Testing APIs using Pact-js

What you'll learn

- Define tests using Pact-js
- Run the tests and push the test report to Xray
- Validate in Jira that the test results are available

Source code for this tutorial

- code is available in GitHub

Overview

The Pact family of frameworks provide support for Consumer Driven Contracts testing.

A Contract is a collection of agreements between a client (Consumer) and an API (Provider) that describes the interactions that can take place between them.

Consumer Driven Contracts is a pattern that drives the development of the Provider from its Consumers point of view.

Pact is a testing tool that guarantees those Contracts are satisfied.

Prerequisites

For this example we will use Pact-js, whith Mocha (test framework) and Chai (assertion library).

You can use the Pact flavour that is most suited with you, for more informations please check their document page.

We will need:

- Node.js environment
- Docker

To start using the Pact please follow the Get Started documentation.

Consumer Validations

Consumer driven contract testing with Pact will allow you to validate the contracts between the consumer and the provider sooner in the pipeline. This approach is driven by the consumer, so the provider development will be defined by the consumer point of view.

In order to demonstrate this approach we have defined an API that consists in a Comic store with operations to get all existent comics available or to fetch one particular comic. We have added also an authentication mechanism based on the authorization header.

From the consumer point of view we are going to define the interactions that the consumer is expecting from the provider using Pact-js. We will then run those interactions against a mocked provider. To achieve these results we have defined the following classes that will represent the consumer API:

./consumer.js

```
const express = require("express")
const request = require("superagent")
const server = express()

const getApiEndpoint = () => process.env.API_HOST || "http://localhost:
8081"
const authHeader = {
  Authorization: "Bearer 1234",
}

// Fetch all comics
// Comics Service
const availableComics = () => {
  return request
    .get(`${getApiEndpoint()}/comics/available`)
    .set(authHeader)
    .then(res => res.body)
}

// Find comics by their ID
const getComicsById = id => {
  return request
    .get(`${getApiEndpoint()}/comics/${id}`)
    .set(authHeader)
    .then(
      res => res.body,
      () => null
    )
}

module.exports = {
  server,
  availableComics,
  getComicsById,
}
```

consumerService.js

```
const { server } = require("./consumer.js")

server.listen(8080, () => {
  console.log("Comics Service listening on http://localhots:8080")
})
```

And using Pact we have defined the expected iterations:

