MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Enhancement of Clusterability and Maintainability of Repour Service

Master's Thesis

ADAM KRÍDL

Brno, Fall 2024

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Enhancement of Clusterability and Maintainability of Repour Service

Master's Thesis

ADAM KRÍDL

Advisor: RNDr. Adam Rambousek, Ph.D.

Software Engineering

Brno, Fall 2024


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Adam Krídl

Advisor: RNDr. Adam Rambousek, Ph.D.

Acknowledgements

Firstly, I would like to thank my advisor RNDr. Adam Rambousek, Ph.D. for his time, support, and all the helpful advices he had provided me.

Secondly, I would like to thank my army of official and unofficial consultants — my colleagues at work — without who I would have no chance to manage the thesis in time: Dustin, who helped me mainly with understanding of Repour internals. Ján, for helping me to get into MPP deployments. Jožo, for helping me to get into OCP-C1 deployments. Yet, the biggest thanks belongs to Honza, who always patiently went through thousands of lines of code during PRs, endless discussions regarding high-level design, and generally, being such a tremendous mentor during all my internship period.

Thirdly, I have to thank Kuba for helping me to find such a suitable thesis, providing me all the time required to work on it, and giving me the opportunity to learn plenty of new stuff during this time.

Last but not least, I would like to thank my family who supported me during the whole time of my studies, it would not be possible without you.

Abstract

This thesis was created as an effort of Red Hat's Project Newcastle to migrate an existing microservice Repour from Python to Quarkus, and enhance some of key cloud-native properties, namely clusterability and maintainability.

As part of the thesis, such a solution was designed and implemented, improving the current solution. The new solution is prepared to be fully migrated once Project Newcastle decides to do so.

Keywords

Red Hat, microservices, Quarkus, build system

Contents

Introduction

In software development, programmers build products by writing source code. This source code is written in specific programming languages and is (often) using technologies/frameworks built upon these programming languages. Probably more often than we would like to, these underlying technologies get outdated and in order to keep the product capable of competition at the market, we are forced to migrate the technologies, sometimes even the programming language itself.

This process is typically not as difficult as the initial creation of the product (since we already have a decent knowledge of what we are going to do, and also hold old source code which can be used during the whole migration process). On the other hand, it is definitely not that trivial to migrate a product, especially in cases when we do change also the programming language itself.

The goal of this thesis is to migrate Repour microservice (which is written in Python) into a new microservice written in Quarkus framework.

The thesis consists of 6 chapters, excluding the conclusion. Chapter 1 brings a reader to the context of the thesis. Chapter 2 introduces the current solution. Chapter 3 proposes a new solution. Chapter 4 compares both implementations with a focus on maintainability, scalability, and clusterability. Finally, chapter 5 describes the deployment of the new implementation.

1 Context

Before diving into the analysis of Repour microservice itself, it is beneficial to gain an insight into the whole system whose part the microservice is, and that is Project Newcastle (PNC).

1.1 PNC Build System

As is clearly described in an IEEE Software article: "Release engineering focuses on building a pipeline that transforms source code into an integrated, compiled, packaged, tested, and signed product that's ready for release. The pipeline's input is the source code developers write to create a product"[1]. Such release engineers are present in any development team that develops an enterprise-ready product. Hence, they are present also in Red Hat Middleware[1] teams (examples of products from Red Hat Middleware are Quarkus[2] or JBoss EAP[3]).

Requirements

In order to make the release engineering pipeline effective, as many tasks as possible need to be automated. Within the middleware productization[4] pipeline, one of these automated tasks is done by the PNC build system. Hence, "customers" of PNC are middleware productization engineers and (non)functional requirements are driven by them. The most notable requirements are:

- **Structured data organization:** Productization engineers can organize their data into several levels: Products (e.g. JBoss EAP) has multiple Product Versions (e.g. *3.8*). Every product version can have multiple Product Milestones.
- **Ability to build from upstream:** Being able to fetch source code directly from upstream, e.g. GitHub repository, and make a build above any valid revision from the repository, e.g. a tag.

[1] `https://developers.redhat.com/middleware`
[2] `https://developers.redhat.com/products/quarkus/overview`
[3] `https://developers.redhat.com/products/eap/overview`
[4] This is what is called the Release Engineering discipline in Red Hat.

- **Re-usability of previously built artifacts:** Previously built artifacts are used by future builds whenever applicable.
- **Build reproducibility:** Ability to re-create previously run build.
- **Temporary builds:** Ability to make temporary builds, i.e., builds which are after some time removed from DB (with all its corresponding data (in case nothing else depends on this), e.g. built artifacts).
- **Version increment and Dependency alignment:** Automatic version increment and alignment of public dependencies to Red Hat versions (suppose for the rest of the thesis that Red Hat version is just a suffix *redhat-xxxxx* for persistent builds and *temporary-redhat-xxxxx* for temporary builds, where *x* represents a digit). The alignment to Red Hat versions has two main reasons:

  1. **Performance:** artifacts built in Red Hat, i.e., those with Red Hat versions, are stored directly in a Red Hat artifact repository, which is closer to the PNC cluster.
  2. **No later modification guarantee:** any artifact with a Red Hat version cannot be changed later. In other words, once the uniquely identified artifact is produced (e.g. *org.foo:bar:1.0.redhat-00001*) its hash remains the same for its whole lifecycle.
     Note: This helps with **build reproducibility** discussed previously.

  Example: Example of version increment together with dependency alignment is shown in Figure 1.1.
- **Automatic build scheduling:** Dependencies of build *B* which are used by *B* and need to be (newly) built, are built before build *B* is started.
- **Support for multiple build tools:** Provide support for Maven, Gradle, NPM, and SBT builds.
- **Ability to re-use build configurations:** Build configuration of the concrete product version can be re-used in its product milestones.

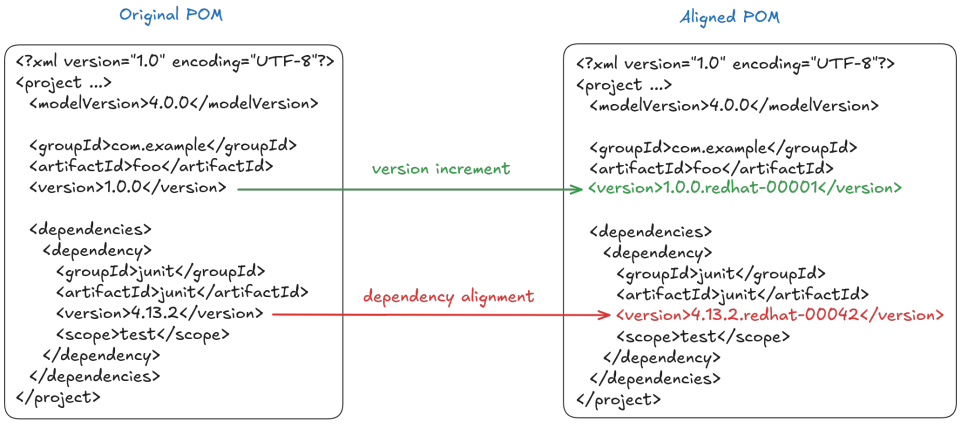

Figure 1.1: Example alignment of maven build

- **Build cancellation:** Ability to cancel running builds.

System architecture

As was already mentioned at the beginning of this chapter, PNC has microservice architecture (with 18 microservices in total).

Besides that, the system uses several external services, e.g. the internal Git SCM repository for storing downstream repositories, or the artifact repository where it stores built artifacts. We will describe the most important of them in order to achieve deeper knowledge of how PNC works.

- **Orchestrator**
  Orchestrator (Orch) is a backend entry point for all user requests. It exposes both REST API and Java client as the way to connect to it. Users interact with PNC either through PNC UI (written in React - in which case, REST API is used to connect to Orch) or Bacon (CLI app written in Java using Picocli[1] - in which case, the Java client is used to connect to Orch).

- **BPM**
  BPM (abbreviation of Business Process Manager) is a microser-

[1] `https://picocli.info/`

vice managing all the processes (e.g. build process, which will be explained in the next section, or repository creation process described in 2.2). It calls all the required endpoints from other microservices. Once the whole computation is executed, it returns the result into Orch which (typically) stores the result in some form.

- **Repour**
  Microservice for source code-related operations: synchronization of upstream repositories (publicly available, e.g. at GitHub or GitLab repository) to downstream repositories (stored in the **internal** GitLab instance). These internal repositories are the only ones, that PNC modifies, i.e., PNC is not allowed to manipulate in any way with upstream repositories.

  Besides this, Repour is also responsible for alignment operation which is delegated to additional tools (PME, GME, Project manipulator), see below.

- **PME / GME / Project Manipulator**
  Manipulators executing alignment operations, specifically:

  - **PME:** Abbreviation of POM Manipulation Extension, which handles alignment in POM files during Maven builds.
  - **GME:** Abbreviation of Gradle Manipulation Extension, which handles alignment in *build.gradle* files during Gradle builds.
  - **Project Manipulator:** Handles alignment in *package.json* files during NPM builds.

  Each of these manipulators is delivered as a standalone JAR, which is used (in the context of PNC) from Repour microservice.

- **DA**
  DA stands for dependency analyzer and is used by manipulators in two ways: firstly, when project increment is being done, DA returns **the next version in the sequence** (e.g. when last used per the requested GAV was *redhat-00041*, it will return *redhat-00042*). Secondly, when dependency alignment is being done,

DA returns **the most feasible version** of any previously built artifact of the requested GAV.

- **Build container**
  PNC builds are run within a minimalistic OCI container containing just enough tools (based on a build configuration) to properly run a build.
- **Environment driver**
  To ensure efficient scaling of PNC, it is essential to automate the deployment of build containers. This process is handled by the environment driver.
- **Artifact repository**
  Once a build is successful, its built artifacts have to be stored in some central place in order to be (potentially) re-used by subsequent builds. PNC uses for this sake the internal Red Hat repository called Indy.

Build process

To finalize our insight into how PNC works, we need to describe the build process from a high-level perspective:

1. **Trigger a build**
   A build is triggered (almost exclusively) either from PNC UI or Bacon CLI. In both cases, this results in Orch's trigger build endpoint being called. Every build is configured by its build configuration containing all the necessary information, e.g.:
   - **repository:** repository with source code, e.g. `https://github.com/wildfly/wildfly`
   - **git reference:** (branch, tag or commit) revision which will be used to clone source code, e.g. *34.0.0.Final*
   - **build type:** either Maven, Gradle, NPM, or SBT
   - **build environment:** the build environment used in build container
   - **build script:** a script executed in the build container, e.g. *mvn clean deploy*

2. **Start an alignment**
   Repour clones specified repository locally into the container environment in which it runs. Subsequently, it starts the alignment operation by calling the corresponding manipulator based on the build type (for instance, for the Maven build type, the PME manipulator is picked).

3. **Alignment process**
   A manipulator configured from Repour communicates with DA to perform both version increment and dependencies alignment.

4. **Push alignment changes**
   Once the manipulator successfully finishes, alignment changes are pushed into a downstream repository.

5. **Create a build container**
   Create the environment for the build itself based on the build environment from the build configuration.

6. **Clone alignment changes**
   Before a build container runs the build itself, it needs to clone alignment changes from the downstream repository pushed by Repour in step 4.

7. **Run a build**
   Run the build script from a build configuration. During a build, a build container communicates with an artifact repository in order to fetch artifacts.

8. **Store built artifacts**
   Once a build succeeds, its produced artifacts need to be stored in the artifact repository in order to be accessible by subsequent builds.

9. **Save the build result into DB**
   Save all the build-related data into the orch database. This holds information like used build configuration, artifacts produced by the build, or artifacts used by the build.

