## Setup Project Architecture

Step 1 – IntelliJ and MySQL server

Step 2 – Import projects consumer(Library) and provider(Course) microservice.

Step 3 – Create DB schema and tables for both services

Step 4 – Update the DB properties in project.

Step 5 – Start the microservices. Library started on 8080, and courses on 8181

Step 6- We will use PACT jvm library + Junit5 to build our tests.

## CONSUMER TEST

We have Library service which is consumer. First we have to write test cases for that by connecting it to pact mock provider. In order to use PACT in our consumer service add below dependency in pom.

<dependency>  
 <groupId>au.com.dius.pact.consumer</groupId>  
 <artifactId>junit5</artifactId>  
 <version>4.2.7</version>  
 <scope>test</scope>  
</dependency>

Then under src/test/java create unit test under class PactConsumerTest.

We have to annotate our class with below annotations.

@SpringBootTest  
@ExtendWith(PactConsumerTestExt.class)  
@PactTestFor(providerName = "CoursesCatalogue") 🡪 we can give any name for provider

First we have to generate response from our pact mock server as per the information provided in pactAllCoursesDetailsConfig

package com.rahulshettyacademy;  
  
import au.com.dius.pact.consumer.MockServer;  
import au.com.dius.pact.consumer.dsl.PactDslJsonArray;  
import au.com.dius.pact.consumer.dsl.PactDslWithProvider;  
import au.com.dius.pact.consumer.junit5.PactConsumerTestExt;  
import au.com.dius.pact.consumer.junit5.PactTestFor;  
import au.com.dius.pact.core.model.RequestResponsePact;  
import au.com.dius.pact.core.model.annotations.Pact;  
import com.fasterxml.jackson.core.JsonProcessingException;  
import com.fasterxml.jackson.databind.ObjectMapper;  
import com.rahulshettyacademy.controller.LibraryController;  
import com.rahulshettyacademy.controller.ProductsPrices;  
import org.junit.jupiter.api.Assertions;  
import org.junit.jupiter.api.Test;  
import org.junit.jupiter.api.extension.ExtendWith;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.context.SpringBootTest;  
  
@SpringBootTest  
@ExtendWith(PactConsumerTestExt.class)  
@PactTestFor(providerName = "CoursesCatalogue")  
public class PactConsumerTest1 {  
  
 @Autowired  
 private LibraryController libraryController;  
  
 //In below method we have defined consumer as booksCatalogue  
 //Using builder object we have construct a dummy response using pact  
 //This response is returned when /allCourseDetails get called,  
 // then status 200 returned with response as defined in body with minimum size of response array as 2 with price value as 12 for both index of array.  
 @Pact(consumer = "BooksCatalogue")  
 public RequestResponsePact pactAllCoursesDetailsConfig(PactDslWithProvider builder){  
 return builder.given("courses exist")  
 .uponReceiving("getting all courses details")  
 .path("/allCourseDetails")  
 .willRespondWith()  
 .status(200)  
 .body(PactDslJsonArray.*arrayMinLike*(2)  
 .stringType("course\_name")  
 .stringType("id")  
 .integerType("price", 12)  
 .stringType("category")  
 .closeObject()  
 ).toPact();  
 }  
  
