Submitted by
Dominik Mascherbauer

Submitted at
Institute for System Software

Supervisor
Prof. Dr. Dr. h.c. Hanspeter Mössenböck

Co-Supervisor
Dr. Christian Wirth
Dipl.-Ing. Paul Wögerer

Oktober 2023

Using Virtualization for Building Images from Native Image Bundles for Deterministic Reproducibility

Bachelor Thesis
to obtain the academic degree of
Bachelor of Science
in the Bachelor's Program
Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

GraalVM Native Image is a tool for the ahead-of-time compilation of Java applications, tailored for specific target systems. This approach operates under a closed-world assumption, yielding a native executable. The native executable also contains an Initial Image Heap consisting of objects created during the image build that are reachable from application code, `java.lang.Class` objects of classes used in the native image and object constants embedded in method code. The Initial Image Heap facilitates a significantly faster startup time compared to running an application on JVM by copying the contents of the Initial Image Heap from the binary. To ensure the deterministic reproduction of this binary, we employ the concept of *Native Image Bundles*. Native Image Bundles are archives encapsulating all requisite components, including Java class files and environment variables.

However, the build time computations happening as a part of the Native Image Build allow access to the host system and network of the host system we are building on. Although uncommon, the potential for unwanted external influences necessitates more stringent controls. To address this concern, we introduce a virtualization-based approach. This entails the creation of a secure container environment wherein explicitly specified resources and the running Native Image Builder are encapsulated. The native image is then constructed within this controlled virtualized environment, safeguarded against interference from external resources. If the build requires additional, not explicitly defined inputs, it will fail.

Native Image Bundles are architecturally similar to Java JAR files, encompassing all class files and application resources. An extension to this bundle paradigm introduces the convenience of executing the enclosed Java application on JVM instead of building a native image first. We introduce a bundle launcher specified as the primary entry point within the bundle. When executed, this launcher dynamically configures an executable environment from the bundle's contents, facilitating straightforward execution of the bundled application. Furthermore, we offer the flexibility of executing the application within a container or attaching the Native Image agent. The Native Image agent is a Java agent for capturing usages of dynamic aspects of Java application executions that need to be specified for building a native executable.

In summary, GraalVM Native Image Bundles introduces a systematic approach to reproduce a native executable under the same conditions and with the same inputs active during bundle creation. This thesis addresses the imperative requirements of determinism, resource isolation, and direct execution, underpinned by the bundle and virtualization features.

Kurzfassung

GraalVM Native Image ist ein Werkzeug für die Ahead-of-Time-Kompilierung von Java-Anwendungen für spezifische Ziel-Systeme. Dieser Ansatz beruht auf der Annahme eines geschlossenen Systems und erzeugt eine nativ ausführbare Datei. Die native ausführbare Datei enthält auch einen so gennanten *Initial Image Heap*, der aus Objekten besteht, die während der Native Image Erzeugung erstellt wurden und die von Anwendungscode erreichbar sind, `java.lang.Class`-Objekten von Klassen, die in der nativ ausführbaren Datei verwendet werden, und in Methodencode eingebetteten Objektkonstanten. Der Initial Image Heap ermöglicht einen erheblich schnelleren Start im Vergleich zur Ausführung einer Anwendung auf der JVM durch das Kopieren des Inhalts des Initial Image Heap aus der Binärdatei. Um die deterministische Reproduzierbarkeit dieser Binärdatei sicherzustellen, verwenden wir das Konzept der *Native Image Bundles*, Archive die alle erforderlichen Komponenten einschließlich Java-Klassendateien und Umgebungsvariablen umfassen.

Während dem Build-Prozess einer nativ ausführbaren Datei behält Native Image jedoch Zugriff auf das gesamte Dateisystem und Netzwerkressourcen. Obwohl unwahrscheinlich, besteht die Möglichkeit unerwünschter externer Einflüsse, die strengere Einschränkungen erfordern. Um dem gerecht zu werden, führen wir einen virtualisierungsbasierten Ansatz ein. Dies beinhaltet die Erstellung einer sicheren virtuellen Umgebung, in der explizit angegebene Ressourcen und die ausgeführte GraalVM eingekapselt sind. Eine nativ ausführbare Datei wird dann in dieser kontrollierten virtualisierten Umgebung erstellt und vor dem Einfluss externer Ressourcen geschützt. Würden zusätzliche, nicht explizit definiert Dateien für den Build benötigt werden, dann würde dieser fehlschlagen.

Native Image Bundles ähneln Java JAR-Dateien und umfassen alle Klassendateien und Ressourcen einer Anwendung. Eine Erweiterung dieses Bundle-Paradigmas führt die Möglichkeit ein, die eingekapselte Anwendung direkt auf eine JVM auszuführen ohne vorher eine native ausführbare Datei dafür zu erzeugen. Wir führen einen Bundle-Launcher ein, der als primärer Einstiegspunkt eines Bundles spezifiziert ist. Wenn dieser Launcher ausgeführt wird, konfiguriert er eine virtuelle Umgebung mit den Inhalten des Bundles und ermöglicht eine unkomplizierte Ausführung der Anwendung. Darüber hinaus bieten wir die Flexibilität, die Anwendung innerhalb eines Containers auszuführen oder den *Native Image Agent* anzuhängen. Der Native Image Agent ist ein Java-Agent zur Erfassung der Verwendungen dynamischer Aspekte währen der ausführung von Java Anwendungen, die für den Aufbau einer nativen ausführbaren Datei angegeben werden müssen.

Zusammenfassend bietet GraalVM Native Image Bundles einen systematischen Ansatz zur Reproduktion einer nativen ausführbaren Datei unter denselben Bedingungen und mit denselben Eingaben als während der Bundle Erstellung. Diese Arbeit behandelt die Anforderungen an Determinismus, Ressourcenisolierung und direkte Ausführung, unterstützt durch Bundles und Virtualisierungswerkzeuge.

Contents

Chapter 1

Introduction

Nowadays, most Java applications are not self-contained and use various libraries, configuration files, and often depend on specific environment variables. This results in multiple files required to run an application, which have to be managed such that we always have the correct version for each dependency. However, versions might change over time, or some dependencies are no longer available that were available during the initial build and deployment of an application. Therefore, the behavior of an application might change even though the application itself remains the same. This is non-deterministic behavior at application build time, which we want to avoid. One measure is to build a compiled executable with GraalVM Native Image that combines the application with all its dependencies, such that it is executable at any time in the same way without worrying about version updates or unavailability of dependencies. However, if we want to make changes to the executable such as hotfixes, security updates in the Native Image Builder, or changing the target architecture, we would need to recompile the executable and end up with the same non-deterministic problem as before.

How to guarantee that we can rebuild a native image at a later point in time, such that we have the same inputs as in the initial build? Native Image Bundles, a feature for GraalVM Native Image, introduces so-called bundles, archives containing all inputs required by the Native Image Builder to build a native image. Each bundle contains a META-INF directory, conforming to the JAR file format. Furthermore, bundles enable remote building of bundles without a dedicated build server, as we have the application and all dependencies in one file, which is easier to manage. However, we still end up with one key issue if we create this bundle on a regular host system. We can not ensure, that there were no external files accessed from the host's file system or even from the network in the background, which did not end up in the bundle. However, this could eventually cause the build of an executable to fail, if something we were not aware of at bundle creation time is not available any longer. We want to provide a solution that all inputs used for the initial build will be used again in subsequent builds from a bundle. Therefore, we create a temporary system with a controlled environment, which we are set up to have full control over what can and can not be accessed during bundle creation. However, we now have to persist the controlled environment or at least a deterministically executable blueprint of it, such that we can enforce the same environment at a later point in time.

Since Native Image Bundles encapsulate all inputs required to build a native image for a given application, they are also a natural fit for executing the bundled application directly on JVM instead of using it as input for the Native Image Builder. Therefore, with some modifications in the bundle creation process, we can inject resources that enable us to run the bundled application on the Java HotSpot VM. Much like a Java JAR file, which is quite similar to a bundle containing an application with its dependencies, can be invoked to execute a stand-alone application. Building up on that, a bundled application may also make use of some external files through the host system's file system or network, which again might not be available at a later point in time or on a different system. Again, a controlled environment that can be tweaked in a way to create a deterministic execution of a bundled application can help us out as well.

Chapter 2

Background

In this chapter, we introduce the tools and artifacts used throughout the thesis. First, Section 2.1 describes GraalVM and its features, including Native Image and Native Image Bundles. Then, in Section 2.2, we will explain virtualization tools and their usage. Finally, in Section 2.3, we will give a brief description of the JAR file specification, which is the archive format used for Native Image Bundles.

2.1 GraalVM

GraalVM [6] is a high-performance *Java Virtual Machine (JVM)* [14], based on *OpenJDK HotSpot* [23]. Furthermore, it provides several improvements and additional features:

1. Advanced just-in-time (JIT) compiler: For better optimized Java applications
2. GraalVM *Truffle* [31]: A language implementation framework, which enables running other languages such as JavaScript, Python, Ruby, and more (see Figure 2.1). With Truffle, Java, and other supported languages can also interoperate with each other, allowing to mix multiple programming languages in one application.
3. *Native Image* [19] tool: A tool to ahead-of-time (AOT) compile Java applications into native executables, which are also called native images. A further extension of Native Image is the *Native Image Bundles* [21] feature, which offers a simple way to archive all required dependencies for building an application into a native image.

In this thesis, the main focus lies on the Native Image Bundles feature.

2.1.1 Native Image

A native image is a stand-alone executable that contains AOT-compiled Java code. It includes compiled code from files necessary to run an application, which are application classes, classes from dependencies, runtime library classes, and statically linked native code from JDK. However, a native image does not run on a JVM but contains native code that can directly be executed on the target architecture. All relevant parts from the JVM, such as memory monitoring, thread scheduling, and more are provided in native code by a purpose-built runtime system, called *Substrate VM* [29]. Compared to executing an application on a JVM, a native image has a significantly faster startup time and lower runtime memory overhead.

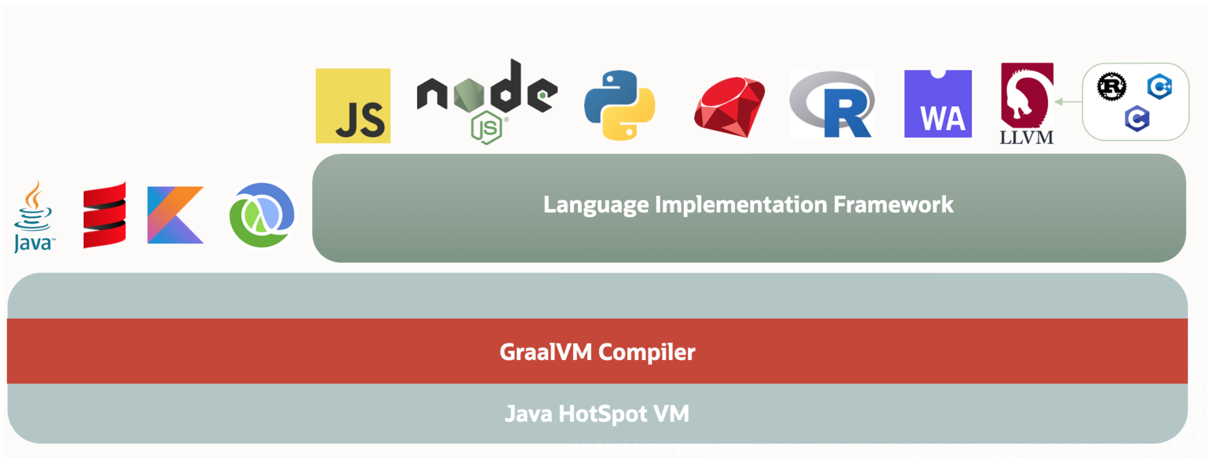

Figure 2.1: GraalVM Architecture [7]

To build a native image, we make use of the Native Image Builder. The builder processes all classes that form a native image, including application classes, classes from dependencies, runtime library classes, and statically linked code from the JDK. All classes are statically analyzed to determine methods and fields that are reachable during application execution, and AOT compiles them into a standalone executable. However, compared to running the application on JVM on any system, a native image is built for a specific operating system and architecture, hence it will only run on systems with the architecture it was created for. This process is called a Native Image Build.