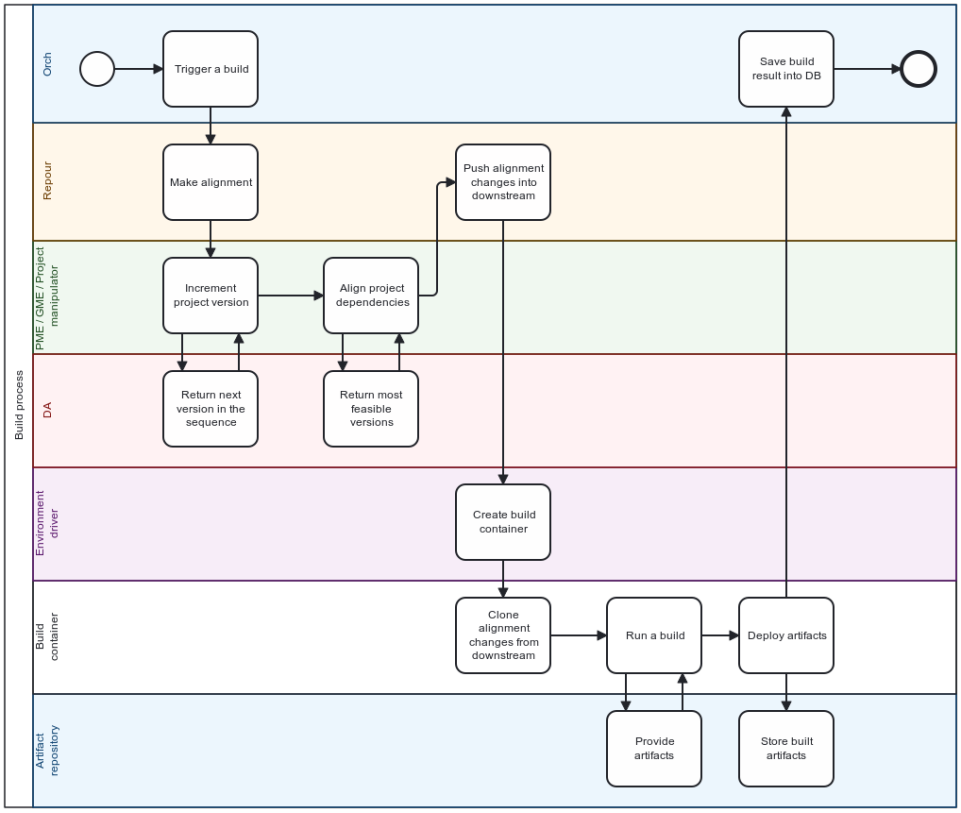

Figure 1.2: BPMN build process diagram

The whole build process is also visualized by BPMN diagram[1] shown in the Figure 1.2.
Note: Even though the BPM microservice manages the whole process, it is shown neither in the text description nor in Figure 1.2 in order to prevent showing all the unnecessary details.

1.2 Quarkus framework

As was previously mentioned in Introduction, the goal of this thesis is to rewrite the existing microservice into a new microservice written in

[1] `https://miro.com/diagramming/what-is-bpmn/`

Quarkus. This makes a need for a brief explanation of what Quarkus is, a necessity.

Jakarta EE

Back before 1999, when one wanted to develop an enterprise application in Java, there was no other choice than to implement everything in Java SE. Even though Java has a relatively extensive standard library, provided abstractions were not sufficient for writing enterprise applications because application developers needed to understand these (quite) low-level abstractions upon which they could build their application.

This led to the creation and rise of Jakarta EE[1]. Citing an official Jakarta EE website: "Jakarta EE is the standard, a set of specifications for enterprise Java application development."[2]. The previous sentence is short and simple, yet precisely defines what Jakarta EE is about.

Each specification is a higher-level abstraction built on top of Java SE, which can be used by application developers to develop their enterprise application without the need to understand all low-level details (this knowledge is delegated to those who implement the specification).

As an example, suppose a connection with a database has to be done from the application. We can use Jakarta Persistence specification[2] providing more higher-level abstractions compared to abstractions provided by *java.sql* package (which we would probably use from Java SE to implement database connection).

Every specification is vendor-neutral and is evolved by Jakarta EE Working Group[3], which is a global community consisting of representatives of leading technology organizations and individuals. For a visual representation of the above explanation, see Figure 1.3.

---

[1] Initially, its name was Java EE and was renamed once Oracle announced its submission under the Eclipse foundation.

[2] `https://jakarta.ee/specifications/persistence/`

[3] `https://jakarta.ee/about/working-group/`

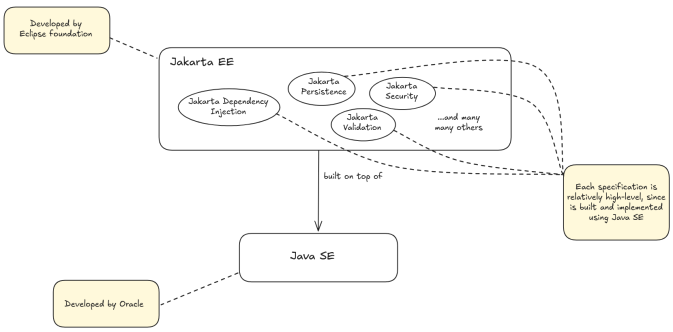

Figure 1.3: Jakarta EE building blocks

Quarkus

Jakarta EE specification process[1] is long-running and formal, which leads to specifications being mature and stable. Although it may not seem so at first glance, this can be quite a limiting disadvantage: typical examples are teams who develop enterprise applications but want to use the latest enterprise standards. Using Jakarta EE, such an integration can take place in a year. This was one of the reasons why Microprofile[2] got popular. Another one being its specialization in optimizing enterprise Java for a microservices architecture.

From the architectural point of view, Microprofile is built on the same principles as Jakarta EE: consists of specifications and once we use Microprofile API, we have access to the functionality of any of its specifications.

Quarkus is an open-source implementation[3] of Microprofile API and has a great focus on microservices: "Quarkus is a Java framework that targets microservices and serverless system development. Quarkus emerged as an alternative to the existing Java microservices stack to provide an application framework that delivers an unmatched performance benefits while still providing a development model utilizing the APIs of popular libraries and Java standards that the Java ecosystem has been practicing for years."[3][4].

---

[1] `https://jakarta.ee/committees/specification/guide/`

[2] `https://microprofile.io/`

[3] `https://github.com/quarkusio/quarkus`

Documentation is separated as a series of guides[1]. When describing implementation details later in the text, we will sometimes reference concrete guides from this series.

[1] `https://quarkus.io/guides/`

2 Repour microservice

In Chapter 1, we introduced the build process and roughly discussed the role of Repour within the PNC Build System. In this chapter, we will explore Repour[5] in more detail — in particular, we will identify all the endpoints present and recognize when are these endpoints called.

2.1 Server intialization

Repour is written in Python using asyncio library[1], which is designed to write concurrent code using the *async/await* syntax. The core of asyncio applications is an event loop[2], which takes care of asynchronous tasks, its callbacks, IO operations, and others.

Repour server is being initialized through several methods — below are listed the most important parts of the initialization (all of these listings are taken[3] from Repour's GitHub using *akridl-thesis* tag as the revision: `https://github.com/project-ncl/repour/blob/akridl-thesis`).

Listing 2.1: Server initialization: *start_server* method

```
1 def start_server(bind, repo_provider, repour_url
    , adjust_provider):
2     loop = asyncio.get_event_loop()
3
4     loop.run_until_complete(
5         init(
6             loop=loop,
7             bind=bind,
8             repo_provider=repo_provider,
9             repour_url=repour_url,
10            adjust_provider=adjust_provider,
```

[1] `https://docs.python.org/3/library/asyncio.html`

[2] `https://docs.python.org/3/library/asyncio-eventloop.html#asyncio-event-loop`

[3] There is not 1:1 correspondence, since pseudocodes covered in this text are simplified.

```
11          )
12      )
13
14      try:
15          loop.run_forever()
16      except KeyboardInterrupt:
17          logger.debug("KeyboardInterrupt")
```

In the listing 2.1 displaying *start_server*()[1], we can see:

- **L2:**
  Obtaining asyncio's event loop.
- **L4-12:**
  Initialize the application as described at the listing 2.2.
- **L14-17:**
  Only once is the initialization of the event loop completed, start the server indefinitely. Allow the server to be killed by a keyboard interrupt.

Listing 2.2: Server intialization: *init* method

```
1 async def init(loop, bind, repo_provider,
    repour_url, adjust_provider):
2     app = web.Application(loop=loop)
3
4     external_to_internal_source = endpoint.
        validated_json_endpoint(
5         shutdown_callbacks,
6         validation.external_to_internal,
7         external_to_internal.translate,
8         repour_url,
9     )
10
11    clone_source = endpoint.
        validated_json_endpoint(
```

[1] https://github.com/project-ncl/repour/blob/akridl-thesis/repour/server/server.py#L99

```
12          shutdown_callbacks,
13          validation.clone,
14          clone.clone,
15          repour_url,
16          send_logs_to_bifrost=False,
17      )
18
19      adjust_source = endpoint.
           validated_json_endpoint(
20          shutdown_callbacks, validation.
               adjust_modeb, adjust.adjust,
               repour_url
21      )
22
23      internal_scm_source = endpoint.
           validated_json_endpoint(
24          shutdown_callbacks,
25          validation.internal_scm,
26          internal_scm.internal_scm,
27          repour_url,
28          send_logs_to_bifrost=False,
29      )
30
31      app.router.add_route("GET", "/", info.
           handle_request)
32      app.router.add_route(
33          "POST", "/git-external-to-internal",
               external_to_internal_source
34      )
35      app.router.add_route("POST", "/clone",
           clone_source)
36      app.router.add_route("POST", "/adjust",
           adjust_source)
37      app.router.add_route("POST", "/internal-scm"
           , internal_scm_source)
38      app.router.add_route("POST", "/cancel/{
           task_id}", cancel.handle_cancel)
```

```
39      app.router.add_route("GET", "/version", info
           .handle_version)
40
41      server = await loop.create_server(
42          app.make_handler(access_log=None), bind[
               "address"], bind["port"]
43      )
```

In the listing 2.2 displaying *init*()[1], we can see:

- **L2:**
  Initialize *app* as a web server.
- **L4-29:**
  Initialize endpoint handlers.
- **L31-40:**
  Register previously initialized endpoint handlers to routes.
- **L42-44:**
  Finally, create a TCP server bound to the specified address and port.

2.2 Endpoints

How endpoints work

Within the listing 2.2, every endpoint is created by *endpoint.validated_json_endpoint*()[2] method, which does (from high-level perspective) the following:

1. **Parse JSON input request**
   In case parsing fails, 400 is returned to the client right away.
2. **Validate parsed request**
   Every endpoint provides a valid format that all the incoming

[1] https://github.com/project-ncl/repour/blob/akridl-thesis/repour/server/server.py#L32

[2] https://github.com/project-ncl/repour/blob/akridl-thesis/repour/server/endpoint/endpoint.py#L99

requests must meet. In case an invalid request is given, 400 is returned.

3. **Call a function**
   Call a function provided as one of the arguments, which is doing the actual computation.

4. **Send a callback**
   In case a callback was provided within the request, send the callback. The method handling the callback sending is implemented to be resilient by using retries and exponential backoff between fails.

What each endpoint does

The listing 2.2 (at lines 31-39) registers all the endpoint handlers into the application router. Hence, it is clearly determined what endpoints Repour has.

- ***POST /git-external-to-internal***
  Translation of external Git repository URL into its corresponding internal counterpart.

  For instance, suppose given is external URL `https://github.com/akridl/empty`. Internal URL corresponding to this external URL could be e.g. `https://gitlab.internal.redhat.com/pnc/akridl/empty`[1].

- ***POST /clone***
  Clone (either partially or fully) from an external Git repository into the corresponding internal Git repository.

- ***POST /internal-scm***
  Creation of a Source Code Management (SCM) repository within the Red Hat internal GitLab instance and PNC workspace.

- ***POST /adjust***
  Trigger an alignment operation (as described in 1.1).

[1] Unlike external URL, internal URL is hidden behind Red Hat's firewall, hence not publicly available. However, for security reasons, this URL does not point to the real Red Hat GitLab instance anyway.

- ***POST /cancel/{taskId}***
  Cancel the operation with the given *taskId*.

- ***GET /version***
  Information about the running JAR in the JSON format containing the following fields:

  - **name:**
    A name of the application
  - **version:**
    A version of the application
  - **commit:**
    A commit upon which was the JAR built
  - **builtOn:**
    Date and time when was the JAR built (in ISO-8601 UTC format[1])
  - **components:**
    An array of additional components in the same format as this JSON being described.

  Listing 2.3: Example of */version* endpoint output

```
{
    "name": "reqour-rest",
    "version": "1.0.0-SNAPSHOT",
    "commit": "17
       f4ad23ee059706154cfb527f72846ef5b8ccc9
       ",
    "builtOn": "2024-11-13T16:18:03Z",
    "components": []
}
```

- ***GET /***
  Minimalistic HTML containing a subset of the information provided by */version* endpoint.

[1] https://www.ietf.org/rfc/rfc3339.txt

When are endpoints called

In this part, we will describe when is each endpoint called.

During repository creation process

In section 1.1, we described the build process. However, that is not the only process PNC can execute (although being run the most often).

Another such process is the process of repository creation. This is used by productization engineers when they want to create an internal SCM repository within the PNC workspace, where PNC is allowed to contribute (e.g. push alignment changes). Such an internal repository is (at least when being created) synchronized to its external repository counterpart.