test/consumer.spec.js

```
const path = require("path")
const chai = require("chai")
const chaiAsPromised = require("chai-as-promised")
const expect = chai.expect
const { Pact, Matchers } = require("@pact-foundation/pact")
const { log } = require("console")
const LOG_LEVEL = process.env.LOG_LEVEL || "WARN"

chai.use(chaiAsPromised)

describe("Pact", () => {
  const provider = new Pact({
    consumer: "e2e Consumer Example",
```

```
    provider: "e2e Provider Example",
    log: path.resolve(process.cwd(), "logs", "mockserver-integration.log"),
    dir: path.resolve(process.cwd(), "pacts"),
    logLevel: LOG_LEVEL,
    spec: 2,
  })

  // Alias flexible matchers for simplicity
  const { eachLike, like, term, iso8601DateTimeWithMillis } = Matchers

  // comic to match
  const comic_to_match =  {
    id: 2,
    title: "Batman: no return",
    pages: 22
  }

  const MIN_COMICS = 2

  const comicBodyExpectation = {
    id: like(1),
    title: like("X-MEN"),
    pages: like(50)
  }

  // Define comics list payload, reusing existing object matcher
  const comicListExpectation = eachLike(comicBodyExpectation, {
    min: MIN_COMICS,
  })

  // Setup a Mock Server before unit tests run.
  // This server acts as a Test Double for the real Provider API.
  // We then call addInteraction() for each test to configure the Mock
Service
  // to act like the Provider
  // It also sets up expectations for what requests are to come, and will
fail
  // if the calls are not seen.
  before(() =>
    provider.setup().then(opts => {
      // Get a dynamic port from the runtime
      process.env.API_HOST = `http://localhost:${opts.port}`
    })
  )

  // After each individual test (one or more interactions)
  // we validate that the correct request came through.
  // This ensures what we _expect_ from the provider, is actually
  // what we've asked for (and is what gets captured in the contract)
  afterEach(() => provider.verify())

  // Configure and import consumer API
  // Note that we update the API endpoint to point at the Mock Service
  const {
    availableComics,
    getComicsById,
  } = require("../consumer")

  // Verify service client works as expected.
  //
  // Note that we don't call the consumer API endpoints directly, but
  // use unit-style tests that test the collaborating function behaviour -
  // we want to test the function that is calling the external service.
  describe("when a call to list all comics from the Comic Service is
made", () => {
    describe("and the user is not authenticated", () => {
      before(() =>
        provider.addInteraction({
          state: "is not authenticated",
          uponReceiving: "a request for all comics",
          withRequest: {
```

```
            method: "GET",
            path: "/comics/available",
          },
          willRespondWith: {
            status: 401,
          },
        })
      )

      it("returns a 401 unauthorized", () => {
        return expect(availableComics(comic_to_match)).to.eventually.be.
rejectedWith(
          "Unauthorized"
        )
      })
    })
    describe("and the user is authenticated", () => {
      describe("and there are comics in the database", () => {
        before(() =>
          provider.addInteraction({
            state: "Has some comics",
            uponReceiving: "a request for all comics",
            withRequest: {
              method: "GET",
              path: "/comics/available",
              headers: { Authorization: "Bearer 1234" },
            },
            willRespondWith: {
              status: 200,
              headers: {
                "Content-Type": "application/json; charset=utf-8",
              },
              body: comicListExpectation,
            },
          })
        )

        it("returns a list of comics", done => {
          const comicsReturned = availableComics()

          expect(comicsReturned)
            .notify(done)
        })
      })
    })
  })

  describe("when a call to the Comic Service is made to retreive a single
comic by ID", () => {
    describe("and there is an comic in the DB with ID 1", () => {
      before(() =>
        provider.addInteraction({
          state: "Has an comic with ID 1",
          uponReceiving: "a request for an comic with ID 1",
          withRequest: {
            method: "GET",
            path: term({ generate: "/comics/1", matcher: "/comics/[0-9]+"
}),
            headers: { Authorization: "Bearer 1234" },
          },
          willRespondWith: {
            status: 200,
            headers: {
              "Content-Type": "application/json; charset=utf-8",
            },
            body: comicBodyExpectation,
          },
        })
      )

      it("returns the animal", done => {
```

```
        const comicsRetuned = getComicsById(11)

        expect(comicsRetuned)
          .to.eventually.have.deep.property("id", 1)
          .notify(done)
      })
    })

    describe("and there no comics in the database", () => {
      before(() =>
        provider.addInteraction({
          state: "Has no comics",
          uponReceiving: "a request for an comic with ID 100",
          withRequest: {
            method: "GET",
            path: "/comics/100",
            headers: { Authorization: "Bearer 1234" },
          },
          willRespondWith: {
            status: 404,
          },
        })
      )

      it("returns a 404", done => {
        const comicReturned = getComicsById(100)

        expect(comicReturned)
          .to.eventually.be.a("null")
          .notify(done)
      })
    })
  })

  // Write pact files
  after(() => {
    return provider.finalize()
  })
})
```

In the above class we have defined a new Pact between a consumer, that we have named: "**e2e Consumer Example**" and a provider named: "**e2e Provider Example**" (notice that we have also defined other parameters such as: the log path and a log level).

```
...
describe("Pact", () => {
  const provider = new Pact({
    consumer: "e2e Consumer Example",
    provider: "e2e Provider Example",
    log: path.resolve(process.cwd(), "logs", "mockserver-integration.log"),
    dir: path.resolve(process.cwd(), "pacts"),
    logLevel: LOG_LEVEL,
    spec: 2,
  })
...
```

Before starting these validations we want to start a mock server (that will represent the provider) and point our client to it.

```
...
before(() =>
    provider.setup().then(opts => {
      // Get a dynamic port from the runtime
      process.env.API_HOST = `http://localhost:${opts.port}`
    })
  )
...
```

Finally we need to define the various interactions this consumer expects from the provider. In our case we have defined 4 interactions:

- when a call to list all comics from the Comic Service is made
    - and the user is not authenticated
    - and the user is authenticated
        - and there are comics in the database
- when a call to the Comic Service is made to retrieve a single comic by ID
    - and there is a comic in the DB with ID 1
    - and there no comics in the database

In each of these we have defined a state that will define a desired state of the provider before executing the test (this state name will be used afterwards in the provider to setup the expected state), we also have defined the request details (such as method, path and possible headers) and the response expected.

```
...
describe("and there no comics in the database", () => {
      before(() =>
        provider.addInteraction({
          state: "Has no comics",
          uponReceiving: "a request for an comic with ID 100",
          withRequest: {
            method: "GET",
            path: "/comics/100",
            headers: { Authorization: "Bearer 1234" },
          },
          willRespondWith: {
            status: 404,
          },
        })
      )
...
```

Finally we will check if the expectation matches the answer obtained.

```
...
it("returns a 404", done => {
        const comicReturned = getComicsById(100)

        expect(comicReturned)
          .to.eventually.be.a("null")
          .notify(done)
      })
...
```

Once the code is implemented we can execute it with the following command:

```
npm run test:consumer
```

This will generate an immediate result in the console showing the status of the tests:

```
> e2e@1.0.0 test:consumer /Users/cristianocunha/Documents/Projects/xray-pact
> mocha test/consumer.spec.js


  Pact
    when a call to list all comics from the Comic Service is made
      and the user is not authenticated
        ✓ returns a 401 unauthorized
      and the user is authenticated
        and there are comics in the database
          ✓ returns a list of comics
    when a call to the Comic Service is made to retrieve a single comic by ID
      and there is a comic in the DB with ID 1
        ✓ returns the animal
      and there no comics in the database
        ✓ returns a 404


  4 passing (1s)
```

A Junit report and the pact file are generated from the execution:

junit_consumer.xml


```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites name="Mocha Tests" time="1.1280" tests="4" failures="0">
  <testsuite name="Root Suite" timestamp="2022-02-16T14:50:35" tests="0"
time="0.0000" failures="0">
  </testsuite>
  <testsuite name="Pact" timestamp="2022-02-16T14:50:35" tests="0" file="
/Users/cristianocunha/Documents/Projects/xray-pact/test/consumer.spec.js"
time="0.0000" failures="0">
  </testsuite>
  <testsuite name="when a call to list all comics from the Comic Service
is made" timestamp="2022-02-16T14:50:36" tests="0" file="/Users
/cristianocunha/Documents/Projects/xray-pact/test/consumer.spec.js" time="
0.0000" failures="0">
  </testsuite>
  <testsuite name="and the user is not authenticated" timestamp="2022-02-
16T14:50:36" tests="1" file="/Users/cristianocunha/Documents/Projects/xray-
pact/test/consumer.spec.js" time="0.0310" failures="0">
    <testcase name="Pact when a call to list all comics from the Comic
Service is made and the user is not authenticated returns a 401
unauthorized" time="0.0120" classname="returns a 401 unauthorized">
    </testcase>
  </testsuite>
  <testsuite name="and the user is authenticated" timestamp="2022-02-16T14:
50:36" tests="0" file="/Users/cristianocunha/Documents/Projects/xray-pact
/test/consumer.spec.js" time="0.0000" failures="0">
  </testsuite>
  <testsuite name="and there are comics in the database" timestamp="2022-
02-16T14:50:36" tests="1" file="/Users/cristianocunha/Documents/Projects
/xray-pact/test/consumer.spec.js" time="0.0170" failures="0">
    <testcase name="Pact when a call to list all comics from the Comic
Service is made and the user is authenticated and there are comics in the
database returns a list of comics" time="0.0050" classname="returns a list
of comics">
    </testcase>
  </testsuite>
  <testsuite name="when a call to the Comic Service is made to retrieve a
single comic by ID" timestamp="2022-02-16T14:50:36" tests="0" file="/Users
/cristianocunha/Documents/Projects/xray-pact/test/consumer.spec.js" time="
0.0000" failures="0">
  </testsuite>
  <testsuite name="and there is a comic in the DB with ID 1" timestamp="
2022-02-16T14:50:36" tests="1" file="/Users/cristianocunha/Documents
/Projects/xray-pact/test/consumer.spec.js" time="0.0150" failures="0">
    <testcase name="Pact when a call to the Comic Service is made to
retrieve a single comic by ID and there is a comic in the DB with ID 1
returns the animal" time="0.0050" classname="returns the animal">
    </testcase>
  </testsuite>
  <testsuite name="and there no comics in the database" timestamp="2022-02-
16T14:50:36" tests="1" file="/Users/cristianocunha/Documents/Projects/xray-
pact/test/consumer.spec.js" time="0.0210" failures="0">
    <testcase name="Pact when a call to the Comic Service is made to
retrieve a single comic by ID and there no comics in the database returns
a 404" time="0.0040" classname="returns a 404">
    </testcase>
  </testsuite>
</testsuites>
```


