Bernhard Reinisch, Bakk. techn.

Construction of a Release Pipeline for Agile Software Development

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Institute for Software Technology

Graz, May 2017

To . . .
my parents Gertrude & Bernhard
lovingly named Mama & Papa.

Abstract

In recent years the software industry has experienced quite an evolution in terms of what customers expect from software and software deliveries. Customers are much more willing and interested to play an active role in the software development process. In order to achieve fast feedback loops, a constant stream of software deliveries is necessary. In Agile Software Development, the key resources are empowered cross functional development teams which rely heavily on an automated infrastructure.

AVL reacted to these market changes by reconsolidating its software development process. An Agile Transformation was initiated. The whole software development process was adapted based on the *Scaled Agile Framework (SAFe)*. These changes also affected the development infrastructure. The infrastructure was not designed to support Agile development teams. Actually, the tool landscape heavily relied on people who manually executed the necessary steps when requested. Automated workflows were only rarely in place.

As part of this thesis, the core infrastructure building blocks were replaced. A modern *Source Control* system is used in order to support isolation for the development teams. Therefore, good support for branching and merging is paramount. The proprietary *Build System* is superseded by the Build vNext system in order to give the teams the ability to run builds by themselves. The *Deployment* package creation used a licensed authoring tool which did not meet the needs for branching. The open source tool Windows Installer XML (WiX) was the replacement.

In addition, an automated testing environment based on virtual machines was established. The software products are installed and tested within these environments at least on a nightly basis.

Keywords:

Release Pipeline, Agile Transformation, Team Foundation Version Control, Branching, Merging, Build vNext, VMWare vSphere, Oneiroi, Artifactory, Windows Installer XML, OneSetup

Kurzfassung

In den vergangenen Jahren hat in der Software Industrie ein Umbruch der Kundenerwartungen in Bezug auf Software- und Softwarelieferungen stattgefunden. Kunden sind mehr denn je dazu bereit und auch interessiert, eine aktivere Rolle in der Software-Entwicklung zu spielen. Um schnelle Rückkopplungsschleifen zu erreichen ist ein konstanter Strom von Softwarelieferungen erforderlich. Die Schlüsselressourcen in der Agilen Softwareentwicklung sind multifunktionale Entwicklungsteams, die sich wiederum stark auf eine automatisierte Infrastruktur verlassen.

Die AVL reagierte auf diese Marktveränderungen durch eine Evaluierung ihres Softwareentwicklungsprozesses. Dieser wurde basierend auf dem *Scaled Agile Framework (SAFe)* angepasst. Diese Änderungen wirkten sich ebenfalls auf die Entwicklungsinfrastruktur aus. Die ursprüngliche Infrastruktur war nicht darauf ausgelegt agile Entwicklungsteams zu unterstützen. Manuelle Arbeitsschritte waren die Regel. Automatisierte Abläufe waren nur selten vorhanden.

Im Rahmen dieser Arbeit wurden Kerninfrastrukurelemente ersetzt. Ein modernes Source Control System wurde eingeführt, damit Entwicklungsteams unbeeinflusst von einander arbeiten können. Das proprietäre *Build System* wurde durch das *Build vNext* System ersetzt. Teams haben so die Möglichkeit, Buildprozesse selbst auszuführen. Zur Erstellung der Installationspakete wird nun Windows Installer XML (WiX) eingesetzt.

Darüber hinaus wurde eine automatisierte Testumgebung unter Verwendung virtueller Maschinen eingerichtet. Die Software wird in diesen Umgebungen nächtlich voll automatisiert installiert und getestet.

Schlagwörter:

Release Pipeline, Agile Transformation, Team Foundation Version Control, Branching, Merging, Build vNext, VMWare vSphere, Oneiroi, Artifactory, Windows Installer XML, OneSetup

Acknowledgements

First and foremost, I would like to thank my supervisor, University Prof. Wolfgang Slany. Although this thesis took quite some time before it was finished, his support and patience made it possible. Secondly, I would like to thank AVL and its representative Dipl.-Ing. Andreas Fischer, who served as my boss and the AVL contact person for this thesis.

I am indebted to many of my student colleagues who supported me over the years. I want to especially mention my *very first hour* colleagues Michael Schneeberger, Martin Schröttner, Bernhard Mayr, Margarete Ortner, and Paul Sprenger who also became my best friends over these years.

As representatives of my work colleagues and part of the working poor, Andreas Kappel, David Bloice, and Christian Scherngell. All of you, but especially you three, were my lab rats in testing all the new tools we introduced. And thank you David for proof-reading this thesis. Your corrections really have proven that I have written this thesis by myself.

My parents Bernhard and Gertrude, my sisters Manuela and Petra, my uncle Franz, my aunt Gabi, and my cousin Markus for supporting me throughout all my studies at University and by simply being there when help was needed. My love Sonja, may you truly become the love of my life. My daughter Anna, you showed me that being a dad is not accompanied by fear. My son Bernhard, you just have arrived in this world and you are already loved so much.

Contents

List of Figures

List of Listings

List of Tables

Glossary

Agile Transformation Enterprise wide change of the software development process from CMMI to Agile software development. vii, ix, 1, 2, 25, 58, 129, 131

ALASKA **A**VL **L**ean **A**gile **S**oftware Development Process with **KA**izen. A development process based on the Scaled Agile Framework (SAFe). 1

Artifactory Universal Artifact Repository Manager from JFrog. vii, ix, xviii, 43, 47, 51, 53–55, 73, 130

AVL The term AVL stands for a particular business unit and includes specific departments (ITS-IM/ITS-XM). xvii, xviii, 1, 4, 7–9, 23, 25, 27, 28, 34, 38, 40, 44, 45, 51, 53, 54, 58, 59, 63, 64, 72, 114, 115, 129

Build vNext Web based, cross platform Team Foundation Server build system. vii, 130

Bundle A Bundle chains together a number of installation packages with a single User Experience. xviii, xxii, 82–84, 86, 87, 95–97, 100, 101, 103

Capability Maturity Model Integration A process-level improvement training and appraisal program. 1

Continuous Delivery Automated tool chain for build/test/deployment. The decision weather a software increment is deployed is manually made. xviii, 1, 2, 25, 37, 38, 131

Continuous Deployment Automated tool chain for build/test/deployment. The decision weather a software increment is deployed is automatically made. xviii, 2, 25, 37, 38

Continuous Integration Automated tool chain for build/test. Automated tests at build time ensures the quality. 2, 25, 37, 131

Dropfolder A central store for data exchange. All build/test results are stored on this file share. 28, 64

ESXi VMware ESXi (formerly ESX) is an enterprise-class, type-1 hypervisor developed by VMware for deploying and serving virtual computers. 27, 38, 42, 45, 46

Harvest CA Harvest Software Change Manager. xiii, xvii, 7–10, 23, 24, 129

InstallShield A proprietary (licensed) software tool from Flexara Software to create install packages. 130

Jira A proprietary issue tracking product, developed by Atlassian. 120, 121

Lab Management Combines the three concepts: Release Definition, ESXi and Oneiroi. xviii, 38, 39, 64

LaTex Is a mark up language specially suited for scientific documents and collaborative writing. xxi, 58–61, 131

Main Main is the name of the trunk in source control. xvii, xviii, 2, 10–16, 18, 19, 33, 39, 65, 66

Oneiroi PowerShell-based script files used in order to prepare machines, install AVL products, and run automated testing packages. vii, ix, xviii, 39, 40, 42–44, 47, 48, 131, 132

OneSetup Deployment platform based on Windows Installer XML (WiX). vii, ix, xviii, xix, 4, 47, 58, 60, 61, 69, 81, 85, 87–91, 95, 98–101, 103, 107, 109, 110, 112, 113, 130, 131

PowerShell PowerShell is a scripting language for task automation and configuration management built on the .NET framework. xxi, 29, 39, 40, 42–44, 46, 47, 49, 50, 59, 64, 73, 131, 132

Release Definition Similar to build definition, workflows for machine roll-outs are defined. xiv, xviii, 38–41, 43, 47, 69, 71

SANTORIN Productline for first/second level host systems. 123

Abbreviations

API Application Programming Interface. 34, 46, 53, 57, 78, 80, 82, 83

BA Bootstrapper Application. xv, xviii, 81–84, 98, 99, 103
BM Baseless Merge. xiv, xvii, xviii, 19–21

CD Continuous Delivery. xviii, 1, 2, 25, 37, 38, 131, *Glossary:* Continuous Delivery
CD^{+} Continuous Deployment. xviii, 2, 25, 37, 38, *Glossary:* Continuous Deployment
CI Continuous Integration. 2, 15, 18, 25, 37, 131, *Glossary:* Continuous Integration
CMMI Capability Maturity Model Integration. 1, *Glossary:* Capability Maturity Model Integration
CPL Common Public License. 79, 80

DISM Deployment Image Servicing and Management. 115
DLL Dynamic Link Library. xix, 51, 90, 99, 102, 107–109, 112
DoD Definition of Done. 58

FI Forward Integration. xvii, 12, 15, 18, 20

GUID Globally Unique Identifier. 102

IDE Integrated development environments. 8, 79
IIS Internet Information Services. xix, xxii, 114–116, 118, 119

MSI Windows Installer. xv, xviii, xxi, xxii, 51, 77–83, 86–90, 100, 102, 103, 109, 123, 125

NUnit NUnit unit-testing framework. 49, 63, 105

RI Reverse Integration. xvii, 12, 15, 16, 18, 20

SAFe Scaled Agile Framework. 1
SSL Secure Sockets Layer. xix, 114, 116, 117
SWR Software Release. 120, 121

TAF Test Automation Framework. *Glossary:* Test Automation Framework
TEA Tiny Encryption Algorithm. 94
TFS Team Foundation Server. xviii, 9, 25, 26, 28, 29, 38, 40, 44, 49, 51, 72, 129–132
TFVC Team Foundation Version Control. vii, ix, xiv, 9–15, 18, 19, 21, 130

UAC User Access Control. 82
UI User Interface. xviii, 82, 90, 94–99, 103, 115, 118
UX User Experience. xxii, 82, 95, 97

WIP Work in Progress. 94
WiX Windows Installer XML. vii, ix, xviii, 79–82, 84, 87, 101, 102, 107, 108, 123, 125, 130
WiX-Toolset Windows Installer XML Toolset. 79–81, 84, 103, 130

Notes

1. Introduction

“Do. Or do not. There is no try."

— Yoda, The Empire Strikes Back

In recent years the software industry has experienced quite a revolution in terms of what customers expect from software and software deliveries. Customers are much more willing and interested to play an active role in the software development process. Requirements are now written and evolve during the software project. With every delivery, the software is examined and re-evaluated. Requirements might change, particular topics gain interest, and others are dropped completely. In order to achieve such fast feedback loops a constant stream of software deliveries is necessary. Over the years a lot of companies have come forward with their success stories. How they managed their process transformation, their *Agile Transformation* needed for the new challenges. They described how they reached this comfortable state of Continuous Delivery (CD).

AVL reacted on these market changes by reconsolidating its software development process. There was already a sophisticated Capability Maturity Model Integration (CMMI)[1]-based software development process in place but it was not suitable for the future challenges. It was designed for long running projects with clear requirement, design, implementation, verification, delivery, and acceptance phases. During a software project, those stages were advanced in one direction and relied very much on paperwork in the early stages of the project. By introducing the **ALASKA**[2] process, which is based on the *Scaled Agile Framework (SAFe)*[1], the course was set for AVLs *Agile Transformation*.

Since *ALASKA* is a rather complex framework for large organisations, the focus of this thesis

[1]"CMMI is a process level improvement training and appraisal program. Administered by the CMMI Institute, a subsidiary of ISACA, it was developed at Carnegie Mellon University (CMU)", http://cmmiinstitute.com/

[2]**A**VL **L**ean **A**gile Software Development Process with **KA**izen

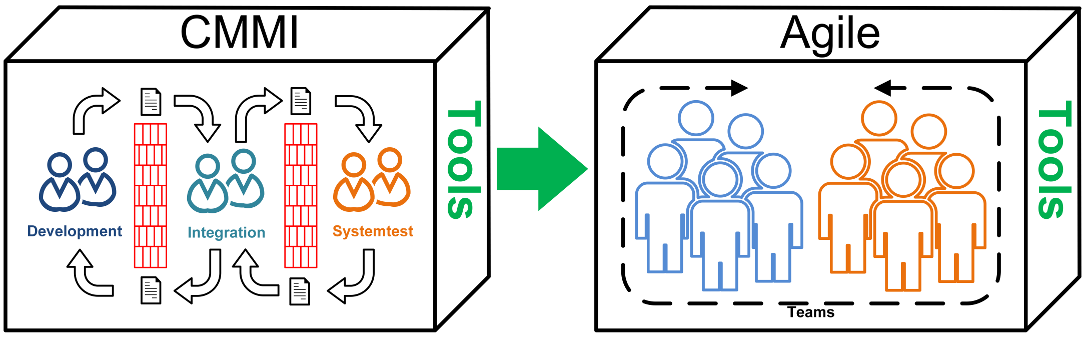


Figure 1.1: Agile Transformation

will be on the team level as shown in Figure 1.1. One of the main ideas was to remove the barriers between the *development, integration, and system test teams.*

At this point it has to be said that bringing these teams together with the **cultural change** needed is by far the most important, most cumbersome, and most challenging part of the Agile Transformation. Organisational units might have grown over centuries. At AVL, the silo mentality[3] is rather high. To overcome these difficulties, strong leadership and a common understanding of the goals as a company are needed. However, bringing everybody together to make it work is the critical part of this transformation, and is still ongoing.

After these organisational changes, the tooling[2] is the second most important aspect in achieving our goals. Everybody needs to be on the same page in respect to *Continuous Integration (CI), Continuous Delivery (CD), and Continuous Deployment (CD$^+$).* Automated testing is one of the key elements[3]. The infrastructure needs to provide the tooling in order to be successfully implemented. Together, metrics need to be defined, and common agreements on actions in case the system breaks down - and once the system is running smoothly it needs to be kept running. CI is not only a principle of thought, it keeps development teams in sync and removes the delays due to integration issues. However, CI is only the first step. Software development is not done by having the code integrated in Main. Bringing an integrated software version into a production environment was a forgotten part within the software development industry leaving a big gap between development and operations[4]. Getting all this working takes effort, but the benefits are profound. Complicated, long running intensive system test phases are no longer needed. Customers see their ideas and requirements arriving much faster and last but not least, emergency calls of operations teams who break systems through upgrades are a thing of the past. Together, development and operations teams get it running - *you build it, you run it.*

1.1 Structure

This thesis focuses on the changes which were/are necessary in different areas of the tooling infrastructure. A short overview of the previous (or in the case of older maintained software still

[3]"Silo mentality is an attitude that is found in some organizations; it occurs when several departments or groups within an organization do not want to share information or knowledge with other individuals in the same organization.", www.investopedia.com/terms/s/silo-mentality.asp

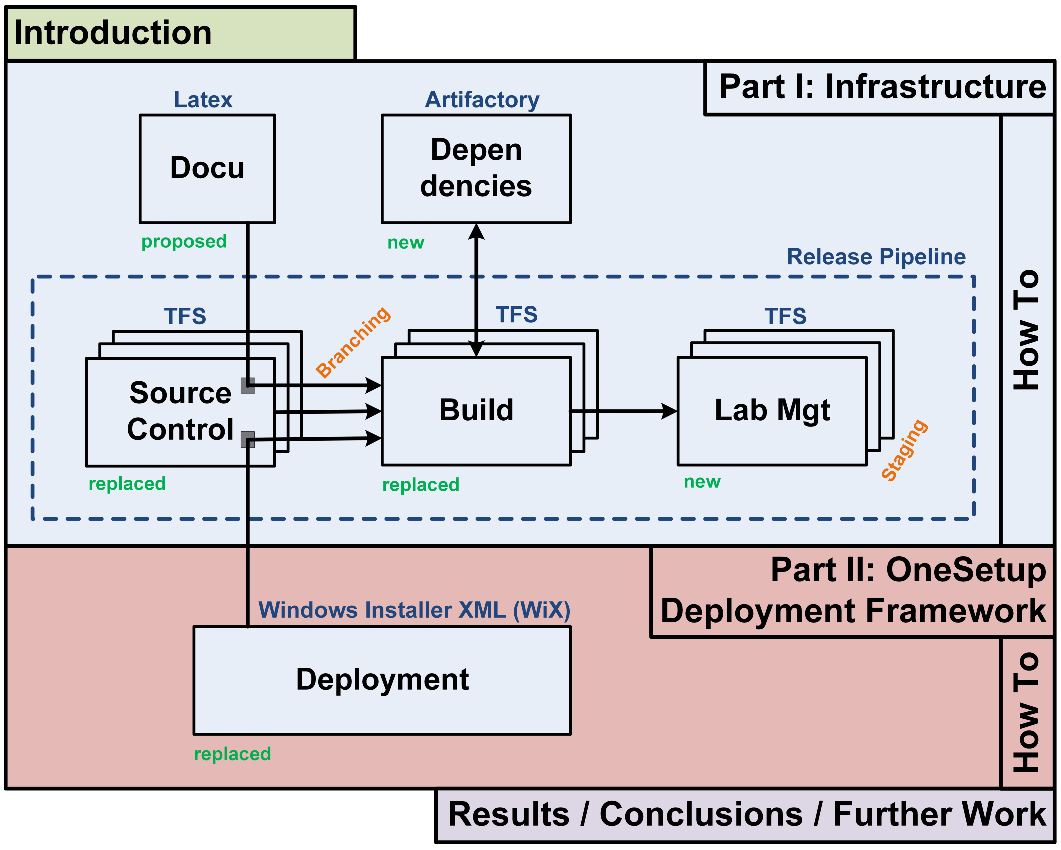


Figure 1.2: The Big Thesis Picture

the current) state is given in order to point out the main implementation flaws in the particular areas needed to be tackled.

At this point there is a **big spoiler alert**: One of the the main tooling flaws was/is that the **teams needed to be empowered** in order to achieve the delivery goals by themselves. Whenever a team is dependent on other teams in order to achieve their goals, priorities must be alignment, friction is generated, and the velocity[4] drops. Wherever it is possible, teams need to be empowered to get their work done on their own.

Figure 1.2 shows the big thesis picture which focuses on the following parts:

- **Introduction:** self reference to this chapter.
- **Part I - Infrastructure:** This first main part describes all the covered infrastructure topics. The release pipeline within the *Team Foundation Server* block is at its core.

  Source Control: Covers the transformation from a change management system into a modern source control system (see Chapter 2).

  Build: Replacement of a proprietary build system (see Chapter 3).

  Dependencies: Inter-product dependencies handling by binary references instead of rebuilding source code (see Chapter 5).

[4]"Velocity is a capacity planning tool sometimes used in agile software development. Velocity tracking is the act of measuring said velocity. The velocity is calculated by counting the number of units of work completed in a certain interval, the length of which is determined at the start of the project.", https://en.wikipedia.org/wiki/Velocity_(software_development)

Lab Management: Moving from test systems located under the testers' desk to a virtualised machine test setup in a data centre (see Chapter 4).

Documentation: Proposal using text based documentation tools integrated in source control (see chapter 6).

How To: Often used workflows within infrastructure part (see Chapter 7).

- **Part II - OneSetup Deployment Framework:** The second part deals with the development and purpose of the OneSetup framework. This framework centralises user interfaces and behavioural installation patterns to be applied by various software products within AVL.

  Introduction: Overall installer concepts (see Chapter 8).

  OneSetup Platform: Covers the reusable, configurable platform part (see Chapter 9).

