**Spring Cloud Contract**

**Consumer-driven contract tests:**

Consumer-driven contract tests are actually integration tests that are targetting your API, whether it’s REST-based or messaging-based. Imagine you’re working on an application that exposes its data using a REST API. Another team is using your exposed data for some functionality that they are providing. In order to guarantee that the functionality of the other team their application doesn’t break if we make changes to our API, we create a contract between the two teams.

This aproach we can generate tests by using those consumer-driven contracts, and verify whether we’re going to break any of our consumers’ applications

Microservices are eating the world! The arrival of this concept changed not only the way we're designing our software architecture, but also how teams are formed, how they're organized and how they work together.

One of the many other challenges that Microservices brings, is the way we test changes made on them. Martin Fowler and James Lewis Introduced on their definition of Microservices the concept of Consumer-Driven Contract Testing:

Executing consumer driven contracts as part of your build increases confidence and provides fast feedback on whether your services are functioning.

In this quick post, we'll briefly define the concept of CDC, as well as testing a Producer and consumer communicating throught HTTP, using Spring Cloud Contract.

**CDC Testing:**

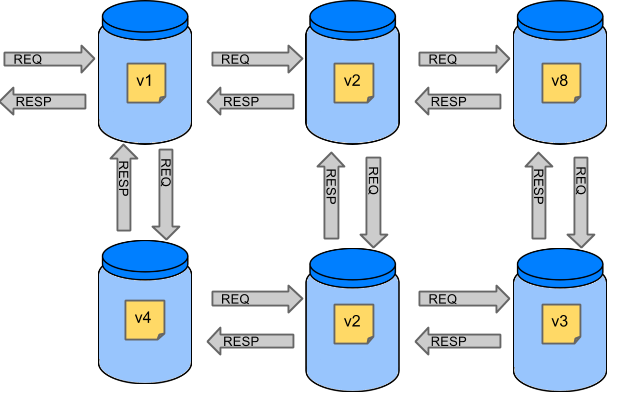
Consumer Driven Contract approach is nothing more than an agreement, to test integration points, between the Server (Consumer) and Client (Provider) about the format of data that they communicate between each other, eliminating the hassle of end to end tests.

Spring Cloud Contract is an amazing framework that facilitates consumer driven contract tests.

Spring Cloud Contract provides us with JVM-based libraries, allowing us to generate Groovy contracts, package them as a jar, and upload them to an artifact repository like Nexus or Artifactory. Great, but that means we can only use these contracts between two parties that are using a JVM languag

**Why a Contract Verifier?**

Assume that we have a system consisting of multiple micro services:

[](https://raw.githubusercontent.com/spring-cloud/spring-cloud-contract/master/docs/src/main/asciidoc/images/Deps.png)

**Testing issues**

If we wanted to test the application in top left corner to determine whether it can communicate with other services, we could do one of two things:

* Deploy all microservices and perform end-to-end tests.
* Mock other microservices in unit/integration tests.

Both have their advantages but also many disadvantages.

Deploy all microservices and perform end to end tests

**Advantages:**

* Simulates production.
* Tests real communication between services.

**Disadvantages:**

* To test one microservice, we have to deploy 6 microservices, a couple of databases, etc.
* The environment where the tests run is locked for a single suite of tests (nobody else would be able to run the tests in the meantime).
* They take a long time to run.
* The feedback comes very late in the process.
* They are extremely hard to debug.

**Mock other microservices in unit/integration tests**

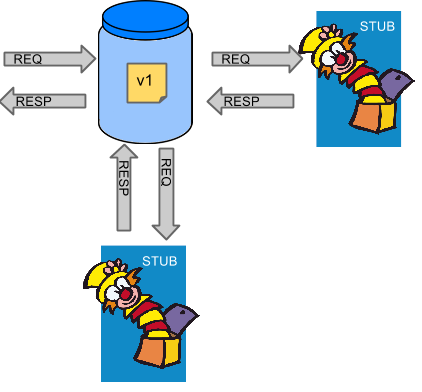
**Advantages:**

* They provide very fast feedback.
* They have no infrastructure requirements.

**Disadvantages:**

* The implementor of the service creates stubs that might have nothing to do with reality.
* You can go to production with passing tests and failing production.

To solve the abovementioned issues, Spring Cloud Contract Verifier with Stub Runner was created. The main idea is to give you very fast feedback, without the need to set up the whole world of microservices. If you work on stubs, then the only applications you need are those that your application directly uses.