e2e_consumer_example-e2e_provider_example.json


```
{
  "consumer": {
    "name": "e2e Consumer Example"
  },
  "provider": {
```

```
    "name": "e2e Provider Example"
  },
  "interactions": [
    {
      "description": "a request for all comics",
      "providerState": "is not authenticated",
      "request": {
        "method": "GET",
        "path": "/comics/available"
      },
      "response": {
        "status": 401,
        "headers": {
        }
      }
    },
    {
      "description": "a request for all comics",
      "providerState": "Has some comics",
      "request": {
        "method": "GET",
        "path": "/comics/available",
        "headers": {
          "Authorization": "Bearer 1234"
        }
      },
      "response": {
        "status": 200,
        "headers": {
          "Content-Type": "application/json; charset=utf-8"
        },
        "body": [
          {
            "id": 1,
            "title": "X-MEN",
            "pages": 50
          },
          {
            "id": 1,
            "title": "X-MEN",
            "pages": 50
          }
        ],
        "matchingRules": {
          "$.body": {
            "min": 2
          },
          "$.body[*].*": {
            "match": "type"
          },
          "$.body[*].id": {
            "match": "type"
          },
          "$.body[*].title": {
            "match": "type"
          },
          "$.body[*].pages": {
            "match": "type"
          }
        }
      }
    },
    {
      "description": "a request for an comic with ID 1",
      "providerState": "Has an comic with ID 1",
      "request": {
        "method": "GET",
        "path": "/comics/1",
        "headers": {
          "Authorization": "Bearer 1234"
        },
```

```
        "matchingRules": {
          "$.path": {
            "match": "regex",
            "regex": "\\/comics\\/[0-9]+"
          }
        }
      },
      "response": {
        "status": 200,
        "headers": {
          "Content-Type": "application/json; charset=utf-8"
        },
        "body": {
          "id": 1,
          "title": "X-MEN",
          "pages": 50
        },
        "matchingRules": {
          "$.body.id": {
            "match": "type"
          },
          "$.body.title": {
            "match": "type"
          },
          "$.body.pages": {
            "match": "type"
          }
        }
      }
    },
    {
      "description": "a request for an comic with ID 100",
      "providerState": "Has no comics",
      "request": {
        "method": "GET",
        "path": "/comics/100",
        "headers": {
          "Authorization": "Bearer 1234"
        }
      },
      "response": {
        "status": 404,
        "headers": {
        }
      }
    }
  ],
  "metadata": {
    "pactSpecification": {
      "version": "2.0.0"
    }
  }
}
```


This concludes the consumer validations, next we need to share the pact file with the provider. There are two ways to do this, either we send the file to the provider (in a shared directory or email) or we can use the Pact Broker for it.

Pact Broker

The Pact broker is an open source tool that enables you to share your pacts and verification results between projects. It is the recommended way forward for serious Pact development.

For our example we will use the Pact broker available in a Docker image. We have included a *docker-compose.yaml* file in the solution, so to start it you just have to use the following command:

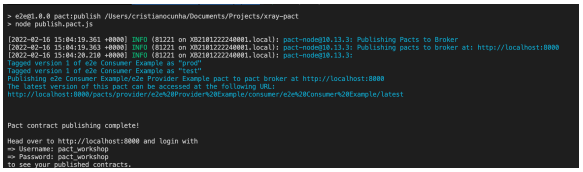
```
docker-compose up
```

This option will require you to deploy, administer and host the broker yourself. If you would prefer a plug-and-play option, you can use Pactflow.