2.1.2 Native Image Build

To build a native image, we first need to add the Native Image tool to GraalVM[1]. Therefore, we make use of the *GraalVM Updater* [11] tool as seen in Listing 2.1. This is required to make the Native Image tool available in the GraalVM bin directory `$GRAALVM_HOME/bin`.

```
$ gu install native-image
```

Listing 2.1: Install the Native Image tool with the GraalVM Updater tool

After the Native Image tool is installed, we can use it as a command-line tool to create native images of applications. To build a native image, we need to specify the classpath and/or module path of an application, as well as provide the name of the main class, a JAR file that specifies the main class in its manifest, or a module with a main class. Whatever way was chosen, the resulting native image will have the same behavior as executing the main class on JVM. Furthermore, we can specify the name of the resulting executable as well as pass some more arguments to the Native Image tool. Commonly used ones are listed and explained in the help text of the Native Image tool[2] (see Listing 2.2). To print the help text, we can use the *--help* option. The most important options for this bachelor's thesis, however, are *--bundle-create* and *--bundle-apply*. These options are explained in the extra help text, shown in Listing 2.3, which is accessible with the command-line option *--help-extra*.

[1]Using GraalVM 22.X [22]
[2]A complete list of options can be found in the GraalVM documentation [20]

```
GraalVM Native Image (https://www.graalvm.org/native-image/)

This tool can ahead-of-time compile Java code to native executables.

Usage: native-image [options] class [imagename] [options]
           (to build an image for a class)
   or  native-image [options] -jar jarfile [imagename] [options]
           (to build an image for a jar file)
   or  native-image [options] -m <module>[/<mainclass>] [options]
       native-image [options] --module <module>[/<mainclass>] [options]
           (to build an image for a module)

where options include:

    @argument files       one or more argument files containing options
    -cp <class search path of directories and zip/jar files>
    -classpath <class search path of directories and zip/jar files>
    --class-path <class search path of directories and zip/jar files>
                          A %pathsep% separated list of directories, JAR
                             archives,
                          and ZIP archives to search for class files.
    -p <module path>
    --module-path <module path>...
                          A %pathsep% separated list of directories, each
                             directory
                          is a directory of modules.
    --add-modules <module name>[,<module name>...]
                          root modules to resolve in addition to the
                             initial module.
                          <module name> can also be ALL-DEFAULT, ALL-SYSTEM
                             ,
                          ALL-MODULE-PATH.
    -D<name>=<value>      set a system property
    -J<flag>              pass <flag> directly to the JVM running the image
        generator
    --diagnostics-mode    enable diagnostics output: class initialization,
       substitutions, etc.
    --enable-preview      allow classes to depend on preview features of
       this release
    --enable-native-access <module name>[,<module name>...]
                          modules that are permitted to perform restricted
                             native operations.
                          <module name> can also be ALL-UNNAMED.
    --verbose             enable verbose output
    --version             print product version and exit
    --help                print this help message
    --help-extra          print help on non-standard options
```

Listing 2.2: Native Image tool help text

```
Non-standard options help:

--exclude-config      exclude configuration for a comma-separated pair of
   classpath/modulepath pattern and resource pattern. For example: '--
   exclude-config foo.jar,META-INF\/native-image\/.*.properties' ignores
   all .properties files in 'META-INF/native-image' in all JARs named 'foo.
   jar'.
--expert-options      lists image build options for experts
```

```
--expert-options-all  lists all image build options for experts (use at
    your own risk). Options marked with [Extra help available] contain help
    that can be shown with --expert-options-detail
--expert-options-detail
                      displays all available help for a comma-separated
                          list of option names. Pass * to show extra help
                          for all options that contain it.

--configurations-path <search path of option-configuration directories>
                      A %pathsep% separated list of directories to be
                          treated as option-configuration directories.
--debug-attach[=<port or host:port (* can be used as host meaning bind to
    all interfaces)>]
                      attach to debugger during image building (default
                          port is 8000)
--diagnostics-mode    Enables logging of image-build information to a
    diagnostics folder.
--dry-run             output the command line that would be used for
    building

--bundle-create[=new-bundle.nib]
                      in addition to image building, create a Native Image
                          bundle file (*.nib file) that allows rebuilding of
                           that image again at a later point. If a bundle-
                          file gets passed, the bundle will be created with
                          the given name. Otherwise, the bundle-file name is
                           derived from the image name. Note both bundle
                          options can be combined with --dry-run to only
                          perform the bundle operations without any actual
                          image building.
--bundle-apply=some-bundle.nib
                      an image will be built from the given bundle file
                          with the exact same arguments and files that have
                          been passed to native-image originally to create
                          the bundle. Note that if an extra --bundle-create
                          gets passed after --bundle-apply, a new bundle
                          will be written based on the given bundle args
                          plus any additional arguments that haven been
                          passed afterwards. For example:
                      > native-image --bundle-apply=app.nib --bundle-create
                          =app_dbg.nib -g creates a new bundle app_dbg.nib
                          based on the given app.nib bundle. Both bundles
                          are the same except the new one also uses the -g
                          option.

-E<env-var-key>[=<env-var-value>]
                      allow native-image to access the given environment
                          variable during image build. If the optional <env-
                          var-value> is not given, the value    of the
                          environment variable will be taken from the
                          environment
                      native-image was invoked from.

-V<key>=<value>       provide values for placeholders in native-image.
    properties files
```

Listing 2.3: Native Image tool extra help text

Furthermore, we can also use the Native Image Builder to build a native shared library. In this case, the native image itself is not directly executable on JVM, but it provides callable entry points. A shared library can be built with a given main class or a JAR file, similar to an executable native image. The shared library will have the main method of the main class as its entry point. However, shared libraries can also be built without specifying a main class. Nevertheless, a shared library needs at least one entry point to be useful, but it also can have multiple entry points. Therefore, entry points can also be defined manually by annotating methods with `@CEntryPoint`.

2.1.3 Native Image Bundles

With the Native Image tool, we can also create a so-called native image bundle. A bundle is an archive, that contains all inputs to successfully reproduce the native image at a later point in time, without relying on the availability of the same environment and dependencies. Therefore, it contains all application class files including class files from all its dependencies, which are required for building a native image or native shared library. All files are archived into one **.nib* file, which is a form of a JAR file with a manifest file and a fixed internal structure.

To create a specific native image bundle, we have to pass the *--bundle-create* option to a Native Image tool command line. This builds a native image and additionally creates the corresponding **.nib* file. This bundle then contains all class files including dependencies, some configuration files, and the native image itself. We can then use the bundle file with the command-line option *--bundle-apply* to rebuild the same native image at a later point in time for any architecture.

Figure 2.2 shows the format of a bundle file. It has a specific structure of its internal directories that contain class files, other JAR files, some configuration files, the built native image, and the manifest:

- **nibundle.properties**: Contains the bundle version info with bundle format version, platform, and architecture and GraalVM and Native Image version used for bundle creation.
- **input/auxiliary**: Contains auxiliary files that are passed to the Native Image Builder via arguments, such as external *JSON* [15] files.
- **input/classes**: Contains all classpath and module path entries passed to the builder. Classpath in *cp* and module path in *p*.
- **build.json**: Full command line for the Native Image tool without *--bundle-create* and *--bundle-create.*
- **environment.json**: Key-Value pair of environment variable names and values used for building the native image, that were specified with the *-E* option.
- **path_canonicalizations.json**: Records path canonicalizations during bundle creation for the input files. Contains mappings from relative paths to their absolute counterparts.
- **path_substitutions.json**: Records path substitutions during bundle creation for the input files. Contains mappings from absolute paths of the host system to internal paths of the native image bundle.

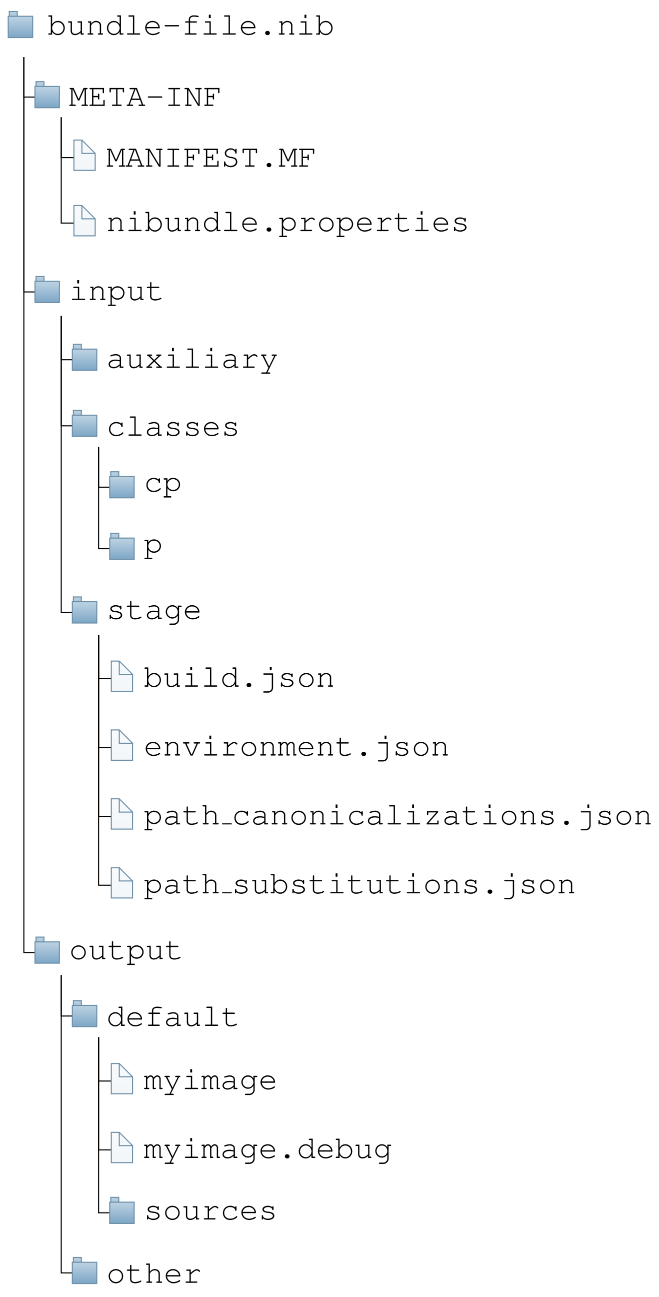

Figure 2.2: Native Image Bundles file format [21]

- **output/default**: Contains all output files that end up on the default output path (specified with the *-H:Path* option) such as the created native image, its debug info, and the debug sources.
- **output/other**: Contains all output files from the Native Image Builder that would have been written to arbitrary paths if no bundle was created.

However, if we add the option *--dry-run*, building the native image is skipped. Therefore, we end up with just the bundle file which also does not contain the native image, as it never got built. In this case, the output directory is omitted, as the Native Image tool with the *--dry-run* option enabled does not produce any image build output.

2.1.4 Native Image Agent

The *Native Image Agent* [30] is a *JVM Tool Interface (JVMTI)* [16] agent that tracks the usage of dynamic features of an execution on a regular JVM. It covers usages of the Java Native Interface (JNI), Java Reflection, Dynamic Proxy objects, and Java resource access [1]. As native images are AOT compiled and rely on the static analysis of the reachable code, the Native Image tool might not be able to predict all the usages of dynamic features. However, we can configure the use of dynamic features with configuration files, which can also be created by the Native Image agent.

The Native Image agent is enabled by providing *-agentlib* if the used JVM is a GraalVM with the Native Image agent installed or *-agentpath* with the path to the Native Image agent on any JVM that supports JVMTI. During the execution of the application, the Native Image agent intercepts dynamic calls and stores the information in the configuration files:

- *jni-config.json*
- *reflect-config.json*
- *proxy-config.json*
- *resource-config.json*

To provide those configuration files to the Native Image tool, we place them in a `META-INF/native-image/.../` directory on the classpath or module path. If Native Image finds any of the configuration files in such directories or their subdirectories, the configuration files are automatically included in a Native Image Build.

2.1.5 GraalVM Container Images

A GraalVM container image is a container image (see Section 2.2) with GraalVM set up in it. There are multiple GraalVM container images provided depending on the architecture, Java version, and the set of GraalVM features. Each of the container images comes with either Oracle Linux 7, 8, or 9 as their Linux distribution. Furthermore, each of the container images is currently provided with Java 17 and Java 20 and for the architectures `amd64` and `arm64`. There are different images based on the feature set as described in Table 2.1.

