Author
Lukas Moritz

Submission
Institute for System Software

Thesis Supervisor
Dipl.-Ing. Dr. Markus Weninger, BSc.

External Supervisor
Dipl.-Ing. Dr. Philipp Lengauer

Month Year
May, 2022

JVM DEPLOYER: AUTOMATED SERVICE FOR PROVIDING STATE-OF-THE-ART JAVA VIRTUAL MACHINES

Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

Abstract

In order to be able to test its products in various software environments, *Dynatrace* continuously stores new releases of *Java Virtual Machines (JVMs)* in a corporate repository. For this purpose, Dynatrace has developed the *JVM-Deployer*, a company-internal tool which automates a number of tasks related to finding, processing and uploading new Java Virtual Machines. While the usage of the JVM-Deployer has made the deployment of these software artifacts significantly easier, there are still a number of inconveniences, such as the necessity to manually start and operate the JVM-Deployer in regular intervals, as well as the uncertainty about how frequently to run the tool and which online resources to check for new Java Virtual Machines when doing so. These problems can be solved by upgrading the JVM-Deployer to an autonomous web service and enabling it to receive and process web requests from the *Tech Currency Tracker*, another tool by Dynatrace which detects new releases of selected technologies (including Java Virtual Machines) and subsequently triggers corresponding actions. With this new approach, it would no longer be necessary for developers to manually download, start and operate the JVM-Deployer, and they would be able to invest their time in other tasks.

Abstract (German)

Um ihre Produkte in verschiedensten Software-Umgebungen testen zu können, speichert *Dynatrace* fortlaufend neu veröffentlichte *Java Virtual Machines (JVMs)* in einem Firmen-Repository. Zu diesen Zweck entwickelte Dynatrace den *JVM-Deployer*, ein firmen-eigenes Tool, welches eine Anzahl an Tätigkeiten automatisiert, die zuständig sind für das Auffinden, Verarbeiten und Hochladen neuer Java Virtual Machines. Zwar hat der JVM-Deployer das Deployen dieser Software-Artifakte deutlich vereinfacht, doch es gibt immer noch gewisse Unannehmlichkeiten, wie zum Beispiel die Notwendigkeit den JVM-Deployer manuell zu starten und zu bedienen, oder auch die Ungewissheit darüber, wie häufig man das Tool ausführen soll und welche Repositories man dabei auf neue Java Virtual Machines überprüfen soll. Diese Probleme könnten gelöst werden, indem man den JVM-Deployer zu einem selbstständigen Webservice aufrüstet, und es ihm ermöglicht, Web-Requests des *Tech Currency Tracker* zu bearbeiten. Der Tech Currency Tracker ist ein weiteres firmen-eigenes Tool, welches für das Auffinden neuer Veröffentlichungen von ausgewählte Technologien (inklusive Java Virtual Machines) und das darauffolgende Auslösen von Aktionen zuständig ist. Mit diesem neuen Ansatz wäre es nicht länger notwendig, dass Entwickler den JVM-Deployer manuell starten und bedienen, und sie könnten ihre Zeit in andere Aufgaben investieren.

Contents

1 Introduction **1**

2 Background **3**

2.1 JVM-Deployer . . . 4

2.1.1 Initial module for fetching and downloading . . . 4

2.1.2 JavaFX and remaining deployment steps . . . 6

2.2 Tech Currency Tracker . . . 8

2.2.1 Trackers . . . 8

2.2.2 Subscriptions . . . 8

2.2.3 Actions . . . 8

3 Web service setup **9**

3.1 External library vs web service . . . 9

3.2 Spring Boot Web in the JVM-Deployer . . . 9

4 TCT integration **11**

4.1 Azul Zulu JDK Tracker . . . 11

4.2 Web Request Actions . . . 11

4.3 Fetching specific releases . . . 12

4.3.1 Obtaining singular releases from GitHub . . . 12

4.3.2 Introducing inheritance to fetcher classes . . . 14

4.3.3 Building Vendor Packs for specified tags . . . 16

5 Adapting the JVM-Deployer structure **17**

5.1 Improve handling of JVM artifacts . . . 17

5.2 Separating JVM processing and GUI . . . 19

5.3 Triggering deployment tasks autonomously . . . 20

6 Reporting deployment problems **21**

6.1 Diagnose errors . . . 21

6.2 Provide download methods . . . 23

7 Lessons Learned **24**

8 Future Work **25**

9 Conclusion **26**

1 Introduction

Preliminary to project start, the context and overall scope was specified as follows:

Dynatrace uses a company-internal repository called Artifactory. One of the purposes of this repository is the storage of JVMs (mainly used for testing purposes) in an organized way, sorted by vendor, operating system, version et cetera. The deployment of these JVMs – looking for new JVMs, downloading them, renaming them and finally uploading them to the Artifactory – had to be done manually until in 2020 Dynatrace developed a company-internal tool called JVM-Deployer. This tool assists the developer with the deployment of JVMs by automating this process. It browses the repositories of various vendors for new JVMs, parses their metadata like version, operating system and bitness and finally uploads them to the Artifactory. Currently the JVM-Deployer is implemented as a local application with a graphical user interface. However, Dynatrace also uses another company-internal tool called Tech Currency Tracker (TCT), which scans for new releases of selected technologies – including JVMs – and can trigger actions in case of findings. The goal of this thesis is to combine the benefit of automatic release detection by TCT with the automated deployment of JVMs. The JVM-Deployer shall be upgraded from a local GUI-based application that has to be manually executed to an automated service that is triggered by TCT whenever a new JVM release was found. Since the human component will be removed from the application after this change, it must be made sure that the JVM-Deployer not only logs successfully deployed JVMs (which it already does), but also is prepared for possible errors, for example during metadata parsing. It has to handle these errors, preferable by creating a JIRA-Ticket for its specific issue.

As stated in the specification, the goal is to transform the *Java Virtual Machine (JVM)*-Deployer into an autonomous web service. The *Tech Currency Tracker (TCT)* requires some adaptations in order to be able to send web requests containing sufficient information to the JVM-Deployer. While the TCT is already able to track JVM releases from GitHub repositories, the vendor *Azul* provides its *Zulu JVM* via an independent REST API, therefore a custom tracker for this vendor will have to be implemented. Despite the TCT already being able to trigger actions in response to JVM findings, there is no possibility to trigger web requests. The *Dynatrace Tooling Team* will provide support in tackling this challenge.

Moreover, the overall architecture of the JVM-Deployer is not an outstanding foundation for the planned project. The JVM-Deployer was primarily designed to function as an JavaFX application with a *Graphical User Interface (GUI)*. As a consequence, there is no proper separation of JavaFX elements and logic for processing and uploading JVMs. The presented approach attempts to solve this by extracting JVM processing logic from GUI related code and partitioning both into separate code modules. Additionally, resulting classes will need to be extended in a way they can be run autonomously and not depend on manual operation by an user.

Furthermore, the web service shall be steadily accessible and therefore will be deployed to an external machine where it can run continuously. This, in turn, will make it more complicated to keep an eye on the service and detect parsing errors and other problems that might arise during JVM deployment. To circumvent this, the new system will put an emphasis on error handling and monitor the service closely: The JVM-Deployer will create log files about occuring deployment problems, while attempting to catch, handle and report exceptions in order to prevent crashes. The resulting log files will then be retrievable via web methods embedded in the REST API of the web service.

