Frankfurt University of Applied Sciences

Project Report

Porting OpenTwin to CMake

Author:
Johannes Krafft, Rohit Kumar

Supervisor:
Prof. Dr. Peter Thoma

Report of the progress towards a Linux Port of OpenTwin

in the

Software Engineering Project
Allgemeine Informatik Master / High Integrity Systems Master

December 2022

Contents

List of Figures

Listings

Abbreviations

DLL	Dynamic Link Library
VCS	Version Control System
CCI	Conan-Center-Index
IIS	Internet Information Services
TLS	Transport Layer Security
CLI	Command Line Interface
IPC	Inter-process Communication
ITK	Insight Toolkit

Chapter 1

Introduction

1.1 Current Build System

Currently, the OpenTwin Components are built with Visual Studio 15 2017. Each library comes with a Visual Studio (`.vcxproj`) that configures how the sources of the library are built. Sources and pre-built binaries of all third-party dependencies are located in a separate directory (`Third_Party_Libraries`) within the repository. In the Visual Studio project, environment variables are used to locate the third-party libraries. To build a library, the Visual Studio Project must be opened via a special Batch Script that sets all the environment variables that are needed to allow the compiler and linker to find the required header-files and Dynamic Link Libraries (DLLs) from other OpenTwin components and third-party libraries.

1.2 Problems with the Current Build System

While the current approach to build the project is fulfilling all requirements for a reliable and fast build, it also comes with some drawbacks that may result in problems when further scaling the project or when porting it to another operating system.

1.2.1 Implicit Dependency Tree

OpenTwin is designed to be a distributed systems made out of relatively small microservices. Each microservice is an expert system in a special field. Together they form a structure that is easy to extend by adding additional services. Common functionality

that has to be available in all services of the system, for example in order to allow a standardized communication protocol, are provided by internal libraries that each service links to. Not all services require the same extend of base functionality though, which is why there is multiple different libraries that a service can depend on. Additionally, services and internal libraries depend on third-party libraries to provide their functionality. The result is a quite complex dependency structure, as illustrated in Figure 1.1.

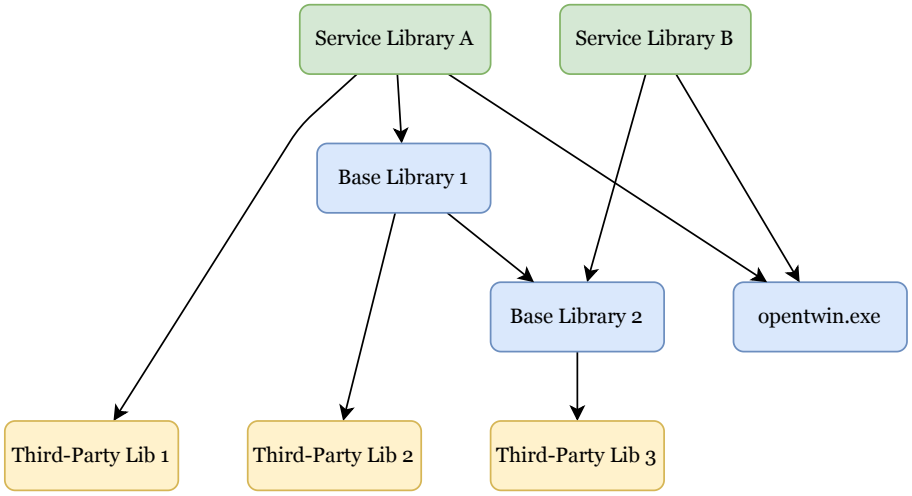

FIGURE 1.1: Illustration of a Possible Dependency Tree
Service libraries (green) can depend on internal system libraries or components (blue) and third-party libraries (yellow) in different configurations. This creates a complex dependency tree.

In the Visual Studio Project, dependencies are implicitly defined in the form of additional include directories and `.lib` files that must be linked to the resulting binary. But the explicit information that a library depends on another library is missing. This means that when building a service and one of the internal libraries that it depends on is not yet compiled, during the build process the linker will complain that the `.lib` file it needs is missing. It is unable to build the required library though, because it doesn't have the information to do so. A similar problem exists for third-party libraries: It is not explicitly stated in the Visual Studio project that a library depends on another third-party library. Instead, the required files are directly referenced, which can lead to confusing error messages when something goes wrong because the referenced files cannot be found.

A more abstract description of dependency relations would make the build process more flexible and would allow for better error handling.

1.2.2 Vendor Lock-in

The current project can only be built with Visual Studio on Windows. It is not possible to compile the codebase on another operating system where Visual Studio is not available. Besides that, all third-party libraries are only available as pre-built binaries for 64 Bit versions of Windows. They would need to be recompiled manually for any other system configuration.

A build configuration that works well for different operating systems, build systems and system architectures would be preferable.

1.2.3 Home-Brewed

The Batch Scripts used for editing, compiling and testing the system components are a custom solution that solves the problem well. But it requires explanation to anybody that is not yet familiar with the project setup because of its custom nature.

A solution that is just based on industry standard tooling could reduce the entrance barrier for new developers.

1.2.4 Large Repository Size

Because the source code repository of OpenTwin contains all of the third-party source code and the related pre-built binaries, the size of the repository is enormous. This takes a lot of space on the developers machine and leads to long waiting times during checkouts, commits or any other action on the Version Control System (VCS). Figure 1.2 shows that most of the disk space in the repository is used up by the third-party libraries.

A reduced repository size would improve the developer experience.

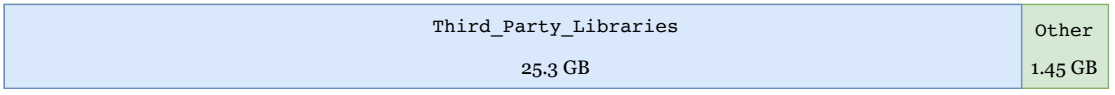

FIGURE 1.2: Current Repository Disk Space Usage:
The majority of the total repository size of $26.75GB$ is required for all the source code and pre-built binaries of the used third-party libraries.

1.3 Proposed Solution

To address the mentioned problems, a new build process is proposed:

1. The project should be configured with CMake, which allows platform and build system indenpendence. Details will be discussed in Section 1.4.

2. Third-party dependencies should be managed with Conan and should be distributed separately from the OpenTwin codebase with a self-hosted Artifactory. This enables easy and automated compilation of all third-party dependencies for any platform configuration and dramatically reduces the required disk space. What Conan is and how it can solve the given problems will be discussed in Section 1.5. Artifactory will be introduced in Section **??**.

3. For Windows builds, the used Visual Studio version should be upgraded to Visual Studio 17 2022. It comes with improved CMake support that allows a streamlined and familiar developer experience despite the radical build system refactoring.

1.4 CMake

CMake is a build configuration system. The difference to a typical build system is that CMake itself cannot build software. It configures other build systems to do so. This allows for a flexible and platform independent build processes. For example, CMake could be used to generate either a Visual Studio Project, a Makefile, a XCode project or Ninja buildfiles, depending on the platform that the code should be compiled on.

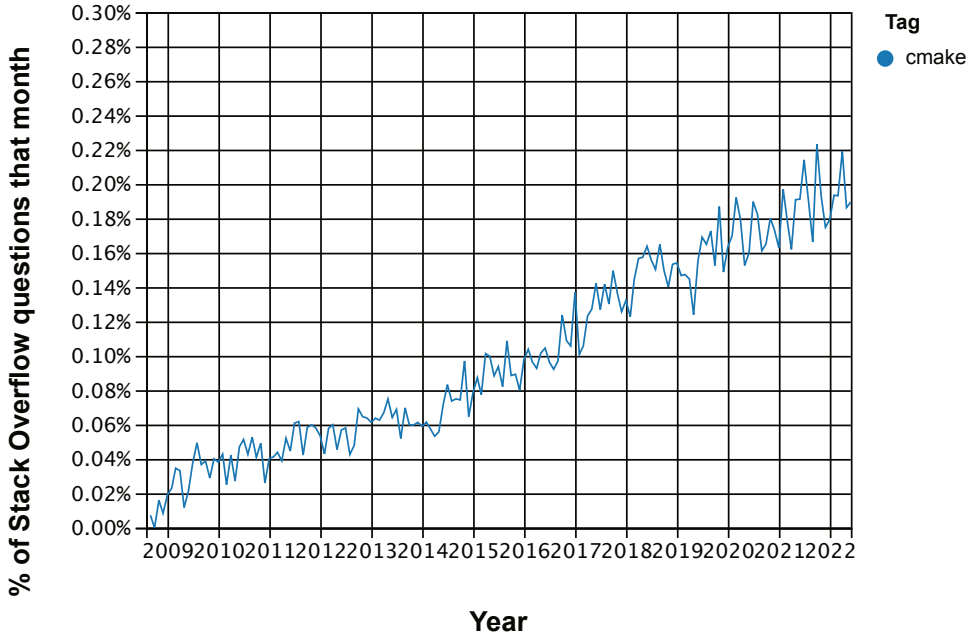

FIGURE 1.3: Percentage of monthly Stack Overflow questions regarding CMake: The percentage of questions is constantly rising since 2009, which may indicate an increase in CMake usage across the industry [1].