All the container images are published to the GitHub Container Registry [9] and can be pulled from there. However, for this bachelor thesis, the *native-image-community*

Package	Description
jdk-community	A size compact GraalVM Community Edition container image with the GraalVM JDK pre-installed.
graalvm-community	A GraalVM Community Edition container image with the *gu* utility [11] to install additional features.
native-image-community	A size compact GraalVM Community Edition container image with the Native Image support.
truffleruby-community	A size compact GraalVM Community Edition container image with the Ruby runtime. It uses a standalone build of TruffleRuby.
nodejs-community	A size compact GraalVM Community Edition container image with the Node.js runtime.
graalpy-community	A size compact GraalVM Community Edition container image with the GraalPy runtime.

Table 2.1: Available GraalVM container images [9]

container images are the most important ones and have the Native Image tool installed. Listing 2.4 depicts how a *native-image-community* container is pulled from the registry using Docker. This container image contains an installation of GraalVM with the Native Image tool available.

```
$ docker pull ghcr.io/graalvm/native-image-community:20
```

Listing 2.4: Pull a *native-image-community* container image [8]

Furthermore, all the Dockerfiles used for creating the container images are published in a GitHub repository [10]. For *native-image-community*, there are two different types of Dockerfiles available. First, one that just contains GraalVM with the Native Image tool installed in Listing 2.5. Second, one that builds up on the first one, but also provides a *musl libc* [18] toolchain which allows us to create fully statically linked executables in Listing 2.6.

```
# LICENSE UPL 1.0
#
# Copyright (c) 2023 Oracle and/or its affiliates. All rights reserved.
#

ARG BASE_IMAGE=container-registry.oracle.com/os/oraclelinux:8-slim

FROM ${BASE_IMAGE}

LABEL \
    org.opencontainers.image.url='https://github.com/graalvm/container' \
    org.opencontainers.image.source='https://github.com/graalvm/container/
        tree/master/native-image-community' \
    org.opencontainers.image.title='Native Image Community Edition' \
    org.opencontainers.image.authors='GraalVM Sustaining Team <graalvm-
        sustaining_ww_grp@oracle.com>' \
    org.opencontainers.image.description='GraalVM Native Image Community Edition ahead of time compilation functionality to generate under closed-world assumption an executable image or a shared object. This resulting binary includes the application, the libraries, the JDK and does not run on the Java VM, but includes necessary components like memory management and thread scheduling etc..'

# Note: If you are behind a web proxy, set the build variables for the build:
#       E.g.:  docker build --build-arg 'https_proxy=...' --build-arg 'http_proxy=...' --build-arg 'no_proxy=...' ...

ARG GRAALVM_VERSION=23.0.0
ARG JAVA_VERSION=20
ARG YUM_REPO=""
ARG YUM_REPO_DEFAULT=https://yum.oracle.com/repo/OracleLinux/OL8/graalvm/community/
ARG TEMP_REGION=""

ENV LANG=en_US.UTF-8 \
    JAVA_HOME=/usr/lib64/graalvm/graalvm-community-java${JAVA_VERSION}

WORKDIR /app

RUN if [ "$YUM_REPO" == "" ]; then YUM_REPO_CURRENT="$YUM_REPO_DEFAULT\$basearch"; else YUM_REPO_CURRENT="$YUM_REPO"; fi \
    && echo -e "\
[ol8_graalvm_community]\n\
name=Oracle Linux 8 graalvm community (\$basearch)\n\
baseurl=$YUM_REPO_CURRENT\n\
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-oracle\n\
gpgcheck=1\n\
enabled=1\
" > /etc/yum.repos.d/ol8_graalvm_community.repo \
    && echo "$TEMP_REGION" > /etc/dnf/vars/ociregion \
    && microdnf --enablerepo ol8_codeready_builder install -y graalvm-community-${JAVA_VERSION}-native-image \
    && rm -rf /var/cache/yum \
    && echo "" > /etc/dnf/vars/ociregion \
    && echo -e "\
[ol8_graalvm_community]\n\
name=Oracle Linux 8 graalvm community (\$basearch)\n\
baseurl=$YUM_REPO_DEFAULT\$basearch\n\
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-oracle\n\
gpgcheck=1\n\
enabled=1\
" > /etc/yum.repos.d/ol8_graalvm_community.repo

ENTRYPOINT [ "native-image" ]
CMD [ "--version" ]
```

Listing 2.5: *native-image-community* Dockerfile based on Oracle Linux 8 and Java 20 for GraalVM 23.0.0 [25]

```
# LICENSE UPL 1.0
#
# Copyright (c) 2015,2022 Oracle and/or its affiliates. All rights reserved.
```

```
#

ARG GRAALVM_VERSION=23.0.0

ARG BASE_IMAGE=ghcr.io/graalvm/native-image-community:20-ol8

FROM ${BASE_IMAGE} as base

FROM base as muslib

LABEL \
    org.opencontainers.image.url='https://github.com/graalvm/container' \
    org.opencontainers.image.source='https://github.com/graalvm/container/
       tree/master/native-image-community' \
    org.opencontainers.image.title='Native Image Community Edition' \
    org.opencontainers.image.authors='GraalVM Sustaining Team <graalvm-
       sustaining_ww_grp@oracle.com>' \
    org.opencontainers.image.description='GraalVM Native Image Community
       Edition ahead of time compilation functionality to generate under
       closed-world assumption an executable image or a shared object. This
        resulting binary includes the application, the libraries, the JDK
       and does not run on the Java VM, but includes necessary components
       like memory management and thread scheduling etc..'

ARG TEMP_REGION=""

ARG MUSL_LOCATION=http://more.musl.cc/10/x86_64-linux-musl/x86_64-linux-
   musl-native.tgz

ARG ZLIB_LOCATION=https://zlib.net/fossils/zlib-1.2.11.tar.gz

ENV TOOLCHAIN_DIR=/usr/local/musl \
    CC=$TOOLCHAIN_DIR/bin/gcc

RUN echo "$TEMP_REGION" > /etc/dnf/vars/ociregion \
    && rm -rf /etc/yum.repos.d/ol8_graalvm_community.repo \
    && mkdir -p $TOOLCHAIN_DIR \
    && microdnf install -y wget tar gzip make \
    && wget $MUSL_LOCATION && tar -xvf  x86_64-linux-musl-native.tgz -C
       $TOOLCHAIN_DIR --strip-components=1  \
    && wget $ZLIB_LOCATION && tar -xvf zlib-1.2.11.tar.gz \
    && cd zlib-1.2.11 \
    && ./configure --prefix=$TOOLCHAIN_DIR --static \
    && make && make install

FROM base as final
COPY --from=muslib /usr/local/musl /usr/local/musl

RUN echo "" > /etc/dnf/vars/ociregion

ENV TOOLCHAIN_DIR=/usr/local/musl \
    CC=$TOOLCHAIN_DIR/bin/gcc

ENV PATH=$TOOLCHAIN_DIR/bin:$PATH

ENTRYPOINT [ "native-image" ]
CMD [ "--version" ]
```

Listing 2.6: *native-image-community* Dockerfile with a *musl libc* toolchain based on Oracle Linux 8 and Java 20 for GraalVM 23.0.0 [26]

2.2 Virtualization Tools

A virtualization tool is used to build and run so-called containers on a host system. It is a form of virtualization which means, the container operates as a separate isolated environment. The main use cases for virtualization tools, such as *Docker* [2] or *Podman* [27], are continuous integration and continuous delivery, where standardized environments are created to allow for a more streamlined development process. Furthermore, we can run multiple containers at the same time, which makes it viable to run multiple services at the same host system, while requiring less overhead than other forms of virtualization.

A container does not rely on the host system's environment, therefore, it contains everything needed to run whichever application it was built for. By default, a container is very well isolated from the host system. However, with virtualization tools, we also can control the level of isolation of the container's network storage or other subsystems. The container may have no network, or storage from the host system is mounted into a container, thus the container is not fully isolated from the host system but rather makes use of the same files if necessary.

The base for a container is a so-called image, which is a template for a container. Therefore, we can use one image to create multiple containers in the same environment. Images are either built from scratch or used from others, that created and published them in so-called registries. Furthermore, we can also base an image on another image and set it up to our needs. For building images, the virtualization tool needs step-by-step instructions on how to do that. For virtualization tools Docker and Podman, we need to create a so-called Dockerfile, which specifies the steps a virtualization tool has to take to create the requested image. As a container image is read-only, we have the same initial state of the container every time we run a container based on an image. Changes made during a container's lifetime then only affect the container rather than the image, which results in the image remaining unchanged and reusable for other containers at the same time. Furthermore, if we want to build an image that already exists, or we make use of an image that already exists on a host system, the existing image is reused.

Dockerfiles have a defined format and contain specific commands, such as `FROM` for creating an image from another base image or `RUN` for executing commands required for the image. The following commands will be used in this thesis:

- FROM <image>[:<tag>] [AS <name>]: A Dockerfile must begin with a `FROM` instruction, and it specifies the parent image which is used for building the image described by the Dockerfile. Each `FROM` creates a new build stage and can be used multiple times in a Dockerfile, however, each `FROM` clears the state created by previous instructions. Furthermore, a build stage may be named and referred to by this name in other build stages.
- LABEL {<key>=<value>}: Adds metadata as a key-value pairs to an image.
- ARG <name>[=<default-value>]: Defines a build-time variable that a user can pass to the builder at image build time. `ARG` is the only command allowed before

`FROM` if the defined argument is used in the `FROM` command.

- ENV {<key>=<value>}: Sets an environment variable to the specified value.

- WORKDIR <path>: Sets the working directory for any instruction. The root directory "/" by default, however, this may already be set to another value by the used parent image. Relative paths change the `WORKDIR` relative to itself.

- RUN <command>: Executes any commands on the current image.

- COPY {<src>} <dest>: Copies files from a source relative to the build context to a destination path in the containers' file system. Adding *--from* allows copying files from previous named build stages.

- ENTRYPOINT ["executable", "param1", "param2"]: Configures a container to run as an executable with an entry point relative to the `WORKDIR`.

- CMD ["executable","param1","param2"]: Only the last `CMD` command in a Dockerfile will take effect. The `CMD` command's main purpose is to provide defaults for executing a container. If an executable was set in `ENTRYPOINT`, the executable in `CMD` may be omitted and default parameter values for the executable in `ENTRYPOINT` are added here.

Overall, the usual workflow for running containers is to first create a Dockerfile based on a parent image, then build the desired image and run the image as a container. Therefore, virtualization tools do at least offer a command-line interface, but there is no standard for virtualization tools. However, the focus of this bachelor's thesis is on Docker and Podman. Listing 2.7 shows example usages of Docker for building an image and running a container according to the command-line interface description for *docker build* [4] and *docker run* [5]. This example creates a container image "myimage" with the build context "." given a Dockerfile named "myDockerfile" which is located in the current working directory. The *docker run* command then runs the container, mounts a read-only directory called input on the current working directory to the container's root directory, and executes its specified command. Additionally, no network access is provided for the container, and it is automatically stopped after the execution has finished.

The tools Docker and Podman share an equivalent command-line interface for building an image and running a container, therefore, Podman may be used as a synonym for Docker. The main difference between Podman and Docker is, that Docker uses a daemon and needs root privileges without the rootless setup. Furthermore, Docker requires a build context to be specified, which is then passed to the Docker daemon. Passing the build context is optional for Podman.

```
$ docker build -f myDockerfile -t myimage .

$ docker run --network=none --rm --mount,type=bind,source=input,target=/
    input,readonly myimage
```

Listing 2.7: Command lines for building images and running containers

2.3 JAR File Specification

JAR stands for Java Archive and is a file format defined in the *JAR File Specification* [12], which is based on the ZIP file format. Similar to a ZIP file, a JAR is used to aggregate multiple files into one, such as class files or resources. Additionally, a JAR file may contain an optional `META-INF` directory. The `META-INF` directory can contain the following special files and directories that are recognized and interpreted by Java to configure applications, class loaders, and services [12]:

- **MANIFEST.MF**: Contains metadata for the contained package.
- **INDEX.LIST**: Contains location information for packages defined in an application and is used by class loaders to speed up class loading.
- **<base-file-name>.SF**: Signature for the JAR file.
- **<base-file-name>.DSA, <base-file-name>.RSA, or <base-file-name>.EC**: Signature block file associated with the signature file.
- **services/**: A directory that stores all service provider configurations for JAR files on the classpath or module path.
- **versions/**: Contains versioned class and resource files for multi-release JAR files.