 //Here in below test method we have to pass pactMethod name,  
 //so that this test will execute as per the rules defined in provided method.  
 @Test  
 @PactTestFor(pactMethod = "pactAllCoursesDetailsConfig", port = "9999")  
 public void testAllProductsSum(MockServer mockServer) throws JsonProcessingException {  
 String expectedJson = "{\"booksPrice\":250,\"coursesPrice\":24}";  
 //booksPrice is hardcoded value, so we kept it as it is.  
 //coursesPrice should actually coming from courses service  
 //but in our case it will come from config method, we set value as 12 for two values of array. so sum is 24.  
 //Make sure to pass expected json in correct format. To remove any spacing issues - https://jsontostring.com/  
 libraryController.setBaseUrl(mockServer.getUrl());  
 ProductsPrices productsPrices = libraryController.getProductPrices();  
 //How to start pact server so that instead of hitting the actual service url  
 //it should hit our pact server and provide response from there which we configured above.  
 //We have to override the setBaseURL method which is present in this service.  
 //When working on actual project, find a way or method where we can pass base url.  
 //We have to provide port in @PactTestFor annotation where pact server should run.  
 //To get the url for pact server, we have to use class MockServer and method name as getUrl()  
 ObjectMapper obj = new ObjectMapper();  
 String actualJson = obj.writeValueAsString(productsPrices);  
 //String actualJson = obj.writerWithDefaultPrettyPrinter().writeValueAsString(productsPrices);  
 //writerWithDefaultPrettyPrinter this method will make json format correctly.  
 Assertions.*assertEquals*(expectedJson, actualJson);  
 //Once our test case get passed this will write pact contract json file in target/pacts folder.  
 //If provider make any changes in response schema then this file will tell them that it can impact consumer unit test cases.  
 }



## PROVIDER TESTS

Now we are ready with our contract file.

Now we can use this contact to execute it in our provider service.

We need to create a folder pacts in src/main/java in provider(Courses) service and paste that contact json file inside that.

Now we need to write a test at provider side to run that json file.

In consumer project we added junit dependency for consumer. But since now we are writing test case for Provider we have to add provider junit dependency.

<dependency>  
 <groupId>au.com.dius.pact.provider</groupId>  
 <artifactId>junit5</artifactId>  
 <version>4.2.1</version>  
</dependency>

package com.rahulshettyacademy.Courses;  
  
import au.com.dius.pact.provider.junit5.HttpTestTarget;  
import au.com.dius.pact.provider.junit5.PactVerificationContext;  
import au.com.dius.pact.provider.junit5.PactVerificationInvocationContextProvider;  
import au.com.dius.pact.provider.junitsupport.Provider;  
import au.com.dius.pact.provider.junitsupport.State;  
import au.com.dius.pact.provider.junitsupport.StateChangeAction;  
import au.com.dius.pact.provider.junitsupport.loader.PactFolder;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.TestTemplate;  
import org.junit.jupiter.api.extension.ExtendWith;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.boot.web.server.LocalServerPort;  
  
//We are stating Provider in its own environment instead of pact environment.  
//Whereas we started Consumer in pact environment.  
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.*RANDOM\_PORT*)  
//We have defined provider name in ConsumerTest class, we have to provide the same name here.  
@Provider("CoursesCatalogue")  
//We have to provide location where our pact contract json file is present  
@PactFolder("pacts")  
public class PactProviderTest1 {  
//@LocalServerPort annotation will read where our server is started  
// and assign that port number to port variable  
//context variable need this port information so that it knows where our service is.  
//We have created setup method where we setTarget by passing url and port to my context.  
 @LocalServerPort  
 public int port;  
  
//We have use @TestTemplate with our method so that it will become only template  
// instead of actual test which can execute multiple times.  
//We have called verifyInteraction method which interact with our contract file.  
//For consumer side we have to extendWith PactConsumerTestExt at class level  
//For provider side we have to extendWith PactVerificationInvocationContextProvider at method level  
  
 @TestTemplate  
 @ExtendWith(PactVerificationInvocationContextProvider.class)  
 public void pactVerificationTest(PactVerificationContext context){  
 context.verifyInteraction();  
 }  
  
 @BeforeEach  
 public void setup(PactVerificationContext context){  
 context.setTarget(new HttpTestTarget("localhost", port));  
 }  
  
 //For which interaction this setup to execute we have to define  
 //So we have to pass the value, this value we get from Consumer given() method of builder.  
 //Value should be same to understand the interaction.  
 //If we don't provide value then this method will execute for all interactions exist in consumer class  
 //Implementation of these setup and teardown method is not needed.  
 //But if we don't define them then it will through compile time error.  
 @State(action = StateChangeAction.*SETUP*, value = "courses exist")  
 public void coursesExistSetup(){  
  
 }  
  
 @State(action = StateChangeAction.*TEARDOWN*, value = "courses exist")  
 public void coursesExistTearDown(){  
  
 }  
  
}

## How contract get failed and how we create contract for only needed fields

1. In our consumer test make minimum items 5 is expected, but in actual our DB has 3 items. So copy this contract to our location and then run provider test. It will fail.
2. When we have price set as integer type earlier and we created contract rules as per integer type. Later at any point of time provider changed it to String. In that case also our contract gets failed.
3. When we change the json path structure.
4. When identity name itself get changed, e.g. earlier it was course\_name now renamed as courseName.