  OneSetup Tools: Useful commissioning/OneSetup tools (see Chapter 10).

  How To: Example usage and common configuration examples (see Chapter 11).
- **Results / Conclusions / Further Work:** Final words about this thesis (see Chapter 12).

Infrastructure

2. Source Control

“TFS ate my source again."

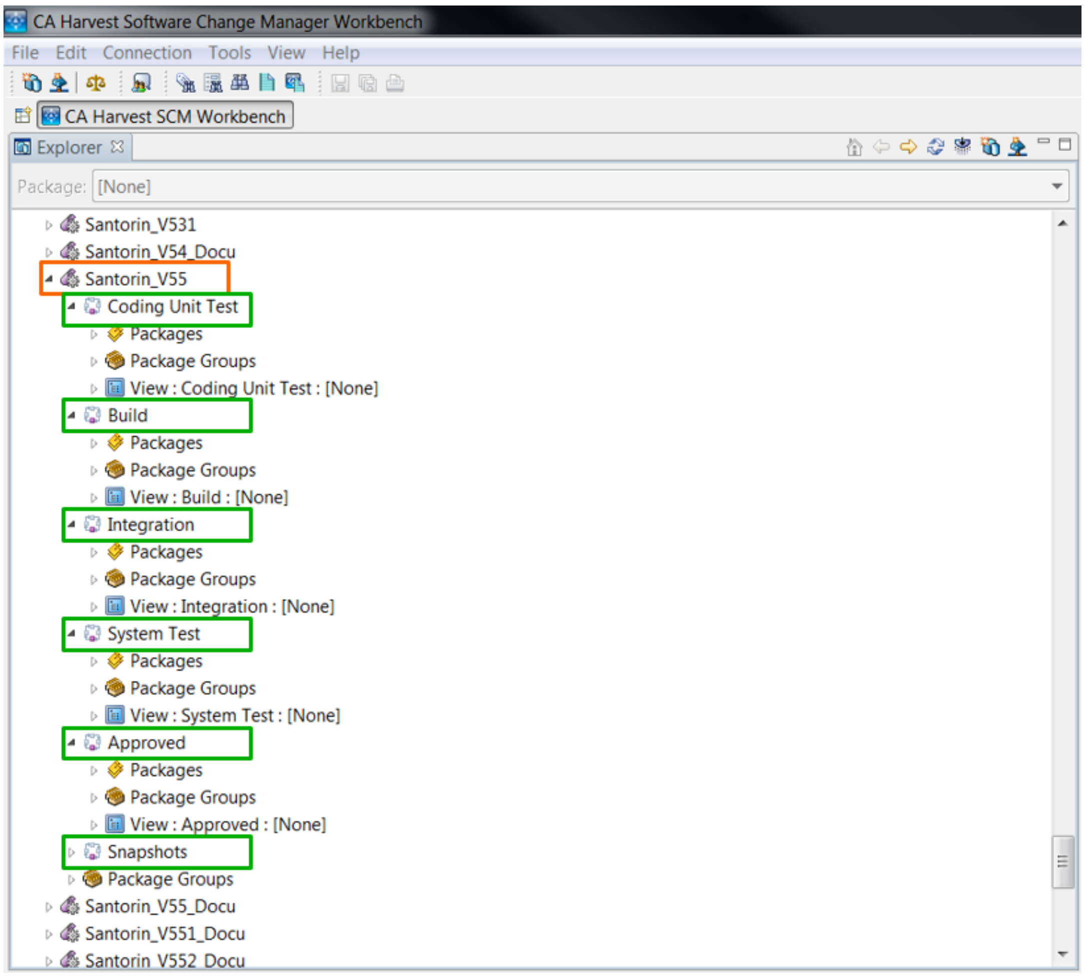
— David Bloice, Feng shui consultant

In AVL *CA Harvest Software Change Manager (SCM)* [5] [6] (internally named Harvest) was used as part of the source control infrastructure. Harvest is used for change management and version control of source code. It originates from the early 70s where it was used for aircraft engine development where robust, traceable, and reliable parallel development was necessary. These three attributes *robust, traceable, and reliable* are the main characteristics of Harvest and have stayed within the product until today. The main features are:

- **Change Packages:** Harvest provides version control and change management. In order to contribute to the source control, the developer creates a *change package*. The developer can then checkout the set of files she might need. The files and the changes are associated with the change package. Once the work has been done, the developer checks in the changes as part of the change package. This is the version control part of Harvest.
- **Life Cycles:** The change package can now be iterated through a predefined life cycle. At AVL those stages are called *Coding Unit Test, Build, Integration, System Test, Approved*, and *Snapshots*. These stages can only be iterated one after the other. At all these stages of this life cycle, the package needs approval from the appropriate user or user groups. These approvals / promotions are stored within Harvest (for audit purposes). At AVL, the movement through the life cycle was mainly done by the *Integration Engineers*. This role / person had to decide which package is to be integrated into what particular software versions and had to provide this software version to the system test.
- **Project (Environments):** Harvest Projects are customisable in order to suit company needs. This includes the process definition within the life cycle, the branching strategy, and access

control.

In Figure 2.1 a typical AVL Harvest environment is illustrated.


CA Harvest Software Change Manager Workbench
File Edit Connection Tools View Help
CA Harvest SCM Workbench
Explorer
Package: [None]

```
Santorin_V531
Santorin_V54_Docu
Santorin_V55
  Coding Unit Test
    Packages
    Package Groups
    View : Coding Unit Test : [None]
  Build
    Packages
    Package Groups
    View : Build : [None]
  Integration
    Packages
    Package Groups
    View : Integration : [None]
  System Test
    Packages
    Package Groups
    View : System Test : [None]
  Approved
    Packages
    Package Groups
    View : Approved : [None]
  Snapshots
  Package Groups
Santorin_V55_Docu
Santorin_V551_Docu
Santorin_V552_Docu
```

Figure 2.1: A typical Harvest Environment used in AVL

The Harvest project here is named “Santorin_V55” and shown in the top box. The project is the root node of the life cycle state model shown here in the boxes below. Within the state model, the change packages are located (within the folder named *Packages*). The packages can be promoted / demoted between the different life cycle stages. The corresponding branching model is shown in Section 2.1.1.

In respect to the needs of agile software development, Harvest has some major drawbacks:

- **Missing IDE Integration:** Harvest does not have a smooth integration in common *Integrated development environments (IDE)*. Anyone who is part of the software development process needs to use the Harvest interface directly.
- **Auditability:** One of the main features of Harvest is that every change can be traced and accounted to a particular person. All changes need to be visible within the system. This requirement is actually key for reliability critical software (e.g. power plant software, aircraft software, ...) but usually not needed for commercial software. This makes it very cumbersome to actually remove files / folders since those changes stay visible as deleted items.

- **Branching:** The life cycle design makes it very hard to support modern branching needs (see Section 2.1.2). During development there is the need for short term feature branches, and during maintenance there is the need to support different service levels in parallel. This can only be modelled by creating additional Harvest environments with their own life cycle. Creating these projects cannot be done by developers themselves which makes it not very useful in the daily routine.

A whole new source control system was needed. Since at AVL software development is very much based on Microsoft technologies, the decision for Microsoft *Team Foundation Server (TFS)* and its source control system *Team Foundation Version Control (TFVC)*[1] was pretty obvious. TFVC is a centralised source control system which scales from small to large projects with millions of files per branch. Access permissions can be defined down to file level. All changesets are checked-in the server side which makes auditing and tracking rather easy.

This new source control provides branching functionality in a rather simple way. Everybody is able to create new branches and therefore can isolate their work from others. On the other hand, by additional branching complexity to the source control is added. To avoid confusion, branches need to be created and reconciled (merged) in an organised way. In the following two chapters, branching patterns are discussed with their advantages and disadvantages. At the end of the day, the product life cycle and teams decide which branching patterns suit them best.

2.1 Branching

In this section different branching models / strategies are discussed. These branching models are very important for the day to day work since they directly translate into where and when feature development or bugfixing are made.

2.1.1 Harvest Branching Model

In Harvest, the branching model is actually directly derived from the *life cycle* model within Harvest itself. Every life cycle step directly translates into a branch (as shown in Figure 2.2).

Change Packages are created in the *Coding Unit Test* branch. Once the code changes have been checked-in, the developer can *promote* the package to the next stage (in this example to the *Build* stage). It depends on the process a team is running which stage actually reflects the state where the source code is being built. Secondly it is also a matter of access rights who is allowed to promote / demote packages within the life cycle. At AVL, a developer is allowed to create change packages in *Coding Unit Test* and can promote the package to build where a build is created nightly. From the build state, on the *integration engineer* takes over and promotes the packages to *Integration*, from where releases are created. As already mentioned, this branching model is very limited when it comes to concurrent support of multiple software versions.

[1]https://www.visualstudio.com/en-us/docs/tfvc/overview

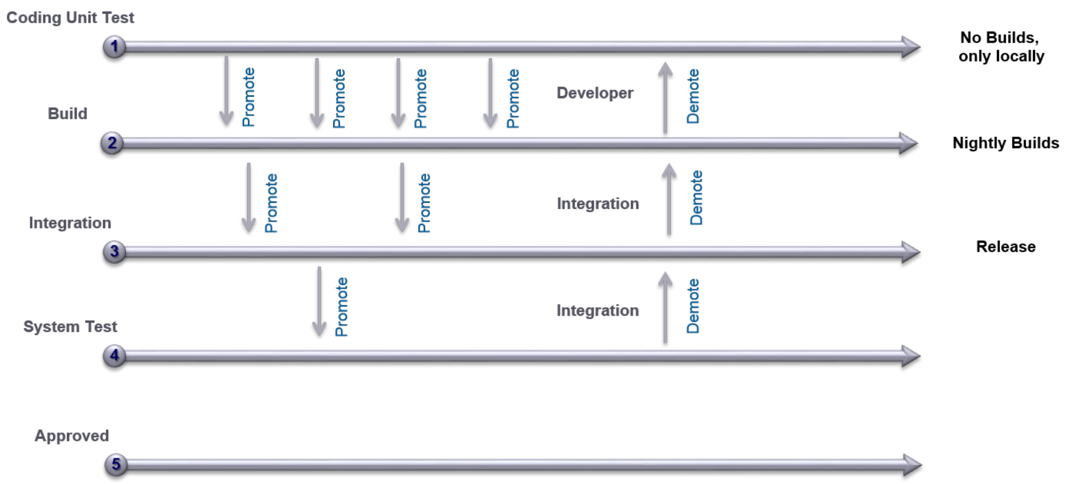


Figure 2.2: Harvest Branching Pattern

2.1.2 TFVC Basic Branching Model

When using TFVC as source control, proper knowledge about branching strategies and models are also needed. Figure 2.3 shows the basic foundation of every branching strategy.

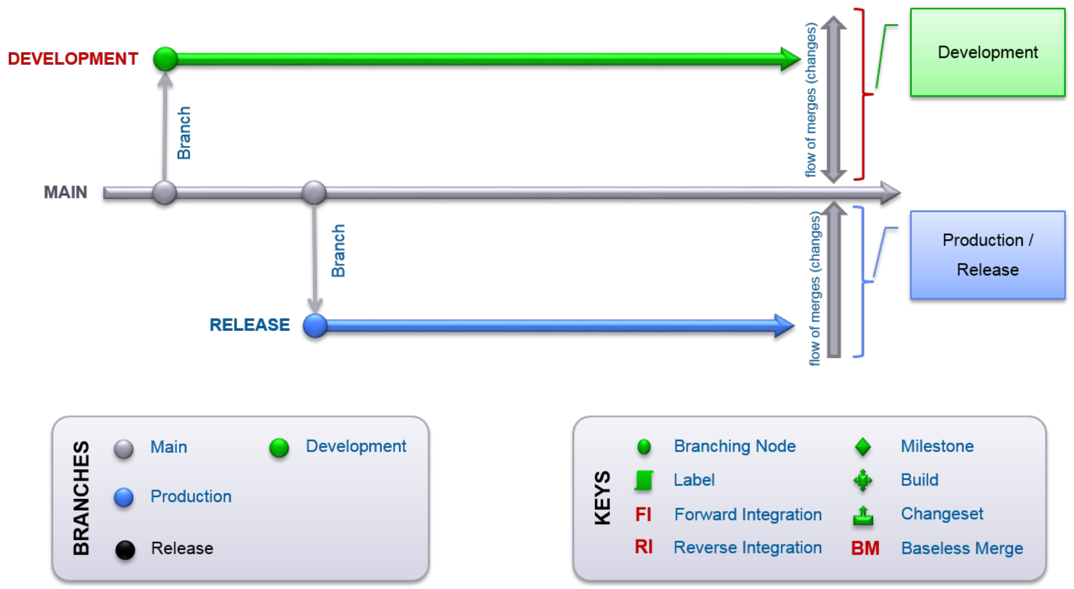


Figure 2.3: Basic Branching Pattern

The Main branch is in the integration centre. This branch contains the next software version which is ready to be released. In agile software development, often the terminology *potentially shippable* is used.

Above the Main branch, the development branch is located where all new feature development takes place. The separation between feature development (*Development* branch) and the current potentially shippable software version (*Main* branch) is needed in order to guarantee that the current feature development does not destabilise the *Main* branch. Once the feature development is done on the *Development* branch, the feature gets merged back into *Main* branch and enhances the current

software version.
Below the Main branch the released software maintenance branches are located. The maintenance branches get updated every time a new release is created and all changes are merged from *Main* branch to *Release* branch. The benefit in having a separate *Release* branch is that any bugfixing (which might be necessary in production) can really happen on the version which was released without having the risk that the latest changes from the *Main* branch might slip into production.
In the following sections this overall pattern - **the separation between development and production with main/trunk in the middle** - will not be broken. The *Main* branch will stay sandwiched between development and production as an anchor between the two.

2.1.3 TFVC Branching - Development Enhancements

The development can be more complicated due to the following challenges:

- **Multiple Teams:** Teams need to work in isolation from each other. Therefore *Team* branches are required.
- **Single Feature Development:** Particular features might take longer or should be developed strictly separated from all other changes. For that purpose, *Feature* branches provide an answer.
- **Quality Assurance: Before** any feature from a team/feature branch can be merged into Main, certain quality criteria have to be met. This guarantees that Main stays at a proper quality level.

Figure 2.4 shows the enhanced development branching.

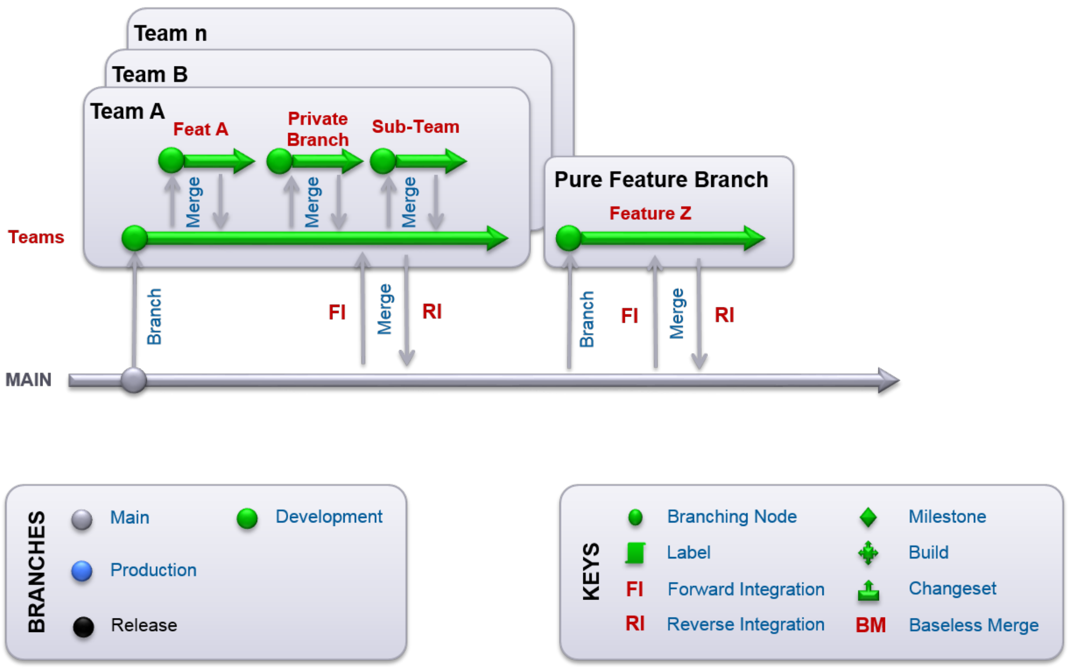


Figure 2.4: Enhanced Branching for Development

The idea is that whole teams are separated from Main, and work on their features on their own team branches. As part of the team scope there might be the necessity that particular teammates have their own private branches or that sub-teams work on a special feature. This strategy supports the development of multiple teams which contribute to one source control (e.g. one product, component). In the case of an epic which affects multiple teams, the concept of the *Pure Feature* branch is shown as well. Such a branch provides the possibility that teams can work together on one epic/feature besides their other team work.

In Figure 2.4, the merge events *Forward Integration (FI)* and *Reverse Integration (RI)* are also shown. The most important thing to mention here is the *FI* event. FI should happen at least on a daily basis to keep the teams aligned with Main and with each other. Merging will be covered in detail in Chapter 2.2.

The branching strategy and the isolation which can be achieved only makes sense if a strong automation chain supports the development. This means that all branches have proper build automation with test automation behind it. The teams need feedback if their changes affect the overall quality of the product / component they are working on. Only then the risk in back merging (Reverse Integration (RI)) new features / changes into main can be minimised. Main remains the *next software version* which can be released at any time.

2.1.4 TFVC Branching - Production/Release Enhancements

For production, the need for an extended branching strategy is required mainly due to:

- Concurrent software versions being released.
- Every release version having different service levels.
- Every service level having to be patched.

Figure 2.5 shows the enhanced production / release branching.

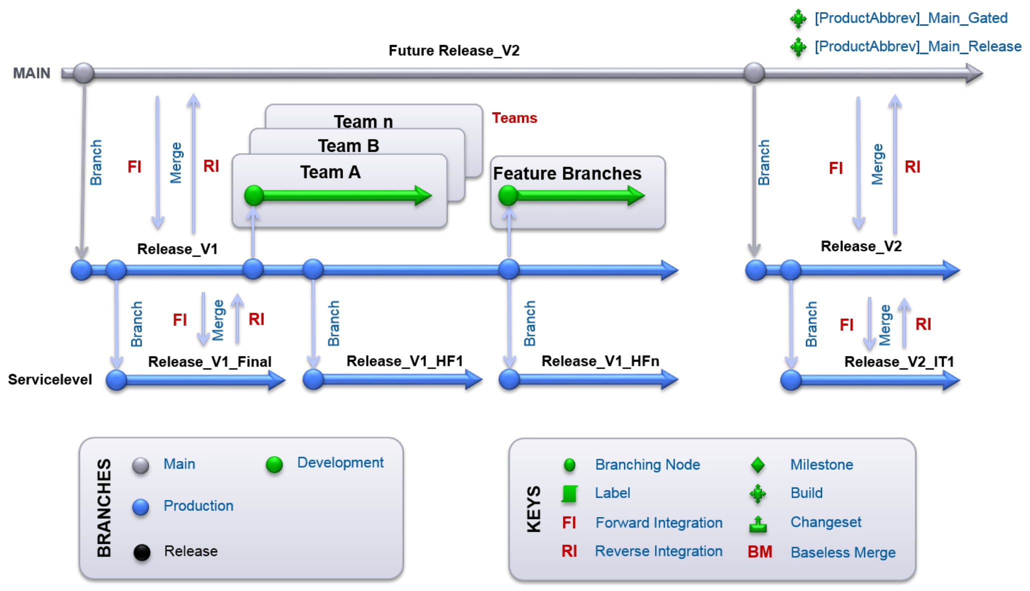


Figure 2.5: Production / Release Branching Enhancements

As part of the software release named *Release_V1*, a release branch is created. A new maintenance branch *Release_V1* is created. Furthermore *Release_V1_Final* will be created as well which represents the actual software level for the final release. *Release_V1* is called the maintenance branch for *V1*. From this branch, all future service releases will be created (e.g. *Release_V1_SL1, Release_V1_SL2, ... Release_V1_SLn*). Therefore, all bugfixing takes place directly on this branch. In case of a critical bug in *Release_V1_Final*, the teams can fix it directly on this branch. This structure extends accordingly to the right for every future release.