The repository creation process works (from a high-level perspective) as follows:

1. **Orch handles *POST /create-and-sync* request**
   This endpoint is triggered either from PNC UI or Bacon CLI.

2. **Orch triggers repository creation process in BPM**
   BPM starts the repository creation process.

3. **BPM calls *POST /git-external-to-internal***
   This is used to validate whether an internal repository with such an internal URL does not exist already.

4. **BPM calls *POST /internal-scm***
   Create a new internal SCM repository within the PNC workspace.

5. **BPM calls *POST /clone***
   Clone an external repository into an internal one.

Visual representation of the repository creation process is depicted in Figure 2.1.

During build process

As was previously described in Figure 1.2, Repour makes the alignment operation, which is triggered by requesting *POST /adjust*.

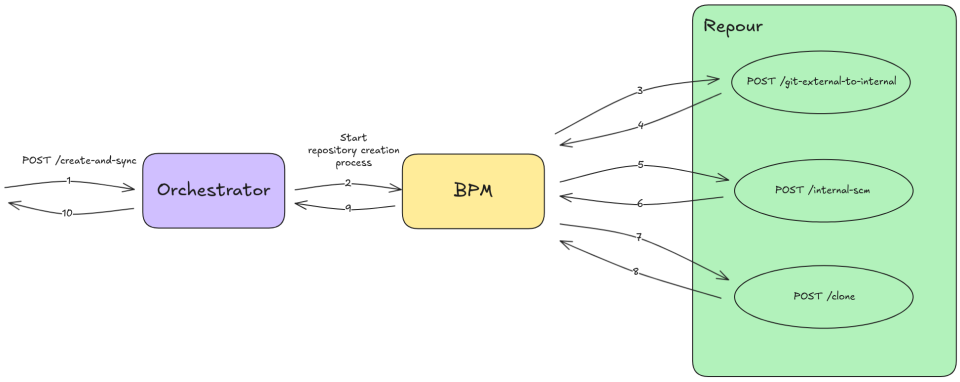

Figure 2.1: Repository creation process

During build cancellation

As was mentioned in 1.1, one of the PNC requirements is the ability to cancel any running build. For this, those PNC microservices that perform long-running operations expose *POST /cancel* endpoint.

An alignment is an example of such a long-running operation during a build, which can be canceled by invoking the *POST /cancel* endpoint.

During debugging

For debugging purposes serve endpoints *GET /* and *GET /version* that are not called as part of any process. Typical usage of these two endpoints is when a PNC developer wants to check what version has been deployed in an environment.

3 Reqour microservice

In the previous chapter, we described the Repour microservice, i.e., the current implementation present in the PNC Build System. Among others, we identified all the endpoints present and observed when is each endpoint called.

This chapter is about the new implementation of Repour, called Reqour[1]. Similarly to Repour's source code, Reqour's source code is publicly available[6]. The current chapter describes every endpoint in more detail than Chapter 2.

3.1 Translation

As the first endpoint, we will explore *POST /git-external-to-internal*, which is in a source code referred as the translation endpoint.

3.1.1 Analysis

Description

The translation endpoint makes the translation from an external Git repository URL (refer to this as an external URL in future reference) into its corresponding internal Git repository URL (refer to this as an internal URL in future reference).

Request DTO

Translation request DTO is JSON of the following format:

```
{
    "externalUrl": "string"
}
```

[1]The name similarity is no coincidence. P to Q change indicates the reimplementation of Repour from Python to Quarkus.

Request validation

For a request to be valid, *externalUrl* field has to match one of the following formats:

- **SCP-like format:**
  Valid SCP-like format meets the regex:
  ^[*protocol://*]*user@host*[*:port*]:[*organization/*]*repository.git$*
  where:
  - **protocol:** an underlying protocol we use when reading the data from the external URL, e.g. *http*, or *ssh*
  - **user:** a specific user performing the operation, e.g. *git*
  - **port:** a port of the target address, e.g. *443*
  - **organization:** a name of the organization owning the repository, e.g. the organization of the external URL of *git@github.com:**wildfly**/wildfly.git* is *wildfly*[1]
  - **repository.git:** a name of the repository with mandatory *.git* suffix
- **Non-SCP-like format:**
  Valid Non-SCP-like format meets the regex:
  ^*protocol://*[*user@*]*host*[*:port*][*/organization*]*/repository*[*.git*]*$*
  where *protocol*, *user*, *host*, *port*, *organization*, and *repository* has the same meaning as in SCP-like format.
- **File-like format:**
  Valid File-like format meets the regex:
  ^*file://path$*
  where *path* is either absolute or relative path.

Note: Entries encapsulated within [] denote optional part.

Examples:

- git@github.com:my-repo.git
  Valid SCP-like format

[1] `https://github.com/wildfly`

- scp://git@gitlab.com:22:repo.git
  Valid SCP-like format
- git@github.com:my-repo
  Invalid SCP-like format: missing mandatory *.git* suffix
- git@github.com/my-repo.git
  Invalid SCP-like format: hostname has to be separated with ':', not '/'
- https://github.com/my-organization/repo
  Valid Non-SCP-like format
- ssh://github.com:22/my-repo
  Valid Non-SCP-like format
- github.com:/project
  Invalid Non-SCP-like format: missing port
- github.com/repo
  Invalid Non-SCP-like format: missing protocol

Response DTO

Translation response DTO is JSON of the following format:

```
{
    "externalUrl": "string",
    "internalUrl": "string"
}
```

where *externalUrl* is the external URL provided in a request DTO and *internalUrl* is the translated URL corresponding to the external one.

Functionality

Once the validation of an external URL is done, we parse out *organization* and *repository* (so that repository does not contain *.git* suffix anymore).

Then the final internal URL is the concatenation of:
git@<Red Hat GitLab URL>:<PNC workspace name>/[organization]/repo.git.

Supposing that:

- Red Hat GitLab URL is *gitlab.redhat.com*, and
- PNC workspace name is *pnc*,

then for the external URL `https://github.com/wildfly/wildfly`, the translated internal URL would be: `git@gitlab.redhat.com:pnc/wildfly/wildfly.git`.

3.1.2 Design

Firstly, there will be an endpoint handler handling all *POST* requests at the path */external-to-internal*.

The endpoint will not contain any business functionality but will delegate the computation to the underlying service. Once the service responds with the result, the endpoint simply returns the resulting DTO.

The above word description is visualized in Figure 3.1.

Translation operation can be synchronous since consisting only of parsing, validation, and final concatenation (none of which takes a non-trivial amount of time).

3.1.3 Implementation

Listing 3.1: Translate request

```
public class TranslateRequest {
    @ValidGitRepositoryURL
    String externalUrl;
}
```

For validation of the request, we use Jakarta Validation API[1]. The translation request[2] has the form as shown in Figure 3.1. Validation is triggered by *@ValidGitRepositoryURL* annotation[3] present also during

[1] `https://jakarta.ee/specifications/bean-validation/`

[2] `https://github.com/project-ncl/pnc-api/blob/akridl-thesis/src/main/java/org/jboss/pnc/api/reqour/dto/TranslateRequest.java`

[3] `https://github.com/project-ncl/pnc-api/blob/akridl-thesis/src/main/java/org/jboss/pnc/api/reqour/dto/validation/ValidGitRepositoryURL.java`

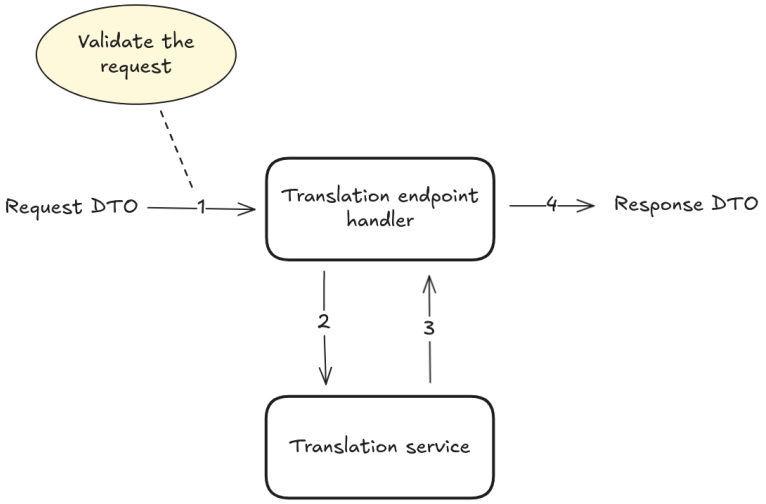

Figure 3.1: Translation design

runtime and being validated by *GitRepositoryURLValidator*[1] during deserialization of the request.

The validator ensures the correct format of *externalUrl* (as described in 3.1.1). In addition, the validator internally uses parsed representation, which is later used by the translation service[2]. Hence, leaving only minor adjustments of parsed tokens and the final concatenation for the service.

Attached PRs

The translation implementation was introduced by the following PRs:

- `https://github.com/project-ncl/pnc-api/pull/216` Introducing request and response DTOs

[1] `https://github.com/project-ncl/pnc-api/blob/akridl-thesis/src/main/java/org/jboss/pnc/api/reqour/dto/validation/GitRepositoryURLValidator.java`

[2] `https://github.com/project-ncl/reqour/blob/akridl-thesis/core/src/main/java/org/jboss/pnc/reqour/service/TranslationServiceImpl.java#L31`

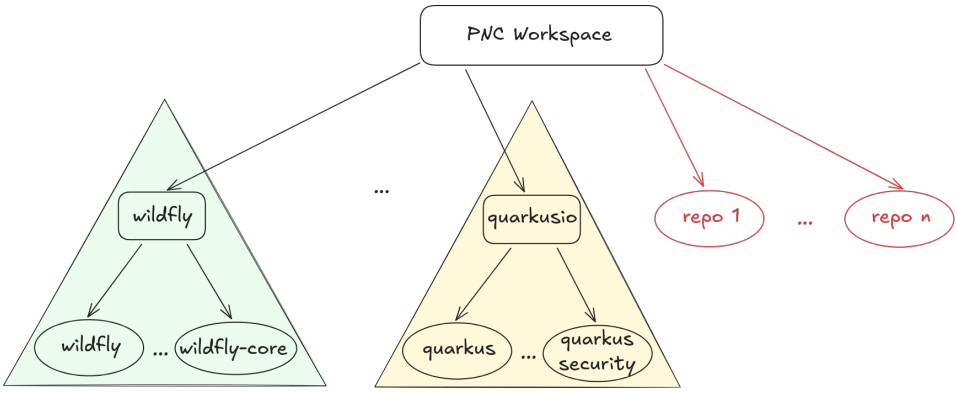

Figure 3.2: Hierarchy under PNC workspace at GitLab

- `https://github.com/project-ncl/pnc-api/pull/218`
  Add the validation using Jakarta Validation API

- `https://github.com/project-ncl/reqour/pull/2`
  Introduce the business logic

3.2 Internal SCM repository creation

This operation is triggered by *POST /internal-scm* call.

3.2.1 Analysis

Description

The endpoint serves for creation of internal SCM repository – such repository is present in the internal Red Hat GitLab instance. In addition, the created repository is under PNC workspace, which is important in order to guarantee that Reqour has write permissions into a repository.

In addition, repositories can be further nested under organization (which is almost exclusively the case, i.e., only very few repositories are directly under PNC workspace). The complete hierarchy is depicted in Figure 3.2.

Request DTO

Internal SCM repository creation request DTO is JSON of the following format:

```
{
  "project": "string",
  "taskId": "string",
  "callback": {
    "method": "string",
    "uri": "string",
    "headers": [
      {
        "name": "string",
        "value": "string"
      }
    ]
  }
}
```

Fields of the DTO have the following meaning:

- **project**
  Name of the repository to be created, e.g. *wildfly*. Optionally, a user can specify also the name of the organization under which should this repository be created. In the latter case, the organization is separated from the repository name by the '/' character, e.g. *wildfly/wildfly*.
- **taskId**
  Task ID to be assigned for this (asynchronous) operation.
- **callback**
  Callback where to send the result once the operation is finished.

Request validation

Request validation is being done as follows:

- **project**
  The only validation taking place in the Reqour codebase is checking for at most one '/' character. Other checks, e.g. for valid

characters in the repository name, take place sooner in the process, e.g. in Orchestrator.

- **taskId**
  Task ID cannot be blank.

- **callback**
  Callback must be present, i.e., cannot be *null*.