**Note** – In point number 3 and 4 are not related to price field and we have created this contract only to verify sum of prices return, so we don’t need them to include in our test. So for that we will create rules only for price in new method in ConsumerTest. We only mention those rules which can impact us.

Make sure before validate below changes first change identity data type or field name

in “AllCoursesData” file for not usable fields as per our contract.

@Pact(consumer = "BooksCatalogue")  
public RequestResponsePact pactAllCoursesDetailsPriceCheck(PactDslWithProvider builder){  
 return builder.given("courses exist")  
 .uponReceiving("getting all courses details")  
 .path("/allCourseDetails")  
 .willRespondWith()  
 .status(200)  
 .body(PactDslJsonArray.*arrayMinLike*(3)  
 .integerType("price", 12)  
 .closeObject()  
 ).toPact();  
}

Now in our consumer test method, update pact method name as “pactAllCoursesDetailsPriceCheck” which we created above.

@Test  
@PactTestFor(pactMethod = "pactAllCoursesDetailsConfig", port = "9999")  
public void testAllProductsSum(MockServer mockServer) throws JsonProcessingException {

Then run Provider test, it will run successfully and it will not validate for other fields.

## Other Scenarios

Scenario 🡪 call api for getProductDetails for only one specified product e.g. appium. This appium product have books present in library as well as courses videos present in Courses service as well. So internally Library service is calling courses service api getCourseByName/appium and it returns the response to product service and product service return concatenated response to frontend.

**Build Consumer Test for contract of price and category details from provider(Courses) microservice.**

**PactConsumerTest1.class**

*/\*\* Test 2\*/* @Pact(consumer = "BooksCatalogue")  
 public RequestResponsePact pactAppiumCourseDetails(PactDslWithProvider builder){  
 return builder.given("appium course exist")  
 .uponReceiving("getting appium course detail")  
 .path("/getCourseByName/Appium")  
 .willRespondWith()  
 .status(200)  
 .body(new PactDslJsonBody()  
 .integerType("price", 450)  
 .stringType("category", "mobile")  
 ).toPact();  
 //In this method we have used PactDslJsonBody because we are getting only Json in response  
 //instead of array of json. We don't need to use close object for this.  
 //We have defined key and value for Price and category in our Pact contract to validate  
 }  
  
 @Test  
 @PactTestFor(pactMethod = "pactAppiumCourseDetails", port = "9999")  
 public void testByProductName(MockServer mockServer) throws JsonProcessingException {  
 String expectedJson = "{\"product\":{\"book\_name\":\"Appium\",\"id\":\"ttefs36\",\"isbn\":\"ttefs\",\"aisle\":36,\"author\":\"Shetty\"},\"price\":450,\"category\":\"mobile\"}";  
 libraryController.setBaseUrl(mockServer.getUrl());  
 SpecificProduct specificProduct = libraryController.getProductFullDetails("Appium");  
 ObjectMapper objectMapper = new ObjectMapper();  
 String actualJson = objectMapper.writeValueAsString(specificProduct);  
 Assertions.*assertEquals*(expectedJson, actualJson);  
 //Here in this method we are using pact method as pactAppiumCourseDetails.  
 //We are overriding the courses actual url with our mock one.  
 //We are asserting if price and category from courses api's  
 // are getting with correct name and values as defined in pact config.  
 }

**Now this will generate contract with 2 interactions one for previous test and one for this.**

**Copy the updated contract file in Provider Pact folder location and run the interaction test from provider. In this class we don’t need to do any changes except for the states.**

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.*RANDOM\_PORT*)  
@Provider("CoursesCatalogue")  
@PactFolder("pacts")  
public class PactProviderTest1 {  
  