In addition to the maintenance branches, the teams can also develop new features as part of a service release. Therefore the enhanced development branching can also be applied *above* the maintenance branch (see also Figure 2.4).

2.1.5 TFVC Branching - Enhanced Pattern

As a consequence of Chapter 2.1.3 and Chapter 2.1.4, the enhanced branching pattern is shown in Figure 2.6.

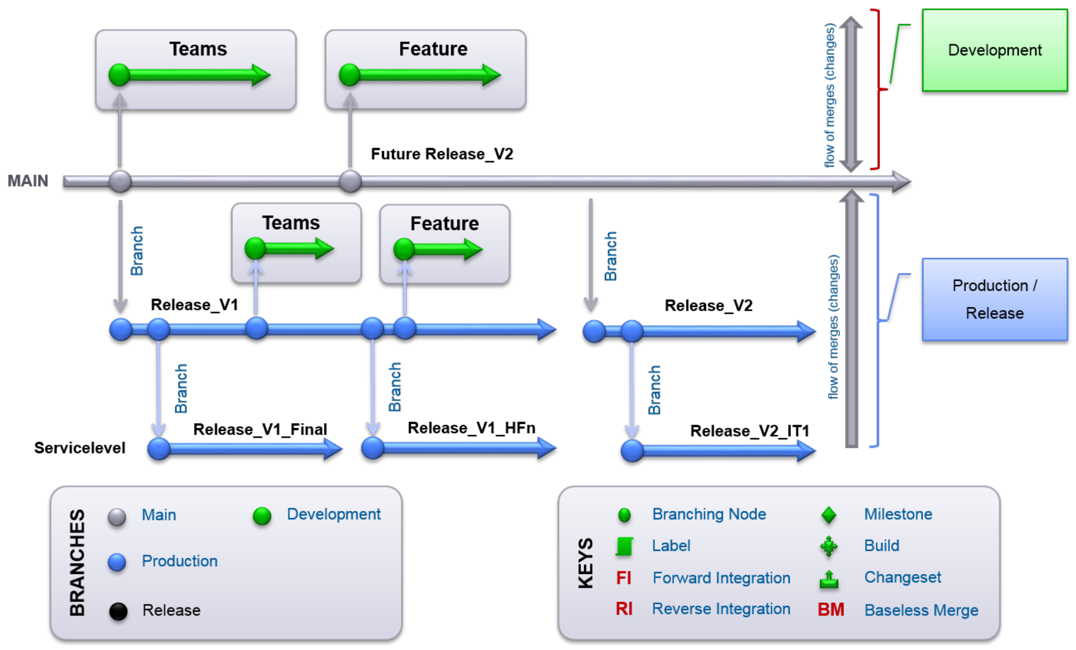


Figure 2.6: Enhanced Branching Pattern

In comparison to the basic branching pattern (see Figure 2.3) the enhanced pattern has the following properties:

- Support for Multiple Teams development
- Support for concurrent maintenance of different released versions
- Main is still in the centre and the anchor for development and maintenance

These branching concepts are actually not limited to TFVC. They are very generic and more or less all modern source control tools support these branching patterns. In any case, the final branching pattern which will be used is usually a variation of the enhanced pattern or a simplified one. In the end, the team setup and the number of contributors are also decisive parameters.

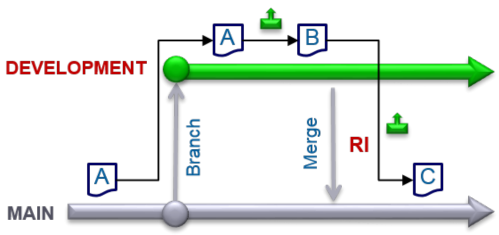


Figure 2.7: File Changes on Development Branch

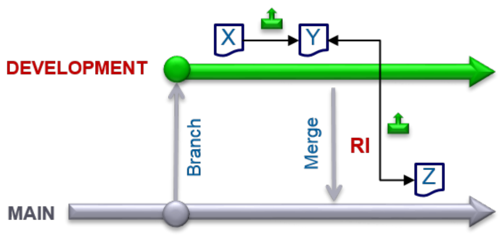


Figure 2.8: New File Created/Changed on Development Branch

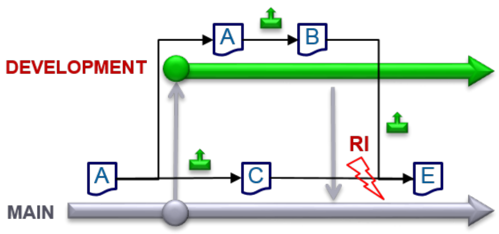


Figure 2.9: Merge Conflict on Main Branch

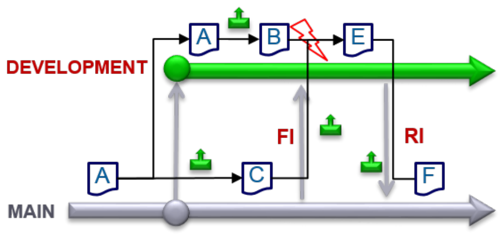


Figure 2.10: Merge Conflict on Development Branch

2.2 Merging

In Section 2.1 different branching possibilities were discussed. Branching provides the possibility to work on features/bugfixes without risking others' work. This isolation comes with at a price. At some point the changes done on development branches need to go back into the Main branch. This can be done by merging. Merging, also called integration in revision control, is a fundamental operation that reconciles multiple changes made to a revision controlled collection of files [7].

Within TFVC, merge operations rely almost entirely on the history (and not the content) of the affected files. This is nothing unique to TFVC since it is part of almost all modern revision control systems. When merging files between branches, two different merge events can occur:

- **Automatic Merge:** Sufficient information in the history of the files is present so that the tooling can automatically resolve the conflicts and reconcile the files.
- **Merge Conflicts:** Files were edited simultaneously on multiple branches. Someone needs to review, compare, and resolve the conflicts.

In Figures 2.7 to 2.10, different basic merge scenarios are shown.

Figure 2.7 and Figure 2.8 show merge scenarios where no conflicts occur. Since in both cases the full history is given, and this history is strictly linear, the user will see the files which are to be merged (and can of course always review the automatic merge) but no immediate user interaction is needed.

Figure 2.9 and Figure 2.10 show merge scenarios where conflicts occur. A particular file gets edited on Main and Development branch. At the time they are merged together again a merge conflict occurs and user interaction is required. Depending on the file types and the content changes, merge tools can provide assistance.

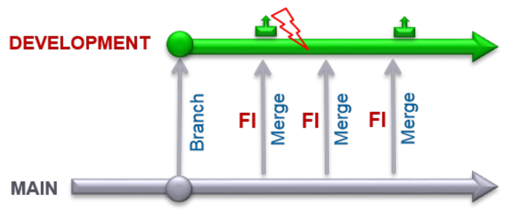


Figure 2.11: Forward Integration from Main to Development

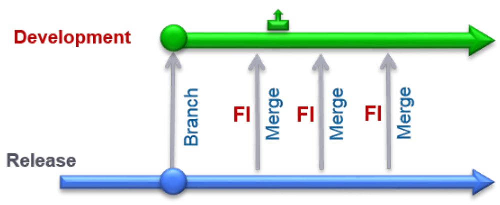


Figure 2.12: Forward Integration from Release to Development

In the following sections, different merge use cases are shown in order to provide a developers cookbook for merging/branching use cases. The used source control is TFVC.

2.2.1 TFVC Merge – Merge Whole Branch

Use Case 1: Forward Integration (FI)

Development branches of teams or features should not differ too much from their anchor branch, e.g. Main. To ensure this, whole branch merges are used to keep the development branch up to date. This is called *Forward Integration (FI)*.

Figure 2.11 and Figure 2.12 show the two uses cases based on the branching model from Section 2.1.5. It is good practice to run such FI merge events on a regular basis (e.g. daily). By doing so, conflicts are resolved early where the memory of the team is still fresh. Side effects can be identified quickly. The team's tool chain stays up and running.

Use Case 2: Reverse Integration (RI)

Merging from a development branch back into Main as shown in Figure 2.13 can mainly be seen as a final step of a feature development. Merging the whole branch ensures that *really all changes* are beeing merged from development to Main. Therefore the whole CI which took place on development branches streams back into Main (no *cherry picking*[2] or selective changeset merging takes place).

In case of maintenance, the whole branch merge (as shown in Figure 2.14) can be used to ensure that all changes from the release branches really stream back into the *next service release* or Main. This guarantees the feature/defect completeness in higher versions - **no downdates!**

Example: Merge whole branch from Main to TeamChicken

In the following a real world example of merging a whole branch from Main to *TeamChicken* is given:

1. Check if both branches are up to date by performing *Get Latest Version* (see Figure 2.15). Conflicts need to be resolved.
2. It is best practice to have no pending changes on neither the source nor the target branch. During the merge, all changes are done locally. Pending changes would then be mixed into the actual merge which might cause additional confusion. Figure 2.16 shows the check-in

[2]"Copy commits from one branch to another using cherry-pick. Unlike a merge or rebase, cherry-pick only brings the changes from the commits you select, instead of all the changes in a branch.", https://www.visualstudio.com/en-us/docs/git/tutorial/cherry-pick

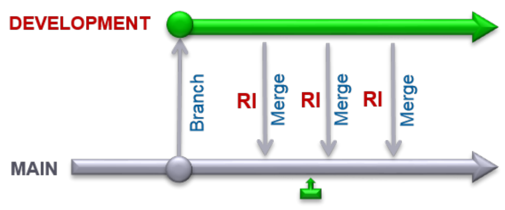


Figure 2.13: Reverse Integration from Development to Main

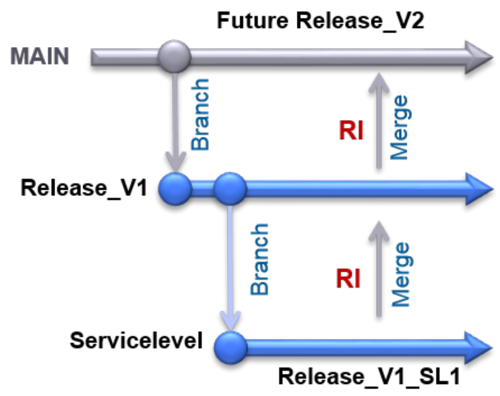


Figure 2.14: Reverse Integration from Servicelevel to Release to Main

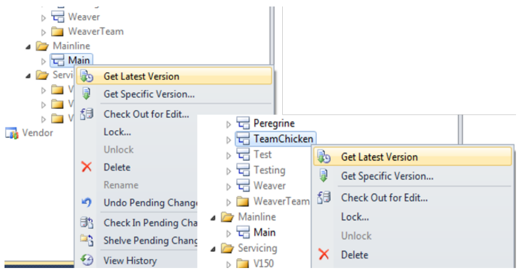
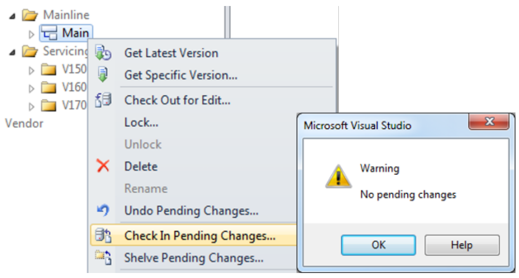


Figure 2.15: Perform *GetLatestVersion* on Main branch and TeamChicken branch



Microsoft Visual Studio

Warning

No pending changes

Figure 2.16: Checkin Pending Changes on both branches.

operation and the warning message in case no pending changes exist.

3. Preparation is done - *Latest Version* and *No Pending Changes* on both branches.
4. The merge is initiated on source branch (Main) as shown in Figure 2.17.
5. The source control wizard (see Figure 2.18) pops up to guide the user through the merge process.

   Check the source branch again
   Select *All changes up to a specific version*
   Select the target branch *TeamChicken*
   Continue with *Next* >
6. Select a proper *Version type*, e.g. *Latest Version* as shown in Figure 2.19. Continue with *Next* >.
7. Final informational note is shown to summarise everything. Continue with *Next* > and the merge operation will take place as shown in Figure 2.20.
8. As a result of the merge operation, conflicts may occur. These conflicts need to be resolved.
9. The whole merge process (including conflict resolution) is executed **locally**. In order to finalise the merge process, the local changes need to be checked-in to the *TeamChicken* branch as shown in Figure 2.21 and Figure 2.22.

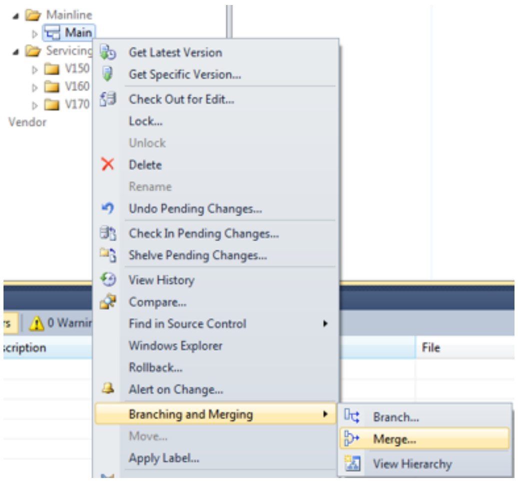
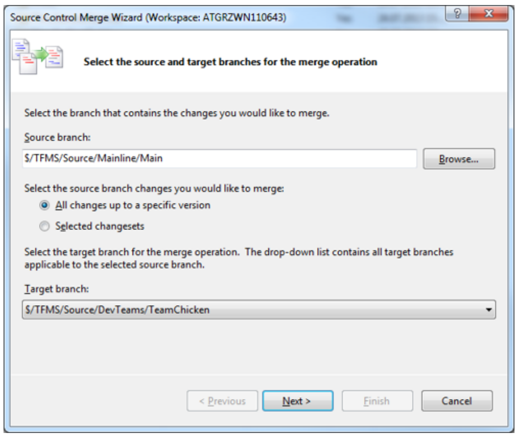


Figure 2.17: Initiating the merge from source branch


Source Control Merge Wizard (Workspace: ATGRZWN110643)

Select the source and target branches for the merge operation

Select the branch that contains the changes you would like to merge.

Source branch: $/TFMS/Source/Mainline/Main [Browse...]

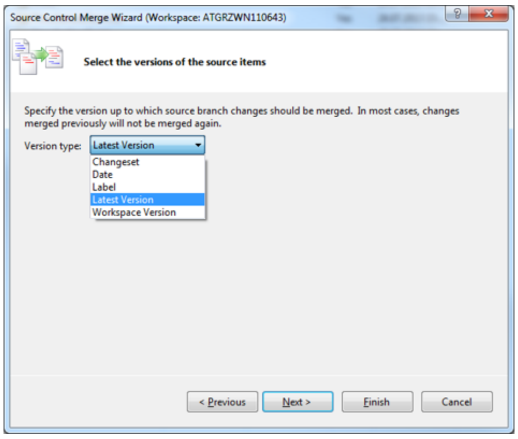
Select the source branch changes you would like to merge:
[✓ All changes up to a specific version]
Selected changesets

Select the target branch for the merge operation. The drop-down list contains all target branches applicable to the selected source branch.

Target branch: $/TFMS/Source/DevTeams/TeamChicken

< Previous | Next > | Finish | Cancel

Figure 2.18: Branch Selection


Source Control Merge Wizard (Workspace: ATGRZWN110643)

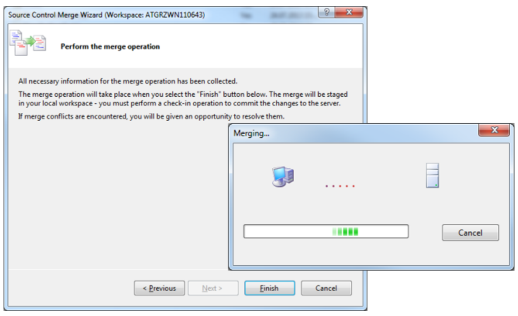
Select the versions of the source items

Specify the version up to which source branch changes should be merged. In most cases, changes merged previously will not be merged again.

Version type: Latest Version
Changeset
Date
Label
[✓ Latest Version]
Workspace Version

< Previous | Next > | Finish | Cancel

Figure 2.19: *Version Type* Selection


Source Control Merge Wizard (Workspace: ATGRZWN110643)

Perform the merge operation

All necessary information for the merge operation has been collected.

The merge operation will take place when you select the "Finish" button below. The merge will be staged in your local workspace - you must perform a check-in operation to commit the changes to the server.

If merge conflicts are encountered, you will be given an opportunity to resolve them.

Merging...