Response DTO

The response DTO (sent as callback payload) is JSON of the following format:

```
{
    "readonlyUrl": "string",
    "readwriteUrl": "string",
    "status": {
        "enum": [
            "SUCCESS_CREATED",
            "SUCCESS_ALREADY_EXISTS",
            "FAILED"
        ],
        "type": "string"
    },
    "callback": {
        "status": {
            "enum": [
                "SUCCESS",
                "FAILED",
                "TIMED_OUT",
                "CANCELLED",
                "SYSTEM_ERROR"
            ],
            "type": "string"
        },
        "id": "string"
    }
}
```

Fields have the following meanings:

- **readonlyUrl**
  Read-only URL of the repository, used e.g. for cloning.
- **readwriteUrl**
  Read-write URL of the repository, used e.g. for pushing.
- **status**:
  Status of the creation operation. This field is used to differentiate whether the requested repository is created or has already existed.
- **callback**:
  Callback describing the resulting status of the Reqour's async task.

3.2.2 Design

Since the operation is creating a new internal SCM repository at the internal GitLab, the whole operation might take several seconds to finish, forcing us to make the operation asynchronous.

The endpoint handler creates the task for this operation. Task logic is done by the underlying service, which parses all the necessary information both from the request and from a configuration. For instance, read-only and read-write URLs are composed by completing the project path (parsed from request's *project* field) into the read-only and read-write templates[1].

Once all the required project-related information is acquired, we create the project using the GitLab API service, which sends the creation request with all the details. Lastly, the task specifies where and what to send in the callback.

The created task is propagated into an asynchronous task executor, which will start the execution on another thread.

Finally, the endpoint handler returns 202 Accepted to inform a client that an operation is being executed.

The design described above is visualized using Figure 3.3.

[1] `https://github.com/project-ncl/reqour/blob/akridl-thesis/core/src/main/resources/application.yaml#L14-L15`

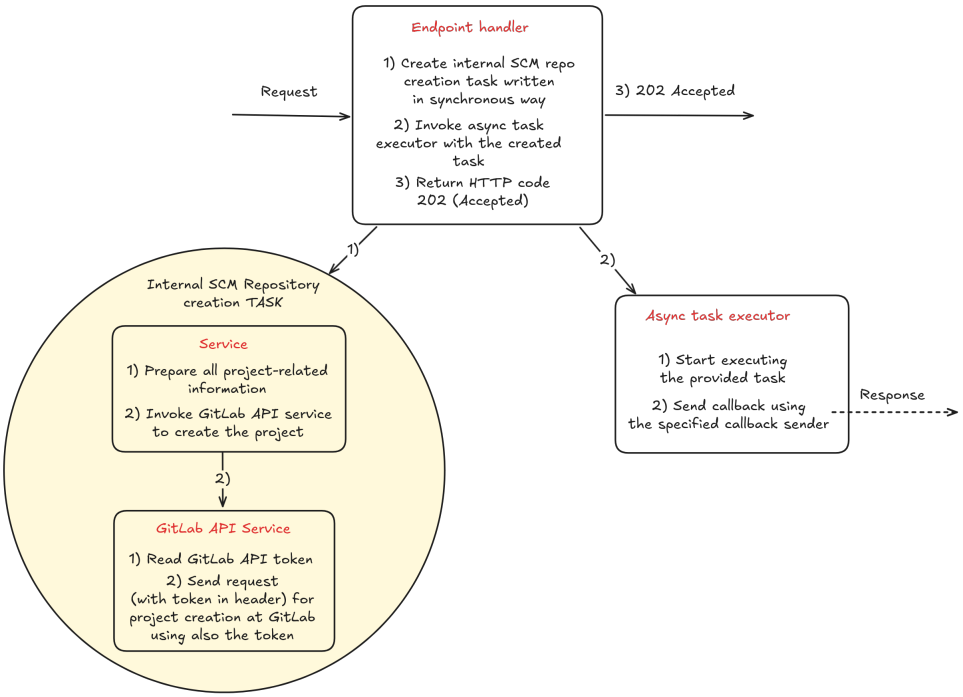

Figure 3.3: Design of internal SCM repository creation

3.2.3 Implementation

Task executor

The core part of an implementation of any asynchronous task within Reqour is **asynchronous task executor**, whose design is depicted in Figure 3.4. Its implementation uses *CompletableFuture*[1] API in order to execute the task asynchronously.

The implementation is shown in Listing 3.2[2]. At line 7, we are using a managed executor[3] which invokes a synchronous function on an underlying thread, hence, not blocking the caller thread. At line 8, in case of an exception, we handle it appropriately. Finally, at line 9, the callback is being sent using the provided callback sender.

Listing 3.2: Asynchronous task executor

```
1  public <T, R> void executeAsync(
2          Request callbackRequest,
3          T request,
4          Function<T, R> syncExecutor,
5          BiFunction<T, Throwable, R> errorHandler
               ,
6          BiConsumer<Request, R> callbackSender) {
7      executor.supplyAsync(() -> syncExecutor.
           apply(request))
8              .exceptionally(t -> errorHandler.
                   apply(request, t))
9              .thenAccept(res -> callbackSender.
                   accept(callbackRequest, res));
10 }
```

[1] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/CompletableFuture.html

[2] https://github.com/project-ncl/reqour/blob/akridl-thesis/core/src/main/java/org/jboss/pnc/reqour/common/executor/task/TaskExecutorImpl.java#L36-L45

[3] https://download.eclipse.org/microprofile/microprofile-context-propagation-1.0/apidocs/org/eclipse/microprofile/context/ManagedExecutor.html

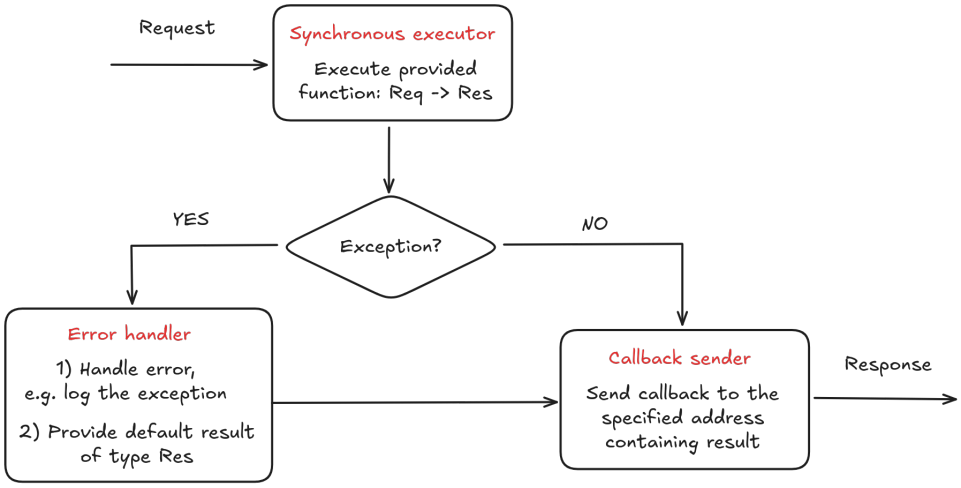

Figure 3.4: High-level overview of asynchronous task executor

Operation implementation

The implementation of internal SCM repository creation itself contains just a call to the asynchronous task executor, see Listing 3.3[1].
Note: Even though the implementation seems to be trivial enough, the most difficult part (once having asynchronous task executor abstraction) is still hidden under *service::createInternalSCMRepository*[2], which contains the whole business logic.

Listing 3.3: Internal SCM repository creation endpoint handler

```
1 public void createInternalSCMRepository(
    InternalSCMCreationRequest creationRequest) {
2     taskExecutor.executeAsync(
3             creationRequest.getCallback(),
4             creationRequest,
```

[1] https://github.com/project-ncl/reqour/blob/akridl-thesis/rest/src/main/java/org/jboss/pnc/reqour/rest/endpoints/InternalSCMRepositoryCreationEndpointImpl.java#L50-L59

[2] https://github.com/project-ncl/reqour/blob/akridl-thesis/core/src/main/java/org/jboss/pnc/reqour/service/GitlabRepositoryCreationService.java#L42-L75

```
5               service::createInternalSCMRepository
                   ,
6               this::handleError,
7               callbackSender::
                   sendInternalSCMRepositoryCreationCallback
                   );
8
9       throw new WebApplicationException(Response.
           Status.ACCEPTED);
10  }
```

Attached PRs

The implementation of internal SCM repository creation was introduced by the following PRs:

- `https://github.com/project-ncl/pnc-api/pull/232`
  Introduce request and response DTOs
- `https://github.com/project-ncl/reqour/pull/5`
  Implement the business logic

3.3 Cloning

Next, we will discuss the cloning operation, triggered by *POST /clone*.

3.3.1 Analysis

Description

The clone operation clones the content of the source repository and pushes it into a target repository.

Request DTO

Clone request DTO is JSON of the following format:

```
{
  "originRepoUrl": "string",
```

```
  "targetRepoUrl": "string",
  "ref": "string",
  "taskId": "string",
  "callback": {
    "method": "string",
    "uri": "string",
    "headers": [
      {
        "name": "string",
        "value": "string"
      }
    ]
  }
}
```

where DTO fields have the following meaning:

- **originRepoUrl**
  URL of the origin repository where to clone from.
- **targetRepoUrl**
  URL of the target repository where to push cloned code.
- **ref**
  Optional reference (either branch, tag, or commit) within the origin repository from which to clone from.
- **taskId, callback**
  Same meaning as in Internal SCM repository creation DTO described in 3.2.1.

Request validation

The request DTO is being validated in the following way:

- **originRepoUrl, targetRepoUrl**
  Both fields are validated in the same way as *externalUrl* of translation request, as was described in 3.1.1.
- **taskId, callback**
  Same validation takes place as within internal SCM repository creation request (described in 3.2.1).

Response DTO

The response DTO (sent as callback payload) is JSON of the following format:

```
{
    "originRepoUrl": "string",
    "targetRepoUrl": "string",
    "ref": "string",
    "callback": {
        "status": {
            "enum": [
                "SUCCESS",
                "FAILED",
                "TIMED_OUT",
                "CANCELLED",
                "SYSTEM_ERROR"
            ],
            "type": "string"
        },
        "id": "string"
    }
}
```

Description of all the fields but *callback* should be obvious from the request DTO. The *callback* denotes the resulting status of the Reqour's async task, same as was the case also in internal SCM repository creation response DTO.

3.3.2 Design

The cloning operation might take several seconds to finish since we need to clone (potentially the whole) upstream repository and push it to the corresponding downstream repository. Because of this, we will use the asynchronous task executor.

Listing 3.4: Cloning the whole repository

```
$ git clone --mirror original-repo.git /path/
   cloned-directory/.git
$ cd /path/cloned-directory
```

```
$ git config --bool core.bare false
```

The operation itself consists of several Git commands. For instance, in case a client does not provide the *ref* field, i.e., cloning of the whole repository takes place, the operation consists of several steps[7] shown in Listing 3.4. Hence, the remaining part is to decide how to execute such git commands. On the one hand, as of today, there already exists a Git client written in Java, which seems to be maintained and ready to be used right away[1]. On the other hand, we are required to process both standard and standard error outputs in real-time in order to inform a user about build progress, so-called **live logs**. In addition, later during the alignment operation (will be described in 3.4), we will need to support yet other commands (in general, any shell command). This led to the creation of our own abstraction, called **Process Executor**.

Process executor

The process executor receives at input a process context, which consists of:

- **command:** command to be executed by the process executor, e.g. *git init –bare*
- **working directory:** the directory where should be the command executed
- **extra environment variables:** additional environment variables just for the process
- **STDOUT consumer:** consumer of the standard output of the executed process
- **STDERR consumer:** consumer of the standard error output of the executed process

Afterwards, it starts executing the provided command and redirecting stdout and stderr into their corresponding consumers. Once the command is finished, the process executor returns its exit code. The design of the process executor is visualized in Figure 3.5.

