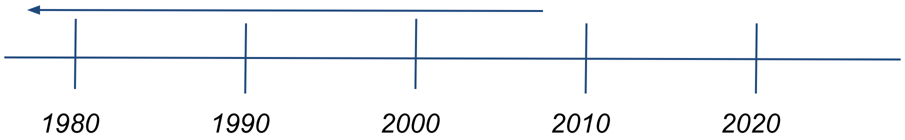
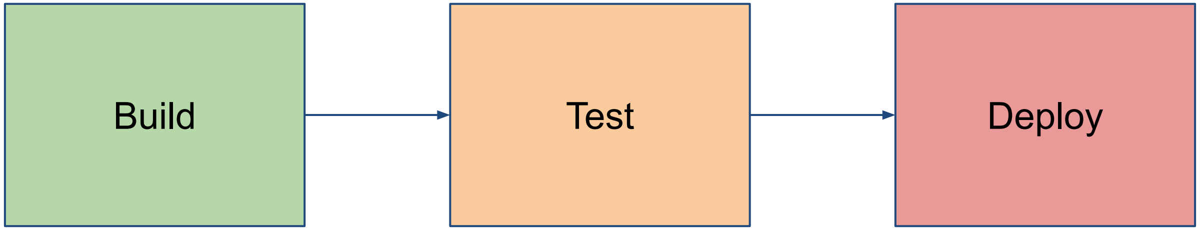
Consumer Driven Contract

Live up to your API promises

kiril.arsov@seavus.com

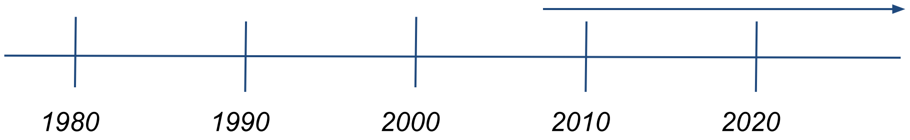
Monoliths

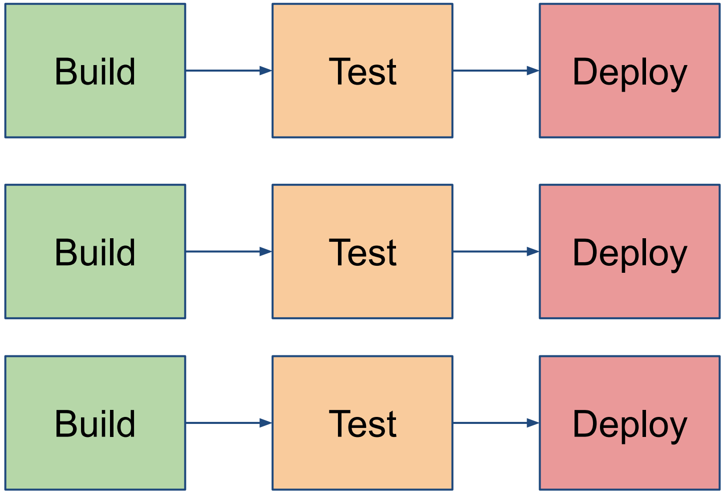




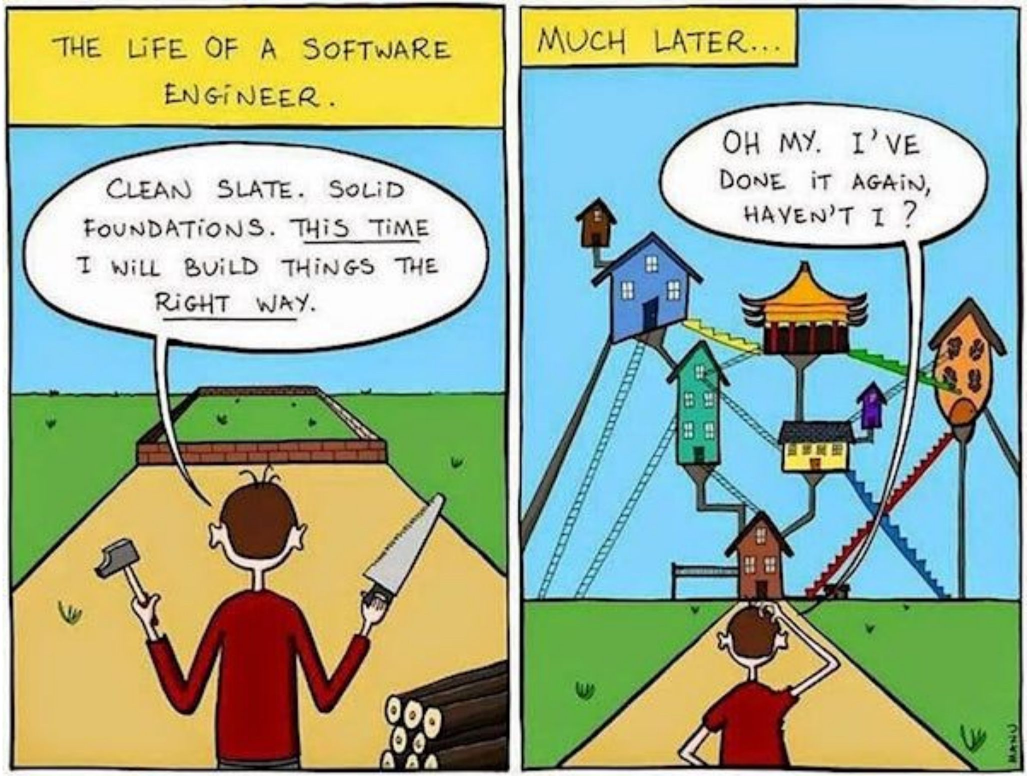
- Simple processes
- Straight-forward testing
- Easy business domain evolution