Once the broker is up you can use the following command to push the pact to the broker:

```
npm run pact:publish
```

We can see in the output if the command was successful or not and an URL of the Broker.



To double check you can access the url of your broker and check that a new Pact was uploaded:

Notice that the column "**Last Verified**" is still empty because the provider has not yet validated this Pact on his side.

Provider Validations

On the provider side we have defined the provider API with the following two classes:

provider.js

```
const express = require("express")
const cors = require("cors")
const bodyParser = require("body-parser")
const Repository = require("./repository")

const server = express()
server.use(cors())
server.use(bodyParser.json())
server.use(
  bodyParser.urlencoded({
    extended: true,
  })
)
server.use((req, res, next) => {
  res.header("Content-Type", "application/json; charset=utf-8")
  next()
})

server.use((req, res, next) => {
  const token = req.headers["authorization"] || ""

  if (token !== "Bearer 1234") {
    res.sendStatus(401).send()
  } else {
    next()
  }
})

const comicRepository = new Repository()

// Load default data into a repository
const importData = () => {
  const data = require("./data/comicsData.json")
  data.reduce((a, v) => {
    v.id = a + 1
    comicRepository.insert(v)
    return a + 1
  }, 0)
}

// Get all comics
server.get("/comics", (req, res) => {
  res.json(comicRepository.fetchAll())
})

// Get all available comics
server.get("/comics/available", (req, res) => {
  res.json(comicRepository.fetchAll())
})

// Find an comic by ID
server.get("/comics/:id", (req, res) => {
  const response = comicRepository.getById(req.params.id)
  if (response) {
    res.end(JSON.stringify(response))
  } else {
    res.writeHead(404)
    res.end()
  }
})

module.exports = {
  server,
  importData,
  comicRepository,
}
```

providerService.js

```
const { server, importData } = require("./provider.js")
importData()

server.listen(8084, () => {
  console.log("Comics Profile Service listening on http://localhost:8084")
})
```

To perform the pact validations we have defined the following class:

provider.spec.js

```
const { Verifier } = require("@pact-foundation/pact")
const chai = require("chai")
const chaiAsPromised = require("chai-as-promised")
chai.use(chaiAsPromised)
const { server, importData, comicRepository } = require("../provider.js")
const path = require("path")

server.listen(8084, () => {
  importData()
  console.log("Comics Service listening on http://localhost:8084")
})

// Verify that the provider meets all consumer expectations
describe("Pact Verification", () => {
  it("validates the expectations of Comics Service", () => {
    let token = "INVALID TOKEN"

    let opts = {
      provider: "e2e Provider Example",
      logLevel: "DEBUG",
      providerBaseUrl: "http://localhost:8084",

      requestFilter: (req, res, next) => {
        console.log(
          "Middleware invoked before provider API - injecting
Authorization token"
        )
        req.headers["MY_SPECIAL_HEADER"] = "my special value"

        // e.g. ADD Bearer token
        req.headers["authorization"] = 'Bearer ' + token
        next()
      },

      stateHandlers: {
        "Has no comics": () => {
          comicRepository.clear()
          token = "1234"
          return Promise.resolve('Comics removed to the db')
        },
        "Has some comics": () => {
          token = "1234"
          importData()
          return Promise.resolve('Comics added to the db')
        },
        "Has an comic with ID 1": () => {
          token = "1234"
          importData()
          return Promise.resolve('Comic added to the db')
        },
        "is not authenticated": () => {
```