Cancel

< Previous | Next > | Finish | Cancel

Figure 2.20: Performing the Merge Operation

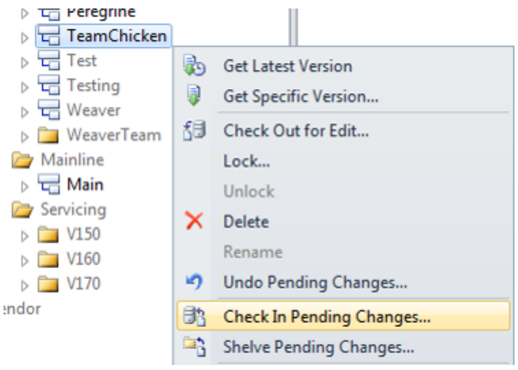
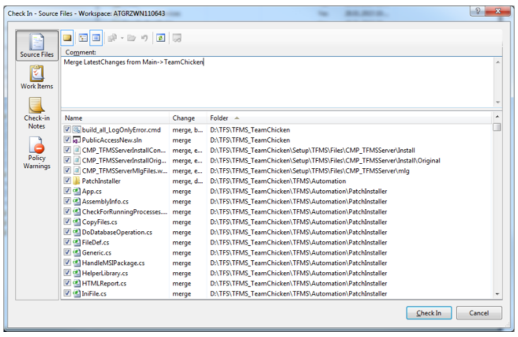


Figure 2.21: Check-in local pending changes after merge


Check In - Source Files - Workspace: ATGRZWN110643

Source Files | Work Items | Check-in Notes | Policy Warnings

Comment:
Merge LatestChanges from Main->TeamChicken

Name	Change	Folder
✓ build_all_LogOnlyError.cmd	merge, b...	D:\TFS\TFMS_TeamChicken
✓ PublicAccessNew.sln	merge	D:\TFS\TFMS_TeamChicken
✓ CMP_TFMSServerInstallCon...	merge, e...	D:\TFS\TFMS_TeamChicken\Setup\TFMS\Files\CMP_TFMSServer\Install
✓ CMP_TFMSServerInstallOrig...	merge, e...	D:\TFS\TFMS_TeamChicken\Setup\TFMS\Files\CMP_TFMSServer\Install\Original
✓ CMP_TFMSServerMlgFiles.w...	merge, e...	D:\TFS\TFMS_TeamChicken\Setup\TFMS\Files\CMP_TFMSServer\mlg
✓ PatchInstaller	merge, d...	D:\TFS\TFMS_TeamChicken\TFMS\Automation
✓ App.cs	merge	D:\TFS\TFMS_TeamChicken\TFMS\Automation\PatchInstaller
✓ AssemblyInfo.cs	merge	D:\TFS\TFMS_TeamChicken\TFMS\Automation\PatchInstaller
✓ CheckForRunningProcesses....	merge	D:\TFS\TFMS_TeamChicken\TFMS\Automation\PatchInstaller
✓ CopyFiles.cs	merge	D:\TFS\TFMS_TeamChicken\TFMS\Automation\PatchInstaller
✓ DoDatabaseOperation.cs	merge	D:\TFS\TFMS_TeamChicken\TFMS\Automation\PatchInstaller
✓ FileDef.cs	merge	D:\TFS\TFMS_TeamChicken\TFMS\Automation\PatchInstaller
✓ Generic.cs	merge	D:\TFS\TFMS_TeamChicken\TFMS\Automation\PatchInstaller
✓ HandleMSIPackage.cs	merge	D:\TFS\TFMS_TeamChicken\TFMS\Automation\PatchInstaller
✓ HelperLibrary.cs	merge	D:\TFS\TFMS_TeamChicken\TFMS\Automation\PatchInstaller
✓ HTMLReport.cs	merge	D:\TFS\TFMS_TeamChicken\TFMS\Automation\PatchInstaller
✓ IniFile.cs	merge	D:\TFS\TFMS_TeamChicken\TFMS\Automation\PatchInstaller

Check In | Cancel

Figure 2.22: Check-in overview of all files merged

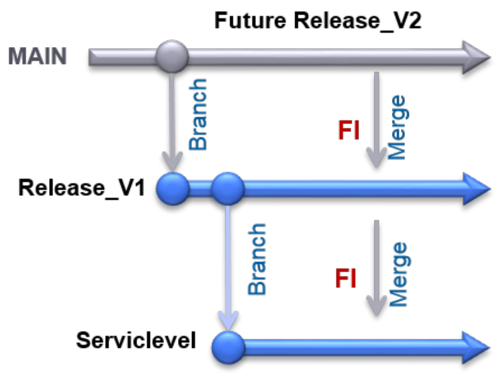


Figure 2.23: Forward Integration from Main -> Release -> Servicelevel

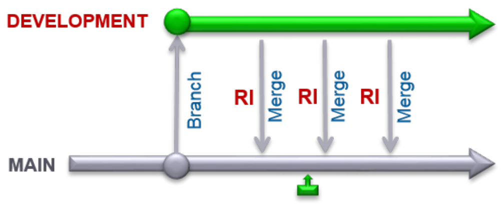


Figure 2.24: Reverse Integration from Development to Main

2.2.2 TFVC Merge – Cherry Picking

In TFVC there are two ways of merging:

1. Merge *all changes up to a specified version* (as already shown in Figure 2.18)
2. or merging *selected changesets*. Particular changesets can be selected from a list of changesets that are in the source branch but not yet merged into the target branch[3].

The second option is called a **Cherry Pick** merge.

By introducing *Cherry Picking*, some problems also arise in respect to CI. The source control sums up all changes over time. Therefore, one changeset is usually not granular. In general, it is not an easy task to identify the changes and the assigned changesets, and merge only the changes/changesets which are related to a specific problem without picking too much or too little.

Additionally, there is always the risk that the target branch will not be buildable after a cherry pick merge. Quite often, some additional files or manual file changes are needed to successfully build the target branch. Sometimes cherry picking is also a symptom of bad work preparation (especially in maintenance). **Teams need to know in advance** where bugfixes should be made in order to achieve proper Reverse Integration in the first place. Otherwise, bugfixes are done in Main and then later, FI is needed to merge the bugfixes into service releases.

Nevertheless, cherry pick merges have at least two valid use-cases:

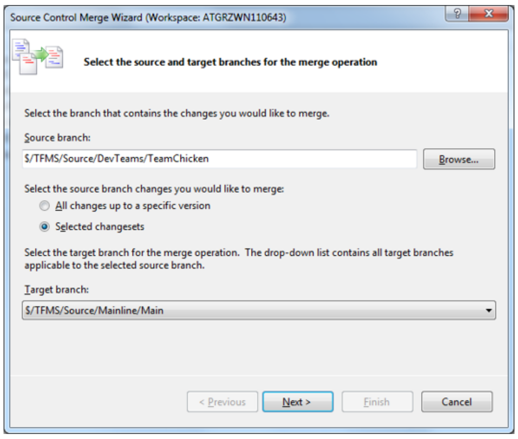
- **Forward Integration** ("Down Merge") of **requested** bugfixes or features into releases / servicelevels which are already solved in higher versions, as shown in Figure 2.23.
- **Reverse Integration** of specific content which is already finished in a team branch and can be released to Main (see Figure 2.24).

Example: Cherry Picking a Specific Changeset

In order to reverse integrate a specific bugfix/feature from *TeamChicken* to *Main*, the following steps are necessary:

1. Precondition: Source and target branches have the latest version and there are no pending changes (see Figures 2.15 and 2.16).

[3]https://blogs.msdn.microsoft.com/billheys/2011/01/19/what-is-a-cherry-pick-merge-and-why-do-you-recommend-against-them/


Source Control Merge Wizard (Workspace: ATGRZWN110643)

Select the source and target branches for the merge operation

Select the branch that contains the changes you would like to merge.

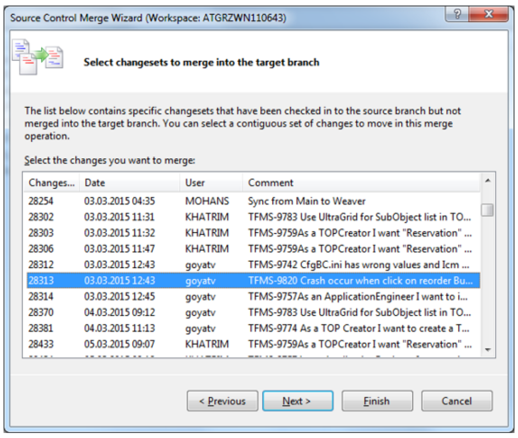
Source branch: $/TFMS/Source/DevTeams/TeamChicken

Select the source branch changes you would like to merge:
( ) All changes up to a specific version
[✓ Selected changesets]

Select the target branch for the merge operation. The drop-down list contains all target branches applicable to the selected source branch.

Target branch: $/TFMS/Source/Mainline/Main

Figure 2.25: Merge Wizard: *Specific Changeset* selected


Source Control Merge Wizard (Workspace: ATGRZWN110643)

Select changesets to merge into the target branch

The list below contains specific changesets that have been checked in to the source branch but not merged into the target branch. You can select a contiguous set of changes to move in this merge operation.

Select the changes you want to merge:

Changes...	Date	User	Comment
28254	03.03.2015 04:35	MOHANS	Sync from Main to Weaver
28302	03.03.2015 11:31	KHATRIM	TFMS-9783 Use UltraGrid for SubObject list in TO...
28303	03.03.2015 11:32	KHATRIM	TFMS-9759As a TOPCreator I want "Reservation" ...
28306	03.03.2015 11:47	KHATRIM	TFMS-9759As a TOPCreator I want "Reservation" ...
28312	03.03.2015 12:43	goyatv	TFMS-9742 CfgBC.ini has wrong values and Icm ...
28313	03.03.2015 12:43	goyatv	TFMS-9820 Crash occur when click on reorder Bu...
28314	03.03.2015 12:45	goyatv	TFMS-9757As an ApplicationEngineer I want to i...
28370	04.03.2015 09:12	goyatv	TFMS-9783 Use UltraGrid for SubObject list in TO...
28381	04.03.2015 11:13	goyatv	TFMS-9774 As a TOP Creator I want to create a T...
28433	05.03.2015 09:07	KHATRIM	TFMS-9759As a TOPCreator I want "Reservation" ...

Figure 2.26: Merge Wizard: *Select a Specific Changeset* to merge

2. Start the merge operation on the *TeamChicken* branch.
3. Select *Selected Changesets* as desired merge content as shown in Figure 2.25. Continue with *Next* >.
4. All changesets which have not yet been merged from *TeamChicken* to *Main* are listed as shown in Figure 2.26. Select the changeset which should be merged and continue with *Next* >.
5. The merge operation takes place locally. In case of conflicts they need to be resolved. Follow the steps as shown in Figures 2.20 to 2.22 to complete the merge operation.

Example: Cherry Picking Specific Folder Content/File(s)

Specific folder and file merges are also a category of cherry picking merges. The following steps are needed:

1. Precondition: Source and target branches have the latest version and there are no pending changes (see Figures 2.15 and 2.16).
2. **For Folder Merge:** Select the folder which should be merged as shown in Figure 2.27 (here *TeamChicken\Setup*).

   Perform the merge operation (see Figures 2.17 to 2.22).
3. **For File Merge:** Select the file which should be merged as shown in figure 2.28 (here *PublicAccessNew.sln*).

   Perform the merge operation (see Figures 2.17 to 2.22).

2.2.3 TFVC Merge – Baseless Merge (BM)

In TFVC, branching creates a relationship between two folders. This relationship is hierarchical. Merge operations are by default only allowed between branches which have a direct *parent-child relationship* with each other. This policy ensures that even in a complicated branching pattern (see Figure 2.6) the integration chain is not broken.

Baseless Merge (BM) however provides the functionality to violate the *parent-child relationship* merge policy. Basically, with a BM merge operation, a merge from any source to any target branch

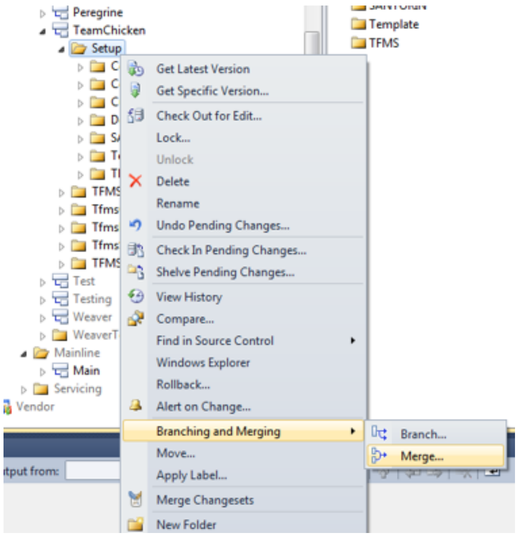


Figure 2.27: Specific Folger merge

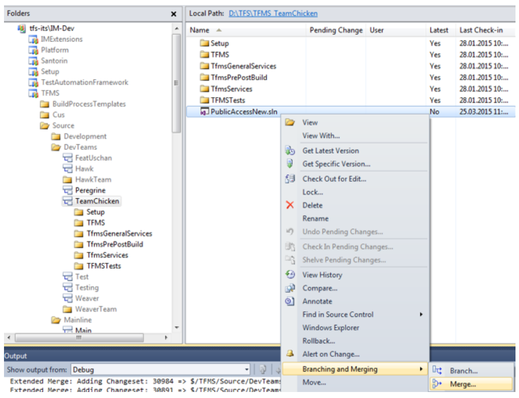


Figure 2.28: Specific File merge

is possible. In Figure 2.29, possible merge operations between branches with no direct relationship are shown.

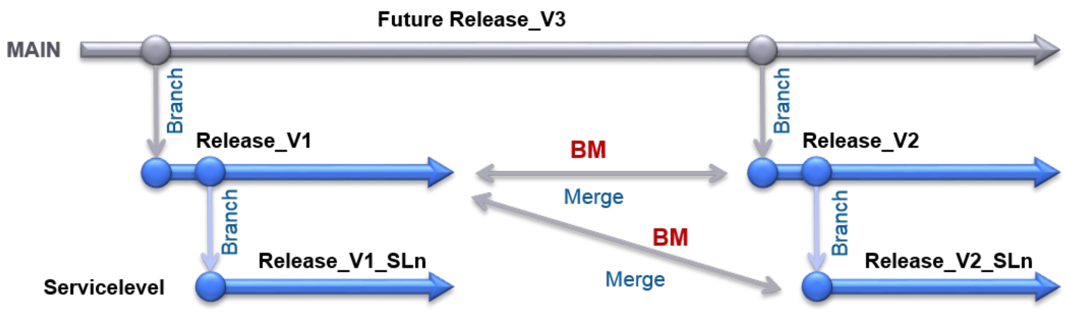


Figure 2.29: Baseless Merge Overview

A real world use case for BM operations is in the case of *Multi Version Bugfixing*. Figure 2.30 shows a standard maintenance use case. Assume that a bugfix is needed in `Release_V1` and in `Release_V2`. The standard answer to this request would be the following:

- Do the bugfixing in `Release_V1`.
- RI of this bugfix into `Future Release_V3`.
- FI of the merged bugfix into `Release_V2`. **This downmerge might cause trouble.**

The codebase in `Release_V3` might have changed dramatically in comparison to `Release_V2` and `Release_V1`. Therefore, merging from future `Release_V3` to `Release_V2` could be very tricky (nearly impossible without merging other changes as well). However the codebase of `Release_V1` and `Release_V2` are more similar to each other. Therefore it is better to merge from `Release_V1` to `Release_V2` directly.

Therefore the preferred procedure here is to use a BM as shown in Figure 2.31:

- Do the bugfixing in `Release_V1`.
- RI of this bugfix into `Future Release_V3`.

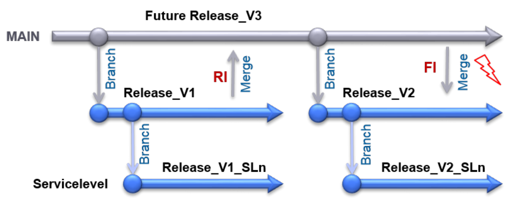


Figure 2.30: Multi version bugfixing without BM

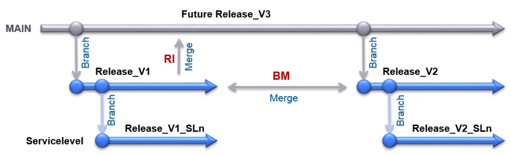


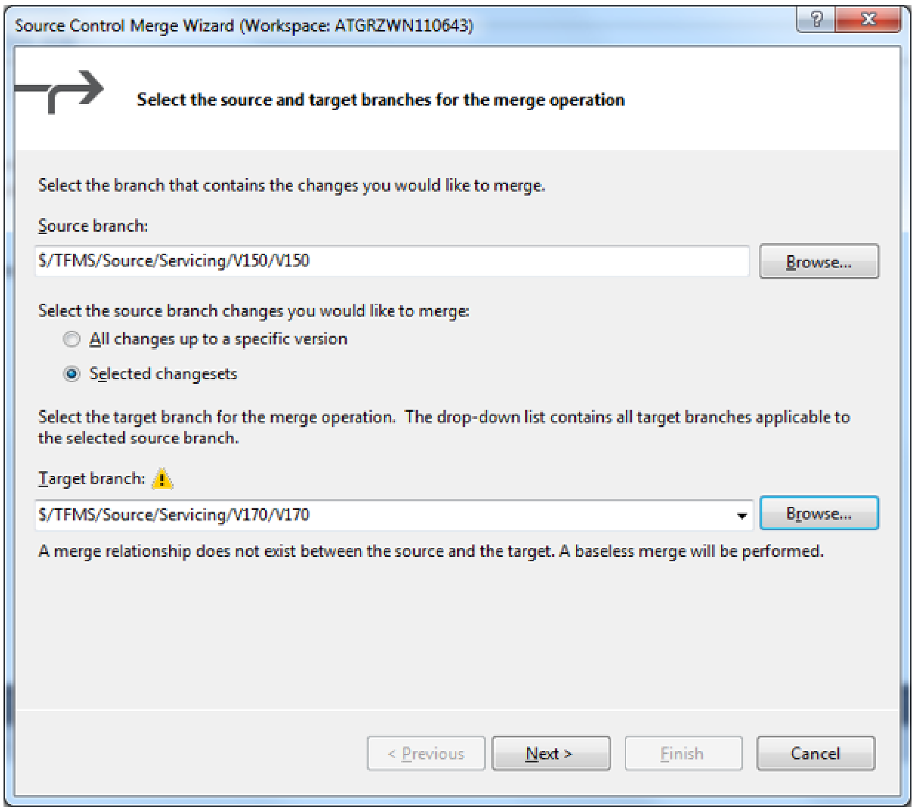
Figure 2.31: Multi version bugfixing with BM

- Merge the bugfix directly from `Release_V1` to `Release_V2` by using a *Baseless Merge*.

Example: TFVC Baseless Merge (BM)

To perform a BM in TFVC the following steps need to be followed:

1. Precondition: Source and target branches have the latest version and there are no pending changes (see Figures 2.15 and 2.16).
2. Start the merge operation on the source branch. Change the target branch by either entering or browsing for it as shown in Figure 2.32. Note: A warning will be shown that no direct merge relationship exists between the source and the target and that a BM will be performed.


Source Control Merge Wizard (Workspace: ATGRZWN110643)

Select the source and target branches for the merge operation

Select the branch that contains the changes you would like to merge.

Source branch:

$/TFMS/Source/Servicing/V150/V150

Select the source branch changes you would like to merge:

All changes up to a specific version

[✓ Selected changesets]

Select the target branch for the merge operation. The drop-down list contains all target branches applicable to the selected source branch.

Target branch:

$/TFMS/Source/Servicing/V170/V170

A merge relationship does not exist between the source and the target. A baseless merge will be performed.

Figure 2.32: Baseless Merge in TFVC

3. Perform the merge as shown in Figures 2.20 to 2.22.

3. Build System

“I would distill my Zirbenschnaps with Build vNext, if there was a build step for that."

— Christian Scherngell, #1 fan of #15

The build system used at AVL was/is a proprietary build system. It was developed in-house by Hermann Schinagl and Michael Karner between the years 2000 and 2002 and is a product of this time. In 2000, there were no standardised build systems available on the market. Companies needed to come up with their own solutions for the build and delivery process. Therefore, the old build system was designed to serve the following (AVL process) needs:

- The source control/change management system used was Harvest (see Chapter 2). The AVL build system needed to be coupled to this source control system.
- At least a nightly build with reporting was necessary.
- *Building source* and *creating deployment packages* are two main blocks and needed to be run separately.
- Interface management was achieved via rebuilding of dependencies.

The solution was a batch script based build system with Perl[1] scripting language at its core. In Listing 3.1, a particular build project `TFMS_V132` with its files is shown.

[1] "Perl is a family of high-level, general-purpose, interpreted, dynamic programming languages.", https://en.wikipedia.org/wiki/Perl

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

D:\Projects\TFMS_V132>dir
 Volume in drive D has no label.
 Volume Serial Number is BCFD-F87B

 Directory of D:\Projects\TFMS_V132

29.03.2016  14:32    <DIR>          .
29.03.2016  14:32    <DIR>          ..
18.09.2012  12:51             1.452 build.bat
01.09.2015  15:09             2.191 PostProcess.bat
02.11.2015  09:23             6.502 TFMS_V132Config.xml
-------- other files got snipped ------------
              40 File(s)        538.036 bytes
               7 Dir(s)  22.803.419.136 bytes free

D:\Projects\TFMS_V132>
```

Listing 3.1: Perl-based build project `TFMS_V132`

The most important files are:

- **build.bat:** With this batch file a whole build can be created. As part of the processing:
  the source is copied to a local drive,
  depending on the XML configuration, build solutions are built and
  PostProcess.bat is executed.
- **PostProcess.bat:** This batch file creates the product runtime on the disk. The deployment packages are created from this runtime which is also part of the postprocessing. In order to support manual modification of the source build, *PostProcess.bat* can also be run separately.