Microservices

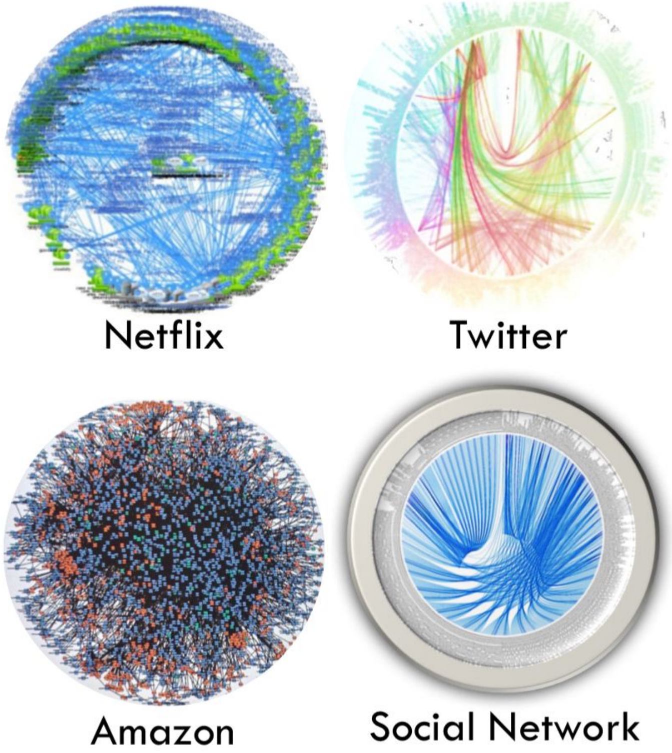




- Orchestration
- Configuration
- Monitoring


THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?
MANU

Microservices in reality



Death Star Pitfalls

- Intense coupling
- Hard to make changes
- Maintenance chaos
- Release nightmare

Is it possible to achieve true release independence ?

Goals

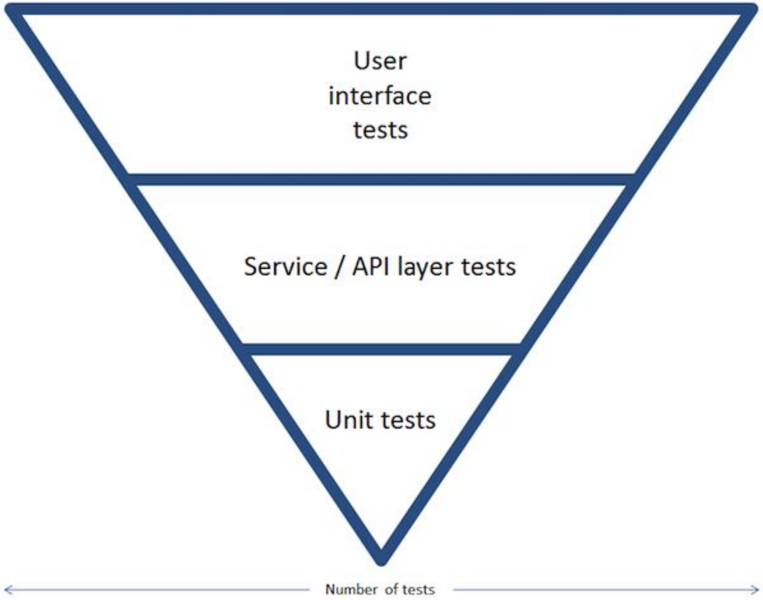
In our deployment pipeline we'd like to shift the failing builds as much to the left as possible. That means that we don't want to wait until the end of the pipeline to see that we have a bug in our algorithm or we have a faulty integration. Our aim is to fail the build of the application in that case.