2 Background

The previous section gave an overview on the scope of this thesis. However, before getting into any implementation details, this section will briefly present the state of the JVM-Deployer and TCT prior to project start, and explain fundamental principles. Figure 1 will serve as a point of orientation throughout this section.

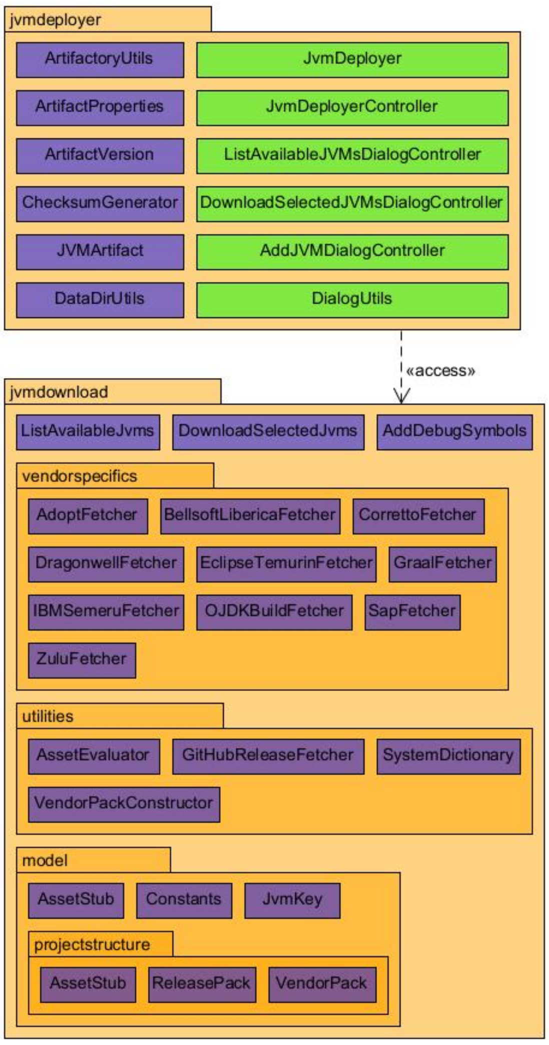


Figure 1: This is the initial structure of the JVM-Deployer. Classes purely responisble for JVM processing are displayed in blue, while the green classes constitute the graphical JavaFX application, as which the JVM-Deployer is being executed.

2.1 JVM-Deployer

The JVM-Deployer was implemented as a gradle project which is only executable as an JavaFX application. It detects, downloads and processes JVM releases before finally deploying them to the Artifactory. The JVM-Deployer achieves this through the execution of 5 consecutive tasks:

1. Fetching: Detects new JVM releases (binaries packaged as ZIP- or TAR.GZ-archives) and parses their metadata.
2. Downloading: Downloads the detected archives and stores them under a standardized file name using the obtained metadata.
3. Debug symbol adding: The JVM-Deployer will also download suitable debug-images and process them by extracting suitable `jvm.pdb` files from the images and inserting them into the associated JVM archives.
4. Preparing: Executes the script `prepareJVM.py` on the resulting archives, which removes unnecessary directories, sets the correct file permissions and ensures that all JVMs are packaged as ZIP-archives.
5. Deploying: Uploads the prepared archives to the Artifactory.

Section 2.1.1 and Section 2.1.2 will describe these tasks in more detail and showcase how they relate to the class structure in Figure 1.

2.1.1 Initial module for fetching and downloading

The `jvmdownload` module forms the foundation of the JVM-Deployer and handles fetching, downloading and debug symbol adding as described before. A more detailed presentation of the individual tasks follows:

- Fetching (cf. Figure 2):
  The fetching is the centerpiece of not only the `jvmdownload` module, but the entire JVM-Deployer. This process begins in the class `ListAvilableJvms` with the method `main(String[] args)`. The JVM-Deployer GUI calls this method, passing a number of user-selected JVM vendors as parameter.

  Taking a look at the package `vendorspecifics`, you will find that it contains a number of fetcher classes, with each class being associated with a JVM vendor.

The `ListAvilableJvms` class will select the right fetchers based on the passed JVM vendors and call their method `getVendorPack()`.

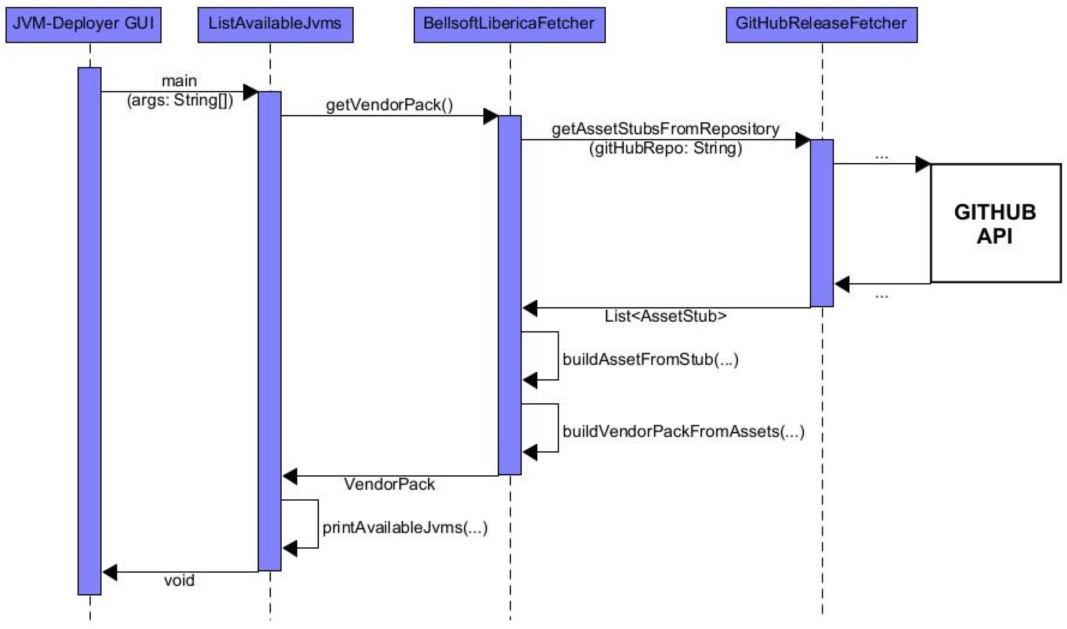


Figure 2: Depiction of the JVM fetching process in reference to the Bellsoft Liberica JVM.

The `getVendorPack()` method in turn calls the `GitHubReleaseFetcher`, which scans GitHub repositories for JVMs (in the form of archived binaries) and returns its findings via the class `AssetStub`. Each asset stub represents one JVM and entails its basic metadata (file name, download URL and publishing date).

Subsequently, the fetcher classes attempt to parse extended metadata (java version, operating system etc.) from the file names of the returned `AssetStub` objects. This is possible because JVMs are usually deployed to GitHub with highly expressive file names (e.g. 'bellsoft-jdk18.0.1+12-linux-amd64.tar.gz'), which include OS and java version. Once the fetchers have parsed said metadata, they store it together with the original `AssetStub` in the class `Asset`. The method responsible for this is `buildAssetFromStub(AssetStub assetStub)`. Finally, the created assets will be packaged into the class `VendorPack`, which sorts the assets by version and OS, and afterwards they are printed to the text file `availablejvms.txt`.