In this section, we will focus on the manifest file, more specifically on how to create a stand-alone executable JAR file, as it is an integral part of the JAR files we make use of in this thesis. The manifest file consists of a main section and optionally a list of sections for individual JAR file entries. The file itself, as well as each section, must end with a `newline` which is defined as either `\CR LF"`, `\LF"`, or just `\CR"`. The main section contains metadata of the JAR file itself, such as the manifest file version, vendor information of the used Java implementation, and classpath URLs of libraries this JAR file requires. Individual sections cover attributes for packages or files contained in the JAR file. Multiple sections for the same file are merged, and the bottommost value is used if an attribute is specified multiple times. Attributes that are not defined are ignored, therefore, a manifest file may contain implementation-specific attributes for applications.

Furthermore, a manifest file's main section also contains metadata for creating an executable JAR file for stand-alone applications. Executable JAR files require a `Main-Class` attribute, which contains the bundled application's main class without a *.class* extension. With the `Main-Class` specified, JAR files can be invoked as seen in Listing 2.8.

```
$ java -jar <jar-name>.jar
```

Listing 2.8: Execute a JAR file

2.4 Problem Statement

The main focus of this thesis is on the Native Image Bundles feature and its ability to recreate a bundle deterministically. Bundles contain all class files of the application, class files from all dependencies, and some configuration files. This ensures that we have all the files required to build a native image at any point in time with this bundle. However, when building a native image, we might inadvertently access some files on the host systems or some files from the web that we are not aware of and are not stored in the bundle. Therefore, if we want to recreate the same native image at a later point in time on a different machine, those files might not be available anymore, hence building a native image from the bundle might fail or we build an invalid or malfunctioning image.

To avoid this behavior of the Native Image Bundles feature, we extend it by providing support for creating the bundle in a controlled environment. This controlled environment is achieved by setting up a container image. When running the container, we can also restrict its network access. We can then also run the Native Image tool inside this container, therefore, it does not have access to the host system and the network at all. With this, we create a bundle that is known to not rely on the host system or the network. However, if it relies on either of them, the bundle creation will fail, as we failed to explicitly specify all necessary inputs required to build a native image fully deterministically. Furthermore, we want to make sure that the bundle also contains the information on how to create the used container image, such that we can recreate the same environment for which it is known to be possible to build a native image out of the bundle.

The secondary goal of this thesis is to implement a feature for executing native image bundles. A bundle is a JAR file, and it is possible to execute a JAR file with a JVM, however, this feature is missing for bundles. As a bundle is designed to be an archive for any application, we need a way to execute any native image bundle, no matter what application is bundled and how this bundled application is executed. To provide the same entry point for all bundle execution, we will have to add a standard main class to every bundle, that uses the files from the bundle to initialize the environment for execution and execute the bundled application.

Chapter 3

Implementation

The implementation of this thesis is based on the already existing Native Image Bundles feature (see Section 2.1.3). It serves as an extension to the Native Image tool's command-line interface, more specifically the Native Image Bundles arguments *--bundle-create* and *--bundle-apply* and the usage of bundle files. The extended functionality of the Native Image Bundles arguments comprises building the native image for the bundle inside a virtualized container and creating a bundle without building a native image. Furthermore, a so-called Bundle Launcher is introduced and injected into a native image bundle. The Bundle Launcher enables executing a bundled application on any JVM by executing the native image bundle as a JAR file. All dependencies are then loaded from within the bundle.

In this section, we will discuss implementation details on the new features introduced to Native Image Bundles. In Section 3.1, we will give an overview of the extensions to the Native Image Bundles file format and their purpose. In Sections 3.2 and 3.3, the Native Image Bundles feature extension for the virtualized Native Image Bundles Build will be introduced. Finally, in Section 3.4, we will discuss the Bundle Launcher extension, which allows us to execute the application archived in a bundle.

3.1 Native Image Bundles Extensions

First, we want to introduce changes made to the created bundle itself, which are limited to the format of Native Image Bundles. To cover the new functionality, a native image bundle permanently stores the configuration files used to build a container image, as well as executing a bundle, such as the command-line arguments needed to run the bundled application as a JAR file. Furthermore, we inject a *Bundle Launcher* (see Section 3.4) into every created bundle.

A native image bundle keeps its general structure and all files as described in 2.1.3. In Figure 3.1 we can see the extensions made to Native Image Bundles file format, with two more configuration files, a Dockerfile, and the compiled Bundle Launcher. The added configuration files are used to capture and store information for the new features:

- **container.json**: If a native image bundle is created and the corresponding native image is built, this configuration file captures the used virtualization tool, its version, and the name of the virtualized image. However, we have two more cases to pay attention to:

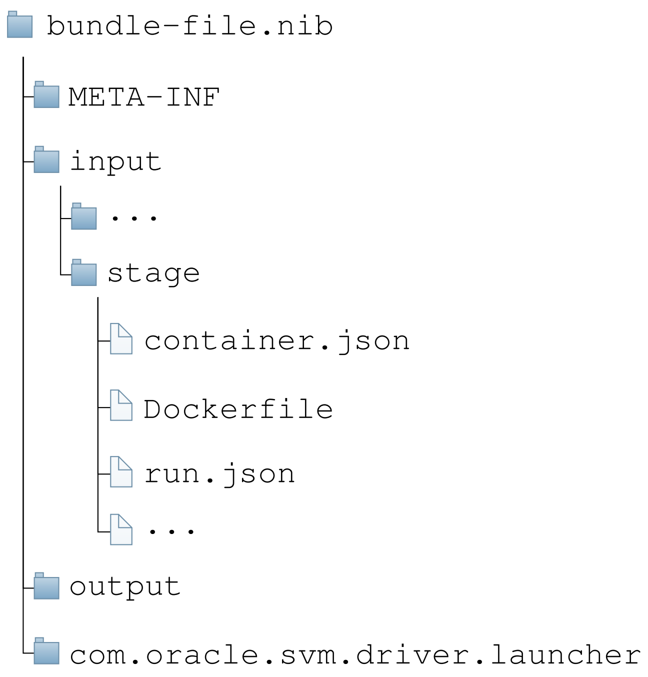

Figure 3.1: Extended Native Image Bundles file format

1. The native image is built without a container: There is no virtualization information available, which means this file is omitted.
2. The native image is not built: This is called a dry-run, where we just create a bundle, but not the actual native image. This may occur if we want to test, whether we can create a native image bundle given a specific input, or if we only want to create a more lightweight bundle, as we do not build and store the native image in the bundle. Therefore, we also do not build a container image, thus we omit to add the file to the bundle. However, we may specify a virtualization tool, which will then be captured in this file.

- **Dockerfile**: The Dockerfile used to create the container image in a virtualized build or the default Dockerfile (see Section 3.3.2) otherwise.
- **run.json**: Contains all command-line arguments necessary to execute a native image bundle as a JAR file, excluding classpath and module path arguments.
- **com.oracle.svm.driver.launcher**: Contains the compiled Bundle Launcher, which is used to execute the bundled application as a JVM application.

3.2 Virtualized Native Image Build

Whenever we create a native image bundle, by default a native image is built as well. However, this bundle and the corresponding native image are built on the host system. This makes the process prone to missed files or configuration setups. For example, the Native Image Build process depends on some file or configuration, but the user is not explicitly aware of that. To identify such cases, we introduce virtualized bundle builds. Therefore, we add the option *container* to the bundle arguments that can be set to trigger a virtualized bundle build. If we use this option, the Native Image tool takes a Dockerfile, either from the user as a path, the existing bundle during *--bundle-apply*, or the default Dockerfile (see Section 3.3.2). Then a container image is created with this Dockerfile as its template. Finally, we run the container and the host system's GraalVM JDK, and all required source files are mounted into the container and the Native Image tool is executed inside.

3.2.1 Parse Extended Native Image Bundles Options

The extension of the Native Image Bundles feature adds three new options. These options are parsed separately from the already existing bundle options (see Listing 3.1) and have to be separated from the main bundle arguments *--bundle-apply* and *--bundle-create* with a comma. The first option *dry-run* is not specific to virtualized builds, it can be used for any bundle build to integrate the *--dry-run* argument of Native Image into the bundle arguments. The other two options control whether the bundle is built in a virtualized environment and how this environment is set up. The extended options have the following implications for bundle builds:

1. **dry-run**: Creates a native image bundle without building the actual native image. As a result, we will get all the files required for building a native image captured in a native image bundle, but not the native image itself. This behavior was previously achieved by using the Native Image argument *--dry-run* combined with one of the bundle arguments. Therefore, this option serves as a stand-in, that integrates the existing way of building bundles without native images into the bundle arguments.

2. **container**: Creates a `ContainerSupport` object, that fetches configuration files from the bundle's stage directory and stores all required information for virtualization. It allows the specification of a virtualization tool according to Section 4.1. If a virtualization tool is specified with this option, we check whether it is supported, otherwise, the bundle creation fails. Furthermore, this option is only allowed once to have one unique virtualization environment for the Native Image Build. This `ContainerSupport` object is then used to create a container image, run the container, and build a native image inside it instead of directly on the host system.

3. **dockerfile**: Builds the container image used for building a native image with the provided custom Dockerfile. May only be specified after *container*, as we need a `ContainerSupport` object to capture the Dockerfile. However, if this option is omitted on bundle creation, we make use of the default Dockerfile described in Section 3.3.2.

```
        switch (optionKey) {
            case DRY_RUN_OPTION -> nativeImage.setDryRun(true);
            case CONTAINER_OPTION -> {
                if (containerSupport != null) {
                    throw NativeImage.showError(String.format("native-image bundle allows option %s to be specified only once.", optionKey));
                }
                containerSupport = new ContainerSupport(stageDir, NativeImage::showError, LogUtils::warning, nativeImage::showMessage);
                useContainer = true;
```

```java
                if (optionValue != null) {
                    if (!ContainerSupport.SUPPORTED_TOOLS.contains(optionValue)) {
                        throw NativeImage.showError(String.format("Container Tool '%s' is not supported, please use one of the following tools: %s", optionValue, ContainerSupport.SUPPORTED_TOOLS));
                    }
                    containerSupport.tool = optionValue;
                }
            }
            case DOCKERFILE_OPTION -> {
                if (containerSupport == null) {
                    throw NativeImage.showError(String.format("native-image bundle option %s is only allowed to be used after option %s.", optionKey, CONTAINER_OPTION));
                }
                if (optionValue != null) {
                    containerSupport.dockerfile = Path.of(optionValue);
                    if (!Files.isReadable(containerSupport.dockerfile)) {
                        throw NativeImage.showError(String.format("Dockerfile '%s' is not readable", containerSupport.dockerfile.toAbsolutePath()));
                    }
                } else {
                    throw NativeImage.showError(String.format("native-image option %s requires a dockerfile argument. E.g. %s=path/to/Dockerfile.", optionKey, optionKey));
                }
            }
            default -> throw NativeImage.showError(String.format("Unknown option %s. Use --help-extra for usage instructions.", optionKey));
        }
```

Listing 3.1: Parsing extended Native Image Bundles options

3.2.2 Check for Virtualization Tool

Virtualized Native Image Builds require a virtualization tool to be installed on the host system. Therefore, we fetch the host system's PATH environment variable and look for an executable virtualization tool. Upon finding it, we know it is available on the system and may be used for the containerized build. In Listing 3.2, we create and iterate over a stream of all paths in the environment variable PATH and look for any location that contains the requested executable. The virtualization tool is either provided by the user or the configuration files of a bundle during *--bundle-apply*. However, if no tool is specified, we check if either of the supported tools is available and make use of it (see Listing 3.3).

```java
    private static boolean isToolAvailable(String toolName) {
        return Arrays.stream(System.getenv("PATH").split(":"))
                        .map(str -> Path.of(str).resolve(toolName))
                        .anyMatch(Files::isExecutable);
    }
```

Listing 3.2: Check virtualization tool availablility

```java
            for (String supportedTool : SUPPORTED_TOOLS) {
                if (isToolAvailable(supportedTool)) {
                    if (supportedTool.equals("docker") && !isRootlessDocker()) {
                        messagePrinter.accept(BundleLauncher.BUNDLE_INFO_MESSAGE_PREFIX + "Rootless context missing for docker.");
                        continue;
                    }
                    tool = supportedTool;
                    toolVersion = getToolVersion();
                    break;
                }
            }
            if (tool == null) {
                throw errorFunction.apply(String.format("Please install one of the following tools before running containerized native image builds: %s", SUPPORTED_TOOLS), null);
            }
```

Listing 3.3: Find a supported virtualization tool

Furthermore, if the used virtualization tool is Docker, we also have to check for the availability of the rootless context, which enables us to use Docker without root privileges (see Section 4.1).

3.2.3 Build Container Image

Before we can run a container from a container image, we either have to build the container image or make use of an existing one, that was built with the same Dockerfile. Each container image that is created by the Native Image tool gets the SHA-1 hash of the used Dockerfile as its name. Therefore, we can check for existing container images by looking for a container image on the host system that has the hash of the current Dockerfile as its name.