In order to fail fast and start getting immediate feedback from our application, we do test driven development and start with unit tests. **That's the best way to start sketching the architecture we'd like to achieve**. We can test functionalities in isolation and get immediate response from those fragments. With unit tests, it's much easier and faster to figure out the reason for a particular bug or malfunctioning.

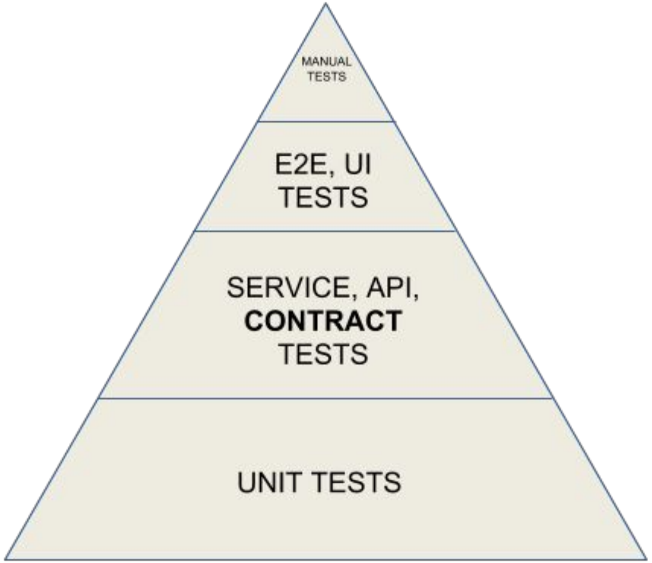
Are unit tests enough? Not really since nothing works in isolation. **We need to integrate the unit-tested components and verify if they can work properly together**. A good example is to assert whether a Spring context can be properly started and all required beans got registered.

Goals

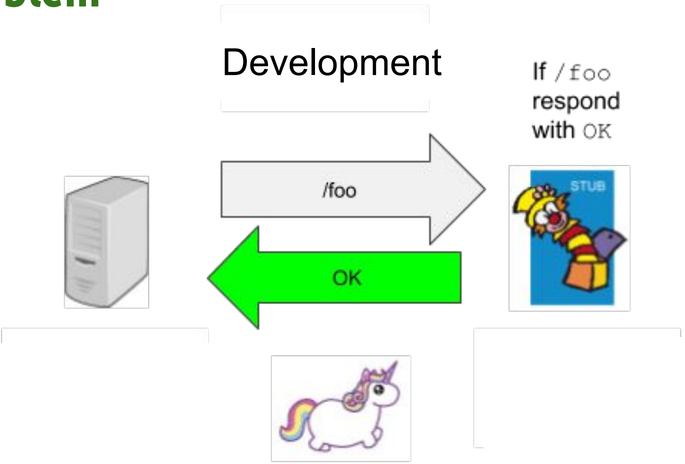
From



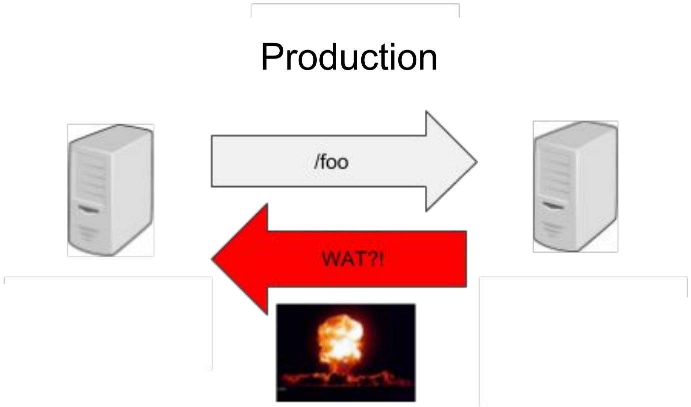
To



The problem


Development

If `/foo` respond with `OK`


Production

What does it actually mean? Why do the test pass whereas the production code fails?! That's happening due to the fact that **the stubs created on the consumer side are not tested against the producer's code**.

That means that we have quite a few false positives. That actually also means that we've wasted time (thus money) on running integration tests that test nothing beneficial (and should be deleted). What is even worse is that we've failed on end-to-end tests and we needed to spend a lot of time to debug the reason for these failures.

Definition

Consumer-Driven Contracts (CDC) is a pattern for evolving services. In Consumer-Driven Contracts, each consumer captures their expectations of the provider in a separate contract. All of these contracts are shared with the provider so they gain insight into the obligations they must fulfill for each individual client.

In other words: To make sure that 2 services are "on the same page"

Contract
The contract is an agreement between the producer and consumer how the API/message will look like.

> What endpoints can we use?
> What input do the endpoints take?
> What does the output look like?