[1] https://github.com/eclipse-jgit/jgit

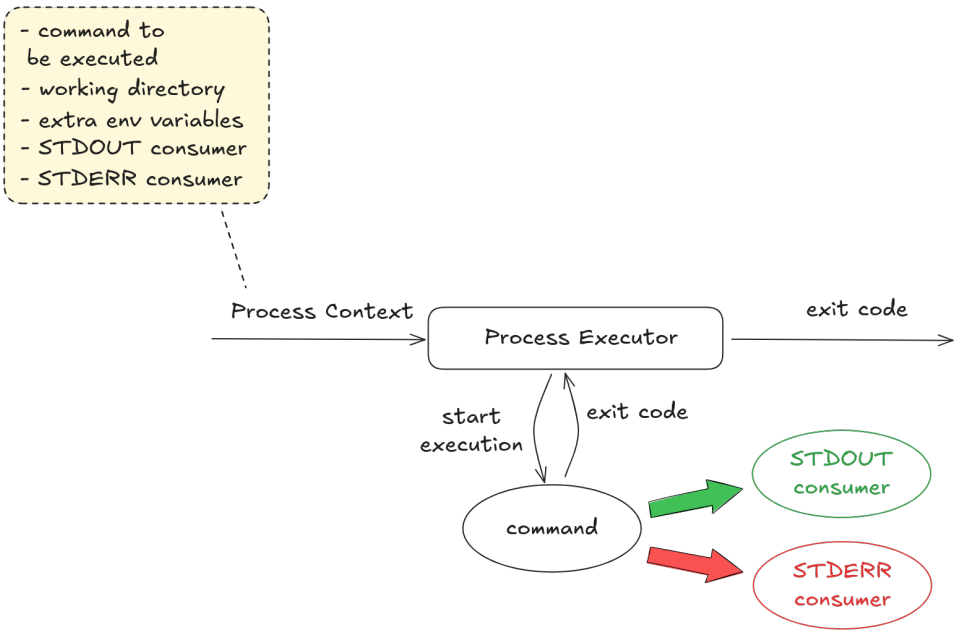

Figure 3.5: High-level overview of process executor

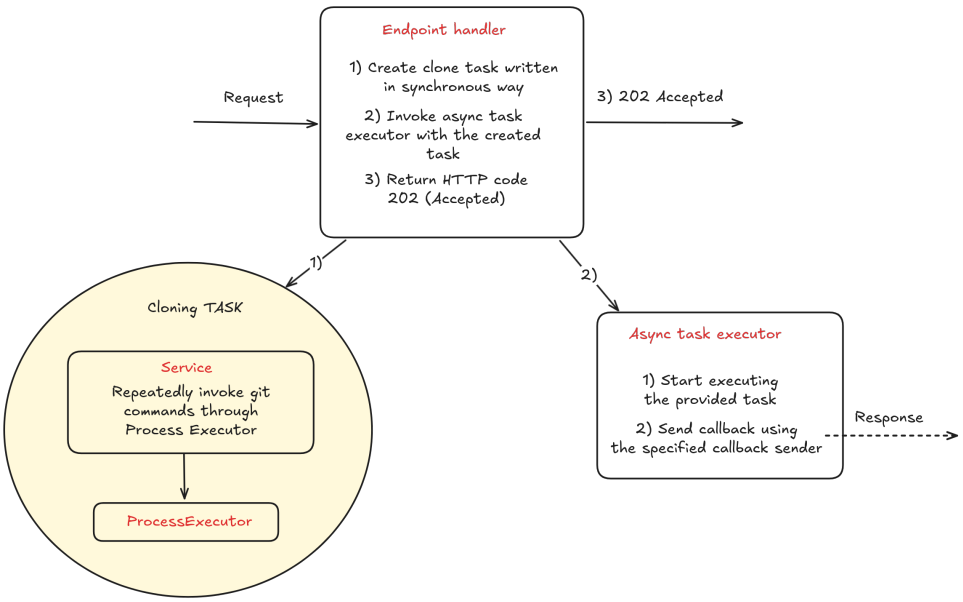

Figure 3.6: Design of the clone operation

Operation design

The design of the clone operation itself is similar to the design of internal SCM repository creation, see Figure 3.6.

3.3.3 Implementation

Process executor

The implementation of the process executor is shown in the Listing 3.5[1]. It uses *ProcessBuilder*[2] to create instances of *Process*[3] class. Furthermore, similarly as asynchronous task executor (see 3.2), it uses completable futures and the container's managed executor in order

[1] https://github.com/project-ncl/reqour/blob/akridl-thesis/core/src/main/java/org/jboss/pnc/reqour/common/executor/process/ProcessExecutorImpl.java#L35-L73

[2] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/ProcessBuilder.html

[3] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Process.html

to create asynchronous tasks. This is hidden under the hood of calls *createOutputConsumerProcess* at lines 7 and 10.

Listing 3.5: Process executor implementation

```
1  public int execute(ProcessContext processContext
       ) {
2      ProcessBuilder processBuilder = new
           ProcessBuilder(processContext.getCommand()
           )
3              .directory(processContext.
                   getWorkingDirectory().toFile());
4      processBuilder.environment().putAll(
           processContext.getExtraEnvVariables());
5
6      Process processStart = processBuilder.start
           ();
7      CompletableFuture<Void>
           stdoutConsumerProcess =
           createOutputConsumerProcess(
8              processStart.inputReader(),
9              processContext.getStdoutConsumer());
10     CompletableFuture<Void>
           stderrConsumerProcess =
           createOutputConsumerProcess(
11             processStart.errorReader(),
12             processContext.getStderrConsumer());
13
14     int exitCode = processStart.waitFor();
15     stdoutConsumerProcess.join();
16     stderrConsumerProcess.join();
17
18     return exitCode;
19 }
```

Operation implementation

The implementation of the whole operation is then very similar to the implementation of the internal SCM repository creation operation

(see 3.3) and is shown in the Listing 3.6[1]. Business logic details are hidden under *service::clone*[2].

Listing 3.6: Clone endpoint handler

```
1 public void clone(RepositoryCloneRequest
    cloneRequest) {
2     taskExecutor.executeAsync(
3             cloneRequest.getCallback(),
4             cloneRequest,
5             service::clone,
6             this::handleError,
7             callbackSender::
                sendRepositoryCloneCallback);
8
9     throw new WebApplicationException(Response.
        Status.ACCEPTED);
10 }
```

Attached PRs

The implementation of the clone operation was introduced by the following PRs:

- `https://github.com/project-ncl/pnc-api/pull/230`
  Introducing request and response DTOs
- `https://github.com/project-ncl/reqour/pull/4`
  Implement the business logic

3.4 Alignment

The alignment operation (triggered by *POST /adjust*) was already briefly mentioned in Section 1.1 where we also saw an example align-

[1] `https://github.com/project-ncl/reqour/blob/akridl-thesis/rest/src/main/java/org/jboss/pnc/reqour/rest/endpoints/CloneEndpointImpl.java#L38-L47`

[2] `https://github.com/project-ncl/reqour/blob/akridl-thesis/core/src/main/java/org/jboss/pnc/reqour/service/GitCloneService.java#L42-L67`

ment operation on some hypothetical *pom.xml* and its possible result (see Figure 1.1). Let us now describe the alignment operation in more detail.

3.4.1 Analysis

Description

The alignment operation workflow is the following: Reqour optionally synchronizes a downstream repository with an upstream repository. Then, it clones the source code from the downstream repository based on the information provided in a request. The cloned repository should contain a project file (e.g. *pom.xml* for the Maven project). This file is passed to a configured manipulator (based on the request and the runtime configuration), which uses DA to align the project file. Once the alignment is done, Reqour pushes aligned changes to the **downstream** repository, parses the manipulator result, and returns a response in a unified format[1]. The workflow is shown in Figure 3.7.

Note: By **alignment operation**, we mean both **version increment** together with **dependency alignment**. The difference between the two is that during a version increment, **next non-existing version** in the sequence is chosen, whilst during the latter, **the most feasible version from existing versions** is chosen.

Example: Suppose that we request DA and all available versions of the artifact *org.foo:bar* are as shown in Figure 3.8. Version increment for a temporary build will be **temporary-redhat-00003**. Dependency alignment for a persistent build will be **redhat-00003**[2].

Finally, the table 3.9 lists build types and their corresponding manipulators used during alignment. Every manipulator but SMEG is distributed as a standalone JAR (SMEG being distributed as a Scala plugin). Maven and Gradle builds make up the vast majority of PNC builds. This results in PME and GME providing more functionalities. For instance, PME is the project maintained by Red Hat Productization.

[1] Unification of the format is Reqour's responsibility since every manipulator returns the result in its custom format.

[2] Supposing that simply the built artifact from the last persistent build is taken. In reality, PNC dependency alignment is a little more involved but it is not important for our context.

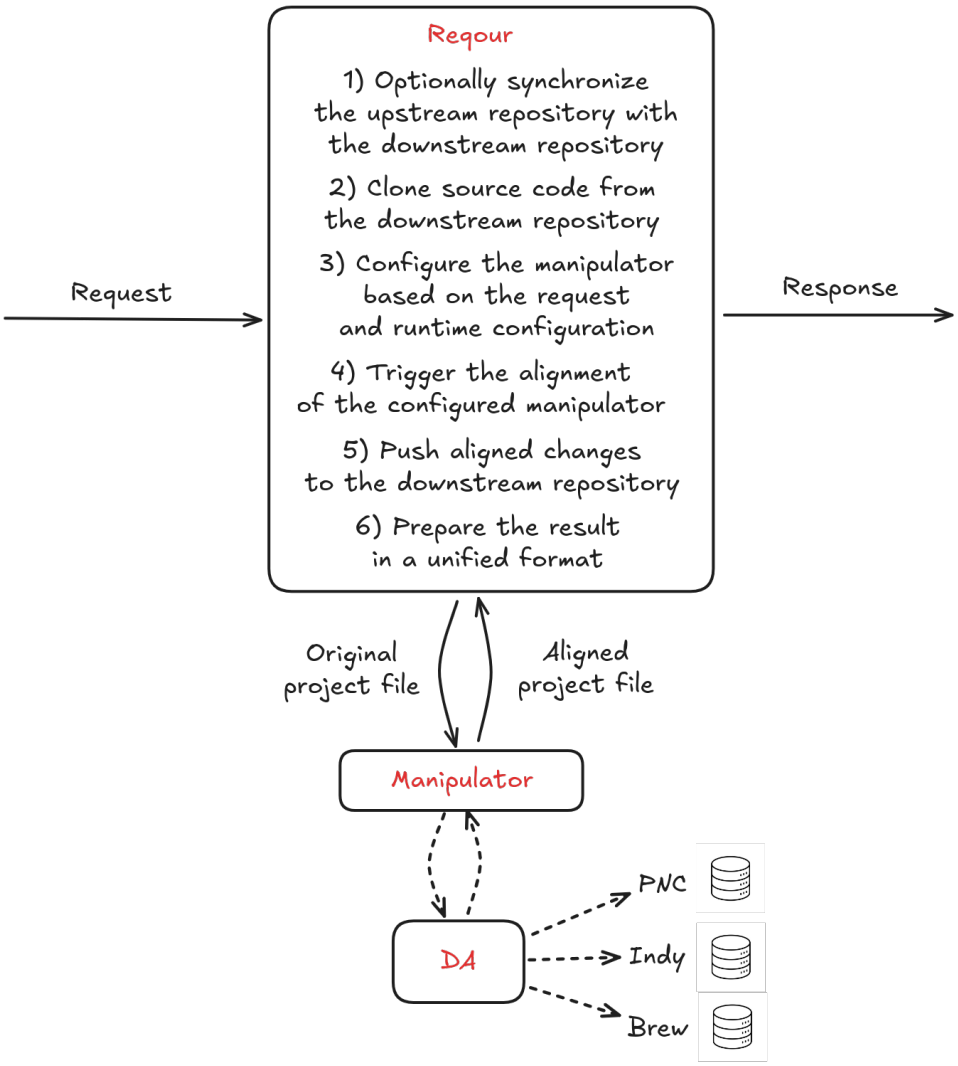

Figure 3.7: High-level overview of alignment process

Figure 3.8: DA and all available versions for the artifact *org.foo:bar*

Build Type	Manipulator
Maven	PME
Gradle	GME
NPM	Project Manipulator
SBT	SMEG

Figure 3.9: Build types and corresponding manipulators

Request DTO

Adjust request DTO is JSON of the following format:

```
{
    "originRepoUrl": "string",
    "ref": "string",
    "sync": boolean,
    "buildType": "string",
    "tempBuild": boolean,
    "internalUrl": {
        "readwriteUrl": "string",
        "readonlyUrl": "string"
    },
    "brewPullActive": boolean,
    "buildConfigParameters": {
        "string",
        "string"
    },
    "pncDefaultAlignmentParameters": "string",
```

```
    "taskId": "string",
    "callback": {
        "method": "string",
        "uri": "string"
    }
}
```

where DTO fields have the following meaning:

- **originRepoUrl**
  External repository URL from which to synchronize to the corresponding internal repository.
- **sync**
  Whether to synchronize the internal (downstream) repository from the external (upstream) repository.
- **buildType**
  Build type, e.g. Maven.
- **tempBuild**
  Whether the build is temporary. Hence, whether the alignment should work with persistent or temporary Red Hat versions.
- **internalUrl**
  Both read-only and read-write URLs of the internal repository.
- **brewPullActive**
  Whether DA should look for built artifacts in Brew (Brew is just another build system among PNC).
- **buildConfigParameters**
  Parameters specified within the build configuration when triggering the build. Alignment might take some of these for configuration.
- **pncDefaultAlignmentParameters**
  Sane defaults set by PNC, which are not possible to override.
- **ref, taskId, callback**
  Same meaning as in Cloning DTO described in 3.3.1.