- Download:
  Subsequently, the user has the opportunity to edit the resulting text file in order to remove unwanted JVMs from the list. Once finished, the user calls the class `DownloadSelectedJvms` from the JVM-Deployer GUI. This class simply reads the file `availablejvms.txt` line by line and downloads the given JVM archives. The downloaded JVMs are then stored under a standardized file name, the Liberica JVM from the previous example would now be named 'jdk-linux-x86_64-18.0.1.12.0.tar.gz' and stored in the folder 'download\bellsoft-liberica'.

- Adding debug symbols:
  Some JVM releases not only provide JVM archives, but also the related debug images. The `DownloadSelectedJvms` class determines for which vendors debug image processing is necessary. The respective JVM archives and their associated debug images are subsequently not stored in the 'download' folder, but in the 'requiresprocessing' folder. Once the downloads are complete, the class `AddDebugSymbols` is triggered automatically, there is no user intervention required.

  The `AddDebugSymbols` class iterates over all files in the 'requiresprocessing' folder. For each suitable JVM archive, there is an attempt to find the associated debug image in the same folder. If such a debug image has been found, the `AddDebugSymbols` class will attempt to find the correct file `jvm.pdb` in the debug image for each associated file `jvm.dll` in the original JVM archive. If all PDB-files have been found, they will be copied into the JVM archive, which subsequently is moved back into the 'download' folder.

2.1.2 JavaFX and remaining deployment steps

The module `jvmdeployer` handles the two remaining steps of the JVM deployment process, which are preparing and uploading JVMs. This is shown in Figure 1. This module furthermore contains the JavaFX classes constituting the JVM-Deployer GUI application. In order to trigger all deployment steps - including the steps mentioned in the previous section - the `jvmdeployer` module accesses the `jvmdownload` module as a project dependency via its gradle configuration.

`JvmDeployer` is the class which at this stage serves as an entry point and starts the JVM-Deployer application. The `DialogUtils` class and the `Controller` classes provide the implementation of the individual JavaFX windows and are responsible for triggering the fetching, download etc. The remaining classes in the `jvmdeployer` module are best explained in the context of their respective deployment steps:

- JVM preparation:
  In order to prepare the downloaded JVM archives for deployment, the method `prepareFrom(...)` in the class `JVMArtifact` calls the aforementioned python script on the JVMs in the 'download' folder. As mentioned before, this script removes unnecessary directories from the JVM archive which inflate the file size. Since some JVMs are provided as TAR.GZ-archives, the script also repackages the JVMs as ZIP-archives and sets the correct file permissions. The prepared archives are stored in the 'prepared' folder.

  The `prepareFrom(...)` method has `CompleteableFuture<JVMArtifact>` as its return type. The `JVMArtifact` class represents one prepared JVM archive and its associated `ArtifactProperties`, which similarly to the `Asset` class contains metadata of the JVM such as java version and operating system. The reason this method returns the `JVMArtifact` wrapped in a `CompleteableFuture` is that the `JvmDeployerController` executes the JVM preparations in parallel, which is easier to facilitate when utilizing `CompleteableFuture` objects.

- Deployment to Artifactory:
  Finally, the class `ArtifactoryUtils` uploads the prepared JVM archives to the Artifactory. For this purpose, it utilizes the class `ChecksumGenerator` as well as methods in the `JVMArtifact` class, which return the correct paths in the Artifactory to deploy to.

This section now should have given a sufficient overview over the JVM-Deployer. Next, Section 2.2 will explain key principles of the TCT.

2.2 Tech Currency Tracker

The TCT is the other component of this project. It will be responsible for tracking JVM releases, triggering the JVM-Deployer web service and providing the information necessary for deployment.

2.2.1 Trackers

For the scope of this thesis, we can imagine the TCT as a web service that handles a collection of trackers. A tracker is assigned an online source to track releases from and scans for new findings multiple times a day. There is a number of predefined tracker templates which handle popular release platforms such as GitHub, Maven, NuGet etc. The user can simply apply these templates to the specific online source to track. This is extremely beneficial as the GitHub template allows us to effortlessly create new trackers for the relevant JVM repositories. However, there are release platforms which the TCT does not support yet, and for this purpose the TCT allows the implementation of custom trackers. This enables us to create a tracker for the REST API of Azul as well.

2.2.2 Subscriptions

The TCT provides the possibility to create subscriptions for existing trackers. These subscriptions are assigned to the respective user who created them, and give an opportunity to specify release adaptation information, filter releases and trigger actions on release detection.

2.2.3 Actions

An *Action* in the TCT describes a task that is triggered in response to new releases. An already implemented example for such a task is the automatic creation of a Jira ticket related to the new releases. However, the *Web Request Action* required by the JVM-Deployer is not implemented yet.

Now that this section provided some basic knowledge on the JVM-Deployer and the TCT, starting with Section 3, this thesis will begin addressing the actual implementation.

3 Web service setup

The first step of the implementation tackled the setup of the JVM-Deployer as a web service. This section will address the decision to transform the JVM-Deployer into a web service as well as details about the implementation with Spring Boot.

3.1 External library vs web service

There exists a multitude of ways to provide the functionality of one software project to another, and it can be difficult to choose the right approach. This was a challenge with the JVM-Deployer as well. Designing the new JVM-Deployer as a web service was the initial idea, and though this approach seemed promising from the start, the question arose whether it would be more beneficial to simply provide the JVM-Deployer's code as an external library.

Using the JVM-Deployer as an external library would have been more straightforward in a number of ways. After all, it is normal practice at Dynatrace to deploy company internal apps as a `.jar` file to the corporate repository, and subsequently import them into the desired projects via gradle. In contrast to a web service, an external library does not require the JVM-Deployer to continuously run on an external machine, and directly calling the JVM-Deployer's methods from the TCT of course is a much more direct solution than sending web requests.

However, there are critical circumstances which made it necessary to resort to the web service approach. The preparation of JVMs mentioned in section Section 2.1.2 is a CPU intensive task, and obviously the download and upload of JVM archives causes a significant load on the network as well. All of this would have ended up putting additional stress on the TCT. Having the JVM-Deployer run as service on a different machine and thereby splitting the load is a much preferable solution in that regard. Since it was also the Dynatrace Tooling Team's request to keep the JVM-Deployer's impact on the TCT minimal, it was ultimately decided to design the JVM-Deployer as a web service.

3.2 Spring Boot Web in the JVM-Deployer

With the decision made that the JVM-Deployer will be implemented as a web service, the next challenge was to actually turn the JVM-Deployer from a local GUI application into a service that processes web requests.

In order to create separation between the JVM processing related logic and the actual web methods, a new module `webservice` was introduced. Implementing web methods completely from scratch is not only burdensome, but also unnecessary considering the availability of libraries exactly for this purpose. For those reasons, *Spring Boot* and *Swagger UI* were imported into the gradle project. The primary reason to go with Spring Boot was that it works out of the box and enables us to create a standalone web service without the need for complicated configuration via additional XML files. Spring Boot handles configuration mostly via annotation processing and comes with an embedded Tomcat implementation, which allows us to forgo launching an external web server.

Listing 1 displays code of the class `JvmDeployerController`, the class which implements the web methods that finally will be called from the TCT. The mappings are specified via annotations. Furthermore a class `WebServiceApp`, which serves as an entry point for the web service application, lies in the same module.

```
...

@RestController
@RequestMapping("/jvm-deployer/v1")
public class JvmDeployerController {

    ...

    @PostMapping("/deploy-github-jvms")
    public void deployGitHubJvms(@RequestBody ...) { ...
        }

    @PostMapping("/deploy-azul-zulu-jvms")
    public void deployAzulZuluJvms(@RequestBody ...) {
       ... }

    ...

}
```

Listing 1: The class `JvmDeployerController` and its web methods.