Definition

Producer/Provider
The producer is a service that exposes an API (e.g. rest endpoint) or sends a message (e.g. Kafka Producer which publishes the message to Kafka Topic)

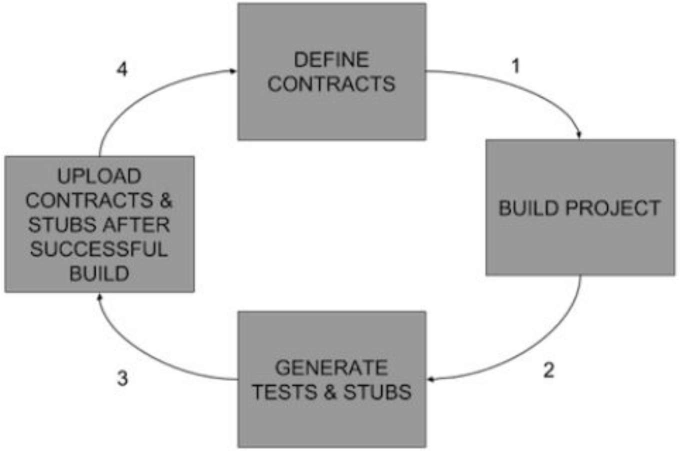
Consumer
The consumer is a service that consumes the API that is exposed by the producer or listens to a message from the producer (e.g. Kafka Consumer which consumes the message from Kafka Topic)

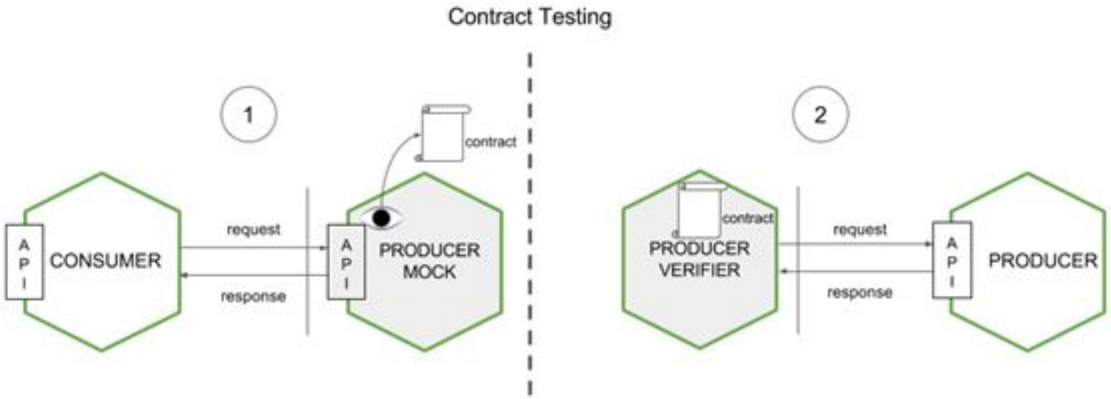
Spring Cloud Contract

A number of tools exist to aid in writing contract tests such as Pact, Pacto, Janus and Spring cloud contract.

Spring Cloud Contract is a project that, simply put, helps us write Consumer Driven Contracts (CDC)

Spring Cloud Contract doesn't require you to actually use Spring. As consumers can call the StubRunner JUnit Rule to download and start stubs.



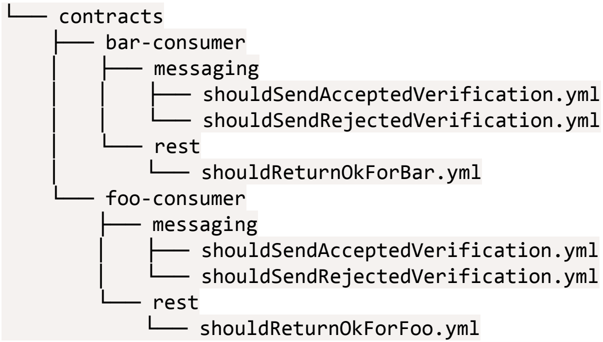


Who writes them?

There are 2 approaches:

- producer driven contract
    - The producer defines the contracts and all consumers need to follow the guidelines defined in the contracts.

- consumer driven contract
    - Consumers create their own set of contracts for a given producer



```
└── contracts
    ├── bar-consumer
    │   ├── messaging
    │   │   ├── shouldSendAcceptedVerification.yml
    │   │   └── shouldSendRejectedVerification.yml
    │   └── rest
    │       └── shouldReturnOkForBar.yml
    └── foo-consumer
        ├── messaging
        │   ├── shouldSendAcceptedVerification.yml
        │   └── shouldSendRejectedVerification.yml
        └── rest
            └── shouldReturnOkForFoo.yml
```

folder structure defined under a producer's repository

Setup

On producer side

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-verifier</artifactId>
    <scope>test</scope>
</dependency>

<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>true</extensions>
</plugin>
```

On Consumer side

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
    <scope>test</scope>
</dependency>
```

Agreement

Let’s imagine that before two applications communicate with each other, they formalize the way they send / receive their messages. What we would like to define are pairs of actual possible conversations that can take place.