 @LocalServerPort  
 public int port;  
  
 @TestTemplate  
 @ExtendWith(PactVerificationInvocationContextProvider.class)  
 public void pactVerificationTest(PactVerificationContext context){  
 context.verifyInteraction();  
 }  
  
 @BeforeEach  
 public void setup(PactVerificationContext context){  
 context.setTarget(new HttpTestTarget("localhost", port));  
 }  
  
 @State(action = StateChangeAction.*SETUP*, value = "courses exist")  
 public void coursesExistSetup(){  
 }  
  
 @State(action = StateChangeAction.*TEARDOWN*, value = "courses exist")  
 public void coursesExistTearDown(){  
 }

@State(action = StateChangeAction.*SETUP*, value = "appium course exist")  
 public void appiumCourseExistSetup(){  
 }  
  
 @State(action = StateChangeAction.*TEARDOWN*, value = "appium course exist")  
 public void appiumCourseExistTearDown(){  
 }  
  
}

## Negative Scenarios

Scenario 🡪 There can be a scenario when course details are not present for particular product then course will return 404, then product service return some customised response with all details of products and a message state that course details not found.

We need to perform contract testing for this scenario because for example instead of returning 404 error with customized message later on Courses service decided to send empty json with 200 code. Then as a owner of library service we should be aware of the impact.

Expected JSON Reponse –

GET - http://localhost:8080/getProductDetails/spring

{

"product": {

"book\_name": "Spring",

"id": "sdfsd22",

"isbn": "sdd",

"aisle": 23,

"author": "Shetty"

},

"msg": "springCategory and price details are not available at this time"

}

Earlier we have written Pact for Appium is present, now for when Appium details not present in courses service.

**Pact ConsumerTest**

@Pact(consumer = "BooksCatalogue")  
public RequestResponsePact pactGetCourseByNameNotExist(PactDslWithProvider builder){  
 return builder.given("appium course not exist", "name", "Appium")  
 .uponReceiving("not getting appium course detail")  
 .path("/getCourseByName/Appium")  
 .willRespondWith()  
 .status(404)  
 .toPact();  
}  
  
@Test  
@PactTestFor(pactMethod = "pactGetCourseByNameNotExist", port = "9999")  
public void testByProductNameNotExist(MockServer mockServer) throws JsonProcessingException {  
 String expectedJson = "{\"product\":{\"book\_name\":\"Appium\",\"id\":\"ttefs36\",\"isbn\":\"ttefs\",\"aisle\":36,\"author\":\"Shetty\"},\"msg\":\"AppiumCategory and price details are not available at this time\"}";  
 libraryController.setBaseUrl(mockServer.getUrl());  
 SpecificProduct specificProduct = libraryController.getProductFullDetails("Appium");  
 ObjectMapper objectMapper = new ObjectMapper();  
 String actualJson = objectMapper.writeValueAsString(specificProduct);  
 Assertions.*assertEquals*(expectedJson, actualJson);  
}

return builder.given("appium course not exist", "name", "Appium")

In above line we are passing key value pair so that in provider test we need not to hardcode value.

**In Pact provider test**, we have to add state methods for setup and teardown.

@State(action = StateChangeAction.*SETUP*, value = "appium course not exist")  
public void appiumCourseNotExistSetup(){  
 //TO delete the appium record in database  
}  
  
@State(action = StateChangeAction.*TEARDOWN*, value = "appium course not exist")  
public void appiumCourseNotExistTearDown(){  
 //Add the appium record again in database  
}

Since we have appium details present in actual database when we run our contract on Provider side, test will fail as it will return 200 success. So we need actual response with appium does not have courses details. So we need to write logic to interact with database like JPA, mysql etc. Then delete that record.