[](https://raw.githubusercontent.com/spring-cloud/spring-cloud-contract/master/docs/src/main/asciidoc/images/Stubs2.png)

Spring Cloud Contract Verifier gives you the certainty that the stubs that you use were created by the service that you’re calling. Also, if you can use them, it means that they were tested against the producer’s side. In short, you can trust those stubs.

**Sample Example:**

Producer side:

Steps:

* Create spring boot project with dependency.
* Create rest Controller using spring rest docs.

Maven Dependency

spring-cloud-starter-contract-verifier

spring-boot-maven-plugin

spring-boot-starter-web

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <project xmlns=*"http://maven.apache.org/POM/4.0.0"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>  <modelVersion>4.0.0</modelVersion>  <groupId>com.consumer.contract.driven</groupId>  <artifactId>person-cloud-contract-producer</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>jar</packaging>  <name>person-cloud-contract-producer</name>  <description>Demo project for Spring Boot</description>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.4.RELEASE</version>  <relativePath /> <!-- lookup parent from repository -->  </parent>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  <spring-cloud.version>Finchley.SR1</spring-cloud.version>  <spring-cloud-contract.version>1.2.1.RELEASE</spring-cloud-contract.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-contract-verifier</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>${spring-cloud.version}</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  <plugin>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-contract-maven-plugin</artifactId>  <version>${spring-cloud-contract.version}</version>  <extensions>true</extensions>  <configuration>  <baseClassForTests>com.consumer.contract.driven.BaseTestClass</baseClassForTests>  </configuration>  </plugin>  </plugins>  </build>  </project> |

[*spring-cloud-contract-maven-plugin*](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-contract-maven-plugin%22) with the name of our base test class, which we’ll describe in the next section:

|  |
| --- |
| <plugin>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-contract-maven-plugin</artifactId>  <version>${spring-cloud-contract.version}</version>  <extensions>true</extensions>  <configuration> <baseClassForTests>com.consumer.contract.driven.BaseTestClass</baseClassForTests>  </configuration>  </plugin> |

**Maven plugin:** it tooks groovy contracts, generates the tests. So this junit tests are executed other junit tests.

Changes happen in producer controller, this time contracts violated.

**Contract grovvy file:**

Groovy file describes the contracts

Spring cloud contract brilliant because it takes the description of contract, and performs two things.

* Generate the junit tests verifies the contracts is implemented..
* Generates wiremock (server) stubs

**Filename:** find\_person\_by\_id.groovy

|  |
| --- |
| import org.springframework.cloud.contract.spec.Contract  Contract.make {  description "should return person by id=2"  request {  url "/person/2"  method GET()  }  response {  status 200  headers {  contentType applicationJson()  }  body (  id: 2,  name: "Nagendra",  surname: "Mekala"  )  }  } |

When we run the build, **the plugin automatically generates a test class named***ContractVerifierTest***that extends our***BaseTestClass* and puts it in */target/generated-test-sources/contracts/*

|  |
| --- |
| **package** com.consumer.contract.driven;  **import** com.consumer.contract.driven.BaseTestClass;  **import** com.jayway.jsonpath.DocumentContext;  **import** com.jayway.jsonpath.JsonPath;  **import** io.restassured.module.mockmvc.specification.MockMvcRequestSpecification;  **import** io.restassured.response.ResponseOptions;  **import** org.junit.Test;  **import** **static** com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson;  **import** **static** io.restassured.module.mockmvc.RestAssuredMockMvc.\*;  **import** **static** org.springframework.cloud.contract.verifier.assertion.SpringCloudContractAssertions.assertThat;  **public** **class** ContractVerifierTest **extends** BaseTestClass {  @Test  **public** **void** validate\_find\_person\_by\_id() **throws** Exception {  // given:  MockMvcRequestSpecification request = given();  // when:  ResponseOptions response = given().spec(request)  .get("/person/2");  // then:  assertThat(response.statusCode()).isEqualTo(200);  assertThat(response.header("Content-Type")).matches("application/json.\*");  // and:  DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());  assertThatJson(parsedJson).field("['name']").isEqualTo("Nagendra");  assertThatJson(parsedJson).field("['surname']").isEqualTo("Mekala");  assertThatJson(parsedJson).field("['id']").isEqualTo(2);  }  } |