In Spring Cloud Contract a contract can be defined either in Groovy, YAML or a Pact file. Let’s look at an example the following YAML contract:

```
Contract.make {
    request {
        method'GET'
        url '/api/cards?accountUuid=account-uuid-1'
        headers {
            header 'Content-Type': 'application/json'
        }
    }
    response {
        status OK()
        body(file('cardResponseDto.json'))
        headers {
            contentType(applicationJson())
        }
    }
}
```

What we’ve managed to achieve is **codify the requirement** of the consumer test that was written against the WireMock stub.

Agreement

In Spring, they take promises seriously, so if one writes a contract ,we generate a test out of it to verify if the producer meets that contract.

The essence of the contract tests is not to assert the functionality. What we want to achieve is to verify the semantics. If the producer and the consumer will be able to successfully communicate on production.

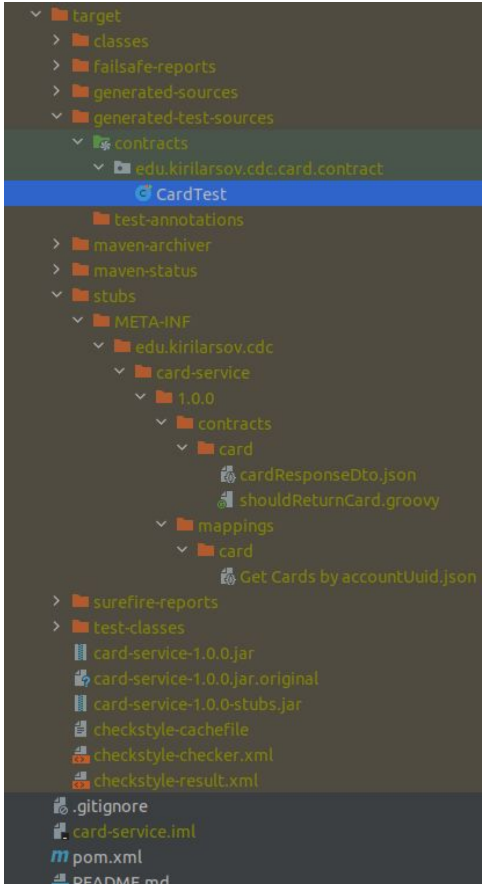
What do those generated tests look like?

```
public class CardTest extends CardBase {

@Test
public void validate_get_Cards_by_accountUuid() throws Exception {

// given:
MockMvcRequestSpecification request = given()
          .header("Content-Type", "application/json");

// when:
ResponseOptions response = given().spec(request)
          .get("/api/cards?accountUuid=account-uuid-1");

// then:
assertThat(response.statusCode()).isEqualTo(200);
assertThat(response.header("Content-Type")).matches("application/json.*");

// and:
DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
assertThatJson(parsedJson).array("['cards']").contains("['accountUuid']").isEqualTo("account-uuid-1");
assertThatJson(parsedJson).array("['cards']").contains("['cardHolderName']").isEqualTo("Tom Jonson");
assertThatJson(parsedJson).array("['cards']").contains("['pan']").isEqualTo("1234-5678-9012-3456");
assertThatJson(parsedJson).array("['cards']").contains("['validTo']").isEqualTo("11/29");
assertThatJson(parsedJson).array("['cards']").contains("['cvc']").isEqualTo("123");
assertThatJson(parsedJson).array("['cards']").contains("['status']").isEqualTo("ACTIVE");
}
}
```

Sign and publish the agreement

After setting up running the generated test we can notice that we’ve gotten stubs in the generated-test-resources folder an additional artifact with a -stubs suffix. That artifact contains the contracts and the stubs. The stub is a standard JSON representation of a WireMock stub

After publishing what would happen is that the fat jar of the application together with the stubs would get uploaded to Nexus / Artifactory. That way we get the reusability of the stubs out of the box, since they are generated, asserted and uploaded only once, after being verified against the producer.