@Autowired  
CoursesRepository coursesRepository;

@State(action = StateChangeAction.*SETUP*, value = "appium course not exist")  
public void appiumCourseNotExistSetup(Map<String, Object> params){  
 String name = (String)params.get("name");  
 //TO delete the appium record in database  
 Optional<AllCourseData> record = coursesRepository.findById(name);  
 if(record.isPresent()){  
 coursesRepository.deleteById("Appium");  
 }  
}  
  
@State(action = StateChangeAction.*TEARDOWN*, value = "appium course not exist")  
public void appiumCourseNotExistTearDown(Map<String, Object> params){  
 String name = (String)params.get("name");  
 //Add the appium record again in database  
 Optional<AllCourseData> record = coursesRepository.findById(name);  
 if(!record.isPresent()){  
 AllCourseData allCourseData = new AllCourseData();  
 allCourseData.setCourse\_name("Appium");  
 allCourseData.setCategory("mobile");  
 allCourseData.setId("12");  
 allCourseData.setPrice(13);  
 coursesRepository.save(allCourseData);  
 }  
}

In below method we are reading the value which we are passing from Consumer Test class.

public void appiumCourseNotExistSetup(Map<String, Object> params){  
 String name = (String)params.get("name");

## Article reading

In this post we'll cover the following, with accompanying videos:

1. Integration testing and the challenges with end-to-end integrated tests
2. Contract testing - what it is and why it helps
3. Pact - how Pact works
4. Demo of contract testing with Pact

#### Integration testing

Before we talk about contract testing, it's first appropriate to talk about why contract testing exists. They exist to help with integration testing - the process by which we build confidence that a system works as a whole.

In a distributed system, integration testing is a process that helps us validate that the various moving parts that communicate remotely - things like microservices, web applications and mobile applications - all work together cohesively.

There are many types of integration testing, but the most commonly relied on approach is what is called “end-to-end integrated testing”, which involves all of the components being deployed together in a real environment - one that closely resembles production - and running a battery of test scenarios against it.

Whilst the concept of contract testing predates Pact, Pact exists because [**end-to-end integrated tests are a scam**](https://pactflow.io/blog/proving-e2e-tests-are-a-scam/).

#### ****The problem with end-to-end integrated tests****

**watch: the problem with end-to-end integrated tests**

Whilst tests at the top of the pyramid more closely represent what the customer would experience, they have several painful drawbacks. They:

* are slow; because they traverse multiple systems and generally must be run serially, each test may take several seconds to several minutes to complete, especially if pre-requisite setup (such as data preparation) must be performed.
* are hard to maintain; end-to-end tests require all systems to be in the correct state before they are run, including the correct version and data.
* can be unreliable or flakey: because of the complexity in orchestrating a test environment, they can often fail causing false-positives, becoming distractions to the team. In many cases, they fail due to a configuration issue unrelated to any code change.
* are hard to fix: when an end-to-end test fails, debugging the issue is usually difficult, because of the distributed and remote nature of the problem.
* scale badly; as more teams' code gets tested, things get more entangled, test suites run exponentially slower and releases get clogged in automation pipelines.
* find bugs too late in the process: because of the complexity of running such testing suites, in many situations these tests are only run on CI after code has been committed - in many cases, by a separate testing team days afterwards. This delay in feedback is extremely costly to modern, agile delivery teams.

Because of these properties, it is advised to keep them to a minimum number e.g. to ensure key business transactions or features are covered.

See [**proving end-to-end tests are a scam**](https://pactflow.io/blog/proving-e2e-tests-are-a-scam/) to dive deeper into this.

#### What is contract testing?

**watch: explanation of contract testing and how Pact works**

Contract testing is a methodology for ensuring that two separate systems (such as two microservices) are compatible and are able to communicate with one other. It captures the interactions that are exchanged between each service, storing them in a contract, which can then be used to verify that both parties adhere to it. Contract testing goes beyond [**schema testing**](https://pactflow.io/blog/contract-testing-using-json-schemas-and-open-api-part-1/), requiring both parties to come to a consensus on the allowed set of interactions and allowing for evolution over time.

What sets this form of testing apart from other approaches that aim to achieve the same thing, is that each system is able to be tested independently from the other and that the contract is generated by the code itself, meaning the contract is always kept up to date with reality.

The following diagram shows the key steps in contract testing:

**how contract testing works**

There are many other important properties that flow on from this, which we'll discuss further below.