Response DTO

The response DTO (sents as callback payload) is JSON of the following format[1]:

```
{
    "downstreamCommit": "string",
    "tag": "string",
    "upstreamCommit": "string",
    "internalUrl": {
        "readwriteUrl": "string",
        "readonlyUrl": "string"
    },
    "isRefRevisionInternal": boolean,
    "manipulatorResult": { ... },
    "callback": { ... }
}
```

Fields have the following meaning:

- **downstreamCommit**
  Commit with alignment changes made by Reqour.
- **tag**
  Tag of the **downstreamCommit**.
- **upstreamCommit**
  Commit from the upstream repository upon which is the alignment being made.
- **internalUrl**
  Same meaning as in the request DTO.
- **isRefRevisionInternal**
  True in case the *ref* from the request DTO is present in the internal repository.

[1]The format is not valid JSON, since *manipulatorResult* and *callback* object types are hidden. The former because of unnecessary details, and the latter is already present in the cloning response DTO shown in 3.3.1

- **manipulatorResult**
  Unified format of the result returned from the Reqour.
  Note: Manipulators themselves do not return result of the unified format. Hence, it is Reqour's responsibility to unify it.

- **callback**
  Same meaning as in the cloning response DTO (see 3.3.1).

3.4.2 Design

The alignment operation is a long-running operation, possibly taking several dozens of minutes, in some cases even exceeding an hour. Hence, there is no doubt that this operation has to be asynchronous.

In essence, everything depicted in Figure 3.7 as Reqour's responsibility, is delegated to an Adjuster which is created per alignment (hence per build) inside the corresponding PNC OpenShift cluster.

The endpoint handler itself is being pretty simple: asynchronously starts execution of some code (in this case creation of Adjuster Job in the OpenShift cluster) and immediately returns 202 Accepted.

Once the created Adjuster is fully initialized, all the work is done in there: optionally synchronize the downstream repository to the upstream repository, clone the source code from the downstream repository, start a manipulator (as *java -jar manipulator.jar*)[1] communicating with DA during the alignment process, and make a callback at the end.

The above word description is visualized in Figure 3.10.

Note: Even though Adjuster is depicted to be created in a separate OpenShift cluster, in fact, all the PNC microservices in the Figure (BPM, DA, Adjuster) are within the very same cluster (actually, even within the very same namespace).

3.4.3 Implementation

Prior to the alignment operation, it made sense to have everything in a single module (since Reqour API is within PNC API project, so clients use PNC API to interact with Reqour) containing all the necessary

[1]The only difference is SMEG which is distributed as an SBT plugin, thus being run directly through SBT.

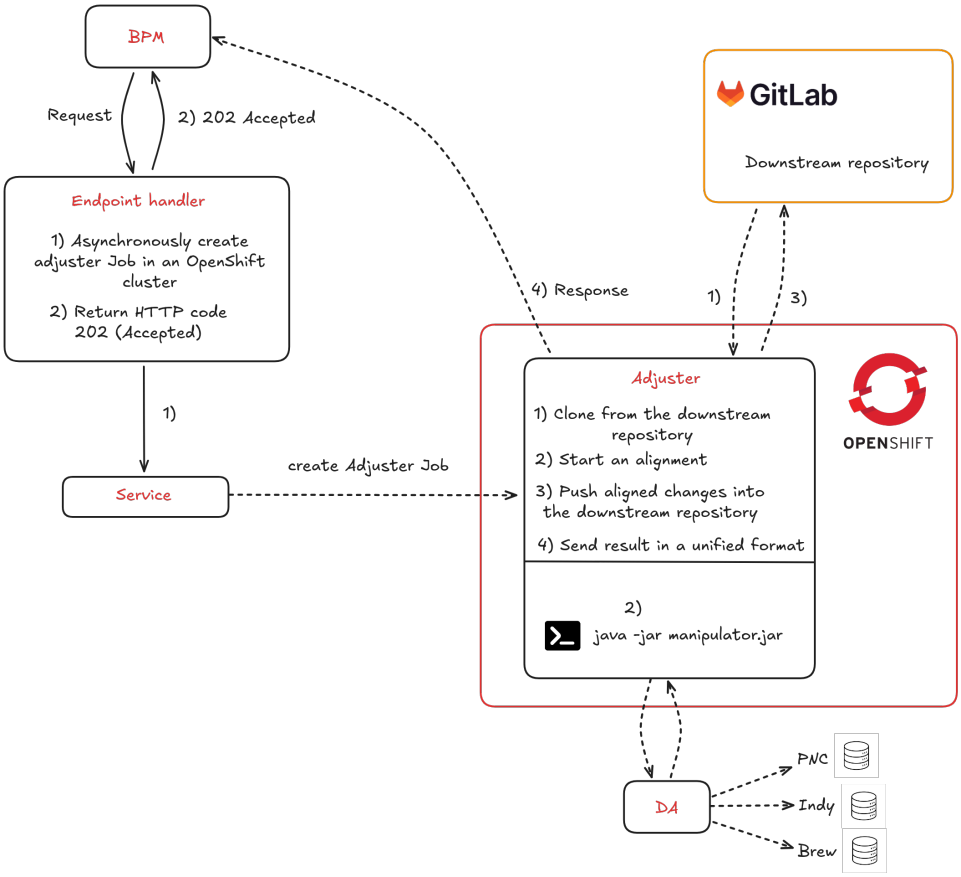

Figure 3.10: Design of the alignment operation

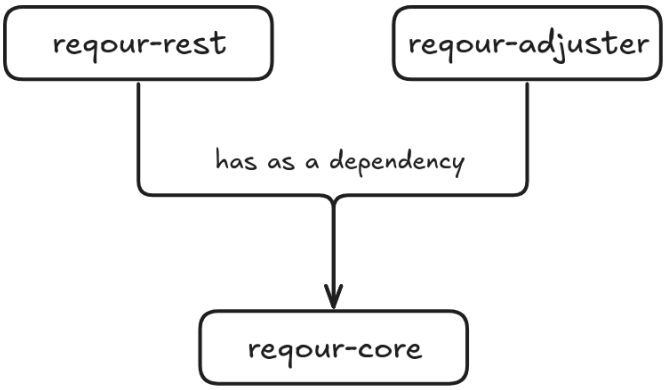

Figure 3.11: Overview of Reqour implementation

parts: endpoint handler implementations, service layer, and common abstractions, e.g. asynchronous task executor. However, business logic for adjusting together with all its configurations is quite independent, needing only very few abstractions from the rest of the project.

Overall modules overview

For a while, we even considered splitting the adjuster into a separate project. But in case of a split, we would need to split the rest of the project into a core (containing all the common abstractions) and everything else (which would use the core). Adjuster would in that case use a core as its dependency. This separation would either mean 3 separate projects (which would lead from our considerations to more drawbacks than benefits) or a project with 2 modules, and a single-module Adjuster project. In the end, separating the whole project into 3 modules (adjuster, core, and everything else) seemed like the most suitable solution, which was in the end chosen. These 3 modules, together with their inter-module dependencies, are shown in Figure 3.11. All 3 modules have separate YAML configurations[1] mapped to Java objects[2].

[1] `https://quarkus.io/guides/config-yaml`
[2] `https://quarkus.io/guides/config-mappings`

Module *reqour-core*

The module containing common abstractions used both from *reqour-rest* and *reqour-adjuster* modules. The most notable abstractions are:

- **TaskExecutor**
  Asynchronous task executor (as described in 3.2.3).
- **ProcessExecutor**
  Executor of shell processes (as described in 3.3.2).
- **GitUtils**[1]
  Utility class for creating git commands.
- **GitCommands**[2]
  Thin wrapper above *ProcessExecutor* to execute git commands (using *GitUtils* to construct the commands).

In addition, the service layer is also part of the module.

Module *reqour-rest*

This module contains REST-related components: endpoint handlers implementing their corresponding endpoint APIs from PNC API. In addition, exception mapping logic written in Jakarta EE way using *@Provider*[3] annotations are present here as well.

Module *reqour-adjuster*

The adjuster module is Quarkus CLI application[4] and contains adjuster logic.

Low-level implementation of the adjuster itself corresponds to design shown in Figure 3.7: optional synchronization of the downstream repository from the upstream repository, together with cloning from

[1] `https://github.com/project-ncl/reqour/blob/akridl-thesis/core/src/main/java/org/jboss/pnc/reqour/common/utils/GitUtils.java`

[2] `https://github.com/project-ncl/reqour/blob/akridl-thesis/core/src/main/java/org/jboss/pnc/reqour/common/GitCommands.java`

[3] `https://jakarta.ee/specifications/restful-ws/3.1/jakarta-restful-ws-spec-3.1#providers`

[4] `https://quarkus.io/guides/command-mode-reference`

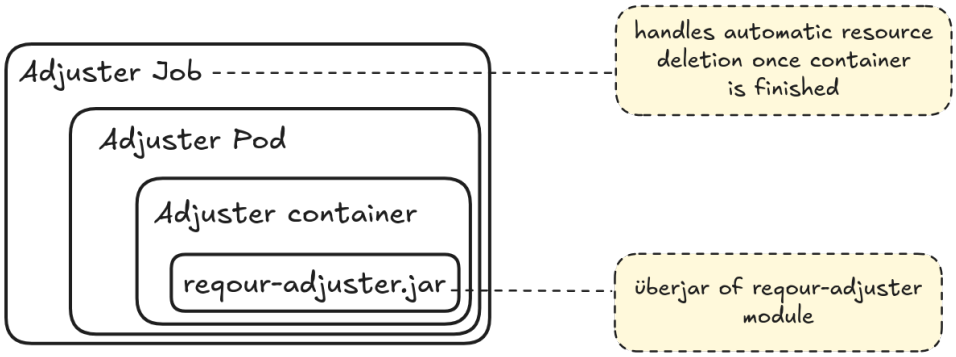

Figure 3.12: Hierarchy of Adjuster deployment

the downstream repository is done by RepositoryFetcher. Construction of the command running a manipulator is done by concrete implementations of AdjustProvider, e.g. MvnProvider (using the adjust request and runtime configuration of the adjuster). Triggering of the alignment is done simply by executing the constructed command by *ProcessExecutor*. The push of aligned changes is done by AdjustmentPusher. Finally, the unified format of the whole alignment operation is represented by AdjustmentResult and being created (either directly or indirectly) in concrete implementations of *AdjustProvider*.

Alignment operation implementation

The last piece of a puzzle is to invoke the creation of the Adjuster. This is done by the Adjust endpoint handler, which asynchronously starts the creation using *OpenShiftAdjusterPodController*[1] and immediately afterward returns 202 Accepted.

OpenShiftAdjusterPodController uses OpenShift API from Quarkus[2]. Concrete details of the deployment are injected during runtime. Adjuster deployment hierarchy is shown in Figure 3.12.

[1] `https://github.com/project-ncl/reqour/blob/akridl-thesis/rest/src/main/java/org/jboss/pnc/reqour/rest/openshift/OpenShiftAdjusterPodController.java`

[2] `https://quarkus.io/guides/kubernetes-client#openshift-client`

Attached PRs

The implementation of the alignment operation was introduced by the following PRs:

- `https://github.com/project-ncl/pnc-api/pull/240`
  Introduce DTOs and endpoint API
- `https://github.com/project-ncl/reqour/pull/10`
  Introduce *reqour-adjuster* module together with creation of Adjuster Job from *reqour-rest*

3.5 Cancel

The last Reqour endpoint containing non-trivial business functionality is *POST /cancel*.

3.5.1 Analysis

Description

One of the requirements of PNC (discussed in 1.1) is the ability to cancel running builds. Reqour's alignment operation (introduced in 3.4) is one of the build phases, which can be long-running (exceeding over an hour in some cases). Quite naturally, the alignment operation has to be cancellable in order to meet the canceling requirement.

Request DTO

The cancel request DTO is JSON of the following format:

```
{
  "taskId": "string",
  "callback": {
    "method": "string",
    "uri": "string"
  }
}
```

where both fields have the same meaning as in Cloning DTO described in 3.3.1.

Response DTO

The response DTO contains just a single field, *callback*, which is the same as in Cloning response DTO (discussed in 3.3.1).

3.5.2 Implementation

The implementation is almost identical to the implementation of the alignment operation (at the REST level): asynchronously start deletion of the Adjuster and immediately return 202 Accepted. Identically to creation, deletion is also handled by *OpenShiftAdjusterPodController*.