```xml
<Project Name = "TFMS"
    VersionControl = "harvest"
    HarvestScheme = "harvest7"
    HarvestEnvironment = "TFMS_V132"
    HarvestRepository = "TFMS_Sys"
    HarvestState = "Build"
    HarvestUser = "buildread"
    HarvestPassword = "buildread"
    WorkSpaceName = "TFMS.sln"
    Configuration = "Release,Debug"
    MailMap = "harvest://TFMS_Sys/All/TFMSMailMap.xml"
    SyncConfig = "harvest://TFMS_Sys/All/TFMSSyncConfig.xml"
    >
    <Dependency Name = "GeneralServices"/>
    <Dependency Name = "CME_V10"/>
</Project>
```

Listing 3.2: Section of Project Configuration File of `TFMS_V132`

- **TFMS_V132Config.xml:** This build project configuration file mainly defines the build dependencies and order. Listing 3.2 shows a section of this configuration file. This section shows one particular solution which is built during the build process. All Harvest related attributes describe the connection to the Harvest source control. The *WorkSpaceName* defines the actual solution file. All dependencies are listed within the *Dependency* nodes.

This small overview of the AVL proprietary build system shows the starting point of the build tooling when the *Agile Transformation* started. The main problems with the old build system are listed as follows:

- *Source Control* and *Build System* are decoupled without any notification system. Modern build systems allow various possibilities to run builds based on check-ins on particular branches or folders - even *Gated-Check-ins*[2] are possible.
- The build scripts are not part of the source control neither are they versioned.
- Creating branches is very cumbersome. Various settings need to be changed.
- No parallel build support.
- **Teams are not able to run/maintain a build by themselves.**

This list could be extended quite easily. It is no wonder the build system is already over 15 years old. In the meantime, big companies have closed the gap and professional build software is on the market.

3.1 Build System

Nowadays the topics *Continuous Integration (CI)*, *Continuous Delivery (CD)*, and *Continuous Deployment (CD$^+$)* are the keywords everybody is talking about. The TFS *Build System* provides a set of tools for achieving these goals. TFS included a build system from its initial release. This build system included build controllers and build agents. The whole process was defined by *Workflow Foundation XAML* files. Although this build system also had its own flaws and issues, also Microsoft came up with a web based build system[3] which also supports Mac OSX and Linux. This latest build system from Microsoft is one of the most powerful in the industry.

3.1.1 Build Agents, Agent Pools, and Queues

The main concepts *Build Agents*, *Agent Pools*, and *Queues* help to manage the build resources within a company. The idea is to manage the resources not individually, but instead to use logical grouping units. Figure 3.1 (inspired by [8]) illustrates the organisational overview of an on-premises installation:

- **Build Machines:** Build Machines represent the actual hardware or virtual machines where build agents are installed.
- **Build Agents:** Build Agents are the smallest building blocks of a build infrastructure. Build agents execute particular build workflows. Multiple build agents can be installed on a single machine. Depending on the machine setup, build agents can have different capabilities in respect to the location, installed licensed vendor software, or available compilers. These capabilities are used for selecting a proper build machine when triggering a new build.
- **Agent Pools:** Agent Pools arbitrarily group together a number of build agents from different build machines. This grouping also defines a shared boundary of all agents within the pool.

[2] A check-in triggers a build. Only if the build is successful are the changes committed to the source control, otherwise they are rejected.

[3] Build vNext, https://msdn.microsoft.com/Library/vs/alm/Build/feature-overview

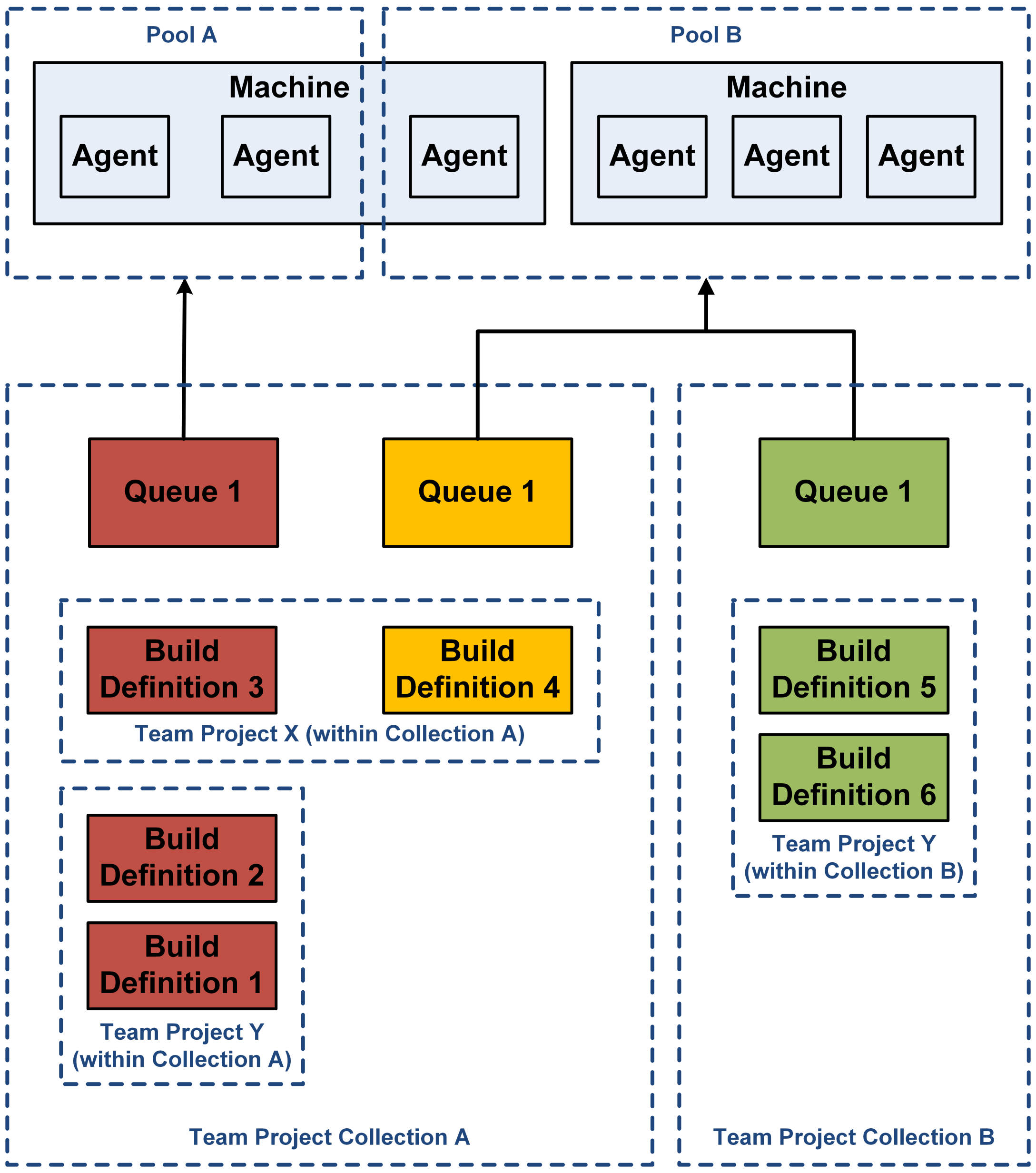


Figure 3.1: TFS 2015 Build system

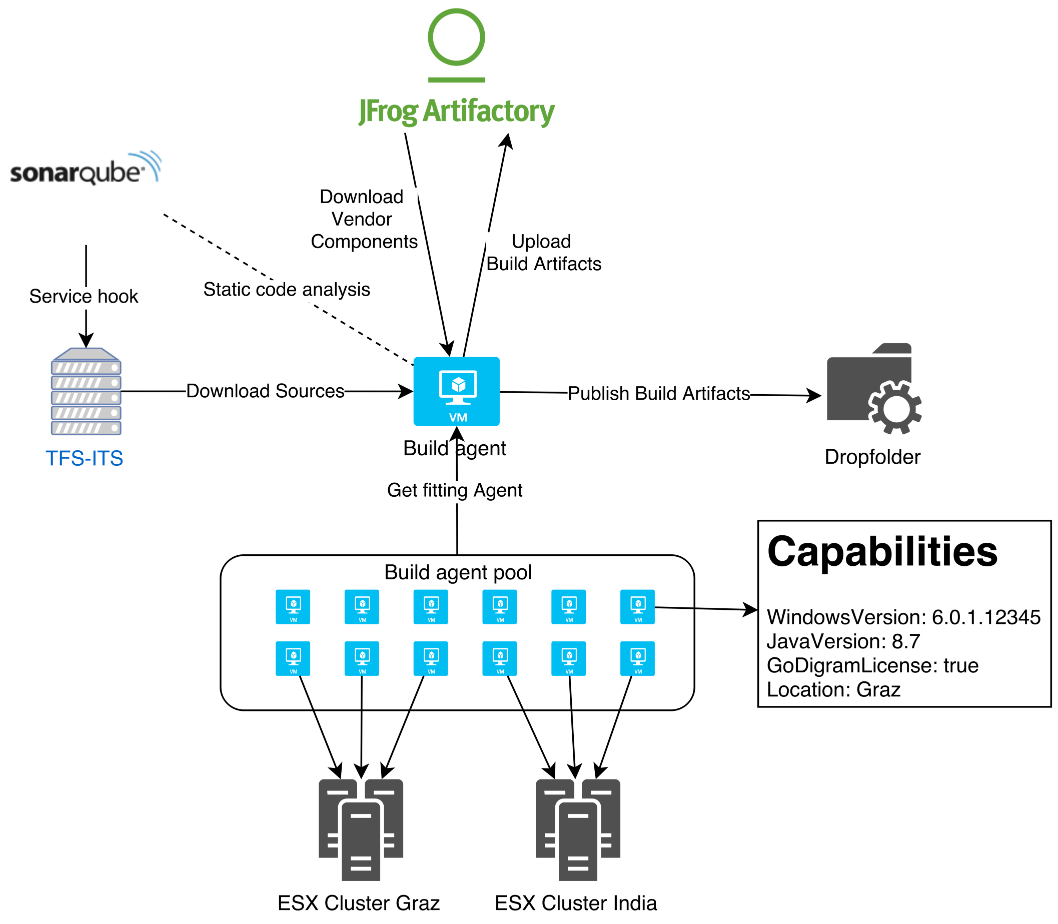


Figure 3.2: AVL Build Infrastructure

Agent pools can be shared over various collections and team projects.

- **Queues:** Queues provide access to particular Agent Pools. On creation of a build definition, an agent pool is chosen. This selection can be changed at any time.

3.2 Build Infrastructure

Following the design patterns previously presented in Section 3.1, a new build infrastructure was constructed. Figure 3.2 shows the current infrastructure at AVL (as a result of an iterative improvement process over the last few years).

The most important thing to mention regarding the AVL build infrastructure is that the *BirdIsland*[4] agent pool consists only of virtual machines which are hosted from two ESXi[5] clusters, one located in Graz and one in Gurgaon (India). For the pool administration and for the actual build process, there is no difference in the workflow if a build is started in Graz or in Gurgaon. As long as the capabilities are met, the build can run successfully on both sites. Despite this, it is simply

[4]Name was derived from the system team name, it is the home of TeamChicken.

[5]"VMware ESXi (formerly ESX) is an enterprise-class, type-1 hypervisor developed by VMware for deploying and serving virtual computers. As a type-1 hypervisor, ESXi is not a software application that one installs as an operating system (OS); instead, it includes and integrates vital OS components, such as a kernel.", https://en.wikipedia.org/wiki/VMware_ESXi

more convenient for the teams that the location where a build is executed, and from where the output can be accessed, is selected. Most of the time a builds serve a purpose and the build output is accessed by the team members. To identify the location of the build agents, the *Location* attribute is provided.

The workflow when a build is triggered is as follows:

1. From the build agent pool a build agent is requested which meets the capabilities defined in the build definition. When an agent is free, it is marked as reserved and assigned to execute the build.
2. The sources are downloaded to the agent's working directory and the steps defined in the build definition are executed.
3. The results of the build process are *published* as *Build Artifacts* to *Dropfolder*.

The *Dropfolder* is a central store for data exchange. Depending on the number of build definitions and their retention settings, the needed storage capacity might be rather high. Therefore, the Dropfolder is set up as a share[6] hosted on storage provided by the central IT of AVL. There is also an auto-extend setting which automatically increases the available capacity in case limits are reached.

Besides the standard build execution shown in Figure 3.2, the two tool service hooks are also shown as part of the build process. Service hooks are the way third party tools can be integrated into the TFS environment and can then be used as part of the build process:

1. **SonarQube[7]:** A tool used for static code analysis. SonarQube runs on its own server with its own database. SonarQube integrates into the build process by having a *Begin Analysis* and an *End Analysis* build step. The actual build and/or unit tests are wrapped from the *Begin/End Analysis* steps (as shown in Figure 3.3), which also define the scope of the static code analysis. Results and trends can then be checked on the SonarCube server.
2. **Artifactory:** See Section 5.3.

3.3 Build Agent Installation

As the teams get used to the new branching concepts, running unit tests during build, having gated check-in builds, continuously running verification builds, and last but not least, adding more code, with more people eventually more build agents are needed to satisfy the needs. Additionally, extending the LAB environment (see Chapter 4) also adds the need for more agents to run the deployment workflows (the same agents can be used for build and lab).

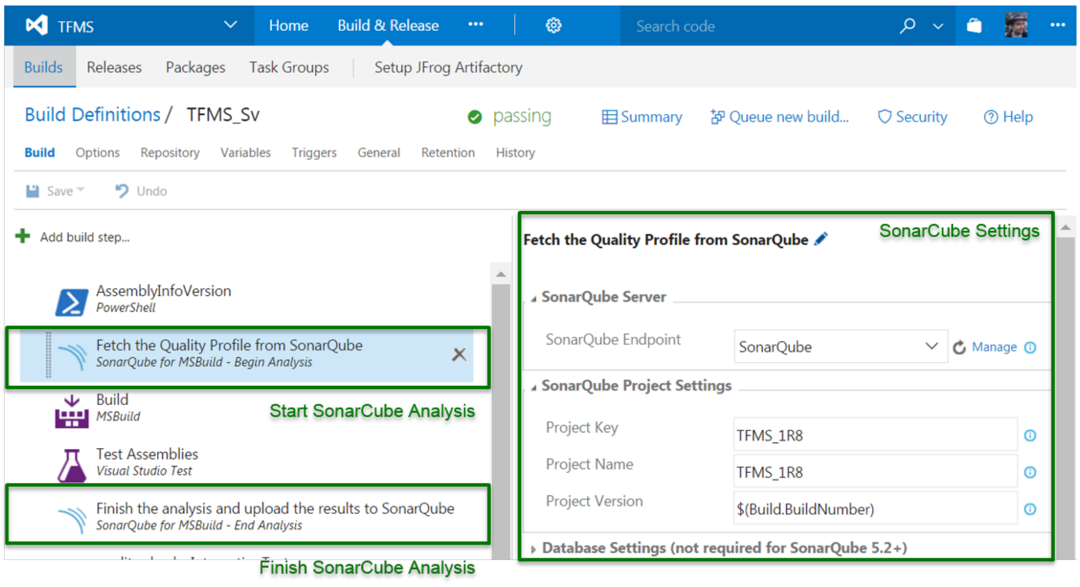
Fortunately, adding Windows build agents is not too complicated and is script based. For installing and registering a new Windows build agent[8], the following steps have to be applied:

- Extract the agent to the C-Drive
- Rename the folder to Agent_GUID

[6]Running on machine: atgrzso1199. Share can be accessed via: \\avl01\atgrz\misc\TFS_Dropfolder. Build outputs are dropped under the following folder structure: <root>\<Builddefinition Name>\<Buildnumber>\<artifactNames>

[7]https://www.sonarqube.org/

[8]The latest version of the build agent can be downloaded from Microsoft: http://go.microsoft.com/fwlink/?LinkID=829054



AssemblyInfoVersion
PowerShell

Fetch the Quality Profile from SonarQube
SonarQube for MSBuild - Begin Analysis

Build
MSBuild

Test Assemblies
Visual Studio Test

Finish the analysis and upload the results to SonarQube
SonarQube for MSBuild - End Analysis

Fetch the Quality Profile from SonarQube

SonarQube Server

SonarQube Endpoint: SonarQube

SonarQube Project Settings

Project Key: TFMS_1R8

Project Name: TFMS_1R8

Project Version: $(Build.BuildNumber)

Database Settings (not required for SonarQube 5.2+)

Figure 3.3: SonarQube steps wrapping the actual Build process

- Call config.cmd from an elevated command line
- Provide the following information

  Server-URL: http://tfs-its:8080/tfs

  Authentication type: Hit Enter (nothing to be specified)

  Agent-Pool: BirdIsland

  Agent-Name: <Machinename>_<GUID>

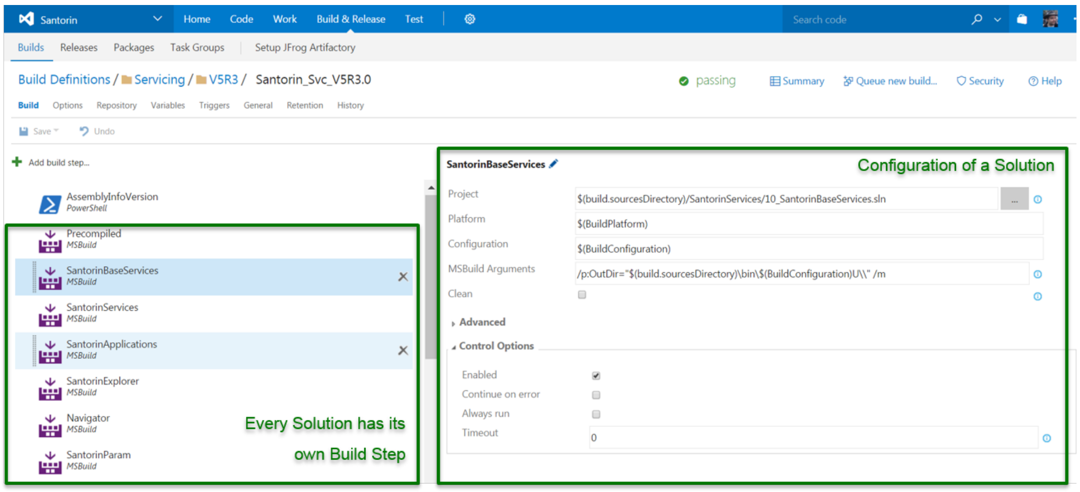
  Work folder: D:\Agent_GUID

  Run as Service: Y

  Provide Username and Password: user avl01\s18f30 is used
- In case the build machine is **not located in Graz**, the build agent needs to be configured to use the TFS proxy on site. Therefore, the Windows registry[9] of the build machine needs to be adapted. The correct settings (for an Indian build machine) are prepared at the location `<dropfolder>\_HowToSetUpABuildMachine\99_Proxy Configuration \IndiaProxy.reg`.

During the operation of a build agent, a lot of log files and temporary files are created but not necessarily always cleaned up properly (e.g. due to failing builds, common temporary folders, build behaviour, etc). It is good practice to clean up the build machine and build agent respectively on a regular basis. Listing 3.3 shows a possible solution for cleaning up the build machine. In its current state, the scripts remove all folders which are known to be generated or filled during the build process. Additionally, this script can be triggered as part of a scheduled nightly task to clean up the build machine. Both the PowerShell script and the scheduled task definition are prepared and located at `<dropfolder>\_HowToSetUpABuildMachine\99_Cleanup Scripts\CleanupBuildmachine.ps1`, and `CleanupBuildmachine.xml`.

[9]Configure Team Foundation Build Service to Use Team Foundation Server Proxy, https://msdn.microsoft.com/en-us/library/cc716770(v=vs.120).aspx



Build Definitions / Servicing / V5R3 / Santorin_Svc_V5R3.0

AssemblyInfoVersion *PowerShell*
Precompiled *MSBuild*
SantorinBaseServices *MSBuild*
SantorinServices *MSBuild*
SantorinApplications *MSBuild*
SantorinExplorer *MSBuild*
Navigator *MSBuild*
SantorinParam *MSBuild*

SantorinBaseServices

Project	$(build.sourcesDirectory)/SantorinServices/10_SantorinBaseServices.sln
Platform	$(BuildPlatform)
Configuration	$(BuildConfiguration)
MSBuild Arguments	/p:OutDir="$(build.sourcesDirectory)\bin\$(BuildConfiguration)U\\" /m
Clean	☐

Advanced

Control Options

Enabled	✓
Continue on error	☐
Always run	☐
Timeout	0

Figure 3.4: Build definition with several solutions