The development of CMake was started by Kitware in 1999 as a flexible tool to allow cross-platform compilation of their Insight Toolkit (ITK) library [2]. Nowadays, CMake is a popular tool used in a variety of large software projects. For example, most components of the KDE project [3] and huge libraries like Gmsh or CGAL are built with CMake [4]. According to *Stack Overflow Trends* the number of questions on the platform regarding CMake has been rising constantly since 2009 (see Fig. 1.3), which may indicate an increase in CMake usage across the industry.

Chapter 2 will discuss in more detail on how CMake was used in this project.

1.5 Conan & Artifactory

Conan is a dependency and package manager. It is open-source and free and works on all platforms. It can be integrated with build systems like CMake or Visual Studio (MSBuild). It is a decentralized package manager having a client-server architecture. Clients can download packages from server as well as host their own packages and binaries privately to server. The Jfrog Artifactory is the recommended Conan server to host own packages privately under your control. It is being used by hundreds of companies like Continental, Audi, Electrolux, Plex and Mercedes-Benz and many thousands of developers around the world [5]. Artifactory is used to optimize storage by making sure that any binaries and its metadata are only stored once on a file system. It will contain binaries for all the dependencies which can be used directly by downloading them and hence cut down the time to download sources and build them.

1.6 Porting Strategy

The porting process involves the following steps:

1. Reverse-engineer the overall dependency tree. How do all the internal components depend on each other?
2. Figure out all of the third-party dependencies and check if they are available on the Conan-Center-Index (CCI).
3. Port all the libraries that are not available on the CCI to Conan.
4. Design a common structure for how the CMake configuration should look like for each of the OpenTwin components.

5. Apply the concept to all of the libraries and executables in the repository.
6. Provide debug run configurations for all of the services in the system.

If all points in the roadmap could be finished successfully and what remains to be done will be discussed in the final outlook in Chapter 6.

Chapter 2

General CMake Structure

This chapter discusses the structure of the new CMake build configuration files. First, an overview over the important build configuration files in the repository is given. Then, the build configuration for a single library is presented in detail and the integration of Rust into CMake is explained. Finally, special files that are required for debugging and testing in Visual Studio are introduced.

2.1 Overview

In contrast to the previous Visual Studio Projects setup, all system components in the repository are now combined in one big `CMakeLists.txt` that includes all of the subfolders and therefore configures build targets of all available libraries and executables at a time (see Fig. 2.1). This comes with the advantage that CMake can handle internal dependencies and automatically build all components that are required for a specific target.

At the heart of the configuration is the root `CMakeLists.txt` file (1). It is the main entry-point for the project configuration and includes the `CMakeLists.txt` files in `Libraries` (4) and `Microservices` (5). It also tells CMake to search for third-party dependencies in the `ThirdPartyLibraries` folder, defines the option `BUILD_TESTING` that allows disabling the configuration of unit-test targets, configures some details about the install process, defines packaging information for *CPack* and generates the `.runsettings` file required for test execution in the Visual Studio Test Explorer (see Section 2.4.2).

The `CMakePresets.json` file (2) defines all parameters that need to be passed to the CMake configuration step. The CMake configuration process must always be started with one of the defined presets (`debug` or `release`) from this file, otherwise a successful

configuration is not guaranteed. In the preset file the build and install directories are configured, and the `PATH` is populated with all possible binary output directories for a successful test execution in Visual Studio.

Before the CMake configuration process is started, the third-party libraries are installed by Conan into the `ThirdPartyLibraries` folder (see Chapter 3). For each dependency a `Find<PackageName>.cmake` file is created by Conan (6). When a dependency is requested somewhere in the configuration process, these files will tell CMake where to find the related binaries and header files.

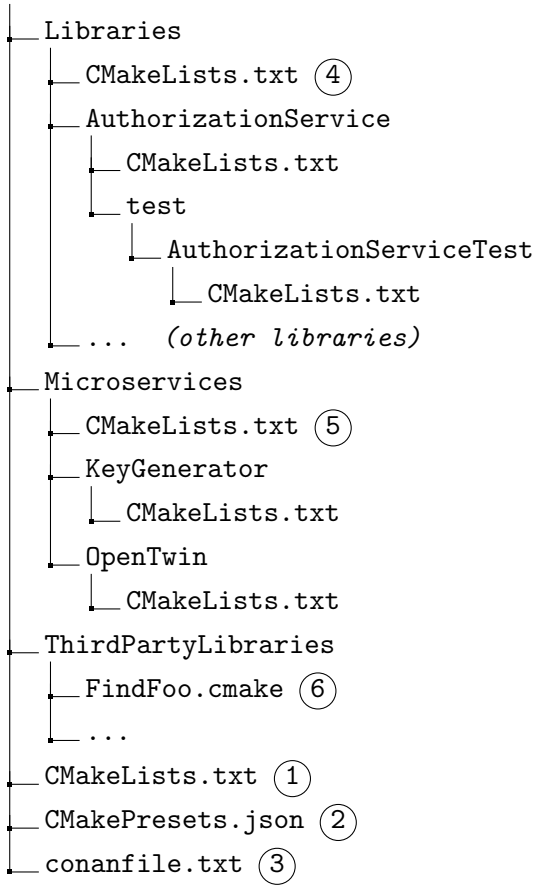

FIGURE 2.1: Overview over the CMake configuration file structure:
A root `CMakeLists.txt` (1) is including the subdirectories `Libraries` und `Microservices`, where again all subfolders with CMake configurations are included (4, 5).

2.2 Library Configuration

Most of the targets ported to CMake are located inside the `Libraries` folder. They all have a similar structure, that will be presented here briefly. Figure 2.2 gives an overview

on the targets that are defined for each library. Is uses the AuthorizationService as an example.

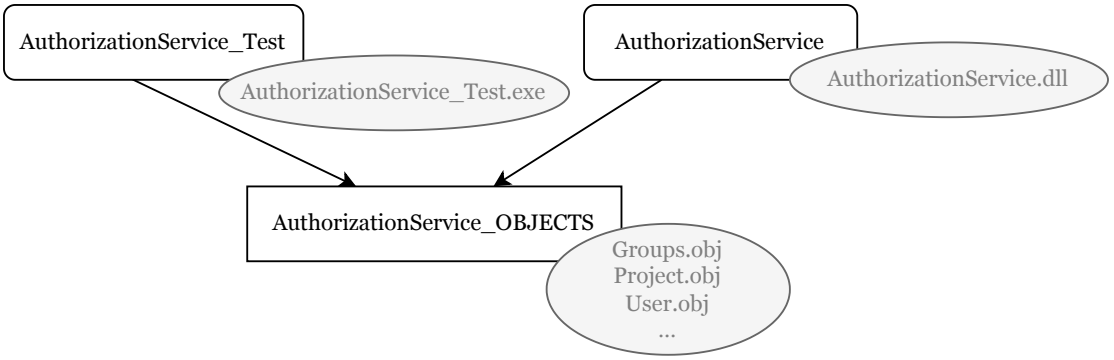

FIGURE 2.2: CMake target hierarchy in a typical OpenTwin library:
A `<LibraryName>_OBJECTS` target is responsible for creating the object files, which are then linked either by the final shared library or by the test executable.

At the base of each library an `OBJECT` target called `<LibraryName>_OBJECTS` is declared. It defines all of the source files and dependencies that are needed for compiling the library. When built, this target outputs an object file for each source file. The object files are not yet linked together to an executable or library in this step.