However, we cannot be sure if the host system contains a different container image with this name. To avoid unintended behavior, or in the worst case malicious use of this feature, we will only use this information to skip redundant log messages from the virtualization tool. Virtualization tools supported by this feature can be used to check if the container image exists or not. Therefore, we build the container image as usual, and it is up to the virtualization tool to reuse an existing container image.

In Listing 3.4 we can see that we first look for a specific container image. The function `getFirstProcessResultLine` returns the first line from the output of a command passed to it in a `ProcessBuilder` or `null` if the response is empty. The *images* command's result is empty if the requested container image name does not exist. Thus, if it returns any string value, we know that there exists a container image that has at least the same name. Most likely, the existing container image is the required one, therefore, we omit the logs of the virtualization tool and replace them with a custom log message. However, even if we found an existing container image, we call the build functionality of the virtualization tool. This leads to the following scenarios:

- Container image name exists: We still execute the build command to make sure that the container image is the correct one. If it is different a new container image is created, if not, which should be the most likely case, the virtualization tool reuses the existing container image.

- Container image name does not exist: We create a new container image for the given Dockerfile and forward the logs of the virtualization tool to the Native Image tool. However, if the virtualization tool has the same container image with a different name, it will reuse this container image and add the new name as an alias to it.

```java
private int createContainer() {
    ProcessBuilder pbCheckForImage = new ProcessBuilder(tool, "images",
         "-q", image + ":latest");
    ProcessBuilder pb = new ProcessBuilder(tool, "build", "-f",
        dockerfile.toString(), "-t", image, ".");

    String imageId = getFirstProcessResultLine(pbCheckForImage);
    if (imageId == null) {
        pb.inheritIO();
    } else {
        messagePrinter.accept(String.format("%sReusing container image
            %s.%n", BundleLauncher.BUNDLE_INFO_MESSAGE_PREFIX, image));
    }

    Process p = null;
    try {
        p = pb.start();
        int status = p.waitFor();
        if (status == 0 && imageId != null && !imageId.equals(
            getFirstProcessResultLine(pbCheckForImage))) {
            try (var processResult = new BufferedReader(new
                InputStreamReader(p.getInputStream()))) {
                messagePrinter.accept(String.format("%sUpdated
                    container image %s.%n", BundleLauncher.
                    BUNDLE_INFO_MESSAGE_PREFIX, image));
                processResult.lines().forEach(messagePrinter);
            }
        }
        return status;
    } catch (IOException | InterruptedException e) {
        throw errorFunction.apply(e.getMessage(), e);
    } finally {
        if (p != null) {
            p.destroy();
        }
    }
}
```

Listing 3.4: Build container image

3.2.4 Run Container for Native Image Build

Finally, after the container image is built and ready to run, we have to create the command line that needs to be executed for starting the container and running the Native Image Build inside it. Listing 3.5 depicts the creation of the command that sets up the virtualized environment for the actual Native Image Build.

By default, a started container runs until it is stopped manually, however, we do not need the container any longer after we finish building the native image. With the option *--rm*, we specify, that the container gets stopped as soon as the command we passed to it finished execution.

To restrict what goes in and comes out of the virtualized environment for the native image built, on the one hand, we can disable network access for the container. With that, we avoid the network as a source of error for non-deterministic native image builds. We can disable network access for a container, by adding the command-line option *--network=none* to the *run* command.

On the other hand, we also want to restrict access to files on the host system. We do not want to fully disable access to the host system, because copying source files into a container and the resulting native image out of the container is inefficient and unnecessary. For creating bundles, we have to define an input directory, and the Native Image tool then also creates an output directory. Furthermore, command line arguments are passed to the Native Image tool with argument files [13]. To keep control of which directories and files will be available to the Native Image tool during the Native Image Build, we declare which directories and files get mapped into the container. This comprises the host system's GraalVM, argument files used for building, and the bundle input and output directories. Therefore, we add one *--mount* argument for each directory and file mounted into the container.

Additionally, we also have to consider the environment variables required for building a native image. One argument for each variable, containing its name and value, is added to the *run* command to inject the required environment variables into the container.

```java
    public List<String> createCommand(Map<String, String> containerEnvironment, Map<Path, TargetPath> mountMapping) {
        List<String> containerCommand = new ArrayList<>();

        // run docker tool without network access and remove container after image build is finished
        containerCommand.add(tool);
        containerCommand.add("run");
        containerCommand.add("--network=none");
        containerCommand.add("--rm");

        // inject environment variables into container
        containerEnvironment.forEach((key, value) -> {
            containerCommand.add("-e");
            containerCommand.add(key + "=" + BundleLauncherUtil.quoteShellArg(value));
        });

        // mount java home, input and output directories and argument files for native image build
        mountMapping.forEach((source, target) -> {
            containerCommand.add("--mount");
            List<String> mountArgs = new ArrayList<>();
            mountArgs.add("type=bind");
            mountArgs.add("source=" + source);
            mountArgs.add("target=" + target.path);
            if (target.readonly) {
                mountArgs.add("readonly");
            }
            containerCommand.add(BundleLauncherUtil.quoteShellArg(String.
                join(",", mountArgs)));
        });

        // specify container name
        containerCommand.add(image);

        return containerCommand;
    }
```

Listing 3.5: Create command for building the native image inside a container

3.3 Capture Virtualization Related Information

After successfully building a native image inside a container, we want to capture and store all related information in the native image bundle. This is necessary to ensure that we can rebuild the same native image at any point in the same virtualized environment, given the same virtualization tool and tool version are available on the host system.

3.3.1 Virtualization Tool

Virtualized builds for Native Image Bundles support the virtualization tools Podman and Docker (see Section 4.1), hence the virtualization tool is another variable for creating a native image inside a container. Therefore, we capture the tool that was used for creating the native image bundle along with its version. Furthermore, the name of the built container image, which is the hash of the contents of the used Dockerfile, is also captured. All this information gets stored in *container.json* in the bundles' stage directory, as seen in Figure 3.1.

Information that is not present during bundle creation is omitted. For example, if we create a bundle without building the corresponding native image, we will not create the container image, therefore we do not have a container image name to add here. If we create the native image bundle without a container, no information is present, thus the whole file is omitted.

Information that got captured can be used later on. If we, for example, create the native image bundle inside a container, we will use the same virtualization tool for applying or executing the bundle by default. However, this can be overruled if another virtualization tool is specified with the *container* option. Additionally, the container image name is used to check whether the required container image is already present on the system that applies or executes a bundle.

3.3.2 Dockerfile

To recall which container image was used for the Native Image Build, we store the Dockerfile that describes the container image. A Dockerfile is stored in every native image bundle, regardless of whether a native image was built inside a container, or even built at all. This means we need to have a default Dockerfile, but it ensures that there is a Dockerfile available when needed, for example, if the bundle is executed virtualized with the Bundle Launcher (see Section 3.4.3).

Listing 3.6 shows the default Dockerfile, which is added to the bundle for non-virtualized builds, and if no Dockerfile was explicitly specified with the Native Image Bundles argument.

The default Dockerfile is based on the Oracle Linux 8 GraalVM native image Dockerfile from the GraalVM container GitHub repository [10]. To base the default Dockerfile on the ones in the container repository, we had the option to choose between Oracle Linux 7, 8, or 9. We selected Oracle Linux 8 because it provides a good middle ground between being more mature compared to Oracle Linux 9 and being supported longer compared to Oracle Linux 7. Compared to the GraalVM container repository, we do not need a fixed Java version. Instead, if the Native Image Builder runs the container image, it mounts itself into the container, such that it is available for building a native image inside.

```
ARG BASE_IMAGE=container-registry.oracle.com/os/oraclelinux:8-slim

FROM ${BASE_IMAGE} as base

RUN microdnf update -y oraclelinux-release-el8 \
    && microdnf --enablerepo ol8_codeready_builder install bzip2-devel ed gcc gcc-c++ gcc-gfortran gzip file fontconfig less libcurl-devel make openssl openssl-devel readline-devel tar glibc-langpack-en \
    vi which xz-devel zlib-devel findutils glibc-static libstdc++ libstdc++-devel libstdc++-static zlib-static \
    && microdnf clean all
RUN fc-cache -f -v

ENV LANG=en_US.UTF-8 \
    JAVA_HOME=/graalvm

WORKDIR /
```

Listing 3.6: Native Image Bundles default Dockerfile

Furthermore, the Native Image tool also supports building *statically linked native images* [28]. Such builds require a *musl* pipeline, *make*, *configure*, and *zlib* to be set up on the system. With containers, we can automate setting up the system for building statically linked native images within a Dockerfile. Therefore, we add an extension to the default Dockerfile that sets up the system in a way to support building statically linked native images, if the command-line arguments *--static* and *--libc=musl* were passed to the Native Image tool. In Listing 3.7 we can see two more build stages. The first fetches and installs all required tools for building a statically linked native image. The second added build stage then copies the installed tools into the default container image. This approach saves storage, as we do not keep files and tools that were only used for installing the pipeline in the container image. This Dockerfile extension is again based on the GraalVM container repository, more specifically the Oracle Linux 8 GraalVM native-image Dockerfile with muslib extension. A native image bundle built for a statically linked native image will not contain any additional files for facilitating such builds in the bundle compared to not statically linked builds. However, the default Dockerfile in Listing 3.6 will be extended by appending the Dockerfile shown in Listing 3.7 to it.

```
FROM base as muslib

ARG TEMP_REGION=""
ARG MUSL_LOCATION=http://more.musl.cc/10/x86_64-linux-musl/x86_64-linux-musl-native.tgz
ARG ZLIB_LOCATION=https://zlib.net/fossils/zlib-1.2.11.tar.gz

ENV TOOLCHAIN_DIR=/usr/local/musl \
    CC=$TOOLCHAIN_DIR/bin/gcc
```

```

RUN echo "$TEMP_REGION" > /etc/dnf/vars/ociregion \
    && rm -rf /etc/yum.repos.d/ol8_graalvm_community.repo \
    && mkdir -p $TOOLCHAIN_DIR \
    && microdnf install -y wget tar gzip make \
    && wget $MUSL_LOCATION && tar -xvf  x86_64-linux-musl-native.tgz -C
       $TOOLCHAIN_DIR --strip-components=1  \
    && wget $ZLIB_LOCATION && tar -xvf zlib-1.2.11.tar.gz \
    && cd zlib-1.2.11 \
    && ./configure --prefix=$TOOLCHAIN_DIR --static \
    && make && make install


FROM base as final

COPY --from=muslib /usr/local/musl /usr/local/musl

RUN echo "" > /etc/dnf/vars/ociregion

ENV TOOLCHAIN_DIR=/usr/local/musl \
    CC=$TOOLCHAIN_DIR/bin/gcc

ENV PATH=$TOOLCHAIN_DIR/bin:$PATH
```
Listing 3.7: Extension of the default Dockerfile for building statically linked native images

3.4 Bundle Launcher

For the second objective of this thesis, we added another feature to Native Image Bundles, which allows us to execute the bundled application. A native image bundle is built up similarly to a JAR file. It contains a *META-INF* directory with a manifest file. The manifest file is set up in a way that each native image bundle can be executed as a JAR file. However, as Native Image Bundles have a specific internal structure and may contain modules or another JAR file as its main application, it was previously not possible to directly execute the bundled application's main class.

To make the bundled application executable, we implemented a new Java application named "Bundle Launcher". It is a self-contained package, which means that it has all the necessary functionality to parse the bundle's configuration files and internal structure, such that it can set up the classpath and/or module path. Furthermore, the Bundle Launcher adds support for special command-line arguments, such as running the bundled application inside a container or attaching a Native Image agent.

To make this feature available for all native image bundles, the Bundle Launcher package is injected into every bundle. To make the bundled application executable, we first need to make the bundle file itself executable and then launch the bundled application from there. Therefore, the Bundle Launcher's main class is set as the value of the `Main-Class` attribute in the bundle's manifest file, thus the bundle file can be executed with any JVM as a JAR file with the Bundle Launcher as its main class.

3.4.1 Bundle Launcher Package

The Bundle Launcher consists of its core part, the `BundleLauncher` class, which we set as the main class in the bundle's manifest file for every native image bundle created.