```
Remove-Item -Path C:\Windows\Temp\ -Recurse -Force -ErrorAction SilentlyContinue -Verbose
Remove-Item -Path $env:TEMP -Recurse -Force -ErrorAction SilentlyContinue -Verbose
Remove-Item -Path "C:\Users\s18f30\AppData\Local\Temp\" -Recurse -Force
           -ErrorAction SilentlyContinue -Verbose
Remove-Item -path "C:\Windows\Microsoft.NET\Framework\v4.0.30319\Temporary ASP.NET Files"
           -Recurse -ErrorAction SilentlyContinue -Verbose
Remove-Item -Path "C:\Users\pbuild\AppData\Local\JetBrains\Transient\InspectCode"
           -Recurse -ErrorAction SilentlyContinue -Verbose
function DeleteFolder($buildAgents)
{
    foreach($buildAgent in $buildAgents)
    {
        $diagFolder = $buildAgent.FullName+"\_diag"
        if(Test-Path $diagFolder)
        {
            Remove-Item $diagFolder -Recurse -Force -ErrorAction SilentlyContinue
                                      -Verbose
        }
    }
}
DeleteFolder (Get-ChildItem C:\ -Filter Agent_*)
DeleteFolder (Get-ChildItem C:\ -Filter GRZ_*)
DeleteFolder (Get-ChildItem C:\ -Filter GRZ*)
```

Listing 3.3: CleanupBuildMachine.ps1

3.4 Building

Creating and maintaining build definitions[10] is well documented and the current implementation follows all the suggested guidelines from Microsoft. Figure 3.4 shows a build section of an example build definition. Every solution has its own build step and can be configured separately.

Although this is quite straightforward, it does have some major drawbacks:

[10]Create and queue a build definition, https://www.visualstudio.com/en-us/docs/build/define/create

- **No parallel build support:** Large software projects tend to have a lot of component solutions. These component solutions do not necessarily need to be built in a consecutive order rather than in parallel.
- **Solution and Build order are not part of source control:** When adding/removing/changing solutions, the build definition also has to be adapted. This cannot be done in one step. When these changes are propagated through the branches, the build definition always needs to be adapted. This leads to a situation where nobody changes solutions due to the effort involved.
- **Cannot be executed locally:** The logic how the project is built is stored in the build definition. To perform a local build, all solutions have to be executed in the same order (manually or scripted). This generates extra effort.

To summarise, it does not scale well. To overcome these limitations, an MSBuild[11] project file can be used. Listing 3.4 partially shows such an MSBuild project file used in the TFMS build process. All solutions are grouped within targets and executed `<Target/>` by `<Target/>` based on the dependencies set:

1. `<Target Name="PreBuild">`**:** Has no dependencies and is executed first. The single `<MSBuild Projects = "$(SolutionRoot)TfmsPrePostBuild\PreBuildStep.sln"` solution is executed.
2. `<Target Name="TfmsGeneralServices"`**:** Dependency is set to `PreBuild` and the single solution is executed only after the first solution has finished.
3. `<Target Name="TfmsServices"`**:** Dependency is set to `TfmsGeneralServices` and is only executed once the first and second solutions have built. This target contains multiple solutions. Since there are no dependencies between the `<MSBuild/>` nodes, they are **executed in parallel**.
4. ... and so on.

[11]The Microsoft Build Engine (MSBuild) is the build platform for .NET and Visual Studio.

```xml
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="TfmsLab" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <SolutionRoot>$(MSBuildProjectDirectory)\</SolutionRoot>
  </PropertyGroup>

  <Target Name="PreBuild">
    <MSBuild Projects = "$(SolutionRoot)TfmsPrePostBuild\PreBuildStep.sln" Targets="Rebuild" Properties="Configuration=Release" />
  </Target>

  <Target Name="TfmsGeneralServices" DependsOnTargets="PreBuild">
    <MSBuild Projects = "$(SolutionRoot)TfmsGeneralServices\TFMSGeneralServices.sln" Targets="Rebuild" Properties="Configuration=Release" />
  </Target>

  <Target Name="TfmsServices" DependsOnTargets="TfmsGeneralServices">
    <MSBuild Projects = "$(SolutionRoot)TfmsServices\TfmsServices_10_All.sln" Targets="Rebuild" Properties="Configuration=Release" />
    <MSBuild Projects = "$(SolutionRoot)TfmsServices\TfmsServices_20_WSTestClient_Tools.sln" Targets="Rebuild" Properties="Configuration=Release" />
    <MSBuild Projects = "$(SolutionRoot)TfmsServices\TfmsServices_30_TestClient.sln" Targets="Rebuild" Properties="Configuration=Release" />
    <MSBuild Projects = "$(SolutionRoot)TfmsServices\TfmsServices_40_PostBuild.sln" Targets="Rebuild" Properties="Configuration=Release" />
  </Target>