For assembling the final shared library, a target with just the library name is defined. It takes care of linking all the object files into one binary and (on Windows) exposing the public symbols in a `.lib` file.

When writing unit-tests for a library, it is often useful to have access to internal symbols that are not publicly exposed by the final shared library. This is why the test target named `<LibraryName>_Test` does not depend on the shared library but on the `<LibraryName>_OBJECTS` target instead. That way the test target can link the intermediate object files directly into the test executable, allowing the test code to access any internal symbol [6].

2.3 Rust Build Integration

The executable `open_twin.exe` that loads the C++ OpenTwin system components and takes care of the Inter-process Communication (IPC) interface over HTTP, is written in Rust. CMake doesn't have native support for building Rust code. That is why the CMake module *Corrosion* was used [7]. When called in `Microservices/OpenTwin/CMakeLists.txt`, the module automatically imports all build targets that are defined in the Rust Cargo file and exposes them as CMake custom targets. This is how targets defined in CMake can explicitly depend on the Cargo package that provides `open_twin.exe`. When a library

that needs the executable for debugging is built with CMake, the Rust build process is automatically initiated first, to ensure that the required executable is present.

2.4 Visual Studio specific files

Apart from the standard files that are related to CMake and Conan and thereby should be mostly platform independent, there is some files that are specific to Visual Studio development on Windows.

2.4.1 `launch.vs.json`

In CMake there is no way to define run configurations for the built executables. This is why in order to debug a CMake executable target in Visual Studio, execution instructions and runtime parameters must be configured in a `launch.vs.json` file [8]. Listing 2.1 shows an excerpt of the launch configurations for OpenTwin. It only shows the debug configuration for the LoggerService, but most other components of OpenTwin can be executed similarly.

Because the LoggerService is a library that needs to be loaded by `open_twin.exe`, in Line 10 the actual executable is referenced. Line 11-16 then sets the runtime parameters, including the DLL that should be loaded. Because the required binaries are located in different directories than the executable, in Line 7 the locations of the required dependencies are set on the `PATH`.

When configured properly, the Service can then be started from Visual Studio as seen in Figure 2.3.

```
1  {
2    "version": "0.2.1",
3    "configurations": [
4      {
5        "type": "dll",
6        "env": {
7          "PATH": "${cmake.binaryDir}\\Libraries\\OpenTwinCommunication;${
     workspaceRoot}\\ThirdPartyLibraries",
8          "RUST_BACKTRACE": "1"
9        },
10       "exe": "${cmake.binaryDir}\\Microservices\\OpenTwin\\open_twin.exe",
11       "args": [
12         "LoggerService.dll",
13         "unused",
14         "127.0.0.1:8096",
15         "unknown",
```

```
16          "unused"
17        ],
18        "project": "CMakeLists.txt",
19        "projectTarget": "LoggerService.dll (Libraries\\LoggerService\\
      LoggerService.dll)",
20        "name": "LoggerService.dll (Libraries\\LoggerService\\LoggerService.dll)"
21      }
22    ]
23  }
```

LISTING 2.1: Excerpt from `launch.vs.json`: Debug configuration for the LoggerService

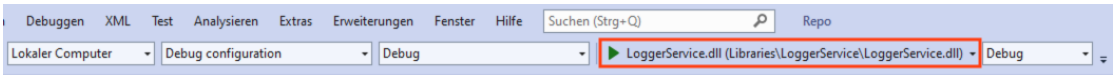

FIGURE 2.3: Running a Service in Debug Mode:
If `launch.vs.json` sets the right parameters for launching an OpenTwin component, it can be debugged from within Visual Studio.

2.4.2 `.runsettings`

Tests in Visual Studio are best executed with the Test-Explorer. When properly configured in CMake, all test suites in the repository are listed there automatically. But on Windows the test execution faces the same problem as the debugging. Because the required binaries are scattered across multiple folders in the build directory, the `PATH` must contain all of the potential directories where a required DLL could be located. Listing 2.2 shows how the list of absolute paths is declared in the `.runsettings` file.

Because the paths in the file have to be absolute, the file is not checked in to the repository directly, but instead is created during the CMake configuration step automatically. The `PATH` in the file will be equal to the one defined in the selected configuration preset defined in `CMakePresets.json`.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <RunSettings>
3   <RunConfiguration>
4     <EnvironmentVariables>
5       <PATH>C:/OpenTwin/Repo/ThirdPartyLibraries;C:/OpenTwin/Repo/build/Debug/
      Libraries/AuthorizationService;C:/OpenTwin/Repo/build/Debug/Libraries/
      CADModelEntities;C:/OpenTwin/Repo/build/Debug/Libraries/CartesianMeshService;
      ...</PATH>
6     </EnvironmentVariables>
7   </RunConfiguration>
8 </RunSettings>
```

LISTING 2.2: Example of a `.runsettings` file for test execution in the Test Explorer

Chapter 3

Dependency Management

This chapter discuss about Conan, the dependency management which we have used in our project. Initially an overview about the Conan is given. Then, the structure of the Conan configuration file is discussed and how we can add new dependencies when required. Finally, need for porting third party libraries to Conan is discussed and how to cache & publish the pre built binaries to Artifactory is explained.

3.1 Overview

As we have already discussed in the section 1.2 about the issues we faced with the current setup of the project and how dependencies are managed. To overcome this issues, we have used Conan as a dependency management for our project. We will have a receipe file (Conanfile.txt) inside the project folder and this file will be used to download the required dependencies from CCI.

3.2 Conan File

We need to create a Conanfile which will be used to download the required dependencies. The Conanfile is created under the root directory of the project with file name `Conanfile.txt`. The Conanfile is mainly consisting of four parts.

- `[requires]` : This section contains list of requirements with its full reference.
- `[generators]` : Generators are specific components that generate the dependency data calculated by Conan in a format suitable for a build system.

- `[options]` : Options section contains list of all the options which are scoped for each package like `<package_name>:<option> = <Value>`.
- `[imports]` : The shared libraries must be put in a folder where the linker or the OS runtime can find them. In import section, we list files which need to be imported to a local directory.

```
[requires]
libcurl/7.80.0
rapidjson/cci.20211112
mongo-cxx-driver/3.6.6
openssl/1.1.1n
zlib/1.2.11
qt/5.15.2
base64/1.0.0@opentwin/thirdparty
TabToolbar/1.0.1@opentwin/thirdparty

[generators]
cmake_find_package

[options]
libcurl:with_ftp=False
libcurl:with_imap=False
libcurl:with_ldap=False
openssl:shared=True
mongo-cxx-driver:shared=True
mongo-c-driver:with_icu=False
zlib:shared=True
qt:shared=True
qt:qtwebsockets=True