4 TCT integration

Even though the JVM-Deployer from here on has appropriate web methods specified and could already be run as a web service, at this point there is no functionality to the methods in Listing 1. As previously stated, the JVM-Deployer is not designed as a webservice, and the TCT also is not yet capable of sending web requests. Therefore it is necessary to enable the TCT to send web requests, and adapt the JVM-Deployer in a way that it can fetch assets specifically for the releases it will receive via the aforementioned web methods.

4.1 Azul Zulu JDK Tracker

As mentioned in Section 2.2.1, the TCT already offers the possibility to comfortably create new trackers for GitHub repositories providing new JVMs. As most of the JVM vendors supported by the JVM-Deployer release their JVMs on GitHub, this is already satisfactory for most of our use cases.

However, the Azul Zulu JVM is not provided via GitHub, but via its own REST API. This requires us to implement a custom tracker in the TCT for this specific case. Fortunately there were already other custom trackers implemented which could be used as a template. Overall the new custom tracker consists of only 3 new classes, with the actual JVM release fetching being done in the method `fetch()` in the class `AzulZuluJdkReleaseFetcher`. This method simply accesses the Azul Zulu REST API and returns the names of all detected JVM entries in the class `FetchResult`.

4.2 Web Request Actions

With all required trackers available, what follows is the creation of a subscription (Section 2.2.2) for each available tracker. As stated before, each subscription can be be set up with actions that will be triggered as a consequence of release detection. While the TCT already is capable of triggering actions concerning Slack, Jira and E-Mail, the required action for our use case - sending web requests - has not been implemented yet.

Initially it was expected that the implementation of *Web Request Actions* would have to be carried out within the scope of this project. Fortunately however, the Dynatrace Tooling Team had already planned to create the possibility to trigger web requests as a new *Action* type. Therefore an agreement was made with the Dynatrace Tooling Team to prepone the implementation of Web Request Actions and ensure that key-value pairs for outgoing request bodies can be set in the web UI. Figure 3 exhibits the creation of new web requests within the TCT web UI.

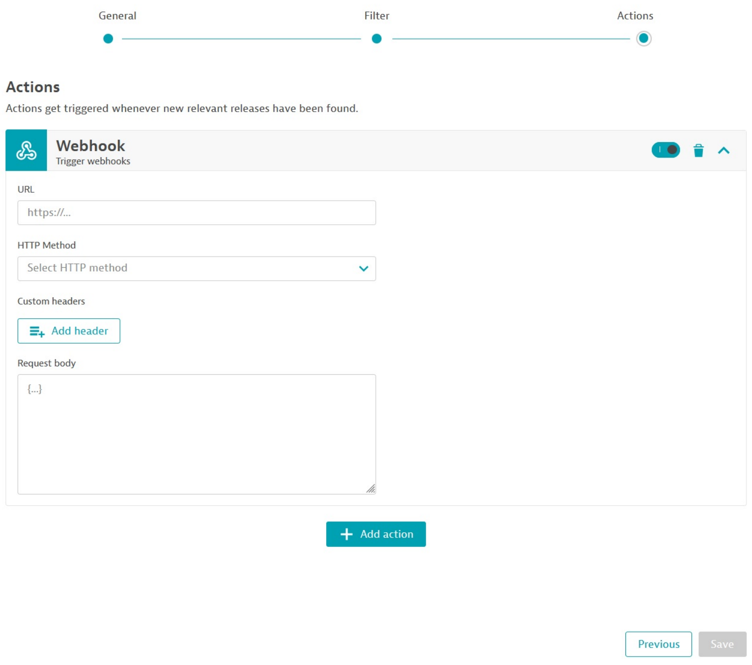


Figure 3: Creation of Web Request Action in TCT web UI

4.3 Fetching specific releases

Since the TCT is now prepared to track JVMs and send web requests to the JVM-Deployer, the subsequent step ensures that the web methods in Listing 1, which receive these requests, are also able to trigger the deployment actions accordingly.

4.3.1 Obtaining singular releases from GitHub

As mentioned in Section 2.1.1, the `GitHubReleaseFetcher` is implemented to create `AssetStub` objects for all available releases of a specified GitHub repository. However, in the future the TCT will provide the web service with a complete list of new releases to fetch, therefore scanning through the entire repository is unnecessary and inefficient.

For this reason, the method `getReleaseForTag(...)` detailed in Listing 2 was implemented in the `GitHubReleaseFetcher`. As is visible in the listing, this method utilizes the GitHub API in a different way and allows to fetch specific releases by providing the repository owner, the name of the repository and the tag of the release to fetch. Note that the method returns a new class `GitHubRelease`, which simply wraps the retrieved `AssetStub` objects with additional information about the release itself.

```
public static GitHubRelease getReleaseForTag(String repoOwner,
    String repoName, String tag) throws ... {
      try (CloseableHttpClient http = HttpClients.createDefault
          ()) {
            String pageUrl = "https://api.github.com/repos/" +
                repoOwner + "/" + repoName + "/releases/tags/" +
                tag;
            ...
            HttpGet get = new HttpGet(pageUrl);
            get.setHeader("Authorization", "token␣" + Constants.
                GITHUB_TOKEN);

            try (CloseableHttpResponse response = http.execute(
                get)) {
                  verifySuccess(response);
                  JsonNode release = new ObjectMapper().readTree(
                      EntityUtils.toString(response.getEntity()));
                  ...
                  return parseRelease(release);
            }
      }
}
```

Listing 2: The method `getReleaseForTag(...)` in class `GitHubReleaseFetcher` fetches `AssetStub` objects for specific releases.

4.3.2 Introducing inheritance to fetcher classes

While the `GitHubReleaseFetcher` from here on is able to create `AssetStub` objects specifically for the new releases provided by the TCT, they still need to undergo parsing in order to obtain java version and other metadata. If you recall Figure 2, until now the vendor-specific fetcher classes each contain only a single public method called `getVendorPack()`. This method fetches all available assets by calling the `GitHubReleaseFetcher` and requesting all available `AssetStub` instances before parsing them into the `Asset` class.

In order to parse `Asset` objects only for specific GitHub releases, the fetchers need to be extended by a method `getVendorPackForTags(...)`. This however exposed a significant flaw in the existing architecture of the fetcher classes: Despite their extreme similarities, the fetchers do not extend a mutual parent class. Since the JVM-Deployer supports 9 JVM vendors who release on GitHub, this design would have required the extension each of the 9 associated fetcher classes with almost identical methods. Additionally, the lack of a common interface requires the developer to manually map JVM vendors to the `getVendorPackForTags(...)` methods of their respective fetcher classes.

On the next page, Figure 4 displays the new and improved design of the fetcher classes . Each fetcher class, including `ZuluFetcher`, now extends the abstract class `Fetcher`. This class stores the associated JVM vendor for each fetcher and defines the abstract method `getVendorPack()`. The `ZuluFetcher` implements this method directly and also introduces the method `getVendorPackForSelectedJvms(...)` for fetching specific JVM releases via the Azul REST API.