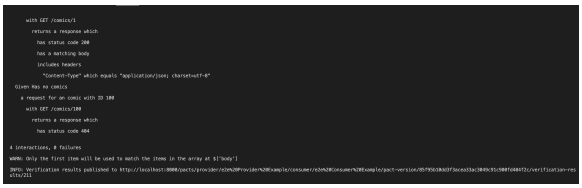
```
          token = ""
          Promise.resolve('Invalid bearer token generated')
        },
      },

      // Fetch pacts from broker
      pactBrokerUrl: "http://localhost:8000",

      // Fetch from broker with given tags
      consumerVersionTags: ["master", "test", "prod"],

      // Enables "pending pacts" feature
      enablePending: true,
      pactBrokerUsername: "pact_workshop",
      pactBrokerPassword: "pact_workshop",
      publishVerificationResult: true,
      providerVersion: "1.0.0",
    }

    return new Verifier(opts).verifyProvider().then(output => {
      console.log("Pact Verification Complete!")
      console.log(output)
    })
  })
})
```

Notice that we have defined that we need to fetch the pact from the broker and that we have defined "*stat e handlers*" that are defining the provider state before the validations (remember that the names used here must match the *providerState* defined in the consumer validations above).

In order to execute the provider validations we run the following command:

```
npm run test:provider
```

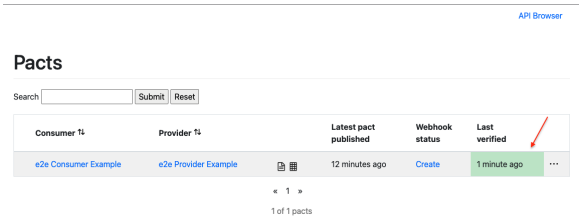
The results appear in the console output as we can see, and are also recorded in a Junit file.



junit_provider.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites name="Mocha Tests" time="0.7920" tests="1" failures="0">
  <testsuite name="Root Suite" timestamp="2022-02-16T15:20:34" tests="0"
time="0.0000" failures="0">
  </testsuite>
  <testsuite name="Pact Verification" timestamp="2022-02-16T15:20:34"
tests="1" file="/Users/cristianocunha/Documents/Projects/xray-pact/test
/provider.spec.js" time="0.7910" failures="0">
    <testcase name="Pact Verification validates the expectations of Comics
Service" time="0.7890" classname="validates the expectations of Comics
Service">
    </testcase>
  </testsuite>
</testsuites>
```

Now when you access the broker you can see that the pact now is validated by the consumer and provider for that combination of versions.



The broker will provide more info that only this view, if you want to know more please check the Pact documentation.

Pact also has available a tool that will use the information in the broker to help decide if we can proceed with the deployment or not called: *can-i-deploy*, more details of this functionality here.

Integrating with Xray

As we saw in the above example, where we are producing Junit reports with the result of the tests, it is now a matter of importing those results to your Jira instance, this can be done by simply submitting automation results to Xray through the REST API, by using one of the available CI/CD plugins (e.g. for Jenkins) or using the Jira interface to do so.

In this particular case, as we have two Junit files we need to repeat the process for each result. The importation of results is usually done in each API pipeline/process.

> As the importation of the Junit results is the same, either if it is from the consumer or the provider side, we will only exemplify the one from the consumer side.

API

API

Once you have the report file available you can upload it to Xray through a request to the REST API endpoint, and for that the first step is to follow the instructions in v1 or v2 (depending on your usage) to include authentication parameters in the following requests.

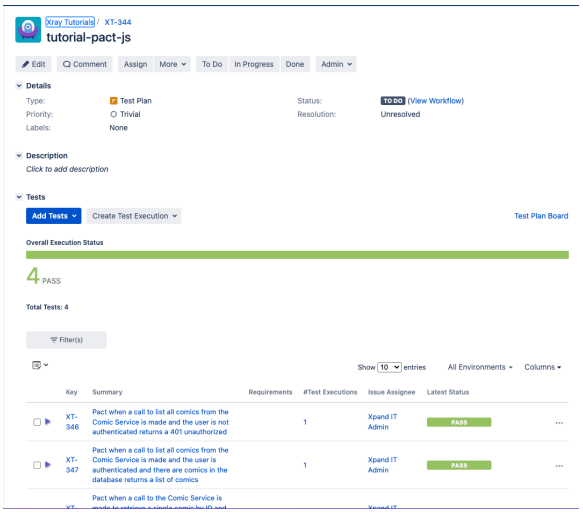
JUnit XML results

We will use the API request with the addition of some parameters that will set the Project to where the results will be uploaded and the Test Plan that will hold the Execution results.

In the first version of the API the authentication uses a login and password (not the token that is used in Cloud).