[imports]
bin, *.dll -> .
lib, *.dll -> .
```

Listing 3.1: Excerpt from `Conanfile.txt`

3.3 Porting Libraries to Conan

Not all libraries that OpenTwin depends on are available on CCI. Those dependencies need to be ported to Conan first, before they can be specified as requirements in the conanfile.

In order to be able to use, build and publish a dependency with Conan, a Python file named `conanfile.py` must be provided. It specifies what targets the library defines, how they can be built and what artifacts (source files, header files or DLLs) need to be distributed as Conan package.

There is two ways to create a package. One way is to distribute the `conanfile.py` file separately from the sources of the library [9]. In the file, the remote location of the sources can then be specified. When initializing a build of the library, the sources would then first be fetched from the remote source before starting the compilation. Another way is to distribute the `conanfile.py` together with the sources of the library [10]. In that case the conanfile does not need to fetch the sources from another remote first, because they are located in the same directory/repository. This is what was done for the dependencies of OpenTwin.

All third-party dependencies of OpenTwin that are not available on CCI are source-controlled in the Github repository `pth68/OpenTwinThirdParty`.

3.4 Caching Pre-Built Binaries

When pulling dependencies directly from the CCI, the installation of dependencies is usually a time consuming process. First the sourcecode is downloaded from the CCI and then the binaries are built for the required system configuration. Depending on the number of dependencies required for the project, this can take lot of time. Also, it takes up lot of storage space for caching the downloaded source files and intermediate build results.

In order to reduce installation time, the binaries are just built once and then cached in our own Artifactory instance. That way new developers can just pull the pre-built binaries without having to compile everything form scratch. Caching the binaries in Artifactory will also reduce the redundant local disk space taken by the installed dependencies, because just the required header files and libraries are downloaded.

The workflow for providing the cached binaries based on the recipes from the CCI is visualized in Figure 3.1. First, the Conan recipe and the sources for the library need to be downloaded. After that the required binaries can be built and pushed to our own Artifactory instance. Developers can then just directly pull the Conan recipe and the related pre-built binaries from `artifactory.opentwin.net`.

The commands that need to be executed to achieve this setup are listed in Listing 3.2. First, the OpenTwin Artifactory needs to be added as a Conan remote (Line 1). In order to be able to publish artifacts, authentication is required. This is why in Line 2 the user with username `foo` is added. The flag `-p` will interactively ask for the password. After that, in Line 3 a library is pulled from the CCI and compiled for the specified profile `default`. If no conan profile exists yet, it can be created like described in Chapter 5.2.2. The resulting package in the local cache now also contains the binaries. The recipe

together with the sources and the pre-built binaries is then published to the Artifactory in Line 4.

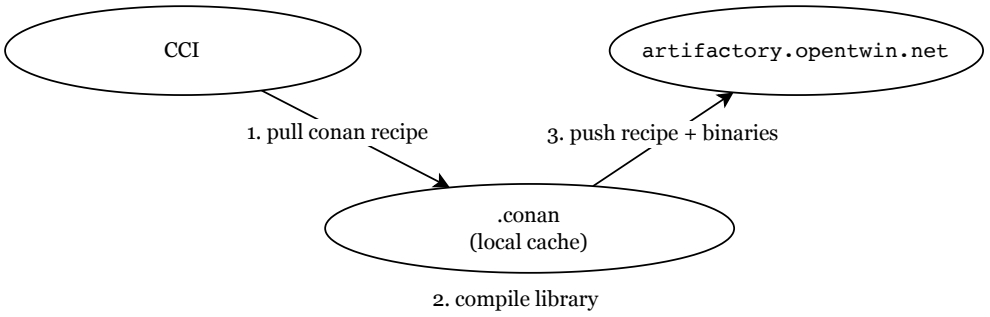

FIGURE 3.1: Workflow for Caching Pre-Built Binaries with Conan:
After fetching the Conan recipe from the CCI (1), the library is built for the required system configuration (2). The resulting binaries are then pushed to our own Artifactory instance (3).

```
1 conan remote add opentwin
      https://artifactory.opentwin.net/artifactory/api/conan/opentwin
2 conan user foo -r opentwin -p
3 conan install foo/1.0.0@_/_ --install-folder ThirdPartyLibraries --build missing
      --profile default --remote conancenter
4 conan upload foo/1.0.0 --all --remote=opentwin
```

LISTING 3.2: Building and publishing library binaries to `artifactory.opentwin.net`

Chapter 4

Artifactory Setup

This chapter documents in detail how the Artifactory server was installed and configured on the `artifactory.opentwin.net` Windows Server. The documentation is aimed at server administrators that want to understand the current server setup or that want to reproduce the install on another machine. Section 4.3 gives an outlook on the maintenance tasks that need to be performed on a regular basis to ensure a safe operation.

4.1 Installation

The installation process consists of three main parts: Installing the Artifactory software and registering it as a service, configuring a reverse proxy to expose the service to the internet and finally enabling TLS for providing encrypted communication.

4.1.1 Installing Artifactory

The Windows installer for the *Artifactory Community Edition for C/C++* was downloaded at `conan.io/downloads`. The downloaded `zip` file was extracted to `C:\jfrog\artifactory` on the server. It is important to choose a short base path that does not contain whitespaces, otherwise the install will not work!

For installing the application as a service, the commands listed in Listing 4.1 were executed as administrator. The Artifactory server was now availabe at `localhost:8082`.

Detailed install instructions for Artifactory are available on `jfrog.com` [11].

```
1 set JFROG_HOME=C:\jfrog\artifactory
2 cd C:\jfrog\artifactory\app\bin
```

```
InstallService.bat
sc start artifactory
```

Listing 4.1: Commands that were executed to install Artifactory as a service

4.1.2 Configuring Reverse Proxy

In the next step, a reverse proxy had to be configured to proxy any request to the domain `artifactory.opentwin.net` to the local port `8082` on the server. For that, a DNS A-record had to be added to the `opentwin.net` domain, configuring that the subdomain `artifactory` forwards to the IP-address of the provided Windows server. On the server, the plugins *URL Rewrite 2.0* and *Application Request Routing 3.0* had to be installed via the Microsoft Web Platform Installer [12]. Once installed, in the Internet Information Services (IIS) the default website was renamed to `Artifactory`. Then the configuration in Listing 4.2 was applied to `C:\inetpub\wwwroot\web.config` by following the setup instructions in [12, 13]. Apart from the reverse proxy, a content length limit of approximately 100 MB was set to allow the upload of large Conan artifacts [14].

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <rewrite>
            <outboundRules>
                <preConditions>
                    <preCondition name="ResponseIsHtml1">
                        <add input="{RESPONSE_CONTENT_TYPE}"
    pattern="^text/html" />
                    </preCondition>
                </preConditions>
            </outboundRules>
            <rules>
                <clear />
                <rule name="ReverseProxyInboundRule1" stopProcessing="true">
                    <match url="(.*)" />
                    <conditions logicalGrouping="MatchAll"
    trackAllCaptures="false" />
                    <action type="Rewrite" url="http://localhost:8082/{R:1}" />
                </rule>
            </rules>
        </rewrite>
        <security>
            <requestFiltering>
                <requestLimits maxAllowedContentLength="100000000" />
            </requestFiltering>
        </security>
    </system.webServer>
</configuration>
```

LISTING 4.2: `web.config` configuration file for the Artifactory reverse proxy

After configuring the reverse proxy, in the global server configuration the proxy had to be enabled by following the steps in Figure 4.1 [13].