#### How does Pact implement contract testing

Pact is a code-first contract testing tool, that requires access to the code on both sides of an integration point. To be able to write Pact tests, you need to be able to write a unit test of the consumer, and to be able to manipulate state (usually within the context of a unit test) on the provider side.

#### How does Pactflow implement contract testing?

Whilst Pactflow supports Pact as its primary contract testing tool, it also enables a broader range of tools to be used in the contract testing process (such as Postman, Dredd or other service virtualisation and mocking tools), providing a way to "upgrade" them into a general contract testing capability. See our blog on [**bi-directional contract testing**](https://pactflow.io/blog/bi-directional-contracts/) for more.

#### What is consumer-driven contract testing?

The "consumer-driven" prefix simply states an additional philosophical position that advocates for better internal microservices design by putting the consumers of such APIs at the heart of the design process. Provider-driven APIs tend to be biased towards the data that is being exposed and the system that is exposing it.

Pact and Spring Cloud Contracts are examples of frameworks that default to this kind of implementation, but this isn't a requirement to get the key benefits of contract-testing and both can be used to achieve the aims of contract testing.

#### What are the benefits of contract testing?

**watch: contract-testing in the test pyramid and how to remove end-to-end integrated tests**

It's helpful first to consider where contract testing sits within the context of a broader automation testing approach. When coming up with a test automation strategy, a good rule of thumb in how you should expend your effort is the approach advocated in Mike Cohn's "Test Pyramid":

Stick to the pyramid shape to come up with a healthy, fast and maintainable test suite: Write lots of small and fast unit tests. Write some more coarse-grained tests and very few high-level tests that test your application from end to end [3]

**From the**[**Practical Test Pyramid**](https://martinfowler.com/articles/practical-test-pyramid.html#TheTestPyramid)

Contract tests fit in the "Service Tests" layer, as they execute quickly and don't need to integrate to external systems to run. Their job is to give you confidence that the systems you integrate with are compatible with your code before you release.

NOTE: The "UI tests" at the top of the pyramid here are often also referred to interchangeably with "integrated end-to-end (e2e) tests".

**The value of contract tests**

Contract tests generally have the opposite properties to e2e integrated tests:

* They run fast, because they don't need to talk to multiple systems.
* They are are easier to maintain: you don't need to understand the entire ecosystem to write your tests.
* They are easy to debug and fix, because the problem is only ever in the component your testing - so you generally get a line number or a specific API endpoint that is failing.
* They are repeatable:
* They scale: because each component can be independently tested, build pipelines don't increase linearly / exponentially in time
* They uncover bugs locally, on developer machines: contract tests can and should run on developer machines prior to pushing code.

From a business point of view, it is well known that the later in a project lifecycle that a bug is found, the more costlier it is to fix.

[**From**](https://www.slideshare.net/bethesque/microservices-test-smarter-not-harder-voxxed-days-2019?qid=e1be4882-f144-4897-9eca-9c89359574a4&v=&b=&from_search=1)**Beth's excellent**[**talk**](https://www.youtube.com/watch?v=79GKBYSqMIo&feature=youtu.be)**on the topic**

By running most of your automation tests as fast unit or local integration tests, you can keep build speeds to a minimum, and prevent queues building up in teams.

It turns out that contract tests also have other secondary, yet positive side-effects:

**Other benefits**

* The ability to develop the consumer (eg. a React Web App) before the API
* The ability to drive out the requirements for your provider first, meaning you implement exactly and only what you need in the provider.
* You get a set of well documented use cases ("Given ... a request for ... will return ...") that show exactly how a provider is being used.
* The ability to see exactly which fields each consumer is interested in, allowing unused fields to be removed, and new fields to be added in the provider API without impacting a consumer.
* The ability to immediately see which consumers will be broken if a change is made to the provider API.

I hope this brief introduction to the topic showed you how you can use contract testing to replace many (or in some cases, all) of your end-to-end integrated tests to speed up your CI pipeline and increase your teams' velocity.

Contract testing is not a silver bullet and no two teams are the same: so think about your test strategy, how it applies to you and what you need to get from it.