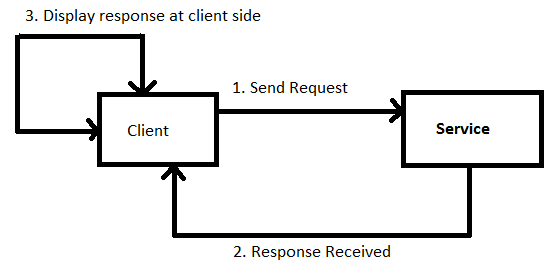
**Consumer Driven Contracts:**

Contract: In simple terms, it is an agreement between the service consumer (Client application) and service provider (WebApi)

Why do we need Consumer driven contract testing?

As we are moving towards micro services architecture the client and provider relies on contracts. Both the applications are loosely coupled. If provider modifies the contract without informing the client then it will break the application at client end.



**Fig 1.1 Overview of an architecture.**

Most of the times, consumer application and provider application owned by different teams. If we want to test the contracts then it will be pain. To mitigate this problem, we need to involve both the teams (Consumer and Service team) for integration testing. But this process is very costly, dependency on other team and time consuming.

To resolve the above issue we can use pact tool for contract testing. <https://www.gitbook.com/book/pact-foundation/pact/details>

Pact: It is a tool for contract testing. For dotnet we can use “PactNet”.

How does it works?

It works in 2 steps. For more details check url (<https://docs.pact.io/documentation/>).

Step 1 - Consumer End:

1. Setup fake service with the help of pact.
2. Mock request and response for the api call.
3. Call the Client method from pact test class.
4. Client calls the fake service and it will return the fake response.
5. It will generate a json file with complete details of request and response.
6. Share this json file with provider.

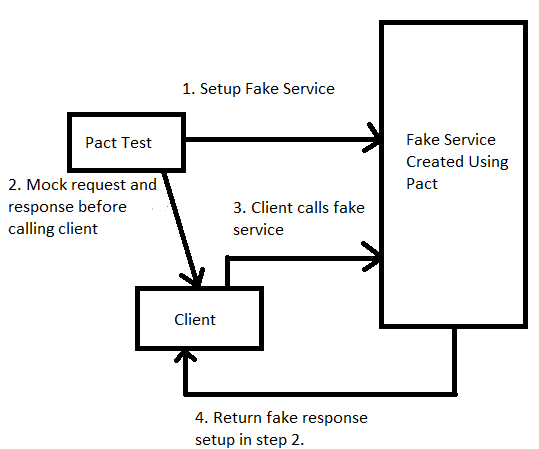


Fig 1.2 Pact overview at consumer end.

Step 2 - Provider End:

1. Read request and response from json files.
2. Pact tests replay all the requests with actual api.
3. Pact will verify all the responses.

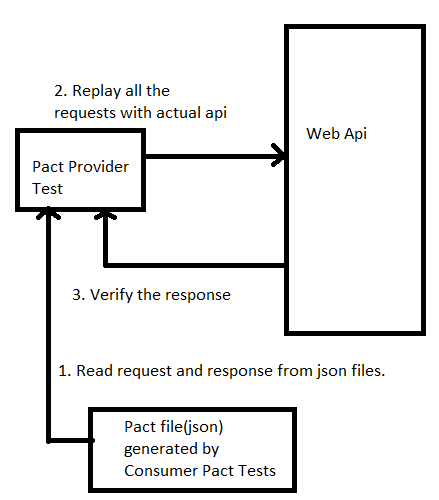


Fig 1.3 Pact overview at provider end.

Pact Implementation

You can download pact application from github (<https://github.com/Head-Strong/Pact.Test.git>).

Solution Overview:

It consist of three projects:-

1. Provider.Test: Provider tests.
2. Test.Pact.App: Consumer tests.
3. WebApi2: Service.

Setup fake service at consumer end:

public ConsumerMyApiPact()

{

// It creates json file and log file at D location

PactBuilder = new PactBuilder(new PactConfig { PactDir = @"D:\Pact", LogDir = @"D:\Pact" }); ;

PactBuilder

.ServiceConsumer(ClientName)

.HasPactWith(ProviderName); // Provider Name

MockProviderService = PactBuilder.MockService(MockServerPort);

//Configure the http mock server

MockProviderService = PactBuilder.MockService(MockServerPort, false);

// By passing true as the second param, you can enabled SSL.