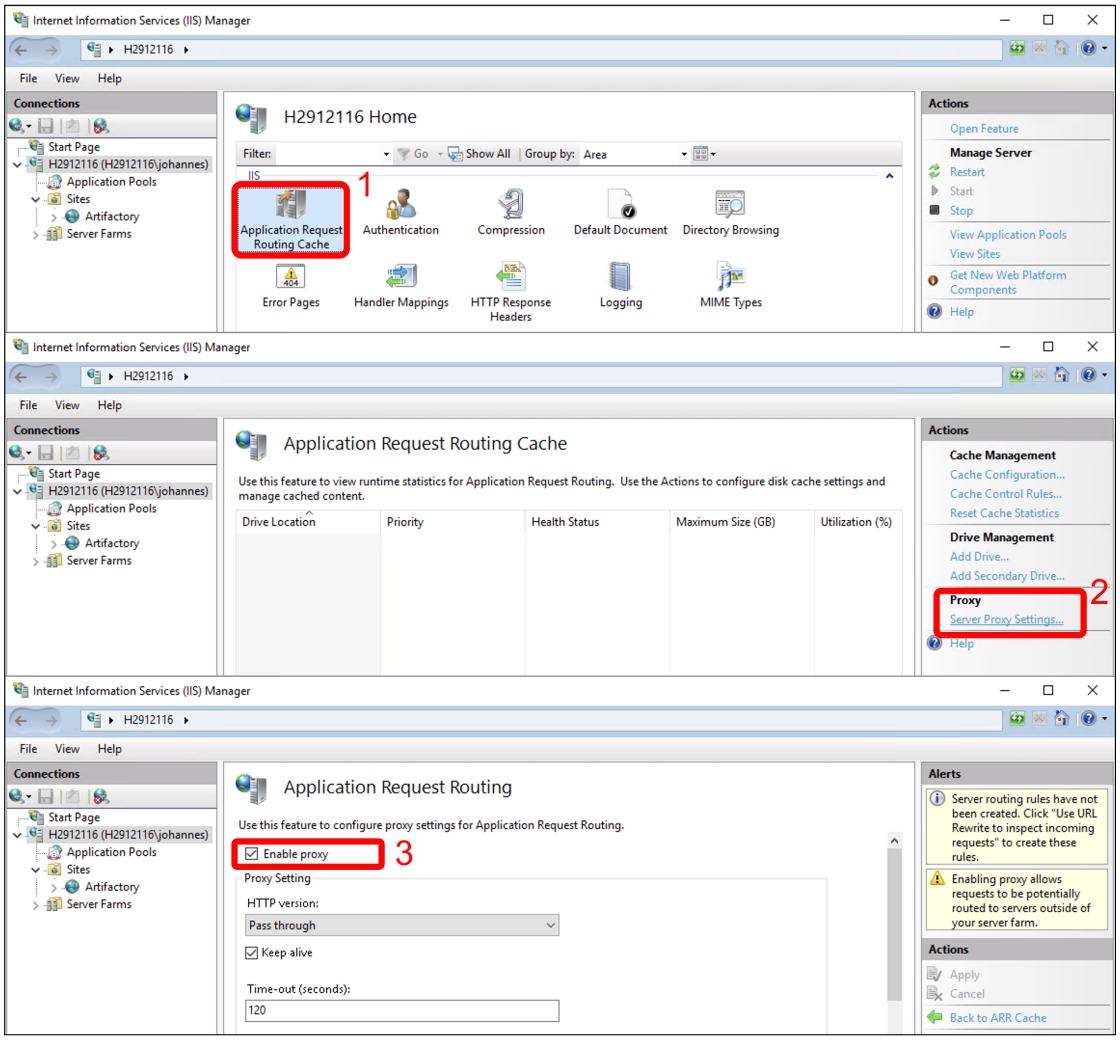

FIGURE 4.1: Enabling the server proxy:
After configuring the reverse proxy for the Artifactory, the global server proxy was enabled.

The server was now available from `artifactory.opentwin.net`, but without providing secure communication via TLS.

4.1.3 Enabling TLS

The software *Win-ACME* was used for configuring the *Let's encrypt* certificate for a secure communication with the server [15, 16]. For that, first the hostname had to be

configured in the IIS as illustrated in Figure 4.2. Only the hostname for port 80 had to be configured. The mapping for port 443 and all other required configuration for providing encrypted communication via `https` was added automatically by *Win-ACME* in the following steps.

Then, the correct version of *Win-ACME* (`win-acme.v2.x.x.xx.x64.trimmed.zip`) was downloaded from GitHub [17] and extracted to `C:\Program Files/win-acme`.

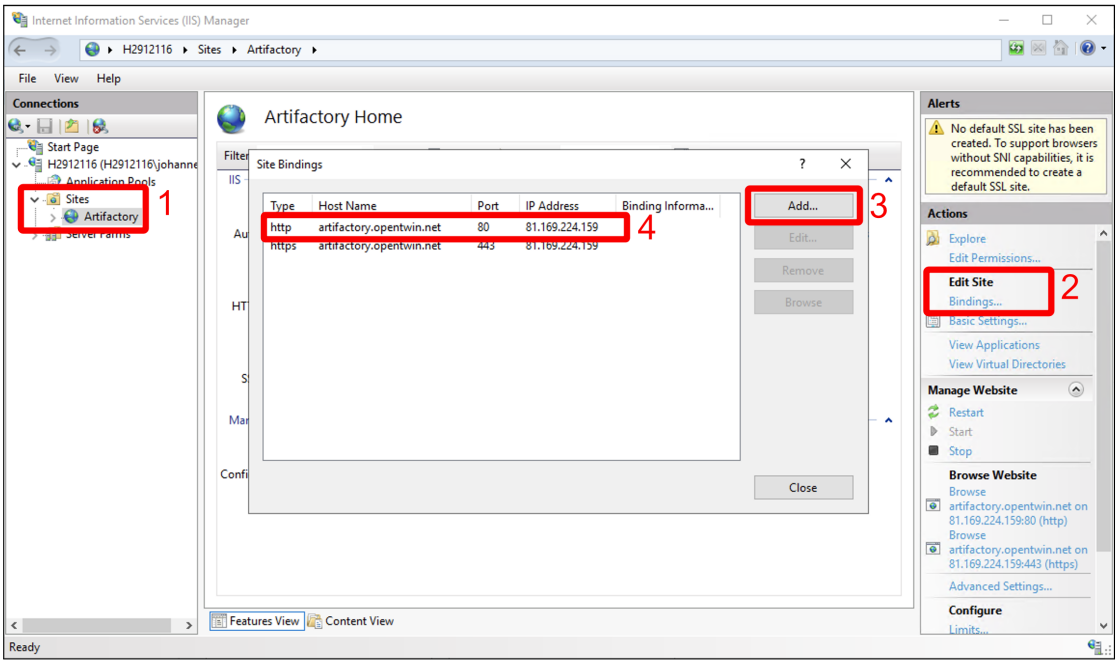

FIGURE 4.2: Configuring the Site hostname:
In the IIS a hostname on port 80 had to be configured for the Artifactory site as a prerequisite for configuring the TLS certificate.

The executable `wacs.exe` was exectued with administrator privileges. The warning from Windows Defender was ignored.

In the Command Line Interface (CLI) a new certificate for `artifactory.opentwin.net` was issued. The E-Mail address `peter.thoma@fb2.fra-uas.de` was given as contact. When the certificate is going to expire, a reminder will be sent to this address.

After finishing the wizard in the `wacs.exe` CLI, the Artifactory could now be accessed via `https` on port 443 from the browser. But it was still also available on port 80 without any transport security. To disable that, a port redirect from port 80 to port 443 had to be configured by adding the rule listed in Listing 4.3 to the sites `web.config` like described in [18].

```
1 <rule name="Http Redirect" stopProcessing="true">
2     <match url="(.*)" />
3     <conditions logicalGrouping="MatchAll" trackAllCaptures="false">
```

```
        <add input="{HTTPS}" pattern="^OFF$" />
    </conditions>
    <action type="Redirect" url="https://{HTTP_HOST}{REQUEST_URI}"
    appendQueryString="false" />
</rule>
```

LISTING 4.3: `http` to `https` redirect rule that was added to `web.config`

When the website is accessed via `http://artifactory.opentwin.net` the user will now automatically be redirected to `https://artifactory.opentwin.net`.

4.2 Configuration

Once the installation was finished, the configuration could be started. First, the initial setup wizard had to be completed. After that, users, repositories and access permissions where configured. Finally, the artifact storage location was configured to be on the `D:\` drive.

4.2.1 Initial Setup wizard

The first login to the Artifactory was possible with the default `admin` password `password`. In the following wizard, a new password for the administrator had to be set and the domain of the application had to be set to `artifactory.opentwin.net`. A default repository for Conan artifacts was created as well, but it was deleted later.

4.2.2 Users, Repositories, Permissions

In the current configuration, there is only one repository with the name `opentwin` and the layout `conan-default` defined.

Anonymous access to the repository was enabled as seen in Figure 4.3, to allow developers to pull any artifact from the repository without having to login. Distinct user accounts are only required for developers that want to publish new artifacts. Artifactory permissions are configured in three stages:

- *Permissions* give a defined list of permissions to a repository. In the current configuration, two *Permissions* are defined:
    1. `Anything` gives read access to the `opentwin` repository.
    2. `Publish` gives full write permissions to the `opentwin` repository.

- *Groups* give a group of users a defined list of permissions. Currently, two groups are defined:

  1. `Publisher` gives users the `Publish` permission.
  2. `readers` gives users the `Anything` permission.

- *Users* can be in a list of groups to inherit the permissions that are defined in a group. A user that is supposed to be able to publish new artifacts, must be in the `Publisher` group. Instructions on how to create a new user can be found in Section 4.3.3.