  <Target Name="TFMS_10" DependsOnTargets="TfmsServices">
    <MSBuild Projects = "$(SolutionRoot)TFMS\TFMS_10_Interfaces.sln" Targets="Rebuild" Properties="Configuration=Release" />
  </Target>
    <!-- A lot of other solutions snipped -->
</Project>
```

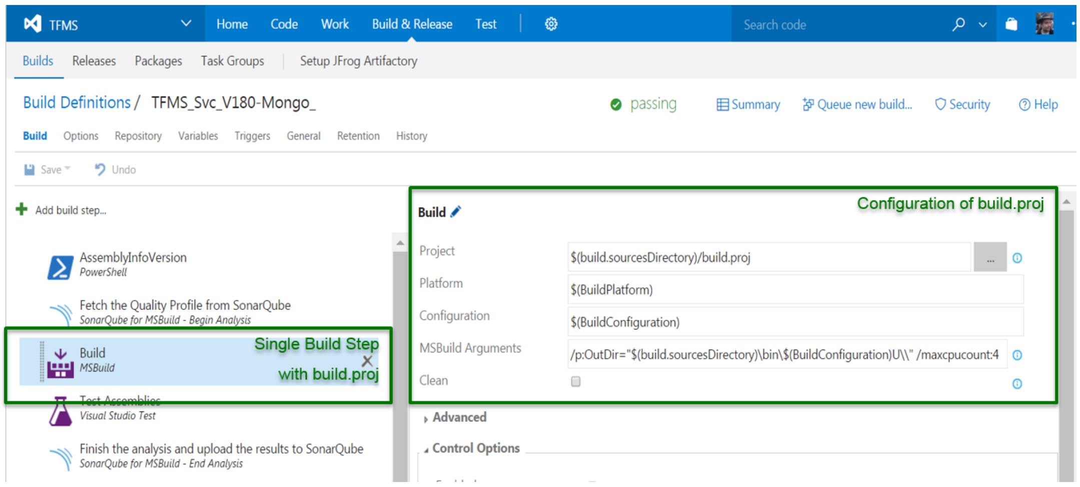
Listing 3.4: Example Build.proj file

Another advantage of using an MSBuild project file is that it is checked-in as part of the source control. Teams can change the build order, add/remove new solutions, and build locally much easier. Even gated check-ins are possible along with the changeset. Figure 3.5 shows the cleaned up build definition using the MSBuild project file. All build steps which dealt with particular solutions are simplified into one build step which executes the *build.proj* file.

3.5 Build Numbering Convention

The old build process used a proprietary binary versioning scheme which actually only relied on a single build number. With the introduction of the enhanced branching model (see Section 2.1.5), a different version number scheme was used to identify different released versions.

Figure 3.6 illustrates the logical continuation of the *enhanced branching pattern* in the case where there are already three software versions released and the fourth is in current development stage. `Release_V1`, `Release_V2`, and `Release_V3` are already in regular maintenance mode. These branches represent the maintenance branches for future service releases of the particular



Build Definitions / TFMS_Svc_V180-Mongo_ — passing

AssemblyInfoVersion — *PowerShell*
Fetch the Quality Profile from SonarQube — *SonarQube for MSBuild - Begin Analysis*
Build — *MSBuild*
Test Assemblies — *Visual Studio Test*
Finish the analysis and upload the results to SonarQube — *SonarQube for MSBuild - End Analysis*

Build

Project	$(build.sourcesDirectory)/build.proj
Platform	$(BuildPlatform)
Configuration	$(BuildConfiguration)
MSBuild Arguments	/p:OutDir="$(build.sourcesDirectory)\bin\$(BuildConfiguration)U\\" /maxcpucount:4
Clean	☐

Figure 3.5: Build definition with build.proj file

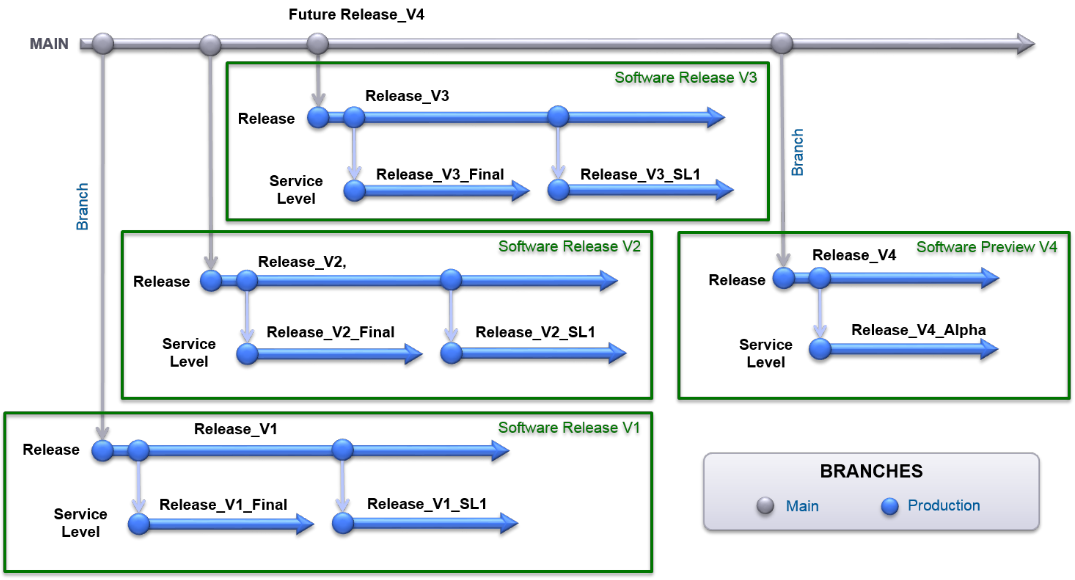


Figure 3.6: Multiple Software Releases with multiple Servicelevels

software versions. Below `Release_V1`, `Release_V2`, and `Release_V3` are the already released servicelevels arranged. In addition, `Release_V4` is currently in development on `Main` branch but pre-releases (alpha, beta, release candidates, etc.) can be maintained using the already created `Release_V4` and `Release_V4_Alpha` branches. For this branching pattern, proper versioning is needed.

The binary version number scheme is based on the standard[12] provided by Microsoft and consists of four parts taking the format `major.minor.build.revision`. All four parts must be positive integers and need to be provided. They have the following meaning:

- **Major:** This part indicates *major differences* between two assemblies with the same name but different major version. A change in the major version is often accompanied with a change of interfaces of the assembly. Backwards compatibility cannot be guaranteed.

[12]Version Class, https://msdn.microsoft.com/en-us/library/system.version(v=vs.110).aspx

- **Minor:** A different minor version indicates that enhancements were introduced with the assembly. These *minor* changes do not break the backwards compatibility of the assembly.
- **Build:** The build part is increased daily and follows the following notation: `-.<LastTwoDigitsOfYearCount><DayOfTheYear>.-` e.g. the build number part on 1st January 2017 is `-.17001.-`, for 31st January 2017 is `-.17001.-`, ... until 31st December 2017 with `-.17365.-`. Coding this time information into the build number is not a necessity but it turned out to be rather convenient in order to immediately identify *the age* of a particular assembly.
- **Revision:** The revision part identifies the daily build number. It is increased with every build and is reset at midnight. The first build of the year 2017 is `-.17001.1`, the second `-.17001.2` and so on.

The combination of all these four parts provides the binary version number which is *strictly monotonous increasing*[13]. In order to compare two binary versions, they have to be compared part-wise beginning with the major part.

The product versioning scheme used at AVL is based on a semantic versioning scheme[14]. The product naming and versioning format used follows the following notation:

`AVL <Product> <Generation>`^{TM}`R<Release>.<Servicelevel>.<Patchlevel>`

The placeholders have the following meaning:

- **Product:** The name of the product line is provided here e.g. *SANTORIN Host, SANTORIN MX, TFMS, Concerto, ...*
- **Generation:** This is the first part of the semantic versioning. The generation identifies big steps within the software product in respect to architectural changes, and/or different technologies with no compatibility. The generation version changes in a timeframe of 5-10 years.
- **Release:** A release signifies a new big increment of the software product. Compatibility is only given in defined areas. New releases are done on a yearly basis.
- **Servicelevel:** Service releases are mainly maintenance releases in order to ship bugfixes (in special occasions, also new features) to the customer. Compatibility within the release is mandatory. Service releases are scheduled every 10-12 weeks.
- **Patchlevel:** Patches are provided from servicelevel branches in order to provide fast bugfixes. They are created irregularly and depend on the priority of the customer requests. Patches are always cummulative.

The product version number does not change with every build but rather with every release and needs to be set manually. Figure 3.7 shows an example (based on SANTORIN) of how the *product versioning* and *binary versioning* can be applied to the given branching pattern. The following details in respect to the binary build number assignments are important:

- **Major:** The major version is set to **5** which corresponds to the generation *Santorin Host* ***5*** of the software product.

[13]Every build has its unique build number and the build number increases with every build. There are now builds with the same build number.

[14]Versioning indicates the compatibility of the APIs provided, http://semver.org/spec/v2.0.0.html

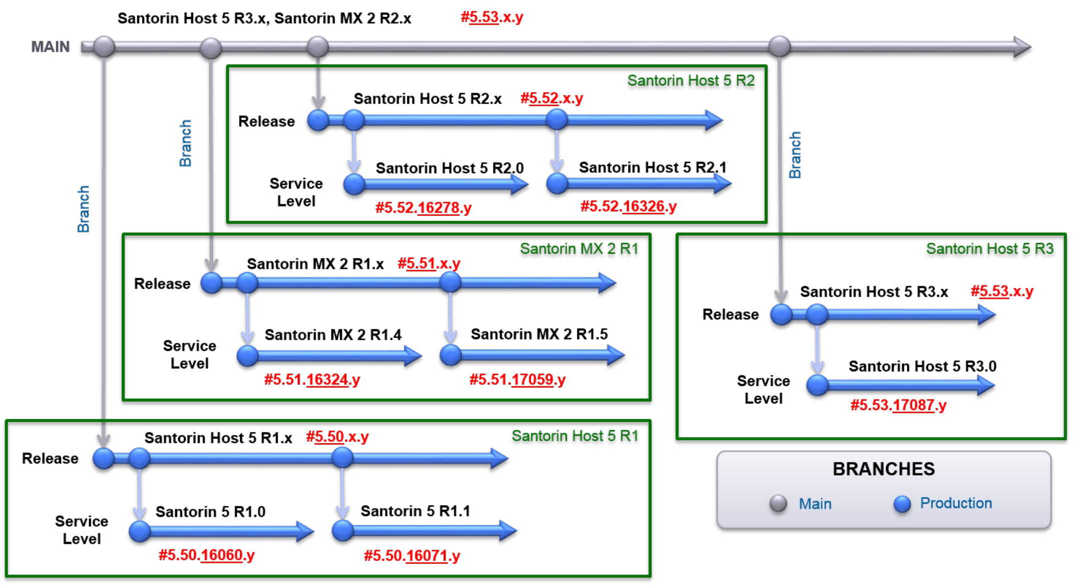


Figure 3.7: Example Santorin: Naming with associated Build Numbers

- **Minor:** The minor version changes with every software release e.g. `R1 -> .50.`, `R2 -> .52.`, `R3 -> .53.`.
- **Build:** The build version gets frozen on the service level.
- **Revision:** The revision is the only part which changes on the servicelevel in order to identify a particular patchlevel.

In addition to Figure 3.7, the Table 3.1 contains the Product/Binary version pairs (with patch-levels).

Product version	**Binary version**	**Remarks**
Santorin Host 5 R1.0	5.50.16060.11	—
Santorin Host 5 R1.1	5.50.16071.10	was released eleven days after R1.0
Santorin Host 5 R1.1.1	5.50.16071.18	—
...	...	—
Santorin Host 5 R1.1.7	5.50.16071.58	latest (7th) cumulative patch for R1.1
Santorin Host 5 R2.0	5.52.16278.3021	first release of R2.0
Santorin Host 5 R2.0.1	5.52.16278.3051	—
Santorin Host 5 R2.1	5.52.16326.10134	—
...	...	—
Santorin Host 5 R2.1.3	5.52.16326.10911	latest (3rd) cumulative patch for R2.1
Santorin Host 5 R3.0	5.53.17087.10991	first release of R3.0

Table 3.1: Example Santorin: List of Product and Binary Versions

4. Lab Management

"It is done, okay it is checked in, I'll test it immediately ... oh, a butterfly."

— Andreas Kappel, The young wolf playing on the green spring meadows.

To recap, the main goal of the tool transformation in the previous two chapters, *Source Control 2* and *Build System 3*, was to implement an infrastructure which applies the principles of *Continuous Integration (CI)*. As a development practice, CI enables developers to integrate their code several times a day. These check-ins are validated immediately by the automated build system, which also includes automated unit testing. Problems/errors can be detected at an early stage in the development and can be located more easily.

Depending on the literature, the term *Continuous Integration (CI)* may not only mean source and build verification, it can also include integration tests on systems which are similar to production environments. In any case, when *Continuous Delivery (CD)* and/or *Continuous Deployment (CD^+)* are the goals, there is no way around having an infrastructure providing a large variety of systems. Figure 4.1 (inspired by [9]) illustrates the idea behind *Continuous Delivery / Continuous Deployment*. Both terms have the same goal: Deploying a product to the production environment and guaranteeing its functionality[10].

Once every code change is deployed to numerous staging environments, tested against various quality parameters, and everything is automated, the confidence in deploying the product automatically to a production environment is high. The question is who exactly should be allowed to deploy to this production environment. The goal, of course, should be in having a CD^+ workflow running where every step is automated, including the decision to deploy to the production environment. On the other hand, there are business use-cases where an automatic deployment is not suitable due to

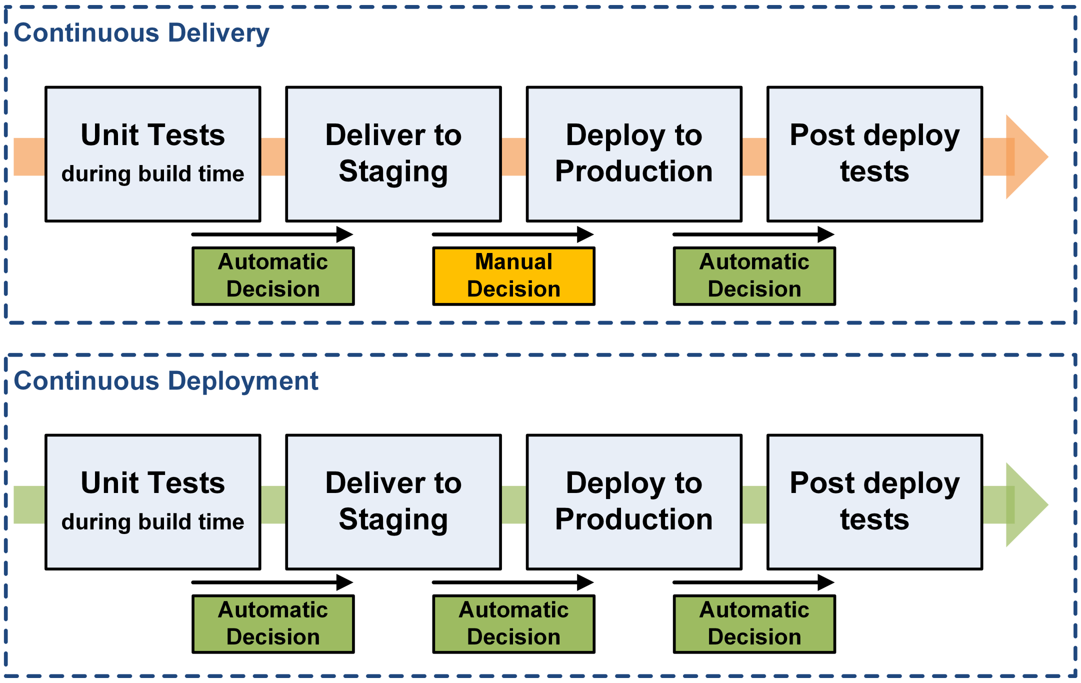


Figure 4.1: Continuous Delivery vs. Continuous Deployment

Service Level Agreements (SLAs)[1] and missing *Feature Toggles*[2].

In order to build up a staging environment, the *Lab Environments / Lab Management* was introduced as part of the development process. Therefore, the TFS *Release Management* [11] service module is used to create workflows. These workflows can be linked to particular builds as event triggers, which allows creating/reverting staging environments (using VMWare ESXi Clusters), and running installation and test scripts, whose results can then be shown at different stages.

To get an overview, Figure 4.2 illustrates all building blocks within the current AVL *Lab Management* solution:

- **Release Definition:** The Release Definition is the fundamental concept within TFS. It defines the whole workflow to be executed when new builds are ready to be tested. The release agent is the actual worker thread which executes the workflow (build agents can be re-used as release agents).
- **VMWare vSphere:** The VMWare vSphere ESXi Cluster provides the virtual machines needed for the various staging environments and the interfaces required in order to create and revert the virtual machines.

  Physical Machines: Physical machines can also be used, but the machine state has to be managed manually or taken into account by the deployment scripts. They are mainly used to

[1]"A service level agreement (SLA) is defined as an official commitment that prevails between a service provider and the customer. Particular aspects of the service quality, availability, and responsibilities are agreed between the service provider and the service user.", https://en.wikipedia.org/wiki/Service-level_agreement, https://www.paloaltonetworks.com/cyberpedia/what-is-a-service-level-agreement-sla

[2]"This technique allows developers to release a version of a product that has unfinished features. These unfinished features are hidden (toggled) so they do not appear in the user interface. This allows many small incremental versions of software to be delivered without the cost of constant branching and merging.", https://en.wikipedia.org/wiki/Feature_toggle

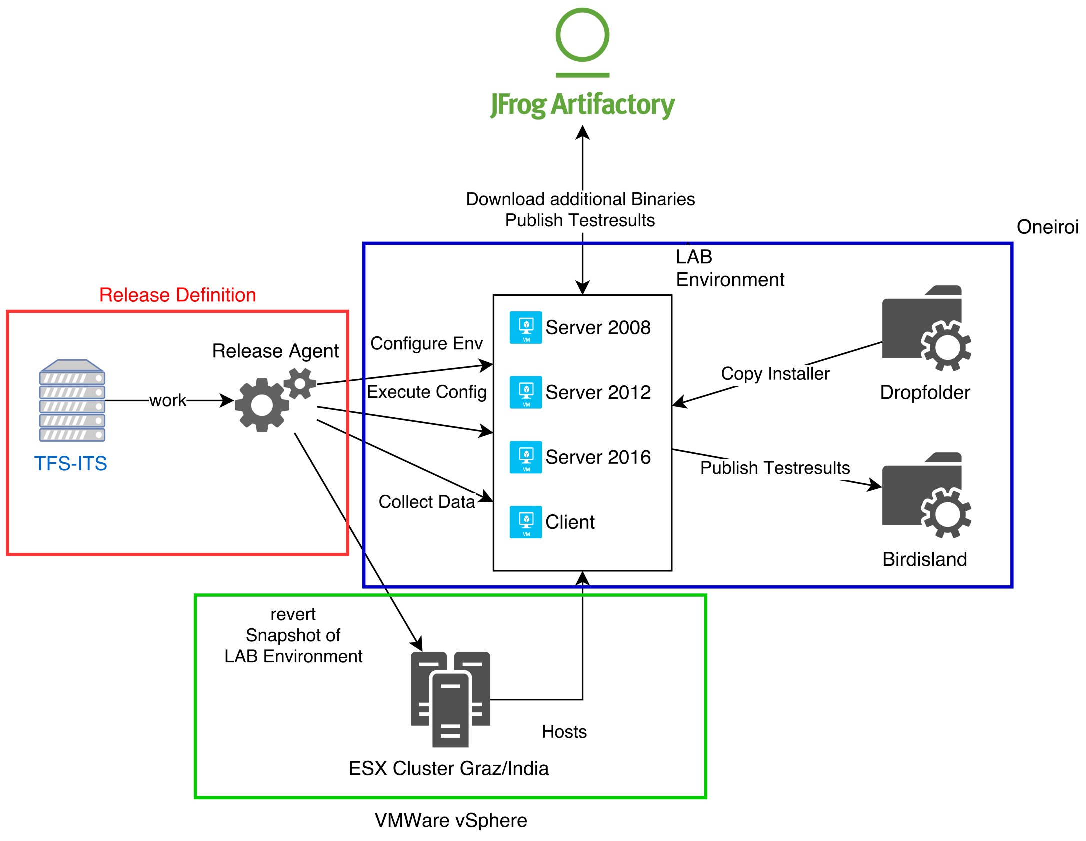


Figure 4.2: Lab Management Infrastructure

run testbed simulators which cannot be run on virtual environments (due to constraints of the real time operating system).

- **Oneiroi:** Oneiroi is the working title of all PowerShell scripts which are used on the particular virtual machines. The virtual machines are reverted to a machine state where no product software or other prerequisites are installed. Depending on the settings provided by the *Release Definition*, the scripts configure the machine, install the software packages, run automated tests, and publish the results.

4.1 Release Definition

The *Release Definition* [12] is the central configuration point. Figure 4.3 shows an example Release Definition for the TFMS product:

- **Stages / Environments:** A stage/environment defines a group of machines which are logically bound together within the stage. All environment specific variables, like machine names to be used, are defined here. A Release Definition can contain multiple stages. Each Stage has its own specific purpose. The Release Definition in Figure 4.3 contains two environments.

  The *Development (Dev)* environment is a highly volatile environment which gets updated whenever a new build is available from the TFMS Main branch and the previous deployment has finished. All automated integration tests are run.

  The *Quality Assurance (QA)* environment is the second stage which has a high availability and deployments are only done when a certain level of stability and content is reached on the development (Dev) environment. This stage is mainly used for manual testing and for System Demos[3].

Additional stages are currently not used but are already in development in order to have additional environments which are closer to customer environments, have realistic database dumps, and furthermore improve the test coverage.

- **Automation Tasks:** A list of tasks and task groups are executed on every machine within the environments. Depending on the scripting (see Section 4.3), actions might be executed differently.

  Task groups: Task groups provide the possibility to group together a selection of steps into a single *reusable* task group. Task groups can then be added to the workflow like any other step to the Release Definition. Currently, the *Download Scripts* and the *Revert Snapshot* task groups are available. These two task groups actually hide some network domain peculiarities of AVL. After reverting virtual machines, it is not always guaranteed that the machines are correctly attached to the AVL domain. Therefore, additional steps for flushing DNS caches are added and all machines are iterated over in order to detect if they are really available within the AVL network.
- **Settings**: Every task has settings to be provided, e.g. settings for the task group *RevertSnapshot*. This particular AVL specific task reverts all machines provided as variable names e.g. `$(Server2008), $(Server2012), ...` (defined on the environment) to a defined state.
- **Artifacts:** A *Release Definition* can take several input artifacts. In this TFMS example, the Release Definition is linked to the *LabRuntime-Artifact* of a particular TFMS build. The *LabRuntime* consists of all install packages created from the build. When a *Release* is created, the versions of the particular artifacts can no longer be changed. This means that at every stage the same artifact package is evaluated.

  Within a Release Definition not only artifacts from a *Build vNext* build can be used. When registered to TFS as a service, other build services like TeamCity and Jenkins can also be used. In other scenarios, artifacts also stored in version control systems like Git can be consumed, processed, and validated.

4.1.1 TFMS Release Definition in Detail

For a better understanding and overview, the six steps within the TFMS release definition are elaborated below:

- **Variables for Release Definition:** For the Release Definition only the PowerShell scripts of *Oneiroi* (see Section 4.3) are parameterised.
- **Variables for Development Environment:** In Figure 4.4, all the machines which are used in this stage are listed by name. In addition, the `Variant` here, `1R5_Stage01_Dev` defines the type of the environment. This variable is used to identify within the Oneiroi scripts which actions are to be performed. The variables `BuildType, MetaVersion, and Stage` are

[3]"The purpose of System Demos is to test and evaluate the full system and to get feedback from the primary stakeholders of the solution under development.", http://www.scaledagileframework.com/system-demo/

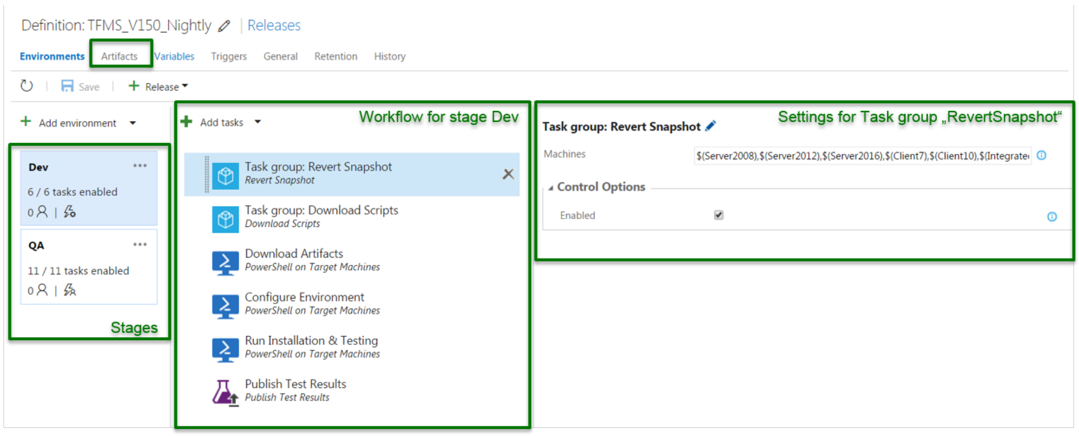
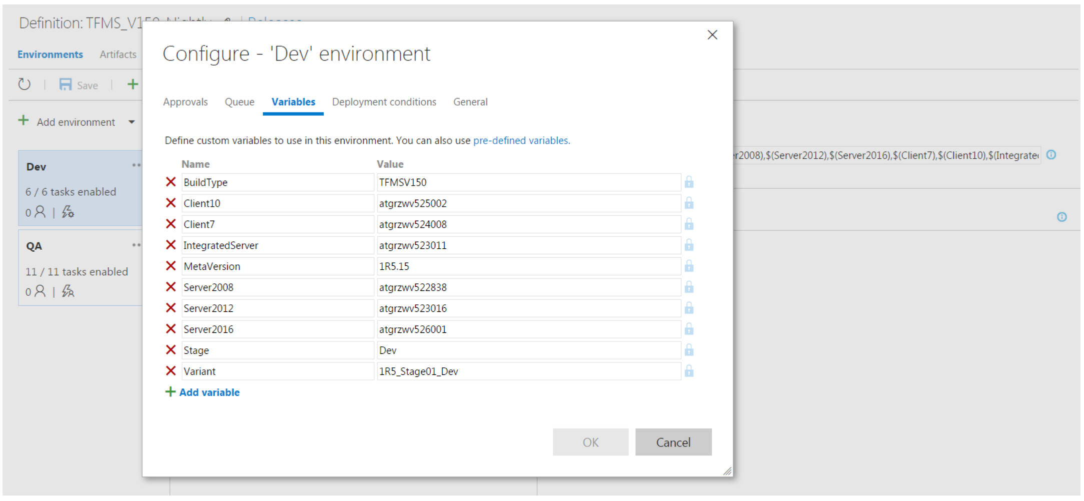

Definition: TFMS_V150_Nightly
Releases
Environments
Artifacts
Variables
Triggers
General
Retention
History
Save
Release
Add environment
Dev
6 / 6 tasks enabled
QA
11 / 11 tasks enabled
Stages
Add tasks
Workflow for stage Dev
Task group: Revert Snapshot
Revert Snapshot
Task group: Download Scripts
Download Scripts
Download Artifacts
PowerShell on Target Machines
Configure Environment
PowerShell on Target Machines
Run Installation & Testing
PowerShell on Target Machines
Publish Test Results
Publish Test Results
Task group: Revert Snapshot
Settings for Task group „RevertSnapshot“
Machines
$(Server2008),$(Server2012),$(Server2016),$(Client7),$(Client10),$(Integrate
Control Options
Enabled

Figure 4.3: Example Release Definition in TFMS


Configure - 'Dev' environment

Approvals | Queue | **Variables** | Deployment conditions | General

Define custom variables to use in this environment. You can also use pre-defined variables.

Name	Value
BuildType	TFMSV150
Client10	atgrzwv525002
Client7	atgrzwv524008
IntegratedServer	atgrzwv523011
MetaVersion	1R5.15
Server2008	atgrzwv522838
Server2012	atgrzwv523016
Server2016	atgrzwv526001
Stage	Dev
Variant	1R5_Stage01_Dev

Figure 4.4: Variables for Development Environment in TFMS

already deprecated but for compatibility reasons are still included.

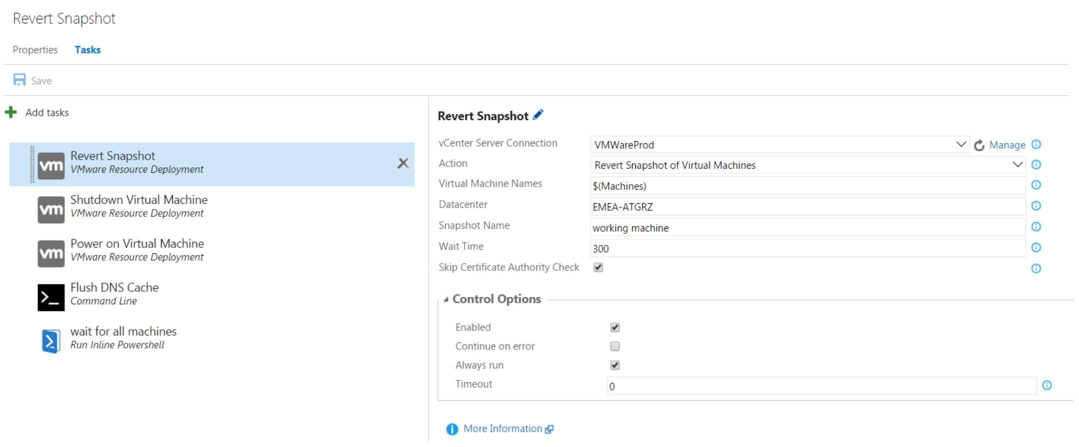
- **Task Group Revert Snapshot:** In Figure 4.5, the five steps of this task group are shown.

  Revert Snapshot: This VMWare task reverts the specified machine on the VMWare ESXi cluster.

  Shutdown Virtual Machine and Power On Virtual Machine: These two steps perform a reboot of the machines in case there are some network connectivity problems which can be solved by a simple reboot (e.g. address resolution issues).

  Flush DNS Cache: This command shell command is executed on the newly restarted machine in order to resolve DNS cache resolution problems.

  PowerShell script *WaitForMachines*: This inline PowerShell script (shown in Listing 4.1) waits until every machine is really accessible from the release agent.


Revert Snapshot

Properties Tasks

Save

Add tasks

Revert Snapshot
VMware Resource Deployment

Shutdown Virtual Machine
VMware Resource Deployment

Power on Virtual Machine
VMware Resource Deployment

Flush DNS Cache
Command Line

wait for all machines
Run Inline Powershell

Revert Snapshot

vCenter Server Connection	VMWareProd
Action	Revert Snapshot of Virtual Machines
Virtual Machine Names	$(Machines)
Datacenter	EMEA-ATGRZ
Snapshot Name	working machine
Wait Time	300
Skip Certificate Authority Check	✓

Control Options

Enabled	✓
Continue on error	☐
Always run	✓
Timeout	0

More Information

Figure 4.5: Task Group *Revert Snapshot*

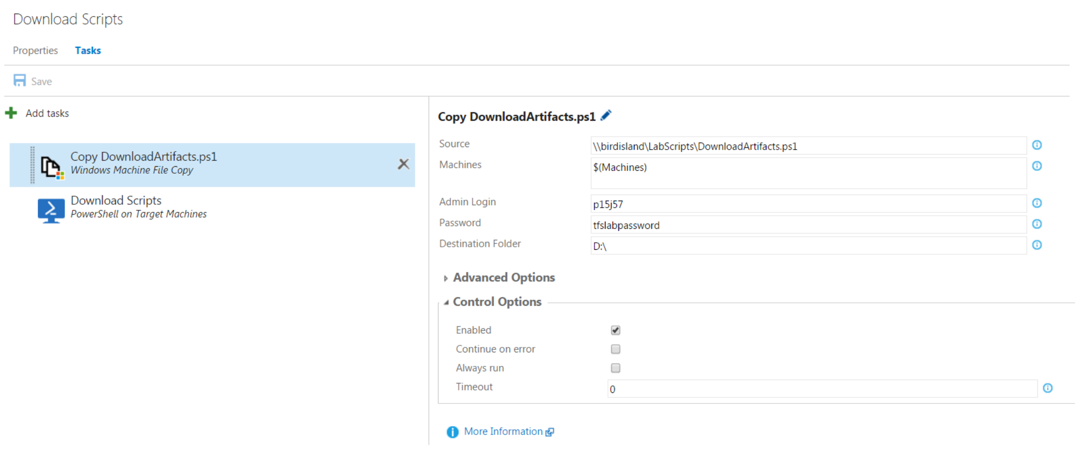
```
param
(
    [string[]]$Computername
)
$time = [System.Diagnostics.Stopwatch]::StartNew()
$password = ConvertTo-SecureString "tfslabPW1!" -AsPlainText -Force
foreach($computer in $Computername)
{
    while(!(Test-Path \\$computer\c$))
    {
        Write-Output "$computer Not available"
        Start-Sleep 1
    }
}
"Elapsed: {0:HH:mm:ss}" -f ([datetime]$time.Elapsed.Ticks)
$time.Stop()
```

Listing 4.1: PowerShell script `WaitForAllMachines.ps1`

- **Task Group Download Scripts:** The `DownloadArtifact.ps1` is used to download the Oneiroi scripting files in order to be used in later steps. However, the script file itself also needs to be distributed to the machines themselves.

  Copy DownloadArtifact.ps1: In order to solve this chicken and egg problem, the `DownloadArtifact.ps1` is copied from a well known location, `\\birdisland\LabScripts\DownloadArtifact.ps1`, and copied to the Lab machine as shown in Figure 4.6.

  Execute DownloadArtifact.ps1: This execution downloads the Oneiroi script file package from Artifactory and extracts it to `D:\_LabRuntime`, as shown in Figure 4.7. Listing 4.2 illustrates the content of the PowerShell script file. It uses the `Invoke-WebRequest` PowerShell command to download the zipped scripting packages and extracts them to `D:\_LabRuntime\`. It's important to notice here that the script file is prepared to also be run on Indian lab machines (depending on the lab machine name, the Artifactory in Graz or in India is used).


Download Scripts

Properties | Tasks

Save

Add tasks

- Copy DownloadArtifacts.ps1 — *Windows Machine File Copy*
- Download Scripts — *PowerShell on Target Machines*

Copy DownloadArtifacts.ps1

Source	\\birdisland\LabScripts\DownloadArtifacts.ps1
Machines	$(Machines)
Admin Login	p15j57
Password	tfslabpassword
Destination Folder	D:\

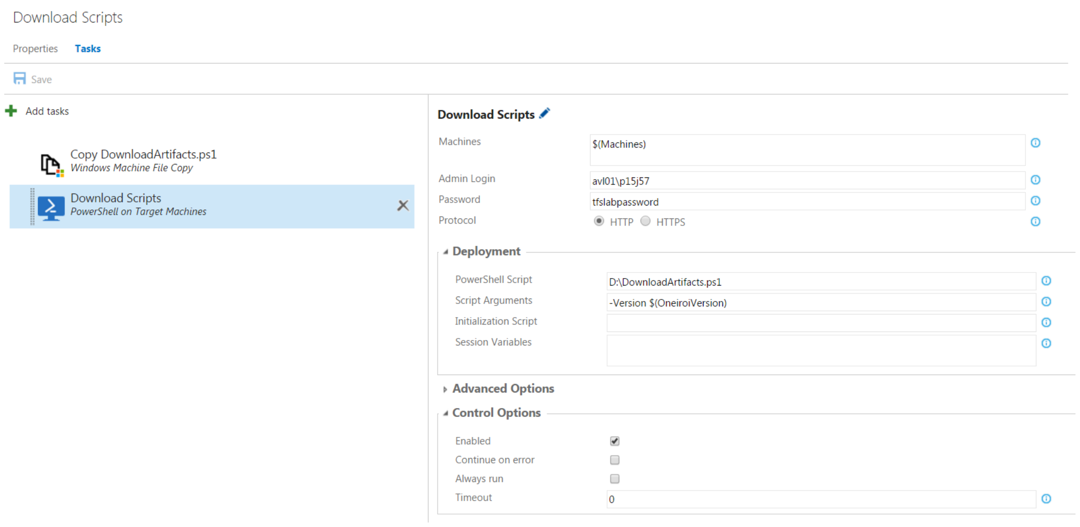
Advanced Options

Control Options

Enabled	✓
Continue on error	☐
Always run	☐
Timeout	0

More Information

Figure 4.6: Copy `DownloadArtifact.ps1` to a particular Lab machine


Download Scripts

Properties | Tasks

Save

Add tasks

- Copy DownloadArtifacts.ps1 — *Windows Machine File Copy*
- Download Scripts — *PowerShell on Target Machines*

Download Scripts

Machines	$(Machines)
Admin Login	avl01\p15j57
Password	tfslabpassword
Protocol	[● HTTP] ○ HTTPS

Deployment

PowerShell Script	D:\DownloadArtifacts.ps1
Script Arguments	-Version $(OneiroiVersion)
Initialization Script	
Session Variables	

Advanced Options

Control Options

Enabled	✓
Continue on error	☐
Always run	☐
Timeout	0

Figure 4.7: Execute `DownloadArtifact.ps1` on a particular Lab machine

Next Improvement: The dependency to birdisland needs to be removed in order to have everything within source control. Therefore, this task group will need to be replaced by inline PowerShell commands.

- **Download Artifacts:** This step downloads the artifacts linked to the Release Definition (see Figure 4.3). Normally, for accessing linked artifacts, existing pre-defined tasks are already available. These tasks, however have one drawback: artifacts are always copied via the release agent to a target machine. For the AVL use-case, those artifacts are quite large (about 10GB including all required install packages) and reserve unnecessarily large disk space on the release agents. Therefore, a custom PowerShell script is used to copy from the artifacts drop folder to the lab machine directly.