3.6 Version

To finalize the description of Reqour endpoints, we are left to describe *GET /version* and *GET /*. Let us start with the version endpoint.

3.6.1 Description

As was already briefly mentioned in 2.2, version endpoint serves for debugging purposes, e.g. in case a PNC developer wants to check that the deployment took place and the newest version of the application has been deployed. The version endpoint is not Reqour-specific but is present in PNC microservices thoroughly.

The endpoint responds with JSON of the following format:

```
{
  "name": "string",
  "version": "string",
  "commit": "string",
  "builtOn": "2022-03-10T12:15:50-04:00",
  "components": [
    "string"
  ]
}
```

where the fields have the following meaning:

- **name:** Name of the application.

- **version:** Version of the application.
- **commit:** SHA of the commit from which the deployment comes from.
- **builtOn:** Time of the build in *ISO8601*[1] format.
- **components:** List of application components in the same format as this DTO.

For example, the current version of the deployment at devel environment is:

```
{
  "name": "reqour-rest",
  "version": "1.0.0-SNAPSHOT",
  "commit": "32739
     fbde8fd403dec912c96ea917690189ccb5e",
  "builtOn": "2024-11-28T10:48:02Z",
  "components": []
}
```

From the above output, we can easily deduce that the current version comes from the commit `https://github.com/project-ncl/reqour/commit/32739fbde8fd403dec912c96ea917690189ccb5e`.

3.6.2 Implementation

The version endpoint handler implementation is straightforward: it just fills the DTO by values generated during build time: *name* is taken from *quarkus.application.name* config property[2]. Another values are generated by *maven-replacer-plugin*[3], which creates *BuildInformationConstants* class containing all values as static fields during build time. Consecutively, these values are taken and pasted into the response

[1] `https://datatracker.ietf.org/doc/html/rfc3339`

[2] `https://quarkus.io/guides/all-config#quarkus-core_quarkus-application-name`

[3] `https://mvnrepository.com/artifact/com.google.code.maven-replacer-plugin/maven-replacer-plugin`

DTO by the version handler. The whole implementation[1] is shown in the Listing 3.7.

Listing 3.7: Implementation of version endpoint

```
1 @ApplicationScoped
2 public class VersionEndpointImpl implements
     VersionEndpoint {
3
4     @ConfigProperty(name = "quarkus.application.
        name")
5     String name;
6
7     @Override
8     public ComponentVersion getVersion() {
9         return ComponentVersion.builder()
10                .name(name)
11                .version(
                     BuildInformationConstants.
                     VERSION)
12                .commit(
                     BuildInformationConstants.
                     COMMIT_HASH)
13                .builtOn(ZonedDateTime.parse(
                     BuildInformationConstants.
                     BUILD_TIME))
14                .build();
15    }
16 }
```

PRs

The implementation of the version endpoint was introduced by the following PRs:

[1] https://github.com/project-ncl/reqour/blob/akridl-thesis/rest/src/main/java/org/jboss/pnc/reqour/rest/endpoints/VersionEndpointImpl.java

- `https://github.com/project-ncl/pnc-api/pull/237`
  Introduce the API

- `https://github.com/project-ncl/reqour/pull/7/files`
  Implement the version endpoint handler

3.7 Root redirect

The root endpoint *GET /* does a redirection (HTTP 303 *See other*) to *GET /version*. The implementation[1] itself is shown in the Listing 3.8 and was introduced alongside *GET /version* implementation in: `https://github.com/project-ncl/reqour/pull/7`.

Listing 3.8: Implementation of the root endpoint handler

```
1 @ApplicationScoped
2 @Path("/")
3 @Slf4j
4 public class BaseUrlEndpoint {
5
6     @GET
7     @Produces(MediaType.APPLICATION_JSON)
8     public ComponentVersion redirectToVersion()
          {
9         log.debug("Redirecting␣request␣to␣base␣
              URL␣at␣version␣endpoint␣handler");
10        throw new RedirectionException(Response.
              Status.SEE_OTHER, URI.create("/version
              "));
11    }
12 }
```

[1] `https://github.com/project-ncl/reqour/blob/akridl-thesis/rest/src/main/java/org/jboss/pnc/reqour/rest/endpoints/BaseUrlEndpoint.java`

3.8 Other parts

To entirely complete a description of Reqour, we finish with two more topics: logging and testing.

3.8.1 Logging

For logging, we use SLF4J[1] for defining a logger in our classes as a static field created by Lombok's *@Slf4j*[2] annotation. SLF4J uses as its logging backend JBoss Log Manager, which is configured by Quarkus by default (and connection to this backend is done through its corresponding adapter[3]).

Since JBoss Log Manager backend supports MDC[4] (Mapped Diagnostic Context), we set this up (based on HTTP Headers of incoming requests to Reqour) and use to further identify our emitted log events.

We use JSON as the logging format[5] (configured automatically when using *quarkus-logging-json* extension).

There are 2 logging handlers: console logging handler printing all the caught log events into a console, and Kafka logging handler provided by *quarkus-logging-kafka*[6] dependency, which sends all the incoming log events into a Kafka topic. Thanks to this, Reqour contributes to logging pipeline and provides so-called **live-logs** from its part of a build.

3.8.2 Testing

Reqour contains extensive test suites in the form of unit tests and integration tests.

Unit tests are used in several different ways:

- Testing a single method without any dependencies:
  For example: IOUtilsTest

[1] `https://www.slf4j.org/`

[2] `https://projectlombok.org/api/lombok/extern/slf4j/Slf4j`

[3] `https://github.com/jboss-logging/slf4j-jboss-logmanager`

[4] `https://javadoc.io/doc/org.jboss.logmanager/jboss-logmanager/latest/org/jboss/logmanager/MDC.html`

[5] `https://quarkus.io/guides/logging#json-logging`

[6] `https://github.com/project-ncl/quarkus-logging-kafka`

- Testing a single method and mocking internal dependencies (using Mockito[1])
  For example: GitlabRepositoryCreationServiceTest
- Testing a single method and mocking the whole underlying layer (using Mockito):
  For example: TranslationEndpointTest
- Testing a single method and mocking external dependencies (using WireMock[2]):
  For example: InternalSCMRepoCreationEndpointTest

Alongside unit tests, integration tests are also present, but are not present for every endpoint, e.g. to test *POST /adjust*, we would need to integrate with an OpenShift cluster. However, whenever it makes sense, an integration test is also used, e.g. TranslationEndpointIT.

[1] `https://site.mockito.org/`
[2] `https://wiremock.org/`

4 Comparision of Repour and Reqour

In Chapter 2, we described Repour, i.e., old implementation. In Chapter 3, we looked into Reqour, i.e., new implementation. However, we never really mentioned their differences and did no comparison so far. That will be a subject of this chapter in which we will specifically focus on the comparison of maintainability, scalability, and clusterability.

4.1 Maintainability

Maintainability can be defined as: "The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment"[8]. Many factors affect the maintainability of software systems. Let us list several of them and compare Repour and Reqour against them:

- **Understanding of used programming language / framework**
  Repour is written in Python. On the one hand, Python is a high-level language with many programmers knowing the basics of it, since it is often taught as one of the first languages, e.g. at universities. On the other hand, having advanced Python knowledge, e.g. understanding internals and correct usage of the asyncio library, is not common among programmers and requires a non-trivial amount of time for being confident with it. Other PNC microservices are written either in Quarkus or plain Jakarta EE. Quite naturally, this led to a situation when many programmers focused on Quarkus and did not want to invest that much time into learning Repour. The result was that there was a single person with a deep understanding of Repour, taking care of its maintenance. By rewriting Repour into Reqour (written in Quarkus), the bus factor (regarding Repour - Reqour maintenance) increased from 1 to the number of all PNC backend developers.

- **Documentation**
  Both Repour and Reqour contain extensive documentation (with only the most trivial parts not being documented). Hence, one could say there is no benefit of Reqour against Repour at this

factor. However, more often than not, long-living systems are being maintained but their documentation gets more and more outdated compared to reality in source code. This was also the case of Repour in many cases. Reqour, since being newly implemented, does not suffer from this defect.

- **Unused code**
  As the system evolves, the amount of unused code typically increases. Repour is no exception and contains unused code. Examples worth mentioning are the old system for storing SCM repositories[1] or a few outdated mechanisms to run the alignment operation.

To conclude, all factors mentioned above are putting Reqour in a better position regarding maintainability. However, we would not like to blame Repour for being impossible to maintain in any way. Predominantly, these reasons come from Repour being a long-living system and suffering from typical problems of such a system.

4.2 Scalability

The difference

Comparing Repour and Reqour with respect to scalability, there is no difference except for one operation, and that is the alignment operation. The difference is shown in Figure 4.1.

On the left side, we can see how the alignment operation works within Repour implementation: there are 2 pods[2] (inside PNC production cluster). The manipulator process itself (*java -jar manipulator.jar*) is running directly inside one of the pods (depends on which pod was picked to handle the request by Kubernetes built-in load-balancer[3]). Since Repour is assigning asynchronous tasks to the asyncio event loop, there might be several running manipulator processes within the very same pod.

[1] This was migrated for an internal GitLab.

[2] `https://docs.openshift.com/container-platform/4.17/rest_api/workloads_apis/pod-v1.html`

[3] `https://docs.openshift.com/container-platform/4.17/rest_api/network_apis/service-v1.html`

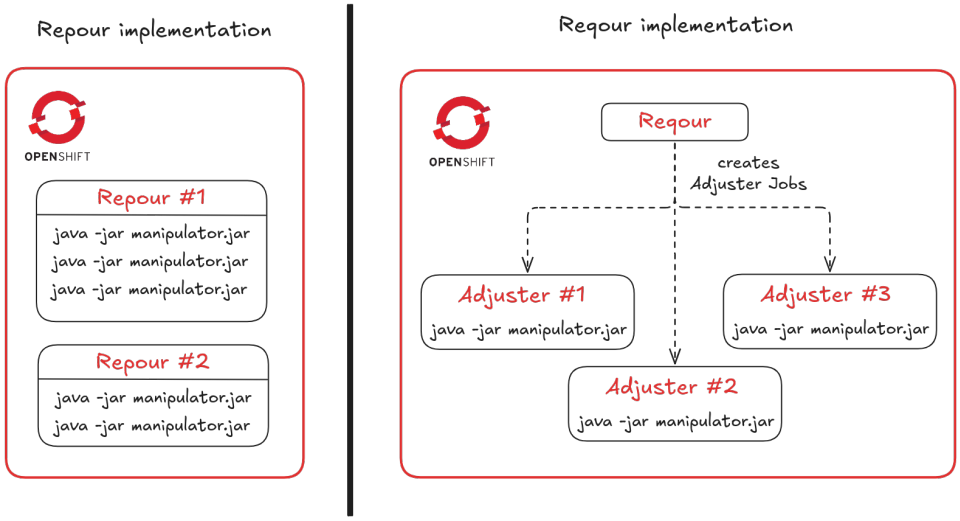

Figure 4.1: Difference between Repour and Reqour alignment operation

On the right side, we can see how alignment works within Reqour implementation: a new Adjuster Job is being started for every new alignment (for further details, see 3.4.2).

Scalability

One of the definitions of Scalability, or more specifically, Load Scalability, is: "Load scalability is the ability of a system to perform gracefully as the offered traffic increases."[9] and can be increased either horizontally (adding more computational resources) or vertically (adding more computation power per resource).

In the case of Repour, we have a number of computational resources (in our case Repour pod) fixed, and that is 2. Hence, the only option to increase the scalability is by vertical scaling. Since these pods run in the PNC production (OpenShift) cluster, vertical scaling is automatically managed by OpenShift itself[1] and is configured by

[1] `https://docs.openshift.com/container-platform/4.17/nodes/pods/nodes-pods-vertical-autoscaler.html`

the Repour deployment. Practically, Repour pod is able to scale up to 5 running alignments in the same time[1].

In the case of Reqour, the situation is different: among OpenShift built-in vertical Pod auto-scaling, there is mainly horizontal scaling (managed by Reqour's alignment endpoint handler) taking place. This allows us to scale up parallel alignments until we have enough resources to create a new Adjuster Job. This should allow us to scale up to a higher number of parallel alignments than in the case of Repour.

Conclusion

To conclude, unlike Repour scaling, Reqour scaling is also (and mainly) horizontal, potentially allowing scaling up to a higher number of parallel alignments than Repour does.

In addition, Reqour automatically supports the backpressure mechanism: in case it cannot create any new alignment, it will simply wait for some preceding one to finish. In contrast, Repour does not allow such a mechanism and a pod simply fails (on OOM error) in case clients are too greedy and want to start too many alignments in parallel.