// This will however require a valid SSL certificate installed and bound

// with netsh (netsh http add sslcert ipport=0.0.0.0:port certhash=thumbprint appid={app-guid})

// on the machine running the test. See https://groups.google.com/forum/#!topic/nancy-web-framework/t75dKyfgzpg

//or

MockProviderService = PactBuilder.MockService(MockServerPort, bindOnAllAdapters: false);

//By passing true as the bindOnAllAdapters parameter the http mock server will be

// able to accept external network requests, but will require admin privileges in order to run

MockProviderService = PactBuilder.MockService(MockServerPort, new JsonSerializerSettings());

//You can also change the default Json serialization settings using this overload

}

It will create json file with name as **“clientname”-“providername”.json.**

Pact Test Scenario at Consumer Side.

var consumerPact = new ConsumerMyApiPact();

\_mockProviderService = consumerPact.MockProviderService;

\_mockProviderServiceBaseUri = consumerPact.MockProviderServiceBaseUri;

consumerPact.MockProviderService.ClearInteractions();

//NOTE: Clears any previously registered interactions before the test is run

\_mockProviderService

.Given("Get user with id '1'")

.UponReceiving("A GET request to retrieve the user")

.With(new ProviderServiceRequest

{

Method = HttpVerb.Get,

Path = "/user/1",

Headers = new Dictionary<string, string>

{

{ "Accept", "application/json" }

}

})

.WillRespondWith(new ProviderServiceResponse

{

Status = 200,

Headers = new Dictionary<string, string>

{

{ "Content-Type", "application/json; charset=utf-8" }

},

Body = new //NOTE: Note the case sensitivity here, the body will be serialised as per the casing defined

{

id = 1,

firstName = "Aditya",

lastName = "Magotra"

}

}); //NOTE: WillRespondWith call must come last as it will register the interaction

var consumer = new UserApiClient(\_mockProviderServiceBaseUri);

//Act

var result = consumer.GetUsers(1);

//Assert

result.Should().NotBeNull();

\_mockProviderService.VerifyInteractions();

//NOTE: Verifies that interactions registered on the mock provider are called once and only once

//NOTE: Dispose will create json file.

consumerPact.Dispose();

Structure of json file generated by pact:

{

"provider": {

"name": "Userservice"

},

"consumer": {

"name": "Userclient"

},

"interactions": [

{

"description": "A GET request to retrieve the user",

"provider\_state": "Get user with id '1'",

"request": {

"method": "get",

"path": "/user/1",

"headers": {

"Accept": "application/json"

}

},

"response": {

"status": 200,

"headers": {

"Content-Type": "application/json; charset=utf-8"

},

"body": {

"id": 1,

"firstName": "Aditya",

"lastName": "Magotra"

}

}

}

],

"metadata": {

"pactSpecificationVersion": "1.1.0"

}

}

Replay scenario at Provider side.

var outputter = new CustomOutputter();

var config = new PactVerifierConfig();

config.ReportOutputters.Add(outputter);

IPactVerifier pactVerifier = new PactVerifier(() => { }, () => { }, config);

pactVerifier.ProviderState("Get user with id '1'");

\_httpClient.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));

\_httpClient.BaseAddress = new System.Uri("http://localhost:61131/api/");

//Act

pactVerifier

.ServiceProvider(ProviderName, \_httpClient)

.HonoursPactWith(ClientName)

.PactUri(string.Format("D:/pact/{0}-{1}.json",ClientName, ProviderName))

.Verify();

// Assert

outputter.Should().NotBeNull();

outputter.Output.Should().NotBeNullOrWhiteSpace();

outputter.Output.Should().Contain(string.Format("Verifying a Pact between {0} and {1}", ClientName, ProviderName));

outputter.Output.Should().Contain("status code 200");

System.Console.ReadLine();

This piece of code reads the json file and replay all the scenarios by calling actual service.