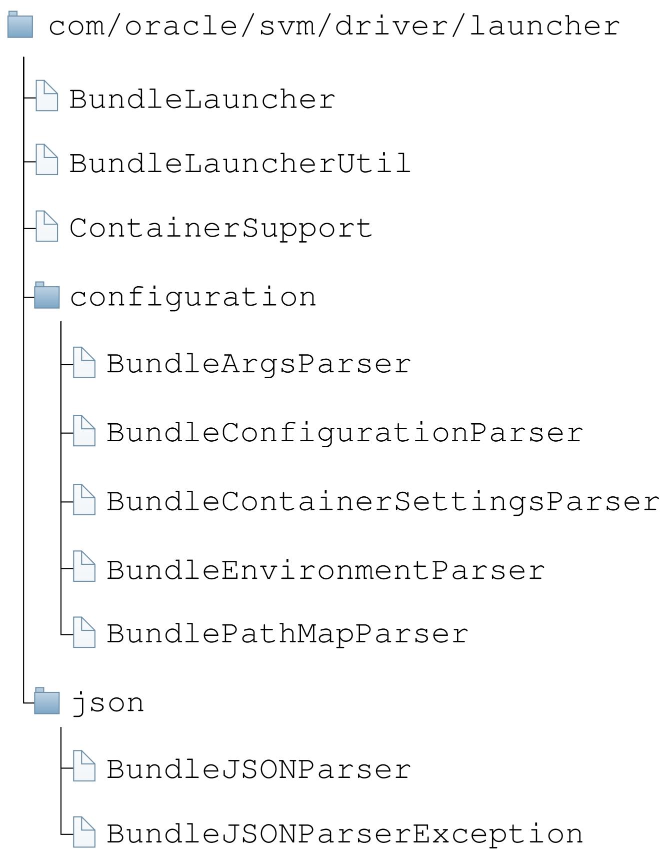

Figure 3.2: Bundle Launcher package structure

The rest of the package is structured according to Figure 3.2. The Bundle Launcher also overlaps with the existing `BundleSupport` class and Native Image Bundles functionality on many occasions. Therefore, common functionality is extracted into the Bundle Launcher to avoid code duplication and still make the Bundle Launcher work stand-alone.

The Bundle Launcher supports virtualized execution of bundled applications, therefore, core additions created for the first part of this thesis are moved into the `ContainerSupport` class. It stores all virtualization information such as the used virtualization tool and version as well as the used Dockerfile and functionality for setting up, building, and running container images.

Although most code for the Native Image Bundles feature implementation is located in `BundleSupport`, this class also makes use of some helper classes from the GraalVM project, such as `JSONParser` and `JSONParserExcpetion`. Therefore, the core parts of those classes need to be extracted and added to the Bundle Launcher, and all references to these helper classes in `BundleSupport` are moved to the newly created classes. The `BundleLauncherUtil` class contains single helper functions from multiple other packages in the project that are required for the `BundleLauncher`.

Each native image bundle contains essential information in configuration files in its stage directory. The most important one for executing a bundled application is the newly introduced *run.json*, which contains all command-line arguments for the bundled application that were passed to the Native Image tool during bundle creation. To parse the JSON files, we make use of the reused `JSONParser`. The raw JSON information gets parsed by one of the bundle configuration parsers, which are all contained within the Bundle Launcher. In total there are 4 implementations of the abstract

`BundleConfigurationParser`:

- `BundleArgsParser`: for *build.json* and *run.json*, which both store command-line arguments
- `BundleContainerSettingsParser`: for *container.json*, which stores virtualization information in named fields
- `BundleEnvironmentParser`: for *environment.json*, which stores environment variables in key-value pairs
- `BundlePathMapParser`: for *path_substitutions.json* and *path_canonicalization.json*, which both store pairs of paths

3.4.2 Inject Bundle Launcher into Native Image Bundles

For executing a native image bundle, we need to make sure that the Bundle Launcher is available in the bundle after creating it. Therefore, we inject the compiled Bundle Launcher into the bundles' root directory. First, the Bundle Launcher is added to the resources when building the Native Image tool itself. Thus, we have the package available when we use the Native Image tool. Then, the package can be fetched from the resources as seen in Listing 3.8 and copied into the bundle.

```java
        Path bundleLauncherFile = Paths.get("/").resolve(BundleLauncher.class.getName().replace(".", "/") + ".class");
        try (FileSystem fs = FileSystems.newFileSystem(BundleSupport.class.getResource(bundleLauncherFile.toString()).toURI(), new HashMap<>());
                        Stream<Path> walk = Files.walk(fs.getPath(bundleLauncherFile.getParent().toString()))) {
            walk.filter(Predicate.not(Files::isDirectory))
                            .map(Path::toString)
                            .forEach(sourcePath -> {
                                Path target = rootDir.resolve(Paths.get("/").relativize(Paths.get(sourcePath)));
                                try (InputStream source = BundleSupport.class.getResourceAsStream(sourcePath)) {
                                    Path bundleFileParent = target.getParent();
                                    if (bundleFileParent != null) {
                                        Files.createDirectories(bundleFileParent);
                                    }
                                    Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);
                                } catch (Exception e) {
                                    throw NativeImage.showError("Failed to write bundle-file " + target, e);
                                }
                            });
        } catch (Exception e) {
            throw NativeImage.showError("Failed to read bundle launcher resources '" + bundleLauncherFile.getParent() + "'", e);
        }
```

Listing 3.8: Fetch the Bundle Launcher from resources and copy it into a native image bundle

3.4.3 Execute Native Image Bundles

We want to be able to execute the application in a native image bundle with any JVM. Therefore, we first have to execute the *.nib* file as if it were a regular JAR file. Listing 3.9 shows the manifest file of an executable bundle, where we added the Bundle Launcher as the `Main-Class` argument, which will be executed as the bundle's main class if we execute it as a JAR file.

```
Manifest-Version: 1.0
Main-Class: com.oracle.svm.driver.launcher.BundleLauncher
```

Listing 3.9: Manifest file for executable native image bundles

The Bundle Launcher main class then contains a main method that extracts the bundle to a temporary location and parses any additional command-line arguments. Then the launcher creates a `ProcessBuilder` with a new command line for executing the bundled application and executes it. After the execution of the bundled application is finished, we will delete the extracted files to avoid any side effects.

The extracted files and additional command-line arguments are used for setting up the command line that executes the bundled application. However, we only need the following parts of the bundle:

- **run.json**: Contains all the arguments that were parsed during bundle creation, that are necessary for running the application. Therefore, we added a new flag to the existing `APIOption` annotation, which marks arguments as needed for running a bundled application. As soon as we register one of the marked options in `NativeImage` we collect them and pass them to Native Image Bundles later. Furthermore, it contains the argument that specifies what to execute. Either a main class, a JAR file, a main module, or a module with a main class.

- **Classpath and Module Path**: For executing a bundled application, we need to recreate the classpath and/or module path used to create the native image bundle. The classpath and module path files and directories are contained in the bundle directories *input/classes/cp* and *input/classes/p* respectively. Thus, we can walk through both directories and add all JAR files and subdirectories to the classpath and module path arguments for executing the bundled application accordingly.

- **environment.json**: All environment variables contained in this file are added to the environment of the `ProcessBuilder` that executes the bundled application.

- **Dockerfile & container.json**: These files are used to build the container image and set up the run command if the bundled application is executed in a container.

Additional command-line arguments are either used by the Bundle Launcher or, if they are not defined as arguments for the Bundle Launcher, passed on to the bundled application.

Furthermore, the Bundle Launcher supports additional command-line options for executing the bundled application with a Native Image agent attached, as well as executing the bundled application inside a container.

- **Execute with Native Image agent**: With this option, we add a command-line argument to the bundled application execution that attaches a Native Image agent to the execution. Additionally, we can add an option to automatically update the bundle file and store the agent's output in the bundles' classpath files. However, this requires the used JDK for executing the bundle to be a GraalVM JDK, with native-image installed and support for Native Image Bundles. After the execution of the bundled application is finished, the Bundle Launcher will create a second command that makes use of the Native Image tool and both bundle arguments, *--bundle-create* and *--bundle-apply*, to create a new bundle from the existing one.

- **Virtualized execution**: First, we check if *container.json* exists and contains any information on virtualization that was captured when creating the bundle. Then, we build a container image or make use of one that already exists on the system and conforms to the Dockerfile in the bundle. Finally, the container image is used to run the bundled application. Therefore, we create a run command for the container, that also contains the command line for the bundled application.

However, we may also bundle a native image that was built as a shared library, which is not executable by itself. Consequently, we also have to check if the bundle can be executed in the first place. For this, we use the *run.json* file, which is only created and inserted into the bundle, if the bundle contains an executable application. Thus, if we don't find a *run.json* file in the bundle, we know that this bundle is not executable, and we can abort the execution (see Listing 3.10).

```
        if (!Files.exists(stageDir.resolve("run.json"))) {
            showMessage(BUNDLE_INFO_MESSAGE_PREFIX + "Bundle " +
               bundleFilePath + " is not executable!");
            System.exit(1);
        }
```

Listing 3.10: Abort execution of shared library bundles

Chapter 4

Limitations

The extensions to Native Image Bundles rely on some restrictions we applied to the used environment. More specifically, we restrict the use of virtualized Native Image Builds and bundle executions by operating systems and supported virtualization tools. In Section 4.1, we will discuss the limitations imposed on the virtualization tools that can be used for virtualized Native Image Bundles Builds. Section 4.2 covers the restriction of supported operating systems for virtualized Native Image Bundles Builds to just Linux.

4.1 Virtualization Tool

The virtualization tools that are supported for the virtualization feature in Native Image Bundles are Podman and Docker. Both work with the same format for Dockerfiles, which means that we only have to store one Dockerfile in a bundle and still be able to use any of the two tools. Furthermore, Podman and Docker feature an almost identical command-line interface. Both tools can handle the same arguments for checking for container images, building container images, and running containers, which comprise the required functionality for the virtualization feature of Native Image Bundles.

With the restriction applied to virtualization tools, we aim to simplify the creation of command lines that make use of the tools. With both tools having almost identical command-line interfaces for our use cases, we simply create one command line and use it with the selected virtualization tool. To support other tools with different command-line interfaces, we would need to add special treatment for each kind of command-line interface. This overcomplicates the implementations with almost no benefits, as the two available tools are already both capable of building and running containers. However, the main difference between the two supported tools is that Podman is daemonless and does not require root privileges without any modification.

Furthermore, the virtualization tool Docker is restricted to only the rootless variant. This allows us to use Docker without root privileges, however, it requires a separate setup after installing Docker. To switch into the rootless mode, we need to install the *rootless* context with the *rootless setuptool* [3] (see Section 5.1) which comes shipped with Docker. After switching to the rootless context, we can execute Docker commands without root privileges.

4.2 Operating System

Docker as well as Podman, and many other virtualization tools are native Linux tools, and they rely on Linux-specific kernel features for working with containers. Although it is possible to set up a Virtual Machine (VM) that is capable of running virtualization tools on other operating systems, this would result in an overhead. Setting up and running a VM and then installing a supported virtualization tool to build a native image inside a container would not be feasible to implement for the Native Image tool.

Therefore, the virtualization feature for creating native images as well as for executing native image bundles is restricted to Linux only. If the *container* option is used with any other operating system, the Native Image tool reports a warning to the user, and virtualization is skipped. The Native Image tool then operates on the host system as if the *container* option was omitted. We decided on this behavior, as the main goal of a user most likely is to build a native image. Furthermore, this allows non-Linux users to create a bundle that contains configuration files for the virtualization feature, such as a custom Dockerfile or a virtualization tool that should be used for applying this bundle.

Chapter 5

Usage and Evaluation

Both features introduced in this thesis can be used with a command-line interface. However, the interface for the virtualized Native Image Build makes use of already existing options of the Native Image tool and serves as an extension to them. The command-line interface for executing native image bundles on the other hand can only be used if a native image bundle is executed on JVM. Therefore, all arguments passed to the main class of the executed bundle are parsed by the Bundle Launcher. The Bundle Launcher then either consumes the arguments if they are defined or passes them on to the bundled application if not.

This chapter covers the usage of the new features added to Native Image Bundles and their evaluation with some real-world examples. First, Section 5.1 describes how to set up a host system for virtualized builds and how to use the updated command-line interface of the Native Image tool. Next, in Section 5.2 we introduce the command-line interface for executing native image bundles, which can be used to execute the bundled application. Then, in Section 5.3 we cover backward compatibility with the previous Native Image Bundles version and the implications of the new features. Finally, the tested real-world examples and the evaluation of the new features are described in Section 5.4.

