */
    // process HTTP response here (HTTP-200)

} catch (AppException e) {
    log.error("Rejected by service provider HTTP-{} {}",
            e.getStatus(), e.getMessage().replace("\n", ""));
    // handle exception here
}
```

In the above example, we are using RPC method. You may also use callback method for handling the HTTP response.

1.15.6 Sending HTTP request body for HTTP PUT, POST and PATCH methods

For most cases, you can just set a HashMap into the request body and specify content-type as JSON or XML. The system will perform serialization properly.

Example code may look like this:

```
AsyncHttpRequest req = new AsyncHttpRequest();
req.setMethod("POST");
req.setHeader("accept", "application/json");
req.setHeader("content-type", "application/json");
req.setUrl("/api/book/new_book/12345");
req.setTargetHost("https://service_provider_host");
req.setBody(keyValues);
// where keyValues is a HashMap
```

1.15.7 Sending HTTP request body as a stream

For larger payload, you may use streaming method. See sample code below:

```
int len;
byte[] buffer = new byte[BUFFER_SIZE];
BufferedInputStream in = new BufferedInputStream(someFileInputStream);
ObjectStreamIO stream = new ObjectStreamIO(timeoutInSeconds);
ObjectStreamWriter out = stream.getOutputStream();
while ((len = in.read(buffer, 0, buffer.length)) != -1) {
    out.write(buffer, 0, len);
}
// closing the output stream would save an EOF mark in the stream
out.close();
// update the AsyncHttpRequest object
req.setStreamRoute(stream.getRoute());
```

1.15.8 Handle HTTP response body stream

If content length is not given, the response body will be received as a stream.

Your application should check if the HTTP response headers contains a "stream" header. Sample code to read the stream may look like this:

```
PostOffice po = PostOffice.getInstance();
AsyncHttpRequest req = new AsyncHttpRequest();
req.setMethod("GET");
req.setHeader("accept", "application/json");
req.setUrl("/api/search?keywords="+body);
req.setTargetHost("https://service_provider_host");
EventEnvelope res = po.request("async.http.request", 5000, req.toMap());
Map<String, String> resHeaders = res.getHeaders();
if (resHeaders.containsKey("stream")) {
    ObjectStreamIO consumer = new ObjectStreamIO(resHeaders.get("stream"));
    /*
    * For demonstration, we are using ByteArrayOutputStream.
    * For production code, you should stream the input to persistent storage
    * or handle the input stream directly.
    */
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    ObjectStreamReader in = consumer.getInputStream(1000);
    for (Object d: in) {
        if (d instanceof byte[]) {
            out.write((byte[]) d);
        }
    }
    // remember to close the input stream
    in.close();
    // handle the result
    byte[] result = out.toByteArray();
}
```

1.15.9 Content length for HTTP request

Important - Do not set the "content-length" HTTP header because the system will automatically compute the correct content-length for small payload. For large payload, it will use the chunking method.

The system may use data compression. Manually setting content length for HTTP request body would result in unintended side effects.

1.16 Version Control

Sometimes, it would be useful to deploy more than one version of the same service in an environment.

This allows us to test new versions without impacting existing functionality.

Version control is supported in the REST automation system and the Post Office event core engine.

To do that, we can add a prefix or suffix to a service route name.

For example, the service route name is originally "hello.world". The versioned services would be:

```
v1.hello.world - the current version
v2.hello.world - the next version of the service with functional improvement or bug fix
```

There are two ways to do versioning:

1. REST automation
2. Route substitution

Let's examine how to do versioning from the REST automation system.

Mercury has its documentation hosted on Read the Docs.