**Creating Rest service using Rest Controller:**

|  |
| --- |
| package com.consumer.contract.driven.controller;  import org.springframework.web.bind.annotation.GetMapping;  import org.springframework.web.bind.annotation.PathVariable;  import org.springframework.web.bind.annotation.RestController;  import com.consumer.contract.driven.model.Person;  import com.consumer.contract.driven.service.PersonService;  @RestController  public class PersonRestController {  private final PersonService personService;  public PersonRestController(PersonService personService) {  this.personService = personService;  }  @GetMapping("/person/{id}")  public Person findPersonById(@PathVariable("id") Long id) {  return personService.findPersonById(id);  }  }  Person model Class:  package com.consumer.contract.driven.model;  public class Person {  public Person(Long id, String name, String surname) {  this.id = id;  this.name = name;  this.surname = surname;  }  private Long id;  private String name;  private String surname;  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public String getSurname() {  return surname;  }  public void setSurname(String surname) {  this.surname = surname;  }  }  Person service:  package com.consumer.contract.driven.service;  import java.util.HashMap;  import java.util.Map;  import org.springframework.stereotype.Service;  import com.consumer.contract.driven.model.Person;  @Service  public class PersonService {  private final Map<Long, Person> personMap;  public PersonService() {  personMap = new HashMap<>();  personMap.put(1L, new Person(1L, "Murali", "Mohan"));  personMap.put(2L, new Person(2L, "Nagendra", "Mekala"));  personMap.put(3L, new Person(3L, "Kumar", "Gaurav"));  }  public Person findPersonById(Long id) {  return personMap.get(id);  }  }  BaseTestClass:  package com.consumer.contract.driven;  import org.junit.Before;  import org.junit.runner.RunWith;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.boot.test.context.SpringBootTest;  import org.springframework.cloud.contract.verifier.messaging.boot.AutoConfigureMessageVerifier;  import org.springframework.test.annotation.DirtiesContext;  import org.springframework.test.context.junit4.SpringRunner;  import com.consumer.contract.driven.controller.PersonRestController;  import io.restassured.module.mockmvc.RestAssuredMockMvc;  @RunWith(SpringRunner.class)  @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)  @DirtiesContext  @AutoConfigureMessageVerifier()  public class BaseTestClass {    @Autowired  public PersonRestController personRestController;    @Before  public void setup() {  RestAssuredMockMvc.standaloneSetup(personRestController);    }  } |

## **Consumer – Client Side**

Steps:

* Create spring boot project with dependency.
* Create rest Controller using spring rest docs.

Maven Dependency:

spring-cloud-starter-contract-stub-runner

spring-restdocs-mockmvc

spring-boot-starter-web

Pom file:

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>  <modelVersion>4.0.0</modelVersion>  <groupId>com.consumer.contract.driven</groupId>  <artifactId>person-cloud-contract-consumer</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>jar</packaging>  <name>person-cloud-contract-consumer</name>  <description>Demo project for Spring Boot</description>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.4.RELEASE</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  <spring-cloud.version>Finchley.SR1</spring-cloud.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework.restdocs</groupId>  <artifactId>spring-restdocs-mockmvc</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>${spring-cloud.version}</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

The consumer side of our CDC will consume stubs generated by the producer **side** through HTTP interaction to maintain the contract, so **any changes on the producer side would break the contract**.

|  |
| --- |
| RestController:  package com.consumer.contract.driven.controller;  import org.springframework.boot.web.client.RestTemplateBuilder;  import org.springframework.web.bind.annotation.PathVariable;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RestController;  import org.springframework.web.client.RestTemplate;  import com.consumer.contract.driven.model.Person;  @RestController  public class PersonConsumerRestController {  private final RestTemplate restTemplate;  PersonConsumerRestController(RestTemplateBuilder restTemplateBuilder) {  this.restTemplate = restTemplateBuilder.build();  }  @RequestMapping("/message/{personId}")  String getMessage(@PathVariable("personId") Long personId) {  Person person = this.restTemplate.getForObject("http://localhost:8000/person/{personId}", Person.class, personId);  return "Hello " + person.getName();  }  }  Create Person model class copy producer model class |

**Stub runner set up:**

Time to configure our stub runner in junit file, which will inform our consumer of the available stubs in our local Maven repository

|  |
| --- |
| @RunWith(SpringRunner.**class**)  @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.***MOCK***)  @AutoConfigureMockMvc  @AutoConfigureJsonTesters  @AutoConfigureStubRunner(ids = "com.consumer.contract.driven:person-cloud-contract-producer:+:stubs:8090", stubsMode=StubsMode.***LOCAL***)  **public** **class** PersonConsumerApplicationTests {    @Test  **public** **void** get\_person\_from\_service\_contract() {  // given:  RestTemplate restTemplate = **new** RestTemplate();  // when:  ResponseEntity<Person> personResponseEntity = restTemplate.getForEntity("http://localhost:8090/person/2", Person.**class**);  // then:  BDDAssertions.*then*(personResponseEntity.getStatusCodeValue()).isEqualTo(200);  BDDAssertions.*then*(personResponseEntity.getBody().getId()).isEqualTo(2l);  BDDAssertions.*then*(personResponseEntity.getBody().getName()).isEqualTo("Nagendra");  BDDAssertions.*then*(personResponseEntity.getBody().getSurname()).isEqualTo("Mekala");  }  } |