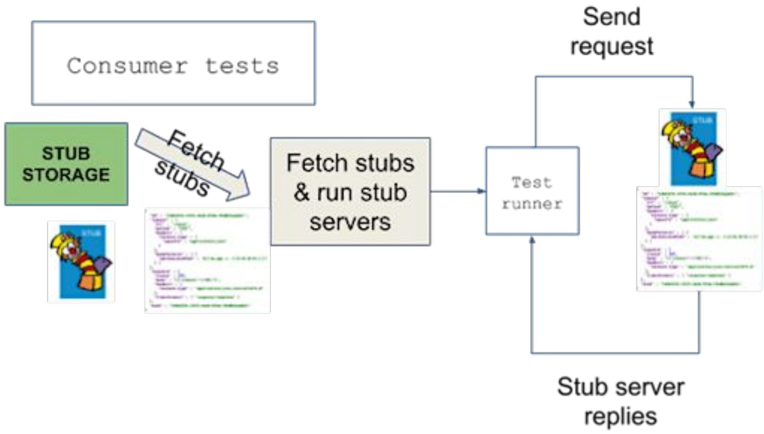
```
target
  classes
  failsafe-reports
  generated-sources
  generated-test-sources
    contracts
      edu.kirilarsov.cdc.card.contract
        CardTest
    test-annotations
  maven-archiver
  maven-status
  stubs
    META-INF
      edu.kirilarsov.cdc
        card-service
          1.0.0
            contracts
              card
                cardResponseDto.json
                shouldReturnCard.groovy
            mappings
              card
                Get Cards by accountUuid.json
  surefire-reports
  test-classes
  card-service-1.0.0.jar
  card-service-1.0.0.jar.original
  card-service-1.0.0-stubs.jar
  checkstyle-cachefile
  checkstyle-checker.xml
  checkstyle-result.xml
.gitignore
card-service.iml
pom.xml
README.md
```

Using the agreement

We can control the stub downloading via the stubsMode switch. The following options can be used to download the stub:-

- StubRunnerProperties.StubsMode.**CLASSPATH** (default value) - This is the default mode. It will scan the classpath and pick stubs from there. We need to add the dependency of the stub with classifier as a stub in the pom.xml with test scope.
- StubRunnerProperties.StubsMode.**LOCAL** - It will pick stubs from a local m2 repository.
- StubRunnerProperties.StubsMode.**REMOTE** - It will pick stubs from a remote location e.g. Nexus. We need to initialize repositoryRoot property with the URL of the remote repository in the AutoConfigureStubRunner annotation.

Using the agreement

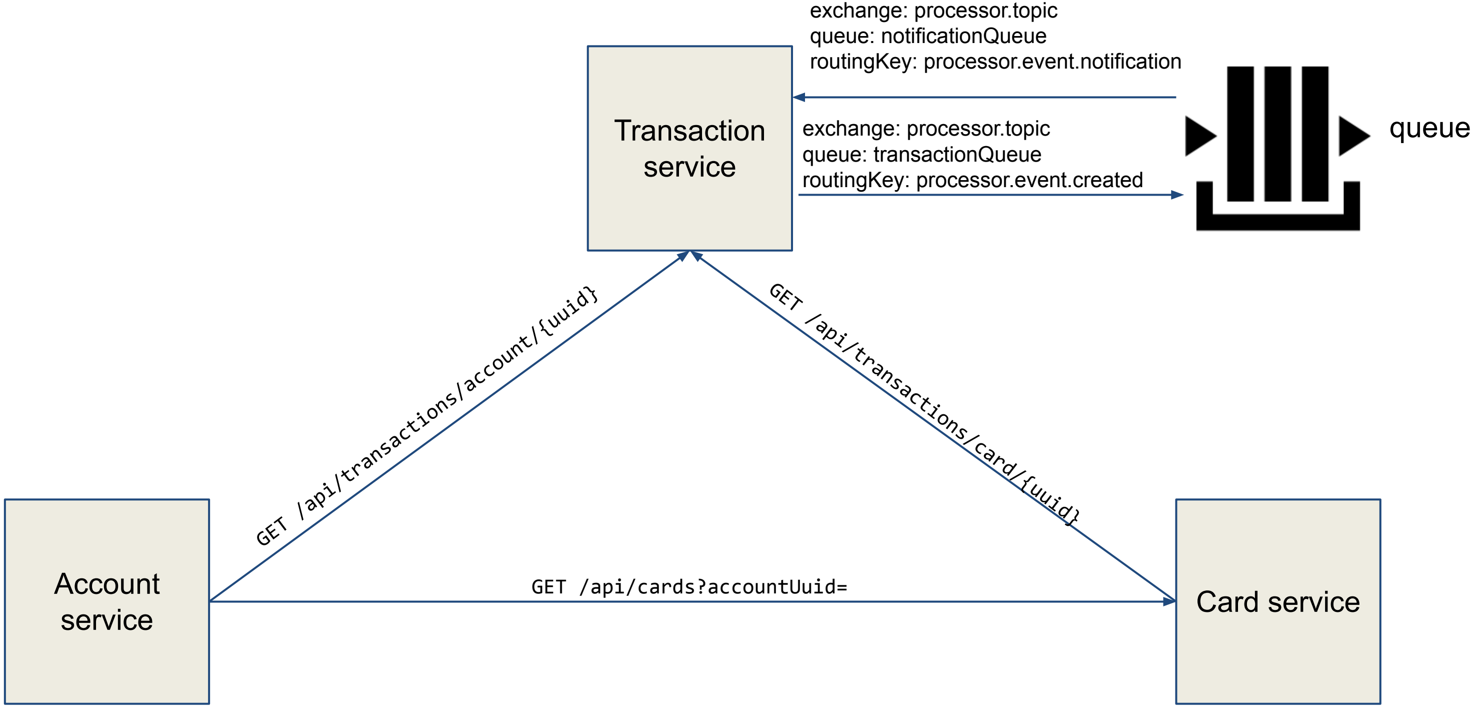


```
@SpringBootTest(
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT,
    properties = {
        "integration.card-service-base-url=" +
            "http://localhost:${stubrunner.runningstubs.card-service.port}"
    }
)
@AutoConfigureStubRunner(
    stubsMode = CLASSPATH,
    ids = {
        "edu.kirilarsov.cdc:card-service:+:stubs"
    }
)

class CardRepositoryContractIT {

    @Autowired
    private CardRepository cardRepository;

    @Test
    void testGetCard() {

        var cardDtos = cardRepository.getCardsByAccount("account-uuid-1");

        assertThat(cardDtos)
            .extracting(CardResponseDto::cards)
            .asList()
            .isNotEmpty()
            .hasSize(1)
            .element(0)
            .hasFieldOrPropertyWithValue("accountUuid", "account-uuid-1")
            .hasFieldOrPropertyWithValue("cardHolderName", "Tom Jonson")
            .hasFieldOrPropertyWithValue("pan", "1234-5678-9012-3456")
            .hasFieldOrPropertyWithValue("validTo", "11/29")
            .hasFieldOrPropertyWithValue("status",
CardResponseDto.CardDto.Status.ACTIVE)
            .hasFieldOrPropertyWithValue("cvc", "123");
    }
}
```