4.3 Clusterability

First things first, we need to unify a terminology: A **cluster** is two or more computers working together to provide higher availability, reliability, and scalability than can be obtained by using a single system. When failure occurs in a cluster, resources are redirected and the workload is redistributed[2]. By **clusterability** of an application, we mean an ability of an application to exist in several replicas (within the same cluster) and provide the following guarantee: during any request, data consistency is preserved.

From the above, it should be clear that stateless applications are predetermined to support clusterability effortlessly (since they do not store any state), unlike stateful applications. As a consequence,

[1] Depending on their size.

[2] `https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc778629(v=ws.10)`

stateless applications are naturally able to support scalability, fault tolerance, and other key properties of cloud-native applications[1].

Repour is stateful: it remembers its running (asynchronous) tasks executed in its asyncio event loop. When *POST /cancel/{taskId}* request is being handled, Repour's load-balancer has to send this request to all its underlying replicas since **at most** one of these replicas is potentially executing the task corresponding to the given *taskId*. On the other hand, Reqour is stateless — hence, it should support clusterability more easily than Repour.

Conclusion

To sum up the whole chapter, we showed that the new solution introduced by Reqour should enhance maintainability, scalability, and also clusterability.

[1] `https://www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless#stateful-vs-stateless`

5 Deployment

Reqour deployment is done through CI/CD[1] pipeline. The pipeline consists of 2 main parts: Jenkins (during which *reqour* and *reqour-adjuster* **JARs** are built and published) and GitLab (during which *reqour* and *reqour-adjuster* **images** are built and published, as well as deployment being done).

All PNC deployment configs are managed as a part of the internal RedHat GitLab instance, hence, not being publicly available. Because of that, only word descriptions together with diagrams will be present in this chapter.

5.1 Jenkins

The pipeline is triggered on push to *main* branch. At first, *SNAPSHOT* job is being started. Using maven, it compiles the source code, runs all the tests (both unit and integration ones), and finally, it checks the formatting based on a defined template, e.g. *eclipse-codeStyle.xml*[2]. In case this succeeds, built artifacts are deployed to JBoss Nexus repository[3]. Some of them will be fetched and used in future stages of the pipeline.

If *SNAPSHOT* job is successful, other 3 jobs are run:

- **reqour-image job:** Triggers *reqour-image* pipeline at GitLab using its REST API[4].
- **reqour-adjuster-image job**: Same as *reqour-image* job, but triggers *reqour-adjuster-image* pipeline.
- **sonarqube job:** Runs static analysis using Sonarqube.

The above description is visualized in Figure 5.1.

---

[1] Where *D* stands for deployment, not delivery. Hence, the pipeline is fully automatized.

[2] `https://github.com/project-ncl/reqour/blob/akridl-thesis/eclipse-codeStyle.xml`

[3] `https://repository.jboss.org/nexus/content/repositories/snapshots/org/jboss/pnc/reqour/`

[4] `https://docs.gitlab.com/ee/ci/triggers/#trigger-a-pipeline`

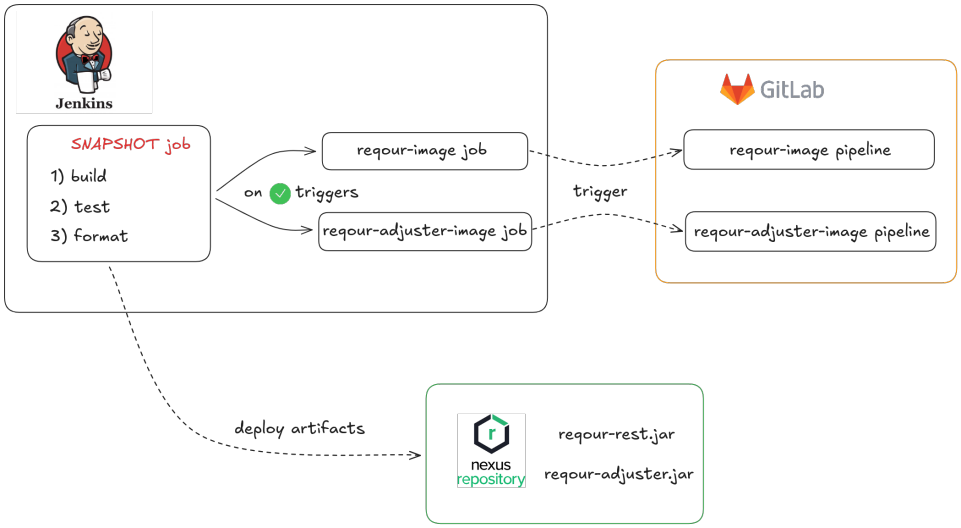

Figure 5.1: Jenkins part of the pipeline

5.2 GitLab

The GitLab part of the pipeline is built of 2 phases:

1. **Building and pushing OCI images**
   Both *reqour* and *reqour-adjuster* images are built (using (among others) the corresponding JAR from Jenkins part). Once built, they are pushed to `https://quay.io/` image repository.

2. **Deploying reqour**
   All PNC microservices run in an OpenShift cluster. Helm is used for templating OpenShift resource files, which allows us to define a resource to all environments in a single file and replace the values for a corresponding environment when creating the chart.

   Created charts are stored in the Helm PNC repository. Once the chart is successfully pushed to the repository, it is installed into the OpenShift cluster. OpenShift updates those resources that were affected by the current pipeline run, hence finishing the deployment.

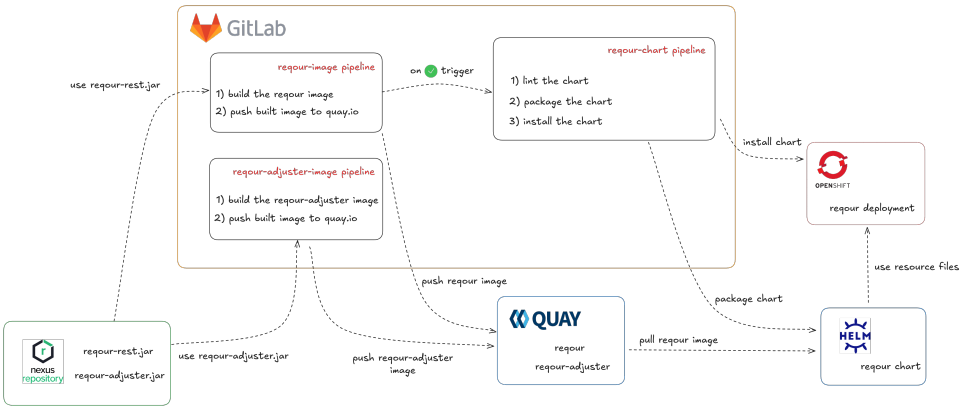

Figure 5.2: GitLab part of the pipeline

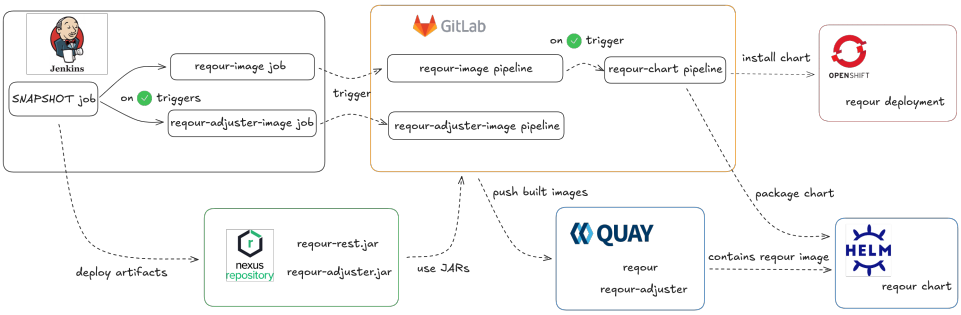

Figure 5.3: Pipeline

The above description is visualized in Figure 5.2.

Both Jenkins and GitLab parts are visualized (in less detail) in Figure 5.3.

Conclusion

The goal of the thesis was to design and implement a new solution of Repour microservice[5] within the PNC build system in Quarkus. In addition, the new solution should enhance clusterability, maintainability, and scalability.

The new solution, Reqour, was designed with desirable enhancements in mind. Based on the design, an implementation (using the Quarkus framework) took place and was continuously being published as a series of pull requests and commits in GitHub. Most importantly, a new repository under the PNC build system organization was created to contain the solution[6]. Apart from this repository, Reqour REST API was contributed to the PNC API repository[1] so that clients can migrate to Reqour later.

Deployment of the Reqour microservice was made and Reqour has been part of the PNC development cluster since August 2024. Most importantly, once PNC developers decide to make deployment also to staging and production environments, these deployments will in essence mean just a few configuration changes.

Regarding future work, an integration of Reqour within the build process inside a new Dingrogu microservice[2] will have to happen (so-called RHPAM initiative). Possibly, in case PNC developers would want to use Reqour before the RHPAM initiative takes place, the BPM codebase would need to be changed to use Reqour instead of Repour[3].

[1] `https://github.com/project-ncl/pnc-api`
[2] `https://github.com/project-ncl/dingrogu`
[3] However, this is more unlikely than likely.

Bibliography

1. ADAMS, Bram; BELLOMO, Stephany; BIRD, Christian; MARSHALL-KEIM, Tamara; KHOMH, Foutse; MOIR, Kim. The Practice and Future of Release Engineering: A Roundtable with Three Release Engineers. *IEEE Software*. 2015, vol. 32, no. 2, pp. 42–49. Available from DOI: 10.1109/MS.2015.52.
2. *Jakarta EE* [online]. Online: Eclipse Foundation, 2024 [visited on 2024-11-10]. Available from: https://jakarta.ee/about/jakarta-ee/.
3. ŠTEFANKO, Martin; MARTIŠKA, Jan. *Quarkus in Action* [online]. Online: Manning Publications, 2021 [visited on 2024-11-11]. Available from: https://www.manning.com/books/quarkus-in-action.
4. KOLEOSO, Tayo. *Beginning Quarkus framework*. 1st ed. Berlin, Germany: APress, 2020.
5. *Repour* [online]. GitHub, 2024 [visited on 2024-11-10]. Available from: https://github.com/project-ncl/repour.
6. *Reqour* [online]. GitHub, 2024 [visited on 2024-11-14]. Available from: https://github.com/project-ncl/reqour.
7. *Git FAQ* [online]. Kernel.org Archive, 2024 [visited on 2024-12-01]. Available from: https://archive.kernel.org/oldwiki/git.wiki.kernel.org/index.php/Git_FAQ.html#How_do_I_clone_a_repository_with_all_remotely_tracked_branches.3F.
8. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*. 1990, pp. 1–84. Available from DOI: 10.1109/IEEESTD.1990.101064.
9. BONDI, André B. Characteristics of scalability and their impact on performance. In: *Proceedings of the Second International Workshop on Software and Performance (WOSP '00)*. ACM, 2000, p. 195. ISBN 158113195X. Available from DOI: 10.1145/350391.350432.

A How to run the application

1. Download the source code
   Use either the thesis archive or Reqour GitHub repository[1].

2. Navigate to the source code root directory
   This is the directory where you can see **adjuster/**, **core/**, and **rest/** directories.

3. Compile the application
   Run **mvn clean install -DskipTests**. Do not forget to link your Maven to Java 21, otherwise the build will not be successful.

4. Navigate to **rest/** directory

5. Run the application in Quarkus dev mode[2]
   Run **quarkus dev**. Reqour uses Quarkus version *3.16.3*[3], so in case of problems, explicitly specify this version.

6. Start requesting the running application
   You can request the application directly from any HTTP client, e.g. *curl*. Because Reqour exposes its API using OpenAPI thanks to *quarkus-smallrye-openapi* extension[4], you can open Swagger UI at the default URL: `http://localhost:8080/q/swagger-ui/`.

   WARNING: Be aware that requests working with external systems (namely: *POST /adjust* and *POST /cancel* — both working with OpenShift, and *POST /internal-scm* — working with GitLab) will **NOT** work out of the box. In both cases are needed own resources (an OpenShift cluster in first 2 cases, and a GitLab workspace in the third case) and corresponding secrets to these resources (OpenShift and GitLab API Group tokens) which are not provided publicly from security reasons.

---

[1] `https://github.com/project-ncl/reqour/tree/akridl-thesis`

[2] `https://quarkus.io/guides/getting-started#development-mode`

[3] `https://github.com/project-ncl/reqour/blob/akridl-thesis/pom.xml#L82`

[4] `https://quarkus.io/guides/openapi-swaggerui`