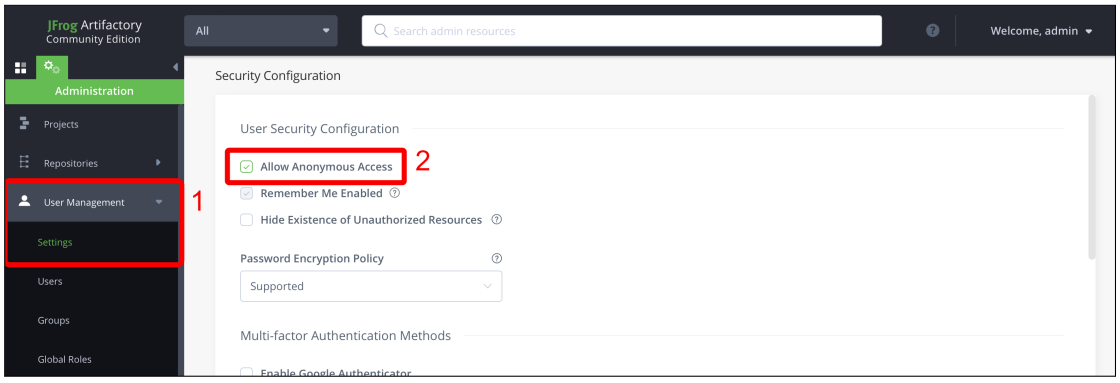

FIGURE 4.3: Enabling anonymous repository access in Artifactory 7: Contrary to the public documentation [19], the anonymous access can be enabled under *User Management* > *Settings*

4.2.3 Artifact Storage Location

By default Artifactory stores any uploaded artifact in the local install folder. In our setup they would be persisted at `C:\jfrog\artifactory\var\data\artifactory\filestore`. This configuration was changed to store all uploaded artifacts in a specified folder on the much larger `D:\` drive of the Windows Server. For that, the configuration file `C:\jfrog\artifactory\var\etc\artifactory\binarystore.xml` had to be changed to the contents in Listing 4.4 [20]. The application database is not affected by these settings. It is still located in the install folder of the application.

```
1 <config version="1">
2     <chain template="file-system"/>
3     <provider id="file-system" type="file-system">
4         <baseDataDir>D:\jfrog\artifactory</baseDataDir>
5     </provider>
6 </config>
```

LISTING 4.4: Contents of `binarystore.xml` that configures the artifact storage location

4.3 Maintenance

There are three maintenance tasks that a server administrator needs to perform on a regular basis to ensure a safe and secure operation of the Artifactory service:

1. The configured *Let's encrypt* certificate must be renewed every 60 - 90 days [21]. Auto-renewal is not (yet) configured.
2. It is recommended to update the Artifactory server on a regular basis.
3. User accounts must be created and deleted for developers that want to publish artifacts to the Conan repository.

It should be noted that the following maintenance documentation has not been fully tested yet and thereby only describes the processes on an abstract level. The server administrator is advised to be cautious when performing the described steps because they may contain errors or misconceptions.

4.3.1 Certificate Renewal

The configured *Let's encrypt* certifacte expires after 90 days. This is why a renewal of the certificate every 60 - 90 days is recommended [21]. Once a renewal is possible, an email notification will be sent to the registered contact. When the certificate has expired the website can still be accessed, but the user will be greeted with a warning by the browser. It is unknown how the Conan CLI responds to an expired certificate. This is why it is crucial to refresh the certificate on time.

The renewal of a certificate can be done with the `wacs.exe` *Win-ACME* CLI. To do so, the executable has to be started as administrator. After that, the letter `r` has to be entered to start the renewal wizard that will guide through the renewal process (see Fig. 4.4).

4.3.2 Artifactory Update

A regular update of Artifactory is recommended because it reduces the risk that the service is affected by a security vulnerability.

To do so, the Artifactory service has to be stopped. After that, the new version can be downloaded and extracted. The contents of `C:\jfrog\artifactory\app` can then be overridden with the new version [22]. It is important to not override or delete the

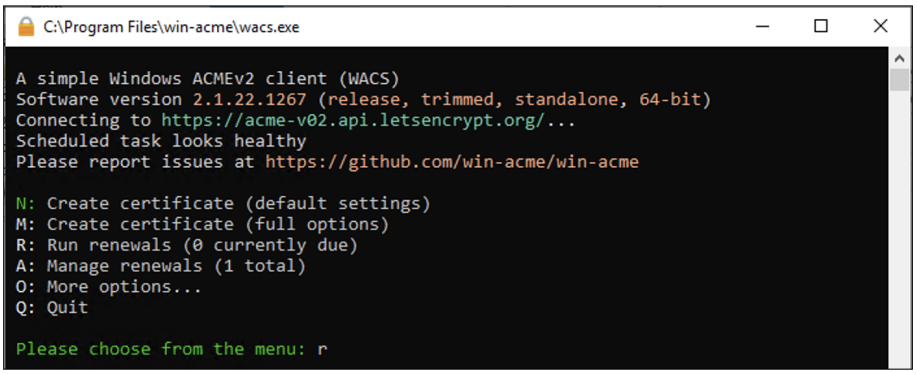

FIGURE 4.4: Renewing the certificate with *Win-ACME*:
By running `wacs.exe` as administrator and then entering the letter `r`, the renewal process can be initiated.

content of `C:\jfrog\artifactory\var`, because it contains the application data and configuration files. When all files have been successfully copied over, the service can be started again.

In any case the official upgrade instructions [22] should be studied before starting the procedure.

4.3.3 Managing User Accounts

When a developer that was in possession of a user account for uploading artifacts leaves the project, the related account should be deleted by the administrator like illustrated in Figure 4.5.

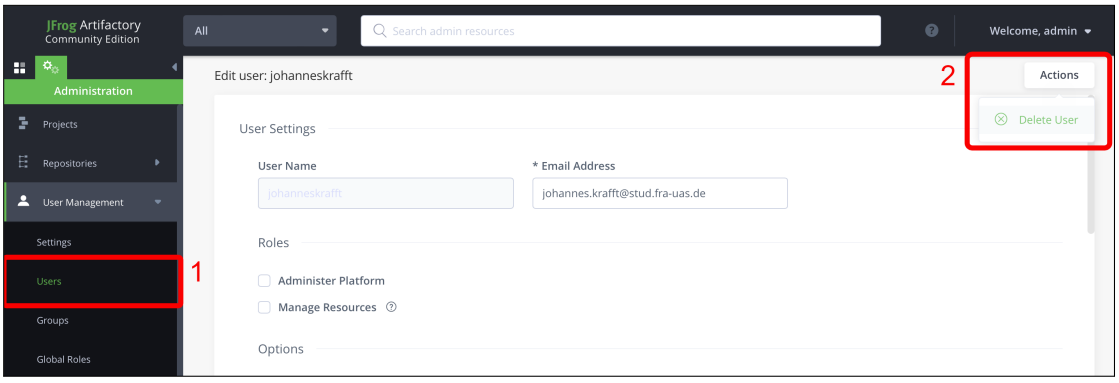

FIGURE 4.5: Deleting an Artifactory user

When creating a new user account, it is important to add it to the groups `readers` and `Publisher` as shown in Figure 4.6.

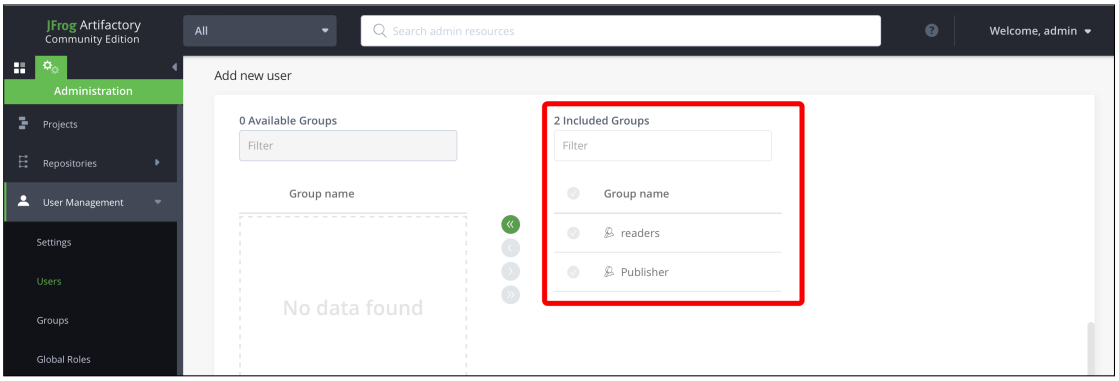

FIGURE 4.6: Creating a new Artifactory User:
It is important to add the user to the groups `readers|` and `Publisher|`.

Chapter 5

Detailed Usage Instructions

This chapter discusses the usage of new implementation and how to build and run the project under windows with Visual Studio.