The fetchers associated with vendors who release on GitHub (for example `BellsoftLibericaFetcher`, `IBMSemeruFetcher` and `SapFetcher`) do not extend the class `Fetcher` directly, as they are children of another abstract class `GitHubFetcher` that lies in between. The `GitHubFetcher` stores information related to the GitHub releases of the respective vendor (repository owner and a list of relevant JVM repositories), and also implements the inherited method `getVendorPack()`. Furthermore, the `GitHubFetcher` introduces a new method `getVendorPackForTags(...)`, which will actually be responsible for fetching and parsing specific JVM releases into `Asset` objects.

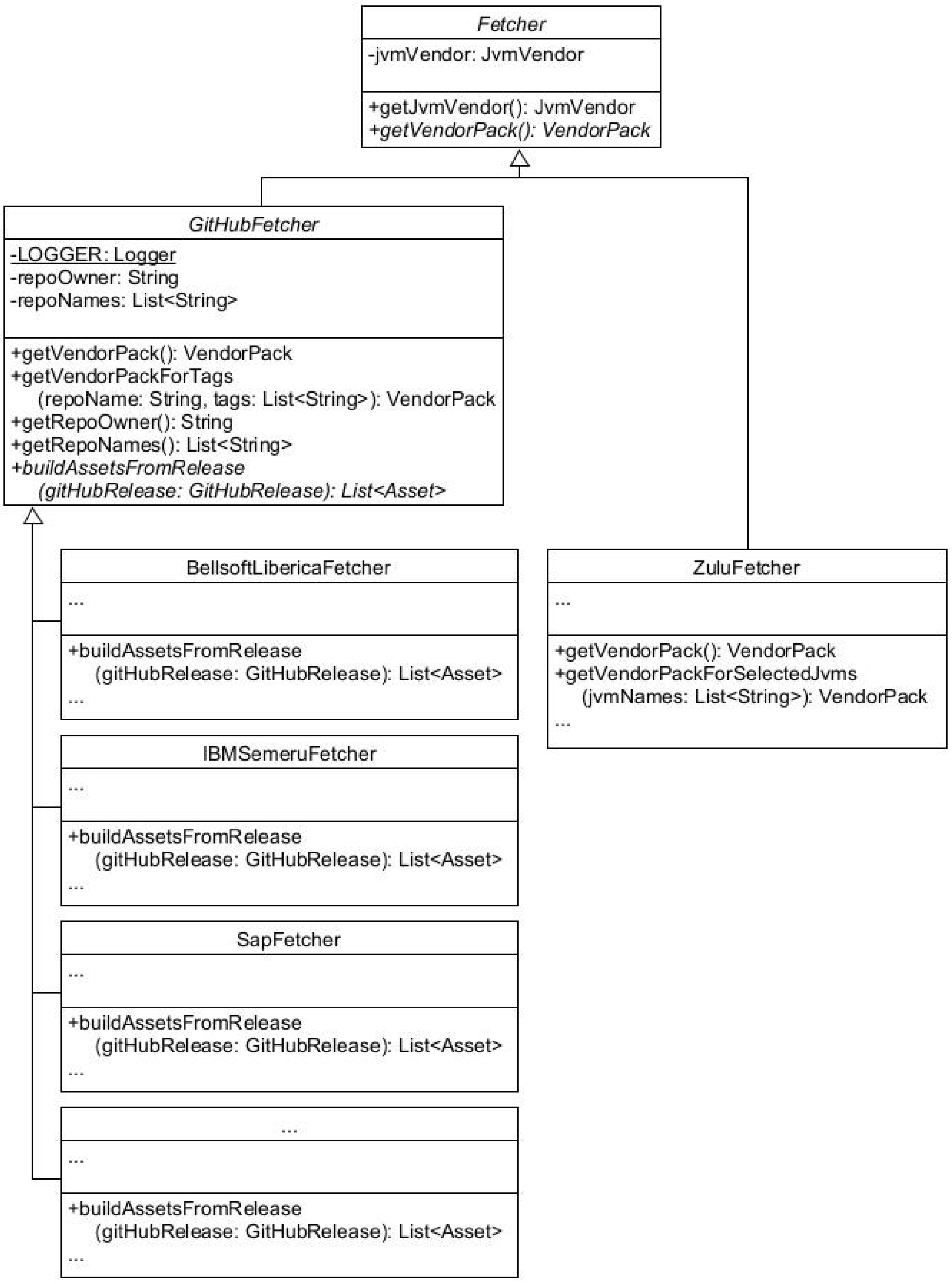

Fetcher
-jvmVendor: JvmVendor
+getJvmVendor(): JvmVendor
+getVendorPack(): VendorPack
GitHubFetcher
-LOGGER: Logger
-repoOwner: String
-repoNames: List<String>
+getVendorPack(): VendorPack
+getVendorPackForTags (repoName: String, tags: List<String>): VendorPack
+getRepoOwner(): String
+getRepoNames(): List<String>
+buildAssetsFromRelease (gitHubRelease: GitHubRelease): List<Asset>
BellsoftLibericaFetcher
...
+buildAssetsFromRelease (gitHubRelease: GitHubRelease): List<Asset>
...
IBMSemeruFetcher
...
+buildAssetsFromRelease (gitHubRelease: GitHubRelease): List<Asset>
...
SapFetcher
...
+buildAssetsFromRelease (gitHubRelease: GitHubRelease): List<Asset>
...
...
...
+buildAssetsFromRelease (gitHubRelease: GitHubRelease): List<Asset>
...
ZuluFetcher
...
+getVendorPack(): VendorPack
+getVendorPackForSelectedJvms (jvmNames: List<String>): VendorPack
...

Figure 4: Improved architecture for the fetcher classes which incorporates inheritance.

4.3.3 Building Vendor Packs for specified tags

The central element of Section 4.3.2 is the method `getVendorPackForTags(...)`. Similarly to the `getVendorPack()` method, it will return a `VendorPack` containing `Asset` objects. This method however will now utilize the `getReleaseForTag(...)` method from the `GitHubReleaseFetcher` in order to fetch `AssetStub` objects - not for all JVMs in the specified directory, but only for the new releases passed by the TCT.

The obtained `AssetStub` objects are subsequently parsed via the abstract method `buildAssetFromRelease(...)`, which the vendor-specific fetcher classes will inherit and implement. Afterwards the class `VendorPackConstructor` packages the parsed `Asset` objects and returns them to the calling method. As it might be seen, the `getVendorPackForTags(...)` method operates very similarly to the already mentioned `getVendorPack()` method, with the main difference being that the `GitHubReleaseFetcher` provides only a selected set of `AssetStub` objects.

At this point, the JVM-Deployer provides all methods necessary to fetch singular GitHub releases. However, the remaining deployment and the overall JVM-Deployer architecture steps still need to be adapted, Section 5 will address this.