```
curl -H "Content-Type: multipart/form-data" -u admin:admin -F
"file=@junit_consumer.xml" http://yourserver/rest/raven/1.0/import
/execution/junit?projectKey=XT&testPlanKey=XT-344
```

With this command we are creating a new Test Execution in the referred Test Plan with a generic summary and four tests with a summary based on the test name.

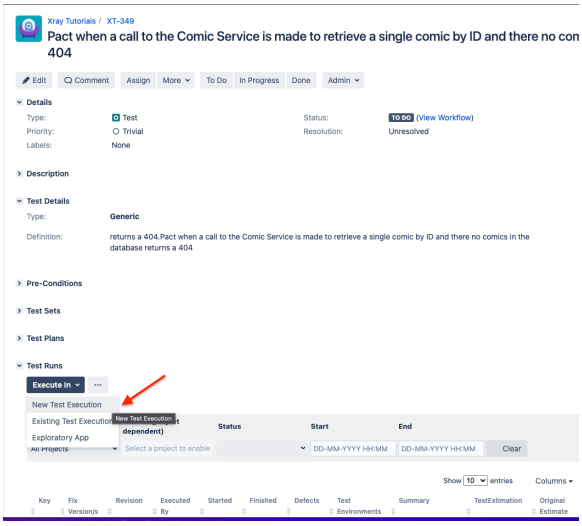


Jira UI

Jira UI

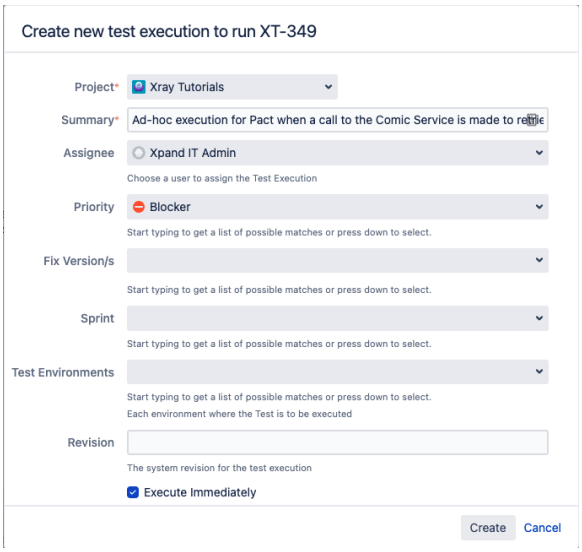
1

Create a Test Execution for the test that you have



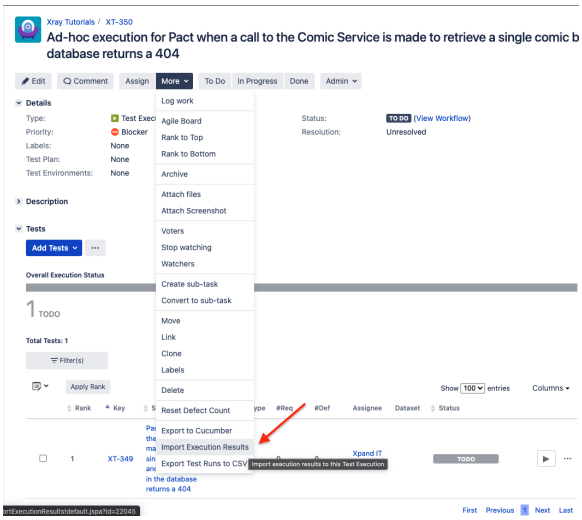
2

Fill in the necessary fields and press "*Create*"



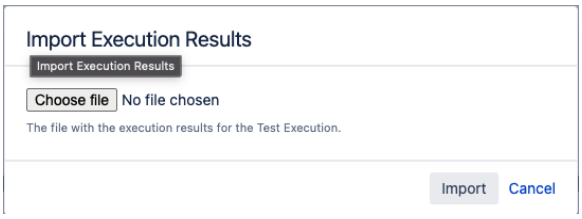
3

Open the Test Execution and import the JUnit report



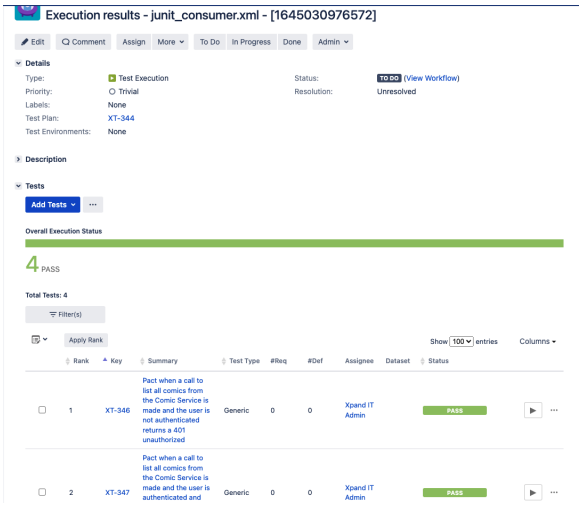
4

Choose the results file and press *"Import"*



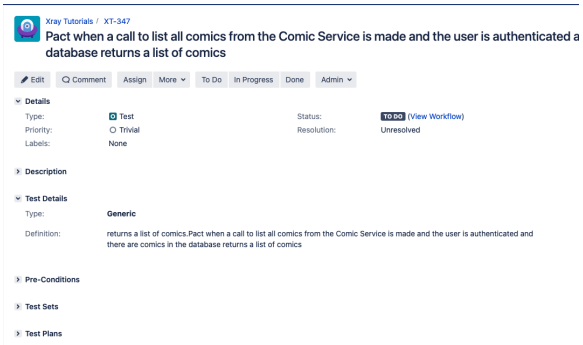
5

The Test Execution is now updated with the test results imported

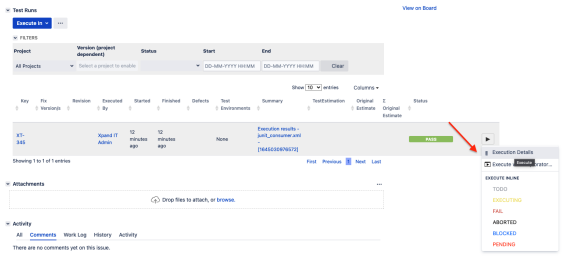


Tests implemented will have a corresponding Test entity in Xray. Once results are uploaded, Test issues corresponding to the tests are auto-provisioned, unless they already exist.

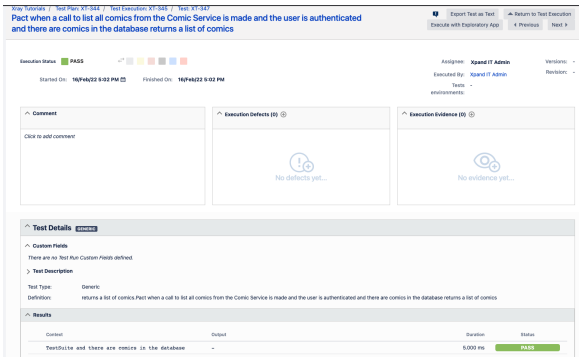
In Xray, results are stored in a Test Execution, usually a new one. The Test Execution contains a Test Run per each test that was executed.



Detailed results, including logs and exceptions reported during execution of the test, can be seen on the execution screen details of each Test Run, accessible through the *Execution details*:



As we can see here:



Tips

- after results are imported, in Jira Tests can be linked to existing requirements/user stories, so you can track the impacts on their coverage.
- results from multiple builds can be linked to an existing Test Plan, to facilitate the analysis of test result trends across builds.
- results can be associated with a Test Environment, in case you want to analyze coverage and test results by that environment later on. A Test Environment can be a testing stage (e.g. dev, staging, prepod, prod) or a identifier of the device/application used to interact with the system (e.g. browser, mobile OS).

References

- https://github.com/pact-foundation/pact-js
- https://docs.pact.io/
- https://docs.pact.io/implementation_guides/javascript
- https://pactflow.io/
- https://github.com/pact-foundation/pact_broker