5.1 Pre-requisites

- Download the OpenTwin project from the Github.
- Install Python 3.10 and its dependencies. Please make sure to tick the option "Download debugging symbols" and "Download debug binaries" in the advanced option field of the python installation. The dependencies can be found in file `requirements.txt` in the project directory.

```
pip install -r requirements.txt
```

- Install Visual Studio 2022

5.2 Conan Setup & Dependencies Installation

5.2.1 Install Conan CLI

Conan can be installed using the installer which can be downloaded from Conan website https://Conan.io/downloads.html. We can also install it using the pip command.

```
pip install Conan
```

5.2.2 Conan Profile

Next step will be to create the Conan profile for both release and debug configuration. Conan does lot of things automatically. It discovers the environment configuration for building projects and create a default profile. Every time we need to do something using Conan, it will use the information contained in the profile. The profile will define the architecture for which the dependencies will be installed.

Note: We will be using x64 Native Tools Command Prompt for VS 2022 for running further commands.

```
1  Conan profile new --detect <profile name>
```

We can create a custom profile also. By default the profile will be created for release configuration. The profile file can be found under the path `<userhome>\.Conan\profiles`. This file can then be used to create the debug configuration by copying the default file and then changing the build type to debug. The content for the profile file for a windows operating system, 64 bit architecture and build type Release will look something like below.

```
1  [settings]
2  os=Windows
3  os_build=Windows
4  arch=x86_64
5  arch_build=x86_64
6  compiler=Visual Studio
7  compiler.version=17
8  build_type=Release
9  [options]
10 [build_requires]
11 [env]
```

LISTING 5.1: Excerpt from `profile file`

The content for the debug profile file will look something like the following.

```
1  [settings]
2  os=Windows
3  os_build=Windows
4  arch=x86_64
5  arch_build=x86_64
6  compiler=Visual Studio
7  compiler.version=17
8  build_type=Debug
9  [options]
10 [build_requires]
11 [env]
```

LISTING 5.2: Excerpt from `profile file`

5.2.3 Add Artifactory

Since we are using Artifactory for the libraries which are not present on the Conan center we need to add the path to our own Artifactory.

```
conan remote add opentwin
  https://artifactory.opentwin.net/artifactory/api/Conan/opentwin
```

5.2.4 Install Dependencies

To install all the thirt party libraries we need to run Conan install command and provide `Conanfile.txt` to get list of all the required libraries.

```
conan install --install-folder <ThirdPartyLibraries Folder> --build missing
  --profile <profile_name> --remote opentwin .
```

5.3 Visual Studio Setup

5.3.1 Configuration using CMake File

Start Visual Studio 2022 and open the project folder in it. The CMake file in the project folder will be detected automatically and it will start the configuration. The configuration settings can be switched between `Release` and `Debug` from the toolbar of Visual Studio. We can see the configuration status in the output section. Once the configuration is finished, we can see the CMake targets for all the libraries in our project.

To see the CMake targets we need to go to solution explorer and click on 'switch between solution and avilable views'. Then we need to select 'CMake Target View'.

5.3.2 Build Targets

The targets can be build in two ways. We can build all the target at once or we can choose to build each target individually.

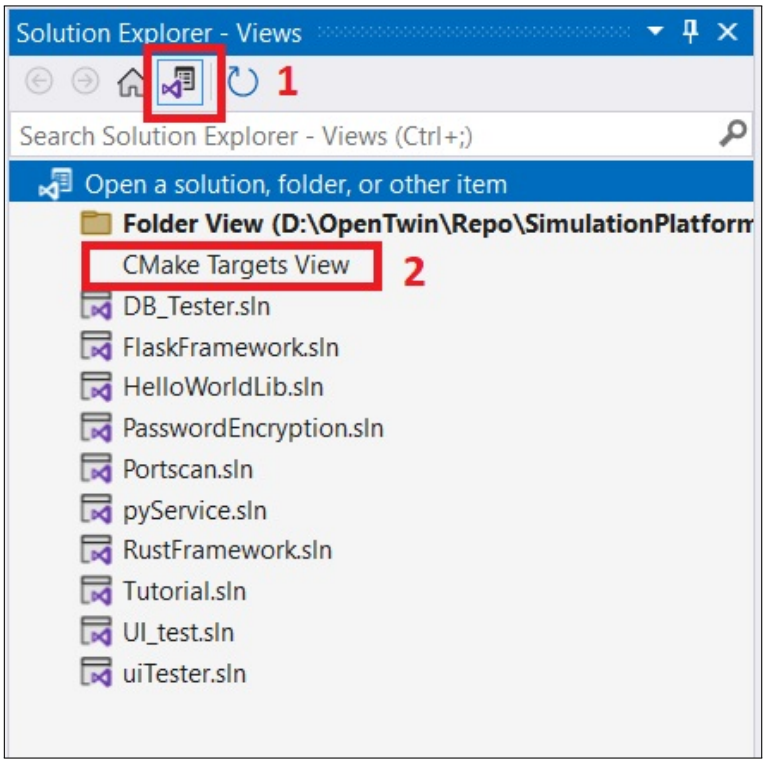

FIGURE 5.1: CMake Target View

- To build all target at once, Right click on the Project name in the CMake Target View (Here OpenTwin Project) and then click on `BuildAll` option.
- To build single target, naviate to `OpenTwin Project -> Libraries -> <LibraryName>`. Right click on the `<LibraryName>` and click `Build`

5.3.3 Build and Run Test

If we build all target at once, Tests for each library also gets build. For building test for individual library, naviate to `OpenTwin Project -> Libraries -> <LibraryName>`. Right click on the `<LibraryName>_test` and click `Build`.

After the build is completed, Test cases can be found in the `Test Explorer` window. Here we can choose either to run all the test at once or individually. This will only work if the RUNSETTINGS file has been configured (Auto Detect runsettings Files).

5.3.4 Debug Mode

To start the Application in the debug mode, first we need to click on the drop down option for startup item(1) and then choose uiFrontend.exe(2)

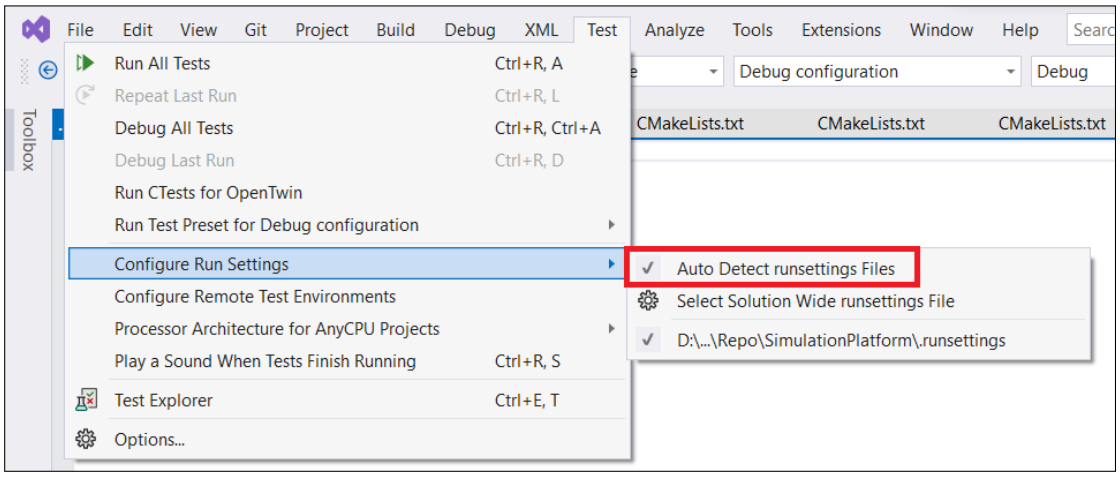

FIGURE 5.2: Test Configuration

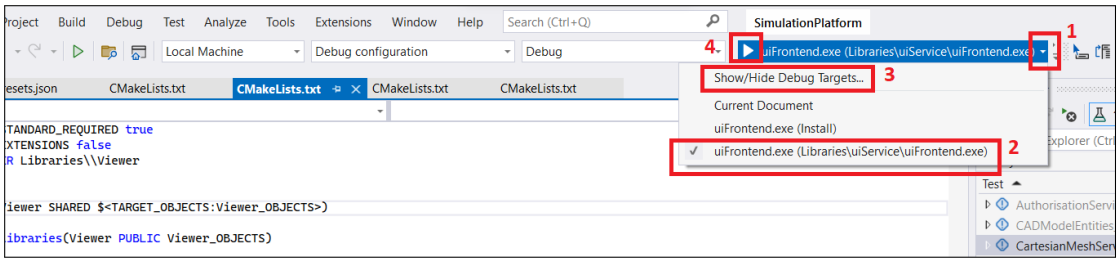

FIGURE 5.3: Debug Mode Configuration

If the above mentioned option is not present, click on `Show/Hide Target`(3) and choose the option from there. Once the configuration is selected, Click on the Run button present on the same select field(4).