5 Adapting the JVM-Deployer structure

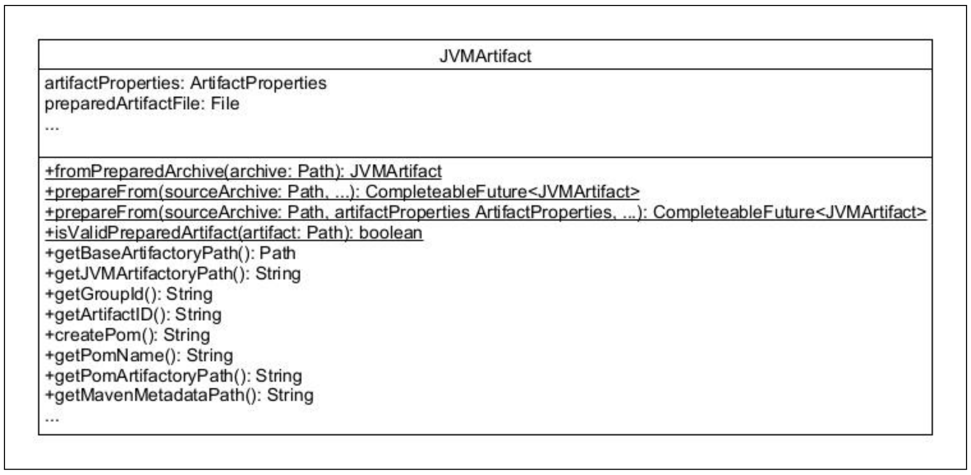
With the changes of Section 4 implemented, the JVM-Deployer is at a point where it can receive web requests from the TCT, fetch the respective releases and parse the required `Asset` objects. Nonetheless, these assets must still undergo download, debug symbol adding, preparation and deployment to Artifactory. The code concerning preparation and deployment of JVMs unfortunately is partly integrated into other classes, and there still is a mixup of JVM processing logic and GUI related code. For this reason, the following sections will address the separation of those code fragments, and the restructuring of the JVM-Deployer into the modules `jvm-deployer-logic` and `jvm-deployer-ui`.

5.1 Improve handling of JVM artifacts

The `JVMArtifact` class represents the result of a successful JVM preparation: A processed JVM archive and the associated properties (java version, operating system etc.). This is displayed in Figure 5. Unfortunately however, the class is crowded with methods that would better be placed in separate classes.

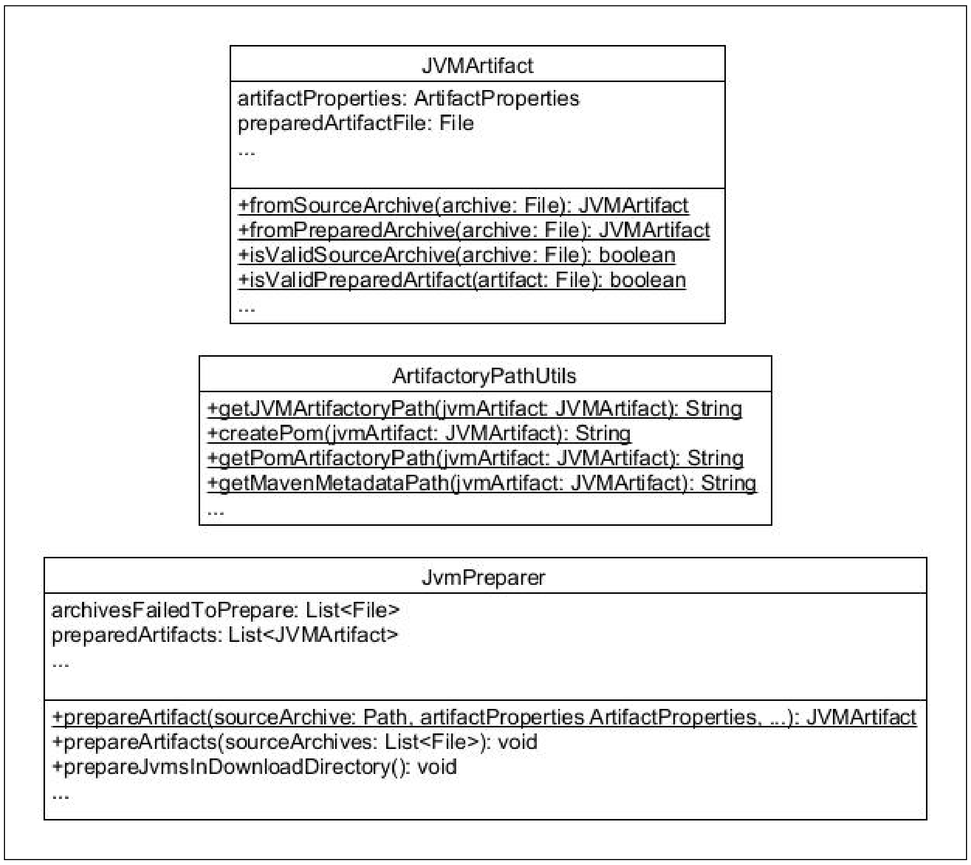
The static method `prepareFrom(...)` executes the actual preparation of JVM archives by calling the python script `prepareJVM.py`. While the return type of these methods is in fact `JVMArtifact` (wrapped in a `CompleteableFuture`), the preparation of JVMs is an important task that should not solely be provided by secondary methods of other classes. Therefore a distinct class `JvmPreparer` was created, which now features the following public methods (also displayed in Figure 6):

- `static Artifact prepareArtifact(...)`:
  Prepares a single artifact from a source archive.
- `void prepareArtifacts(...)`:
  Prepares artifacts from a list of JVM source-archive files. This is an instance method as it stores the results of the JVM preparation into the fields `archivesFailedToPrepare` and `preparedArtifacts`.
- `void prepareJvmsInDownloadDirectory()`:
  Just as in the previous method, prepares a list of JVM source-archive files and stores the results into the aforementioned fields. This method however is not directly passed a list of archives, but scans the 'download' folder of the JVM-Deployer instead.



```
JVMArtifact
artifactProperties: ArtifactProperties
preparedArtifactFile: File
...
+fromPreparedArchive(archive: Path): JVMArtifact
+prepareFrom(sourceArchive: Path, ...): CompleteableFuture<JVMArtifact>
+prepareFrom(sourceArchive: Path, artifactProperties ArtifactProperties, ...): CompleteableFuture<JVMArtifact>
+isValidPreparedArtifact(artifact: Path): boolean
+getBaseArtifactoryPath(): Path
+getJVMArtifactoryPath(): String
+getGroupId(): String
+getArtifactID(): String
+createPom(): String
+getPomName(): String
+getPomArtifactoryPath(): String
+getMavenMetadataPath(): String
...
```

Figure 5: The original `JVMArtifact` attempted to handle a multitude of tasks in a single class.



```
JVMArtifact
artifactProperties: ArtifactProperties
preparedArtifactFile: File
...
+fromSourceArchive(archive: File): JVMArtifact
+fromPreparedArchive(archive: File): JVMArtifact
+isValidSourceArchive(archive: File): boolean
+isValidPreparedArtifact(artifact: File): boolean
...

ArtifactoryPathUtils
+getJVMArtifactoryPath(jvmArtifact: JVMArtifact): String
+createPom(jvmArtifact: JVMArtifact): String
+getPomArtifactoryPath(jvmArtifact: JVMArtifact): String
+getMavenMetadataPath(jvmArtifact: JVMArtifact): String
...

JvmPreparer
archivesFailedToPrepare: List<File>
preparedArtifacts: List<JVMArtifact>
...
+prepareArtifact(sourceArchive: Path, artifactProperties ArtifactProperties, ...): JVMArtifact
+prepareArtifacts(sourceArchives: List<File>): void
+prepareJvmsInDownloadDirectory(): void
...
```

Figure 6: Splitting up `JVMArtifact` into 3 separate classes provides a much better understanding of the individual methods' contexts.

Furthermore, the methods related to parsing Artifactory paths and POM-files (`getJVMArtifactoryPath(...)`, `createPom(...)` etc.) were moved into a separate class `ArtifactoryPathUtils`. While these methods are entirely based on parsing from `JVMArtifact` objects, providing them in a distinct class that supports Artifactory operations provides a much improved class structure.