Note that the ids property of the @AutoConfigureStubRunner annotation specifies:

* com.baeldung.spring.cloud — the groupId of our artifact
* spring-cloud-contract-producer — the artifactId of the producer stub jar
* 8090 — the port on which the generated stubs will run
* stubsMode=StubsMode.LOCAL – refreing stub in mavem repository
* stubsMode=StubsMode.CLASSPATH– refreing stub in classpath
* stubsMode=StubsMode.REMOTE – refreing stub in remote

**When the Contract is Broken:**

If we make any changes on the producer side that directly impact the contract without updating the consumer side, this can result in contract failure.

Note: producer/consumer build all junit test case successful then only contrat approved. If any test, cases are

Failure, contract is broken producer/consumer apllication

WireMock

WireMock is a simulator for HTTP-based APIs. Some might consider it a **service virtualization** tool or a **mock server**.

Spring Boot application that uses Tomcat as an embedded server, for example (the default with spring-boot-starter-web), then you can simply add spring-cloud-contract-wiremock to your classpath and add @AutoConfigureWireMock in order to be able to use Wiremock in your tests. Wiremock runs as a stub server and you can register stub behaviour using a Java API or via static JSON declarations as part of your test. Here’s a simple example:

|  |
| --- |
| **import** org.assertj.core.api.BDDAssertions;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.springframework.boot.test.context.SpringBootTest;  **import** org.springframework.cloud.contract.wiremock.AutoConfigureWireMock;  **import** org.springframework.http.ResponseEntity;  **import** org.springframework.test.context.junit4.SpringRunner;  **import** org.springframework.web.client.RestTemplate;  **import** com.github.tomakehurst.wiremock.client.WireMock;  @RunWith(SpringRunner.**class**)  @SpringBootTest  @AutoConfigureWireMock(port=8081)  **public** **class** CarRentalApplicationTests {  @Test  **public** **void** test\_should\_return\_all\_fradus() {  String json = "[\"Nagendra\",\"Mekala\"]";  WireMock.*stubFor*(WireMock.*get*(WireMock.*urlEqualTo*("/frauds")).willReturn(WireMock.*aResponse*().withBody(json).withStatus(201)));    RestTemplate restTemplate = **new** RestTemplate();  ResponseEntity<String> entity = restTemplate.getForEntity("http://localhost:8081/frauds", String.**class**);    BDDAssertions.*then*(entity.getStatusCodeValue()).isEqualTo(201);  BDDAssertions.*then*(entity.getBody()).isEqualTo(json);  }  } |

**Spring Cloud Stream**

**What is Spring Cloud Streaming?**

Spring Cloud Stream is a framework built upon Spring Boot for building message-driven microservices.

**What is Kafka?**

Kafka is a popular high performant and horizontally scalable messaging platform originally developed by LinkedIn.

**What is Lombok?**

Lombok is a java framework that automatically generates getters, setters, toString(), builders, loggers, etc. in the code.

**Dependency:**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <project xmlns=*"http://maven.apache.org/POM/4.0.0"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>  <modelVersion>4.0.0</modelVersion>  <groupId>com.spring.cloud.kafka.stream</groupId>  <artifactId>Spring-Cloud-Stream-Kafka</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>jar</packaging>  <name>Spring-Cloud-Stream-Kafka</name>  <description>Demo project for Spring Boot</description>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.4.RELEASE</version>  <relativePath /> <!-- lookup parent from repository -->  </parent>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  <spring-cloud.version>Finchley.SR1</spring-cloud.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-stream</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-stream-kafka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  <scope>runtime</scope>  </dependency>  <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  <optional>true</optional>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-stream-test-support</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>${spring-cloud.version}</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

**Define the Kafka streams:**

|  |
| --- |
| package com.spring.cloud.kafka.stream.message;  import org.springframework.cloud.stream.annotation.Input;  import org.springframework.cloud.stream.annotation.Output;  import org.springframework.messaging.MessageChannel;  import org.springframework.messaging.SubscribableChannel;  public interface GreetingsStreams {  String INPUT = "greetings-in";  String OUTPUT = "greetings-out";    @Input(INPUT)  SubscribableChannel inboundGreetings();    @Output(OUTPUT)  MessageChannel outboundGreetings();  } |

In order for our application to be able to communicate with Kafka, we’ll need to define an outbound stream to write messages to a Kafka topic, and an inbound stream to read messages from a Kafka topic.