5.1 Virtualized Image Build

Virtualized image builds can be triggered via extensions of the native-image command-line interface. It can be used with both Native Image Bundles options *--bundle-create* and *--bundle-apply*. As described in Section 4.1, this extension only supports Podman and rootless Docker. Therefore, either Podman or Docker has to be installed on the system. However, Docker is additionally required to be in rootless mode, which can be achieved by running the *rootless setup tool* provided by Docker [3]. After switching to rootless mode (Docker runs in the "rootless" context), Docker may be used for virtualized builds.

```
$ docker context show
> default

$ dockerd-rootless-setuptool.sh install
> [INFO] Creating /home/dominik/.config/systemd/user/docker.service
> [INFO] starting systemd service docker.service
> + systemctl --user start docker.service
> + sleep 3
> + systemctl --user --no-pager --full status docker.service
> ...
> + DOCKER_HOST=unix:///run/user/1000/docker.sock /usr/bin/docker version
> ...
> + systemctl --user enable docker.service
> Created symlink /home/dominik/.config/systemd/user/default.target.wants/
    docker.service -> /home/dominik/.config/systemd/user/docker.service.
> [INFO] Installed docker.service successfully.
> [INFO] To control docker.service, run: `systemctl --user (start|stop|
    restart) docker.service`
> [INFO] To run docker.service on system startup, run: `sudo loginctl
    enable-linger dominik`

> [INFO] Creating CLI context "rootless"
> Successfully created context "rootless"
> [INFO] Using CLI context "rootless"
> Current context is now "rootless"

> [INFO] Make sure the following environment variable(s) are set (or add
    them to ~/.bashrc):
> export PATH=/usr/bin:$PATH

> [INFO] Some applications may require the following environment variable
    too:
> export DOCKER_HOST=unix:///run/user/1000/docker.sock

$ docker context show
> rootless
```

Listing 5.1: Switch to rootless Docker with dockerd-rootless-setuptool.sh

A virtualized build with the option *--bundle-create* builds a native image inside a container and stores it in the created native image bundle. With *--bundle-apply*, we can build a native image with Native Image Bundles in a virtualized environment. If the native image in a bundle was built inside a container, the bundle gets marked as a container build. Virtualized builds are sticky, which means that if a bundle is marked as a container build, applying the bundle with the option *--bundle-apply* will also result in a virtualized build, even if the option was not explicitly specified.

5.1.1 Command-Line Interface

```
$ native-image --bundle-create[=<bundle-name>][,dry-run][,container[=<
    container-tool>][,dockerfile=<dockerfile-path>]]
$ native-image --bundle-apply=<bundle-name>[,dry-run][,container[=<
    container-tool>][,dockerfile=<dockerfile-path>]]
```

Listing 5.2: Command-line interface extensions for Native Image Bundles

The extensions to the existing command-line interface as seen in Listing 5.2 are optional. The Native Image tool with bundle options can be used as before, without any restrictions. However, we can provide the following extended options, separated by a comma, with the updated command-line interface:

- **dry-run**: Runs the Native Image tool with the specified bundle option and also activates the *--dry-run* option. This means no native image is built, however, a native image bundle is created if *--bundle-create* was specified.

- **container**[=<**container-tool**>]: If a native image is built, first a container is built and run and the Native Image Build is executed inside the container. Allows to explicitly specify one of the two supported tools, Podman and Docker. Otherwise, it looks first for Podman, then for Docker on the `PATH` and uses the first one that is found.
- **dockerfile=**<**dockerfile-path**>: Allows to specify a path to a custom Dockerfile. It may be an absolute path or relative to the bundle location. This Dockerfile is then used to build the container image for virtualized builds. However, the host system's GraalVM and the bundle's input and output directories will still get mounted and used for building a native image. Furthermore, if a native image bundle is created, this Dockerfile will also be persisted there.

Combining *--bundle-apply* and *--bundle-create*, and therefore creating a bundle out of an existing one also supports these extensions. However, due to both bundle options being handled by the same object, extended options also apply globally to both bundle arguments. Therefore, the extended options need to be specified only once in any of the bundle arguments. However, the Native Image tool will fail if the *container* option is specified multiple times or the *dockerfile* option is specified without the *container* option.

5.1.2 Updating Virtualization Information in Native Image Bundles

A native image bundle, which was built inside a container also contains additional information about the used virtualization tool the container image, which was built. Therefore, we introduced a new JSON file for Native Image Bundles *container.json* (see Section 3.1).

The Dockerfile, which was used to build the container image, is copied into the bundle. Therefore, we can reuse the same Dockerfile with a bundle, or extract it from the bundle and tinker with it. If we, for example, want to make changes to an existing Dockerfile in a bundle with the name *mybundle.nib*, we can use the command in Listing 5.3. Therefore, we first extract the existing Dockerfile from the bundle and make changes to it. The command in Listing 5.3 then uses the existing bundle for creating the updated bundle. In this process, we can specify the new Dockerfile that should be used for the new bundle. We have to keep in mind that this would also trigger a Native Image Build, however, we can avoid this by adding the option *dry-run*.

```
$ native-image --bundle-apply=mybundle.nib,container,dockerfile=
    myDockerfile,dry-run --bundle-create
```

Listing 5.3: Update the bundled Dockerfile

5.2 Execute Native Image Bundles

Every native image bundle that contains an executable application can be executed with the Bundle Launcher. Therefore, the Bundle Launcher is injected into every created bundle. We also add a new file *run.json* in the bundle's stage directory. However, we are also able to build native image bundles for shared libraries. As shared libraries are not executable on their own, it would also not make sense to execute a bundled shared library. Thus, for shared libraries, we omit creating the *run.json* file as we can not execute a bundled shared library. If we execute the bundle, the Bundle Launcher can not find the *run.json* file. Therefore, we can stop trying to execute the bundle and output an error message to the user.

5.2.1 Command-Line Interface

```
$ java -jar myBundle.nib [options] [-- bundled-application-options]
```

Listing 5.4: Command-line interface extensions for the Bundle Launcher

For executing a bundle, the Bundle Launcher requires a JVM and can be called as seen in Listing 5.4. The Bundle Launcher receives all options and parses them. If an option is defined in the Bundle Launcher, it consumes the option and acts accordingly. However, if the Bundle Launcher parses an option that is not defined, it adds it to a list of arguments that are later passed on to the bundled application.

Options for the Bundle Launcher can be separated from options that are meant for the bundled application. Therefore, we have to add -- as an argument after the options that are meant for the Bundle Launcher. Every argument after -- will get added to the list of arguments that will get passed on to the bundled application later.

The Bundle Launcher supports the following four arguments that will be further explained in the following sections, some of which contain some additional optional parts:

- **--with-native-image-agent**[**,update-bundle**[=<**new-bundle-name**>]]
- **--container**[=<**container-tool**>][**,dockerfile**=<**dockerfile-path**>]
- **--verbose**
- **--help**

5.2.2 Attach Native Image Agent

With the new executable bundle feature, we can also execute the bundle with an attached Native Image agent (see Section 2.1.4). A Native Image agent is specified with the option *--agentlib* or *--agentpath* with some sort of output path passed to the option. Additionally, this also requires the Native Image agent to be available on the host system's GraalVM.

If we create a bundle with the Native Image tool without the option *dry-run*, a native image is created and put into an output directory. The same happens for building a native image from a bundle. To keep consistent with the current behavior, we decided to also move the Native Image agent's output into the bundle's output directory, as seen in Figure 5.1. This directory is set as the Native Image agent's *config-output-dir*, which means it is overwritten on successive bundle applications with the attached agent.

This Native Image agent output can then be used equally to the Native Image agent output produced by running it with the bundled application. Furthermore, we added the option to add the Native Image agent output to the bundle. Therefore, we have to add *update-bundle* to the command-line argument. However, compared to the rest of the Bundle Launcher, this requires the used JVM to be a GraalVM with a Native Image agent available. Accordingly, all actions that create or change bundles directly are performed by Native Image to make sure every bundle is created in precisely the same way. Without

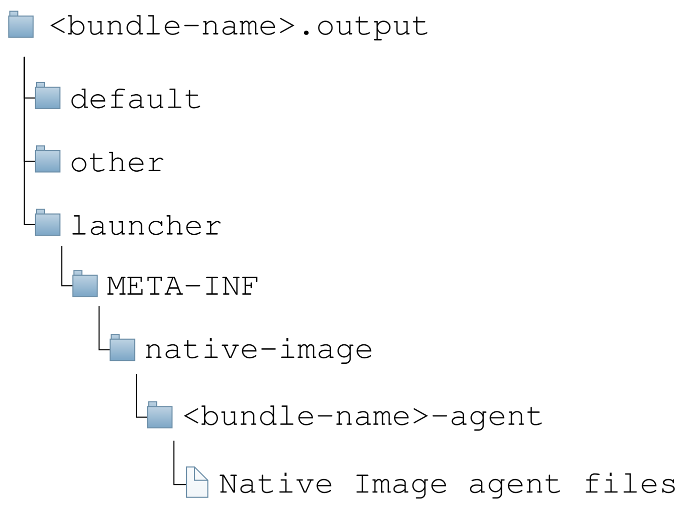

Figure 5.1: Native Image agent bundle output directory

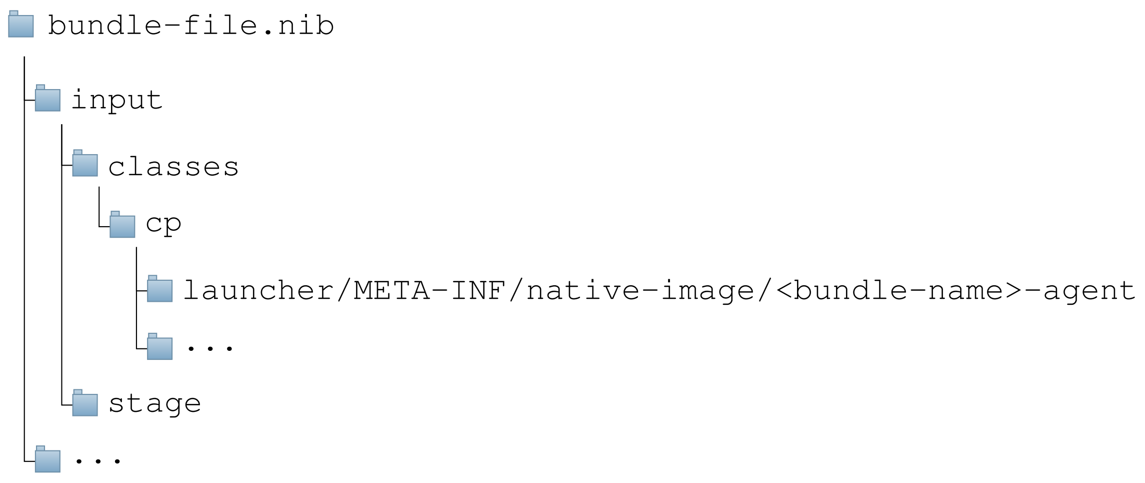

Figure 5.2: Updated Native Image Bundles format

any further option, *update-bundle* updates the existing bundle by adding the Native Image agent's output to the classpath (see Figure 5.2). However, we can also leave the existing bundle the same and create a new updated bundle by specifying a new bundle name that is different from the existing one.

5.2.3 Virtualized Bundle Execution

The *--container* option builds a container image and executes the bundled application inside the container instead of the host system. This addition works similarly to the virtualized Native Image Bundles builds and makes use of the same implementation. However, virtualized bundle executions run the bundled application in the container compared to the virtualized Native Image Builds, which just use the container for building a native image. Therefore, virtualized bundle execution does not make use of the Native Image tool, hence it does not require the JVM to be a GraalVM. Nevertheless, it still requires a supported virtualization tool to be available on the host system to build and run the container.

As described in Section 4.1, this feature only supports Podman and Docker as virtualization tools. If no virtualization tool is explicitly specified, we first check for Podman, then for Docker, and use the first available one. However, if we add *=<container-tool>* to the option, we can force the Bundle Launcher to use a specific tool. If the specified tool is not available, the bundle execution will fail.

Furthermore, executing a bundle virtualized makes use of the already existing Dockerfile inside a bundle. As described in Section 3.3.2, every bundle, no matter if it contains a native image that was built inside a container or not, does contain a Dockerfile. This Dockerfile either is the default Dockerfile from the virtualized Native Image build feature, or it was manually specified on bundle creation. However, for virtualized bundle executions it is still possible to override the bundled Dockerfile file, by specifying the path with *,dockerfile=<dockerfile-path>*. This ensures that the container image for the application execution is built with a custom Dockerfile. If the custom Dockerfile does not exist, the execution of the bundled application will fail.