5.3.5 Install OpenTwin

To install the project, Click on `Build` from menu bar and then click `Install OpenTwin`.

Install option will create an install folder inside the project which will contain all the required binaries to run the project.

5.4 Start Frontend

To start the frontend. Navigate to `OpenTwin\Repo\install\Debug\bin` and double click on uiFrontend.exe.

FIGURE 5.4: Home screen after startup

Chapter 6

Conclusion

The CMake port was finished successfully. It is available in the branch `feature/linux-port`. This chapter will present the major achievements that were met, and give an overview over what the next steps might be.

6.1 Achievements

The issues presented in Chapter 1 could be addressed successfully. The build process of OpenTwin is now more lightweight, standardized and platform-independent.

6.1.1 Repository Size Reduction

By moving all third-party dependencies out of the repository into Artifactory, the repository size could be reduced to just 1,45 GB, given that a shallow clone is done that does not include the complete git history. This is a 94% decrease in size. The third-party dependencies of course still need to be downloaded on the developers machine. But because just the pre-built binaries need to be fetched, only approximately 1.7 GB of disk space are occupied by all the third-party dependencies required for a `debug` build on Windows.

6.1.2 Industry Standard Tooling

No custom scripts where required to achieve the configuration, debugging and testing of all system components that have been ported to CMake. All technologies used (CMake, Conan, Visual Studio) can be considered well known industry standard tooling.

6.1.3 Platform Independence

In theory, the build system is now platform independent. CMake can be used to build with Visual Studio on Windows, Makefiles on Linux or XCode on macOS.

6.2 Outlook

While the results of the CMake port at this point can be considered successful, there is still things to improve.

6.2.1 Finish Open Tasks

There is still open tasks that could not be fully completed in the available time.

First of all, the debug configurations defined in `launch.vs.json` are not complete. There should be a configuration provided for each system component. Also, the install time of the Conan packages could be further improved by optimizing the configuration for the packages. For example, currently the entire Boost library is installed. By just pre-building and distributing the components that are really needed by OpenTwin and it's dependencies, the package size could be reduced by a lot.

6.2.2 Knowledge transfer and Feedback Collection

At the moment of writing this report, the new build process was not yet tested by other developers working on OpenTwin. Before merging the changes in `feature/linux-port` into the `master` branch, knowledge transfer has to happen to a much greater extend than what this report can achieve. Also, feedback must be collected by other developers to figure out if there are unsolved problems or if the workflow can be optimized in certain aspects.

6.2.3 Porting to Linux

The next big step would then be to try to build OpenTwin on another operating system. This would for sure require adjustments to both the source-code and build configuration. While the CMake configuration is by design platform independent, it likely makes assumptions about the build process that will not work on other operating systems.

Appendix A

Task Distribution

- **Johannes Krafft**
  - Drafted the overall porting strategy. Acted as technical lead, giving guidance on how CMake and Conan should be applied to solve the problem.
  - Decided to use Conan for package management and figured out how it can be integrated into the configuration process.
  - Figured out how to integrate the `open_twin.exe` Rust component into CMake.
  - Developed the concept on how debugging and testing of libraries can be achieved in Visual Studio.
  - Installed and configured the Artifactory Server.
  - Ported third-party dependencies that are not available on the CCI to Conan.
  - Applied the CMake porting concept to some of the system components and assisted troubleshooting some problems that occurred during the porting of some system components.

.

- **Rohit Kumar**
  - Reverse-engineered the dependency tree of the OpenTwin project. Created a list of all internal and external dependencies for each component in the system
  - Applied the CMake porting concept to most of the system components.
  - Isolated and troubleshooted most problems that occurred during the porting of each system component.
  - Researched if the required third-party libraries are available on the CCI and how they need to be integrated.

Bibliography

[1] Stack overflow trends: cmake. [Online]. Available: https://insights.stackoverflow.com/trends?tags=cmake (Visited on 2022-08-04)

[2] The history of cmake. [Online]. Available: https://cmake.org/cmake/help/book/mastering-cmake/chapter/Why%20CMake.html#the-history-of-cmake (Visited on 2022-07-31)

[3] Guidelines and howtos/cmake. [Online]. Available: https://community.kde.org/Guidelines_and_HOWTOs/CMake (Visited on 2022-07-31)

[4] Cmake homepage. [Online]. Available: https://cmake.org (Visited on 2022-07-31)

[5] Conan - introduction. [Online]. Available: https://docs.conan.io/en/latest/introduction.html (Visited on 2022-08-10)

[6] Kuga2. (2021) Unit testing internal functions of a dll. [Online]. Available: https://discourse.cmake.org/t/unit-testing-internal-functions-of-a-dll/3805 (Visited on 2022-07-31)

[7] (2022) Github - corrosion-rs/corrosion. [Online]. Available: https://github.com/corrosion-rs/corrosion (Visited on 2022-07-31)

[8] (2022) launch.vs.json schema reference (c++). [Online]. Available: https://docs.microsoft.com/en-us/cpp/build/launch-vs-schema-reference-cpp?view=msvc-170 (Visited on 2022-07-31)

[9] Conan - recipe and sources in a different repo. [Online]. Available: https://docs.conan.io/en/latest/creating_packages/external_repo.html (Visited on 2022-08-10)

[10] Conan - recipe and sources in the same repo. [Online]. Available: https://docs.conan.io/en/latest/creating_packages/package_repo.html (Visited on 2022-08-10)

[11] A. O. Aleksandrowicz. (2022) Installing artifactory. [Online]. Available: https://www.jfrog.com/confluence/display/JFROG/Installing+Artifactory#InstallingArtifactory-WindowsInstallation (Visited on 2022-07-31)

[12] P. Cociuba. (2019) Setup iis with url rewrite as a reverse proxy for real world apps. [Online]. Available: https://techcommunity.microsoft.com/t5/iis-support-blog/setup-iis-with-url-rewrite-as-a-reverse-proxy-for-real-world/ba-p/846222 (Visited on 2022-07-31)

[13] Guster. (2020) How to configure reverse proxy on windows iis. [Online]. Available: https://medium.com/@gusterwoei/how-to-configure-reverse-proxy-on-windows-iis-52a48b90163a (Visited on 2022-07-31)

[14] (2022) Request limits <requestlimits>. [Online]. Available: https://docs.microsoft.com/en-us/iis/configuration/system.webserver/security/requestfiltering/requestlimits/ (Visited on 2022-07-31)

[15] Install let's encrypt with iis on windows server 2019. [Online]. Available: https://www.snel.com/support/how-to-install-lets-encrypt-with-iis-on-windows-server-2019/ (Visited on 2022-07-31)

[16] Winacme - getting started. [Online]. Available: https://www.win-acme.com/manual/getting-started (Visited on 2022-07-31)

[17] Github - win-acme release v2.1.22.1289. [Online]. Available: https://github.com/win-acme/win-acme/releases/tag/v2.1.22.1289 (Visited on 2022-07-31)

[18] (2020) Setting up an http/https redirect in iis. [Online]. Available: https://www.namecheap.com/support/knowledgebase/article.aspx/9953/38/iis-redirect-http-to-https/ (Visited on 2022-07-31)

[19] B. Tova. Artifactory: How to grant an anonymous user access to specific repositories.

[20] A. Atzmony. Setting up artifactory - configuring the filestore.

[21] Let's encrypt - faq. [Online]. Available: https://letsencrypt.org/docs/faq/ (Visited on 2022-07-31)

[22] A. O. Aleksandrowicz. (2022) Upgrading artifactory. [Online]. Available: https://www.jfrog.com/confluence/display/JFROG/Upgrading+Artifactory#UpgradingArtifactory-WindowsUpgrade.1 (Visited on 2022-07-31)