5.2 Separating JVM processing and GUI

While the decomposition of the `JVMArtifact` class provides much more clarity regarding the preparation of JVMs, the code module design of the JVM-Deployer is still questionable. If you recall Figure 1 from the beginning of this thesis, you might remember that the inital code modules do not split up the classes into JVM processing and GUI, but rather into one module for JVM fetching and download, and one module for 'the rest'.

The subpar code module design not only might cause confusion for developers unfamiliar with the JVM-Deployer, it also has negative effects considering the web service implementation in Section 3. The class `JvmDeployerController`, which will be responsible for triggering JVM deployments, is located in a distinct module `webservice`. In order to trigger all necessary deployment tasks, the `webservice` module needs to access both the `jvmdownload` module and the `jvmdeployer` module as project dependencies in gradle. This inherently will cause the web service to load JavaFX related classes and libraries from the `jvmdeployer` module, despite them never being used during web service execution.

To circumvent the aforementioned problems, the JVM-Deployer was restructured into 3 different code modules, as can be seen in Figure 7. This creates independence between the `webservice` and the `jvm-deployer-ui` module, making the resulting web service a bit more lightweight. Figure 7 also highlights the 2 resulting execution modes for the JVM-Deployer:

- Web service: Execution by starting the class `WebServiceApp` in module `webservice`. The `JvmDeployerController` class processes the TCT's web requests and in consequence calls the required methods in the `jvm-deployer-logic` module.
- JavaFX application: Started by `JvmDeployer` class in `jvm-deployer-ui` module. The associated `Controller` classes make calls to the JVM processing logic in module `jvm-deployer-logic`.

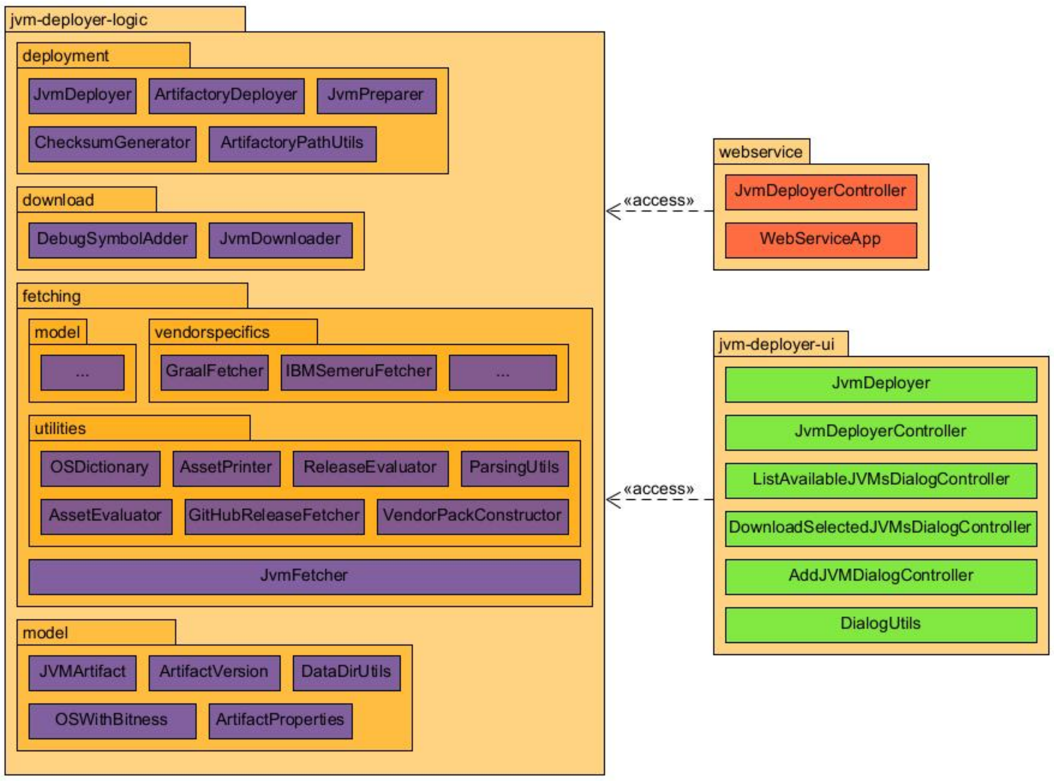


Figure 7: The new and improved structure of the JVM-Deployer (compare to Figure 1). Classes purely responisble for JVM processing are displayed in blue, green classes constitute the graphical JavaFX application, and red classes make up the web service. Logic and UI are separate now and the webservice module does not depend on JavaFX classes.

5.3 Triggering deployment tasks autonomously

The `jvm-deployer-logic` module now is all set up to execute deployment tasks triggered by the web methods in the class `JvmDeployerController`. In order to now coordinate the deployment tasks properly, a new class `AutonomousDeployments` was implemented. This class calls the required deployment jobs (`JvmFetcher`, `JvmDownloader` etc.) successively, passing the result of one task as the input for the subsequent one. All that is left to do for the web methods is to call the methods `deployGitHubJvms(...)` or `deployAzulZuluJvms(...)` in the `AutonomousDeployments` class.

6 Reporting deployment problems

Even though the result of the previous section's achievement is a complete and functional web service that can perform JVM deployments, one issue that was previously mentioned still persists. As the web service will run continuously on an external machine, it will be rather complicated to check produced logs or monitor the service in a reasonable way. Even worse, the service might crash and it may go unnoticed, which would result in the TCT sending web requests to an unreachable URL. The JVM-Deployer would never execute the requested deployments, and as a result developers would have to start the JVM-Deployer GUI and manually deploy the JVMs that were released in the respective time period.

In order to combat the aforementioned issues, it was decided to enhance the handling of arising deployment problems by catching exceptions and parsing errors, before centrally reporting them to the class `AutonomousDeployments`. Section 6.1 will address this decision. Subsequently, the changes in Section 6.2 make these deployment problems accessible from external machines by providing suitable web methods.

6.1 Diagnose errors

When run as a web service, one can view the JVM-Deployer as a number of web methods which trigger the `AutonomousDeployments` class. In turn, this class triggers the distinct deployment tasks by calling the `JvmFetcher`, the `JvmDownloader` and so forth. In order to monitor deployment problems, each of these classes needs to be extended by a `List` of the type `DeploymentProblem`.

`DeploymentProblem` is a new class which represents a problem encountered during the execution of the JVM-Deployer. This class encompasses the following fields:

- `deploymentProblemType`:
  Represents the encountered problem by the means of the `DeploymentProblem-Type` enum. Some examples for this enum are `URL_FETCHING_FAILED`, `PREPA-RATION_FAILED` and `VERSION_NOT_PARSABLE`.

- `description`:
  A `String` providing additional information on the encountered problem. This for example might be the URL for which fetching was unsuccessful or the file name of the JVM archive that could not be prepared.