5.2.4 Other Options

The remaining options for the Bundle Launcher are *--verbose* and *--help*.

The *verbose* option enables verbose output for the Bundle Launcher, which adds some logs during the execution of the Bundle Launcher. For example, printing the final command that was executed by the Bundle Launcher, including all arguments such as classpath and module path entries. However, this does not enable verbose output for the bundled application. Therefore, if the bundled application also has a *--verbose* option defined, and we want to get a verbose output from the bundled application, we have to specify *--verbose* after --. Otherwise, the Bundle Launcher identifies and consumes the *--verbose* option. Listing 5.5 depicts an example that uses verbose output for the Bundle Launcher and the bundled application. The first *--verbose* is consumed by the Bundle Launcher, and the second *--verbose* is passed on to the bundled application.

```
$ java -jar mybundle.nib --verbose -- --verbose
```

Listing 5.5: Use verbose output for the Bundle Launcher and the bundled application

The *help* option prints the help text, displayed in Listing 5.6, for a native image bundle without executing the bundled application. Similar to the *verbose* option, if the help text of the bundled application is also accessible with *--help*, we need to add -- before the *--help* option to pass the help option on to the bundled application.

```
This native image bundle can be used to launch the bundled application.

Usage: java -jar bundle-file [options] [bundle-application-options]

where options include:

    --with-native-image-agent[,update-bundle[=<new-bundle-name>]]
                runs the application with a native-image-agent attached
                'update-bundle' adds the agents output to the bundle-files
                    classpath.
                '=<new-bundle-name>' creates a new bundle with the agent
                    output instead.
                Note 'update-bundle' requires native-image to be installed

    --container[=<container-tool>][,dockerfile=<Dockerfile>]
                sets up a container image for execution and executes the
                    bundled application
                from inside that container. Requires podman or rootless docker to be installed.
                If available, 'podman' is preferred and rootless 'docker' is the fallback. Specifying
                one or the other as '=<container-tool>' forces the use of a specific tool.
                'dockerfile=<Dockerfile>': Use a user provided 'Dockerfile' instead of the Dockerfile
                bundled with the application

    --verbose   enable verbose output
    --help      print this help message
```

Listing 5.6: Bundle Launcher help text

5.3 Backward Compatibility

The additions to Native Image Bundles are introduced in the *GraalVM for Java 21* [24] release, with the previous version of Native Image Bundles as a foundation for the new features (see Section 2.1.3). Although we made changes to the format of a bundle file, the main internal structure of bundles is still the same. Therefore, bundles containing the new features can still be used to create native images with the previous version of the Native Image tool.

For older bundles, on the one hand, it is possible to build a native image in a container. On the other hand, it is not possible to execute bundles created with the previous version of the Native Image tool, as they do not contain the Bundle Launcher. However, we can update old existing bundles by using it to create a new one with the updated Native Image tool as shown in Listing 5.7. This creates a new bundle and injects the Bundle Launcher, a Dockerfile, and the required configuration files into the bundle. Additionally, the Bundle Launcher is added as `Main-Class` in the bundle's manifest file. Afterward, the bundle may be used to execute the bundled application.

```
$ native-image --bundle-apply=oldbundle.nib --bundle-create=newbundle.nib
```

Listing 5.7: Use an existing old bundle to create a bundle that supports the new features

5.4 Evaluation

The features added to Native Image Bundles do not aim for any improvements in the performance of Native Image Bundles creation, nor the file size of a bundle. Instead, these features focus on archiving, the determinism of building a native image from a bundle, and the usability of a bundle. However, we do create an overhead due to building a container image and adding some additional files, which results in slower Native Image Bundles build times and an increase in the file size of a native image bundle.

Build Performance

The time required for creating a native image bundle with the Native Image tool is only increased if we make use of the virtualization feature. Otherwise, the Native Image tool operates as usual with no added overhead at native image build time. However, using the virtualization feature adds the overhead of setting up a container for building the native image and creating the native image bundle. The time added to a Native Image Bundles Build is influenced by multiple factors:

- Dockerfile: Building the container image may work slower or faster, depending on the complexity and the number of build steps specified in the Dockerfile, that is used for the Native Image Bundles Build.
- Network: Fetching a base image for a container image or installing tools in the container image relies on the network. A slower network connection also slows down the container image build and therefore the Native Image Bundles Build.
- Available container images: If we can reuse the container image we want to use, the virtualization overhead for Native Image Bundles Builds is significantly reduced.

Furthermore, the added time to a Native Image Bundles Build is constant and is the same for any application. This means it does not affect the time for building the actual Native Image Build.

Native Image Bundles and Native Image File Sizes

Whenever a native image bundle is built with virtualization support enabled, we also need to store the virtual environment in the bundle. This is required to make use of the same environment for building a native image from a bundle. Therefore, we add a Dockerfile and some configuration files to each native image bundle. Furthermore, for executing a bundle, we also need to add the Bundle Launcher and some additional configuration files. In total, this adds up to around 100 KB. However, the overhead is independent of the underlying application for which the bundle is created. Therefore, for real-world applications, which result in bundles with much bigger file sizes than a few hundred KB, the overhead presented by the new features is neglectable.

Furthermore, the added features are only part of Native Image Bundles, this means, the resulting Native Image still does not have any additional inputs. However, there might be a difference in native image file sizes between virtualized and non-virtualized builds. We make use of a virtual environment that may have other versions of tools and libraries used by the installed Native Image tool, that get built into a native image.

Real-World Use Cases & Examples

A possible use case for the new features, and Native Image Bundles in general, is for archiving sources of a microservice that is built as a native image. For example, we can create a bundle for every update to the native image in production. Therefore, we can rebuild a native image at any point in time and be sure that the build will succeed deterministically. This is why we used some real-world examples for microservices to test the new features for correctness, such as *Spring PetClinic* [32] or some of the *micronaut guides* [17]. Therefore, we executed the microservice on JVM, as a native image and as a native image built in a virtualized environment as well as with Native Image Bundles on JVM. All runs showed the same behavior, we could also observe that the new features perform similarly to their existing counterparts (native image versus native image built in a virtualized environment and microservice on JVM versus Native Image Bundles on JVM).

Chapter 6

Conclusion

GraalVM Native Image is a tool that creates a native executable for a Java application. Furthermore, it provides a feature called Native Image Bundles. Native Image Bundles can be used to create a so-called bundle for a native image, which contains all inputs required to build that native image, which should result in a reproducible build of this native image at any point in time. However, during the Native Image Build process, we have full access to the host system, thus we might make use of a specific set-up environment or some resources on the host system that we are not aware of. Thus, we might fail to build the native image from a bundle on a different system or at a later point in time.

In this bachelor thesis, we implemented two features for Native Image Bundles, addressing the problem with the reproducibility of Native Image Builds and the usability of bundles:

- **Deterministic builds**: To control which files might end up in a bundle and which environment is used to build it, we made use of virtualization tools. These tools enable us to create a so-called container, which essentially is a fully controlled environment for creating a native image bundle. We are able to restrict access to the network and the host system's files. Furthermore, we can save the template for a container, such that we are able to reproduce and reuse the same container at any given time.

  However, we use the virtualization tools by interacting with their command line implementation. This makes using this feature simpler, as we just have to install the virtualization tool. The problem with this approach is, that virtualization tools come with different command-line interfaces. This is why we restricted the support for virtualization tools to just Docker and Podman as both feature an almost identical command line interface. Furthermore, the command-line implementations for the virtualization tools are only available for Linux in the way we need them, therefore we decided to also restrict this feature to only work with Linux.

- **Usability**: A bundle is a JAR file that contains everything to create a native image for an application. Therefore, we also have everything to execute the bundled application. This is why we added the Bundle Launcher, which we had to inject into each bundle. Therefore, we provided the Bundle Launcher as a resource to the Native Image tool, giving it access to class files for the Bundle Launcher during bundle creation. Using the Native Image tool's resources, we then can copy the Bundle Launcher into the bundle.

With the Bundle Launcher, we can execute the bundle as a JAR file on a JVM, and the Bundle Launcher loads the application and its dependencies from the bundle. We then create and run the command that executes the application. We also extended this Bundle Launcher to parse additional command-line arguments, such that we can execute the bundle in a container with the virtualization feature or attach a Native Image agent to the application execution. However, command-line arguments that are not defined in the Bundle Launcher are passed on to the executed application.

List of Figures

Listings

Bibliography

[1] "Assisted Configuration with Tracing Agent." [Online]. Available: https://docs.oracle.com/en/graalvm/enterprise/21/docs/reference-manual/native-image/Agent/#assisted-configuration-with-tracing-agent

[2] "Docker overview." [Online]. Available: https://docs.docker.com/get-started/overview/

[3] "Docker rootless setup tool." [Online]. Available: https://docs.docker.com/engine/security/rootless/

[4] "Docker *build* command line reference." [Online]. Available: https://docs.docker.com/engine/reference/commandline/build/

[5] "Docker *run* command line reference." [Online]. Available: https://docs.docker.com/engine/reference/commandline/run/

[6] "GraalVM." [Online]. Available: https://www.graalvm.org/

[7] "GraalVM Architecture Overview." [Online]. Available: https://www.graalvm.org/latest/docs/introduction/

[8] "GraalVM Community Edition Container Images." [Online]. Available: https://www.graalvm.org/latest/docs/getting-started/container-images/

[9] "GraalVM GitHub Container Registry." [Online]. Available: https://github.com/orgs/graalvm/packages

[10] "Graalvm github container repository." [Online]. Available: https://github.com/graalvm/container/tree/master

[11] "GraalVM Updater." [Online]. Available: https://www.graalvm.org/latest/reference-manual/graalvm-updater/

[12] "JAR File Specification." [Online]. Available: https://docs.oracle.com/en/java/javase/20/docs/specs/jar/jar.html

[13] "Java Command-Line Argument Files," publisher: October2021. [Online]. Available: https://docs.oracle.com/en/java/javase/11/tools/java.html#GUID-4856361B-8BFD-4964-AE84-121F5F6CF111

[14] "The Java® Virtual Machine Specification." [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se20/html/index.html

[15] "JSON." [Online]. Available: https://www.json.org/json-en.html

[16] "JVM Tool Interface 1.2.3." [Online]. Available: https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html

[17] "Micronaut Guides | Micronaut Guides | Micronaut Framework." [Online]. Available: https://guides.micronaut.io/latest/index.html

[18] "musl libc." [Online]. Available: https://musl.libc.org/

[19] "Native Image." [Online]. Available: https://www.graalvm.org/latest/reference-manual/native-image/

[20] "Native Image Build Options." [Online]. Available: https://www.graalvm.org/latest/reference-manual/native-image/overview/BuildOptions/

[21] "Native Image Bundles." [Online]. Available: https://www.graalvm.org/latest/reference-manual/native-image/overview/Bundles/

[22] "Native Image for graalvm 22." [Online]. Available: https://www.graalvm.org/22.3/reference-manual/native-image/

[23] "OpenJDK, HotSpot Runtime Overview." [Online]. Available: https://openjdk.org/groups/hotspot/docs/RuntimeOverview.html

[24] "Oracle GraalVM Release Calendar." [Online]. Available: https://docs.oracle.com/en/graalvm/enterprise/21/docs/release-calendar/#oracle-graalvm-release-calendar

[25] "Oracle linux 8, graalvm native-image dockerfile, java 20." [Online]. Available: https://github.com/graalvm/container/blob/master/rpm-compact/native-image-community/Dockerfile.ol8-java20

[26] "Oracle linux 8, graalvm native-image dockerfile, java 20 muslib." [Online]. Available: https://github.com/graalvm/container/blob/master/rpm-compact/native-image-community/Dockerfile.ol8-java20-muslib

[27] "Podman Introduction." [Online]. Available: https://docs.podman.io/en/latest/Introduction.html

[28] "Static and Mostly Static Images." [Online]. Available: https://www.graalvm.org/latest/reference-manual/native-image/StaticImages/

[29] "Substrate VM." [Online]. Available: https://docs.oracle.com/en/graalvm/enterprise/20/docs/reference-manual/native-image/SubstrateVM/#the-substrate-vm-project

[30] "Tracing Agent." [Online]. Available: https://www.graalvm.org/latest/reference-manual/native-image/metadata/AutomaticMetadataCollection/

[31] "Truffle Language Implementation Framework." [Online]. Available: https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/

[32] "Spring PetClinic Sample Application," Sep. 2023, original-date: 2013-01-09T09:05:18Z. [Online]. Available: https://github.com/spring-projects/spring-petclinic