Key takeaways

- As consumers, we will fail fast if we are incapable of communicating with the producer
- As producers, we can see if our code changes are not breaking the contracts that we've agreed upon with our clients

Demo project setup



exchange: processor.topic
queue: notificationQueue
routingKey: processor.event.notification

Transaction service

exchange: processor.topic
queue: transactionQueue
routingKey: processor.event.created

queue

GET /api/transactions/account/{uuid}

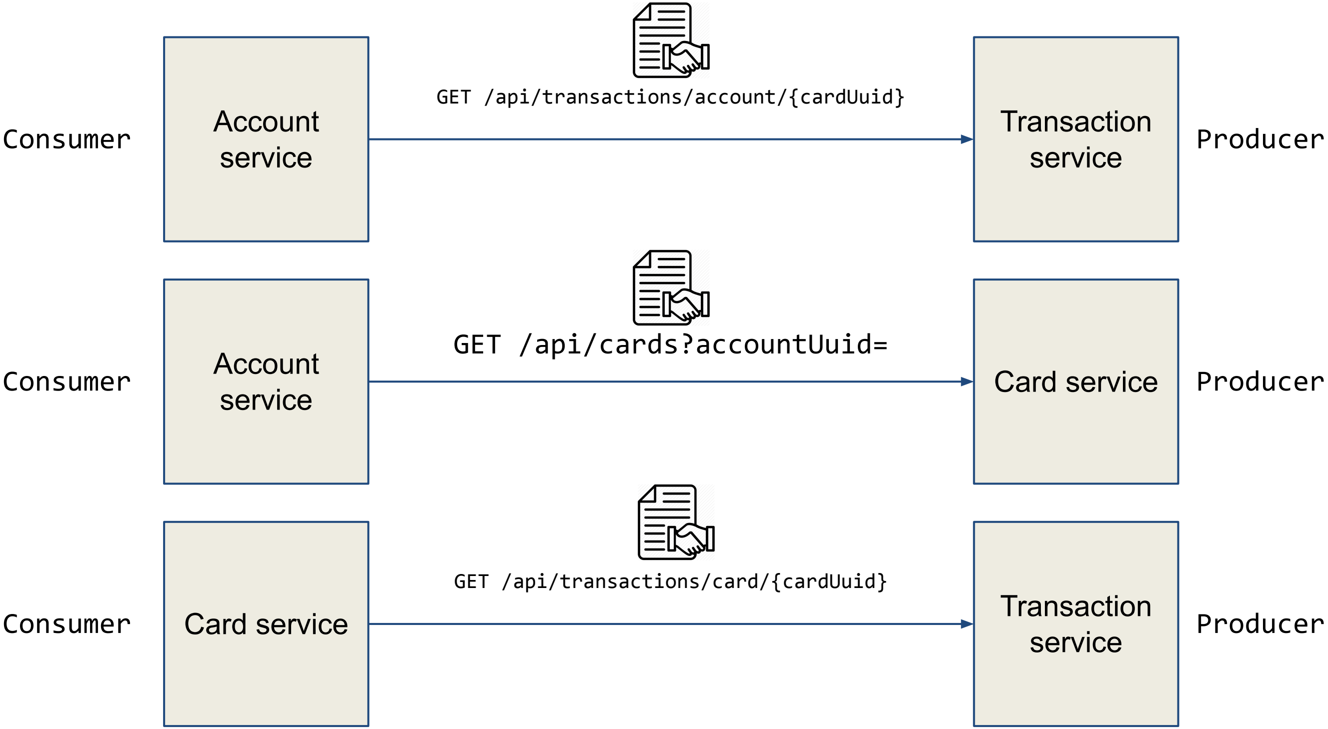
GET /api/transactions/card/{uuid}

Account service

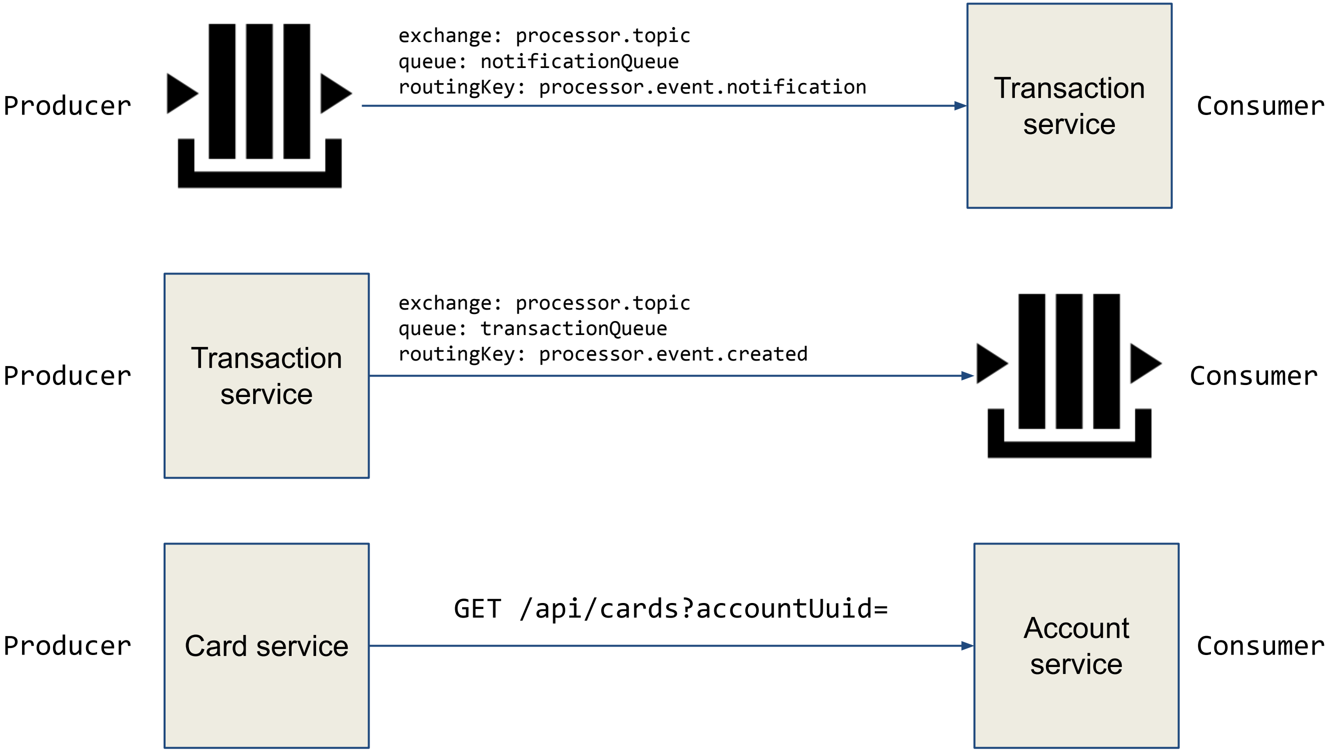
GET /api/cards?accountUuid=

Card service

Project setup



Project setup



Producer

exchange: processor.topic
queue: notificationQueue
routingKey: processor.event.notification

Transaction service

Consumer

Producer

Transaction service

exchange: processor.topic
queue: transactionQueue
routingKey: processor.event.created

Consumer

Producer

Card service

GET /api/cards?accountUuid=

Account service

Consumer

References

- https://piotrminkowski.wordpress.com/tag/spring-cloud-contract/
- https://martinfowler.com/articles/consumerDrivenContracts.html
- https://www.infoq.com/articles/contract-testing-spring-cloud-contract/
- https://novotnyr.github.io/scrolls/enforcing-spring-cloud-contracts-over-amqp/
- https://konghq.com/videos/live-up-to-your-api-promises-contract-testing-apis-so-they-dont-break-apart/

Q&A

Thank you!

kiril.arsov@seavus.com