Spring Cloud provides a convenient way to do this by simply creating an interface that defines a separate method for each stream.

The inboundGreetings() method defines the inbound stream to read from Kafka and outboundGreetings() method defines the outbound stream to write to Kafka.

**Configure Spring Cloud StreamConfigure Spring Cloud Stream:**

Our next step is to configure Spring Cloud Stream to bind to our streams in the GreetingsStreams interface. This can be done by creating a @Configuration class com.kaviddiss.streamkafka.config.StreamsConfig with below code:

|  |
| --- |
| package com.spring.cloud.kafka.stream;  import org.springframework.cloud.stream.annotation.EnableBinding;  import com.spring.cloud.kafka.stream.message.GreetingsStreams;  @EnableBinding(GreetingsStreams.class)  public class StreamsConfig {  } |

Binding the streams is done using the @EnableBinding annotation where the GreatingsService interface is passed.

**Configuration properties for Kafka:**

YAML format as it’s less verbose and allows to keep both common and environment-specific properties in the same file.

|  |
| --- |
| spring:  cloud:  stream:  kafka:  binder:  brokers: localhost:9092  bindings:  greetings-in:  destination: greetings  contentType: application/json  greetings-out:  destination: greetings  contentType: application/json |

**Define service and controller:**

|  |
| --- |
| Model:  @Getter @Setter @ToString @Builder  **public** **class** Greetings {  **private** **long** timestamp;  **private** String message;  }  Controller:  @Service  *@Slf4j*  **public** **class** GreetingsService {  **private** GreetingsStreams greetingsStreams;    **public** GreetingsService(GreetingsStreams greetingsStreams) {  **this**.greetingsStreams = greetingsStreams;  }  **public** **void** sendGreetings(**final** Greetings greetings) {  ***log***.info("Sending greetings: {}",greetings);    MessageChannel messageChannel = greetingsStreams.outboundGreetings();  messageChannel.send(MessageBuilder  .*withPayload*(greetings)  .setHeader(MessageHeaders.***CONTENT\_TYPE***, MimeTypeUtils.***APPLICATION\_JSON***)  .build());  }  }  Listener:  @Component  *@Slf4j*  **public** **class** GreetingsListener {  @StreamListener(GreetingsStreams.***INPUT***)  **public** **void** handleMessage(@Payload Greetings greetings) {  ***log***.info("Received greetings: {}", greetings);  }  }  **Service:**  @Service  *@Slf4j*  **public** **class** GreetingsService {  **private** GreetingsStreams greetingsStreams;    **public** GreetingsService(GreetingsStreams greetingsStreams) {  **this**.greetingsStreams = greetingsStreams;  }  **public** **void** sendGreetings(**final** Greetings greetings) {  ***log***.info("Sending greetings: {}",greetings);    MessageChannel messageChannel = greetingsStreams.outboundGreetings();  messageChannel.send(MessageBuilder  .*withPayload*(greetings)  .setHeader(MessageHeaders.***CONTENT\_TYPE***, MimeTypeUtils.***APPLICATION\_JSON***)  .build());  }  } |

**Spring Boot Actuator**

Spring Boot Actuator module helps you monitor and manage your Spring Boot application by providing production-ready features like health check-up, auditing, metrics gathering, HTTP tracing etc. All of these features can be accessed over JMX or HTTP endpoints.

Spring Boot Actuator Endpoints are exposed over [JMX](https://www.journaldev.com/1352/what-is-jmx-mbean-jconsole-tutorial) and HTTP, most of the times we use HTTP based Actuator endpoints because they are easy to access over the browser, CURL command, shell scripts etc.

Some of the useful actuator endpoints are:

1. beans: this endpoint returns the list of all the beans configured in our application.
2. env: provides information about the Spring Environment properties.
3. health: Shows application health
4. info: Displays application information, we can configure this in Spring environment properties.
5. mappings: Displays the list of all [@RequestMapping](https://www.journaldev.com/3358/spring-requestmapping-requestparam-pathvariable-example) paths.
6. shutdown: allows us to gracefully shutdown the application.
7. threaddump: provides the thread dump of the application.

You can get the complete list of spring actuator endpoints from [here](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html).

Reference sites:

<https://www.pluralsight.com/courses/spring-cloud-contract-introduction>

<https://www.baeldung.com/spring-cloud-contract>

spring cloud

<http://kaviddiss.com/2018/03/03/spring-cloud-stream-kafka/>

<https://blog.knoldus.com/rdd-sparks-fault-tolerant-in-memory-weapon/>.