The diagnosis of the respective deployment problems differs greatly between the individual JVM-Deployer classes. Nonetheless, in order to give an example, Listing 3 displays the adapted `buildAssetFromStub(...)` method of the `DragonwellFetcher` class.

```
private Asset buildAssetFromStub(AssetStub assetStub) {
    try {
        Matcher assetNameMatcher = getAssetNameMatcher(...);
        OSWithBitness os = OSDictionary.translateOS(
          assetNameMatcher.group("system"));
        ArtifactVersion version = parseVersion(
          assetNameMatcher.group("version"));
        ...
    } catch (AssetParsingException e) {
        addDeploymentProblemFromException(e, ...);
        ...
    }
}
```

Listing 3: The method `buildAssetFromStub(...)` calls a number of methods which might throw an `AssetParsingException`. This exception is reported by adding the respective deployment problem to the class' list of deployment problems via the method `addDeploymentProblemFromException(...)`

The implementation in Listing 3 allows the `DragonwellFetcher` class to perform the `AssetStub` parsing without risking an uncaught exception that might crash the JVM-Deployer. In addition, the method `getDeploymentProblems()` can easily retrieve deployment problems that occured from the fetchers. This way, the `AutonomousDeployments` class is able to accumulate `DeploymentProblem` instances from all triggered deployment tasks. After each JVM deployment run, the collected `DeploymentProblem` instances are printed to a file 'deployment-problems_{datetime}.txt', which subsequently will be stored in a distinct folder called 'deployment-problems'.

However, since the resulting text files again are located on the external machine executing the JVM-Deployer web service, it is necessary to provide a possibility for developers to fetch these files in an uncomplicated manner. Section 6.2 will address this issue.

6.2 Provide download methods

```
@RequestMapping("/download/deployment-problems")
public void downloadDeploymentProblemsArchive(@RequestParam(
    name = "maxAge") int maxAgeInWeeks, ...) {
     File deploymentProblemsArchive = archiveFiles(...,
         maxAgeInWeeks);

     provideFileDownload(...);
     deploymentProblemsArchive.delete();
}

private static void provideFileDownload(...) { ... }

private File archiveFiles(..., int maxAgeInWeeks) { ... }
```

Listing 4: The method `downloadDeploymentProblemsArchive(..., int maxAgeInWeeks)` offers the download of available deployment-problem text files. The available files are filtered based on the passed parameter `"maxAge"` and packaged into a ZIP-archive, which subsequently will be provided to the developer. Afterwards the created ZIP-archive is deleted from the file system.

The chosen approach for providing the deployment problem logs to external machines is to again utilize the `JvmDeployerController` class and implement sufficient web methods. Listing 4 displays the methods that were added to said class for this purpose. With these methods implemented, the JVM-Deployer is ready for deployment as a web service. It is able to handle requests from the TCT and subsequently fetch, process and deploy the JVMs from the specified releases. Additionally the service offers web methods which provide text files containing reported deployment problems. Therewith, the JVM-Deployer fulfills all requirements for the scope of this project.

The subsequent sections offer insight on the lessons learned during the planning and implementation this project, the future possibilities and potential improvements for the JVM-Deployer, and finally there will be a conclusion on this thesis.

7 Lessons Learned

During the implementation of this project, we encountered a number of problems.

One of the challenges was the rework of the JVM-Deployer structure. It was initially planned to perform this task at the end of this project, only after the implementation of deployment problem reporting. The realization of the project also was conducted according to this plan initially, and there were little to no problems with the adaptation of fetcher classes to process individual JVM releases.

When it came to reporting the deployment problems however, the project came to a halt. The inconsistent design of the JVM-Deployer unexpectedly required us to constantly find work arounds and adaptations. There were no standalone classes for preparation and deployment of JVMs, and a significant amount of the code necessary to make these tasks work was embedded into the JavaFX `Controller` classes.

After a number of failures we decided to postpone deployment problem reporting and prioritize restructuring the JVM-Deployer. Attempting to diagnose issues accordingly while fiddling around with disjointed code segments in the JavaFX classes was pointless. This situation could have been avoided by being more precise during the design phase of this project and recognising straightaway that the incoherent implementation of deployment tasks would lead to problems.

Another fault was the rushed decision to implement the JVM-Deployer as a webservice. While it was ultimately agreed that this would be the most suitable approach, there were some doubts that only arose during implementation. There was a certain component of wanting to find the most impressive solution for the task at hand, and this lead to the web service proposal without even considering first to simply design the service as an external library.

Overall the lessons learned during this project were to be more careful in the design phase, take time to evaluate alternative solutions, and attempt to anticipate possible obstacles before starting implementation.

8 Future Work

JVM deployments are a recurring task that - despite the great support from the inital JVM-Deployer GUI application - is rather inconvenient, so the realization of this project is a huge benefit already.

However, the general approach of the JVM-Deployer to process web requests by the TCT certainly carries potential for further improvements. As already established, the JVMs deployed to the Artifactory are mainly used for testing purposes. Unfortunately, the JVM-Deployer web service will deploy these JVMs somewhat blindly, meaning there is no verification or initial JVM testing. This could lead to avoidable problems in Dynatrace's test overview.

Furthermore, the established gradle project could comfortably host further modules for other tasks, possibly even ones that are not related to JVMs. The `webservice` module can be extended with other `RestController` classes which process additional web requests by the TCT. These classes in turn could make calls to new modules that handle various tasks related to new releases of certain technologies. The JVM-Deployer web service could be the starting point for significant improvements in the way Dynatrace handles adaptive maintenance.

9 Conclusion

To round off this thesis, this final section will give an overview on the features implemented and the general approach that was taken.

- In the beginning, it was ensured that the JVM-Deployer can even run as a web service and handle respective web requests. For this purpose, Spring Boot was imported into the gradle project as an external dependency. Spring Boot enabled us to create a standalone web service with simple configuration via annotations. All that was left to do was to implement a `RestController` class with according web methods, and the class `WebServiceApp` as a starting point.

- The next chapter addressed JVM fetching. Up to that point, the `GitHubRelease-Fetcher` was solely able to scan entire repositories for JVM releases, the possibility to only parse specific releases passed by the TCT was not given yet. Therefore, this class was extended by a method `getReleaseForTag(...)` that would allow to fetch the `AssetStub` objects for specific releases. Now the fetcher classes needed to utilize this method in order to create the selected `Asset` objects. To facilitate this, an improved class structure that utilizes inheritance was applied to the fetchers. The result were improved fetcher classes, with each one providing the method `getVendorPackForTags(...)`.

- Before proceeding, it was necessary to adapt the overall design of JVM-Deployer code modules. First, the `JVMArtifact` class was split up in order clarify the context of its individual methods. Next was the separation of JVM processing related code and JavaFX elements, and thereby dedicated classes for preparation and deployment of JVMs could be created. Subsequently, all classes were partitioned into a `jvm-deployer-logic` module and a `jvm-deployer-ui` module. This detached the web service from JavaFX elements. To facilitate the coordination of deployment tasks, the class `AutonomousDeployments` was implemented.

- The last step was to enable the JVM-Deployer to report deployment problems to the `AutonomousDeployments` class, and subsequently provide them to developers. This was achieved by improving the exception handling in respective classes and creating `DeploymentProblem` objects for corresponding events. Afterwards, methods were implemented to print these problems to text files, before providing them for download via additional web methods.

ACKNOWLEDGEMENTS

Many thanks to Dr. Philipp Lengauer for supervising me during the development of this project. Also thanks a lot to Peter Lang and Stefan Hauser for assisting me in this endeavor with their advice and their code reviews.

Without the TCT and the Web Request Actions, the project would have never even come about, so many thanks to the Dynatrace Tooling Team as well, especially to Christian Gusenbauer and Dietmar Rammerstorfer, who really took time and assisted me greatly in working with the TCT.

Lastly I want to say thank you Dr. Markus Weninger for assisting me with the written part of this thesis and of course to Dynatrace for providing me with this opportunity and allowing me to implement this project during office hours.