```
param
(
    [string]$Repository="repo-qa",
    [string]$Version
)
$SplitVersion = $Version.Split("{.}")
$MajorMinor = $SplitVersion[0]+"."+$SplitVersion[1]
$BuildRevision = $SplitVersion[2]+"."+$SplitVersion[3]

$uri = "https://artifactory.avl.com/artifactory/$Repository/AVL/OneSetup/$MajorMinor/
$BuildRevision/OneSetup-$MajorMinor-$BuildRevision-Oneiroi.zip"
if($env:COMPUTERNAME.startswith("INGUR"))
{
    $uri = "https://gur-artifactory.avl.com/artifactory/$Repository/AVL/OneSetup/$MajorMinor/
$BuildRevision/OneSetup-$MajorMinor-$BuildRevision-Oneiroi.zip"
}
Invoke-WebRequest -Uri $uri -OutFile D:\Oneroi.zip
Add-Type -Assembly System.IO.Compression.FileSystem
Remove-Item "D:\_LabRuntime\" -Recurse -Force -ErrorAction SilentlyContinue
[System.IO.Compression.ZipFile]::ExtractToDirectory("D:\Oneroi.zip", "D:\_LabRuntime")
Remove-Item "D:\Oneroi.zip" -Recurse -Force -ErrorAction SilentlyContinue
```

Listing 4.2: PowerShell script `DownloadArtifacts.ps1`

- **Configure Environment:** For creating and adapting the particular machine configuration, the `ConfigureConfig.ps1` is essential. In Oneiroi, the configuration for all environments and installation variants is already predefined. In order to activate the predefined configurations, particular settings are adapted.

  `-InstallationVariant:` Sets the installation variant to `$(Variant)`

  `-ReplacesString`: Replaces the machine names `Machine=$(Machine)` within the installation variant.

  `-Stage`: The Name of the stage `$(Stage)` used for the reporting.

  `-DefinitionName:` The name of the release definition (TFS variable) `$(Release.DefinitionName)` which is also used for reporting.
- **Installation & Testing:** After all preparation steps, everything is set up and ready to simply run `Run.ps1`. Since all configuration is stored in different files, no parameters are needed for this script file. This is also very convenient for debugging purposes. As a future goal, the configuration should also be created by a wizard and, in addition, the whole installation can be done by `Run.ps1`, e.g. on customer installations.
- **Publish Test Results:** The results generated during the installation and testing phase are stored on a central network share, `\\birdisland\TestReports\Tfms\$(Release.DefinitionName)\` `$(Build.Buildnumber)\$(Stage)` `\<TimeStamp>_<ProductVersion>_<Environment>_<Machinename>_` `<NunitProjectFilename>.xml/.html`. The *Publish Test Results* step collects all results from the file share in order to store them within the created release. Additionally, an overall stage test report is created by this step for analysis purposes.

4.2 VMWare vSphere

vSphere is the visualisation platform of VMWare. At AVL, the infrastructure is provided by the central IT department. In order to administrate, the vSphere client is needed which is provided by AVL Software Center. The rights management is done by the central IT department. Currently, the following vSphere servers are available:

- **atgrzsw1616:** Productive vSphere, Version 5.x India Lab and build machines
- **atgrzso8101:** Productive vSphere, Version 6.x Graz Lab and build machines
- **atgrzso2213:** Non-productive (Playground) vSphere, Version 6.x

The VMWare vSphere infrastructure is structured in the following organisational units:

- **Data Centre:** Consists of all building blocks, e.g. virtualisation servers, data storages, IP networks, management servers, and desktop clients.
- **Cluster:** Combines multiple ESXi servers in a cluster system.
- **Hosts:** The actual ESXi machines. In the current setup, two host machines are provided, `atgrzsw2444` and `atgrzsw2492`. Both are equipped with 256 GB of RAM and an Intel Xeon E5-2690v4 CPU. The used data storage, `atgrzso1493_esx_tfs_its_02_nobck`, is configured to run without backups, and when failures occur, machine states will be lost. Particular machine states are not critical since they can be setup automatically.
- **Virtual Machine:** Virtual machines instantiated from the particular ESXi cluster.

4.2.1 Virtual Machine Templates

For every operating system, there are self-created/maintained preconfigured templates available. The naming convention follows the pattern `T-ATGRZ-HW11-<Cluster>-<OS/Usage>`. Currently, the following templates are available:

- **T-ATGRZ-HW11-TFS_ITS-Build**: Template to create a new Windows Server 2012 build machine
- **T-ATGRZ-HW11-TFS_ITS-Win10_Ent_x64_EN**: Template to create a new Windows 10 lab machine
- **T-ATGRZ-HW11-TFS_ITS-Win2008_Std_R2_SP1_x64_EN**: Template to create a new Windows Server 2008 lab machine
- **T-ATGRZ-HW11-TFS_ITS-Win2012_Std_R2_x64_EN**: Template to create a new Windows Server 2012 lab machine
- **T-ATGRZ-HW11-TFS_ITS-Win2016_Std_EN**: Template to create a new Windows Server 2016 lab machine
- **T-ATGRZ-HW11-TFS_ITS-Win7_Ent_x64_EN**: Template to create a new Windows 7 lab machine

All templates have the same local administrator account with the username `atfsits`. Various software packages like Oracle Server/Client, remote debuggers, text editors, and other tools used by the teams are installed. In order to save memory and CPU power on the ESXi Host, the default settings are 4GB RAM and two CPU cores.

Virtual Machine Template Edit

From time to time the templates need to be updated, e.g. Windows Updates need to be applied or additional software installed, changed, or removed. Therefore, the following steps are performed:

1. Select the template to be updated,
2. Select `Convert to Virtual Machine` in the context menu,
3. Select the cluster where the virtual machine should run, e.g. ATGRZ-MBK-TFS-ITS,
4. Select the resource pool to be used e.g. ATGRZ-MBK-TFS-ITS,
5. Confirm by using the `Finish` Button,
6. Start the virtual machine,
7. Update domain settings (see Section 4.2.2),
8. Shut down the virtual machine,
9. Select the virtual machine,
10. Execute `Template` and `Convert to Template` in the context menu,
11. Template update was successful.

4.2.2 Virtual Machine Settings

As already stated, the virtual machines are set up with the smallest amount of resources as possible in order to provide a large testing infrastructure. The settings are sufficient for running automated tests but most of the time it is very tedious to work with them interactively. In order to increase RAM and CPU cores, two possibilities are given:

1. **vSphere Client:** To edit a virtual machine, select `Edit Settings` in the context menu of the virtual machine and change the value of Memory and CPUs.
2. **PowerShell script `BoostMe.ps1`:** This PowerShell script calls the vSphere API directly and changes the settings on the fly. Start the PowerShell script (located on the desktop) either via the context menu `Run with PowerShell` or with the PowerShell console.

Both options only work in cases where the virtual machine (guest) operating system is either Windows 7, Windows 10, Windows Server 2008 R2, or Windows Server 2012 R2. On Windows Server 2016 it is not possible to change its settings while the VM is running.

Domain Settings

Domain admin users are allowed to add machines to the AVL01 domain. Open the system properties in vSphere and select `Change settings`. Click the Change Settings link next to “Computer name, domain and workgroup settings” in the system information window to access the system properties window, which allows you to join or leave a domain. To join the domain, enter `avl01.avlcorp.lan` in the domain text box. A dialogue will ask to provide admin credentials (e.g. a17v16).

4.2.3 Troubleshooting

Error 1396 while copying DownloadArtifacts.ps1 to Lab Machine

This error indicates that the machine has not correctly joined the domain or other IP related problems are present. There are two possible solutions to this problem:

- Recreate the virtual machine mentioned in the error message or
- Re-add the virtual machine to the AVL01 domain (see Section 4.2.2).

4.3 Oneiroi

Oneiroi[4] is the working title for all PowerShell based scripts which run directly on the lab machines. The Oneiroi scripts are part of the OneSetup source control. The scripts are located within the folder `R20Y` with the build solution `R20Y.sln`. The scripts are divided into common library scripts and AVL product specific scripts. As part of the OneSetup releases, the Oneiroi package `OneSetup-<Major>.<Minor>-<Build>.<Revision>-Oneiroi.zip` is provided as well. It contains all script files which are also part of the source control.

In Section 4.1, the workflow of a Release Definition was shown. Figure 4.8 shows this workflow from the scripting point of view. The main focus is on the following parts:

- **ConfigureConfig.ps1** This configuration PowerShell script is executed as the first step. This scrip prepares/adapts the machine for the particular machine setup, e.g. stops all Oracle services to free up memory, updates tnsnames.ora entries, and prepares the `LabConfig.xml`.
- **Installation Variants:** An *Installation Variant* is represented by an XML file which lists the tasks (and their parameters) which will be executed on an environment on various machines. All available installation variants can be found at `<R20YRoot>/AVL/Common/LAB/ConfigurationVariants/`. The structure within an installation variant is shown in Listing 4.3.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Variants>
  <RunThisTaskEverywhere Install="*" />
  <RunFirstTaskOnMachineA Install="%MachineA%" ParameterA="Value" />
  <RunSecondTaskOnMachineA Install="%MachineA%" ParameterA="Value" />
  <RunThisTaskOnMachineB Install="%MachineB%" ParameterB="Value" />
  <RunThisTaskOnMachineAandB Install="%MachineA%,%MachineB%" ParameterC="Value" />
</Variants>
```

Listing 4.3: Example installation variant XML file

- **LabConfig.xml:** This file provides all global settings within the workflow. It contains values such as the License Server, Artifactory links, database users, and passwords. The variables can be accessed as shown in Listing 4.4 using `$labConfig.LabConfig.LicenseServer`.

```powershell
[xml]$labConfig = Get-Content $PSScriptRoot\LabConfig.xml
Write-Verbose $labConfig.LabConfig.LicenseServer
```

Listing 4.4: Accessing variables in `LabConfig.xml`

- **Run.ps1:** This script is a wrapper around the installation variants definitions and the executes the methods. It loads all installation tasks from the `LabConfig.xml`. It only loads the tasks

[4]"In Greek mythology, the Oneiroi or Oneiri ("Dreams") were various gods and demigods that ruled over dreams, nightmares and oneiromatic symbols.", https://en.wikipedia.org/wiki/Oneiroi

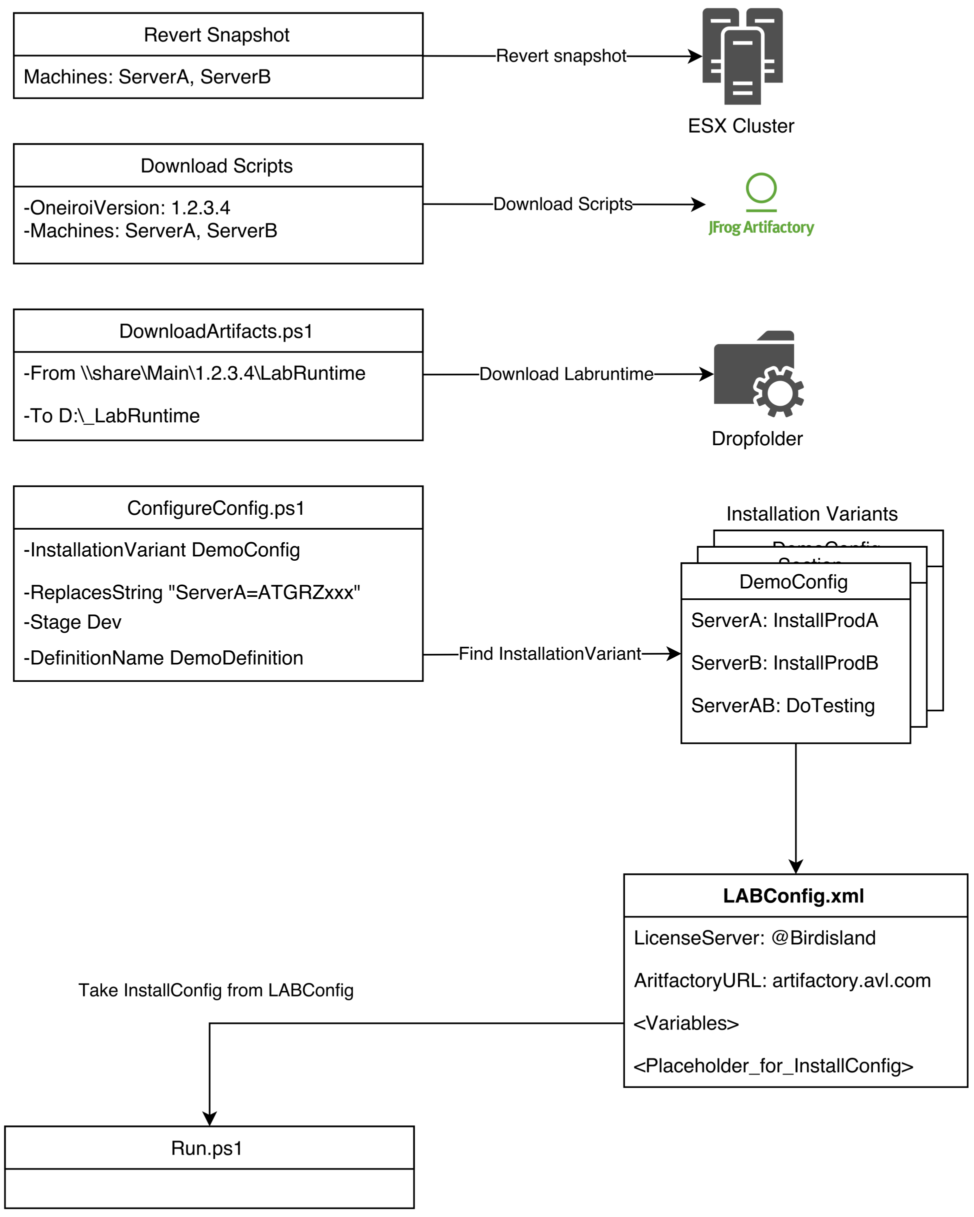


Figure 4.8: Oneiroi Workflow

which are applicable for the current machine. Listing 4.5 shows the switch case structure used.

```
$myTasks = Get-MyTasks
foreach($task in $myTasks)
{
  switch{$task}
  {
    "RunThisTaskEverywhere"
    {
      Run-ThisTaskEverywhere
      break
    }
    "RunFirstTaskOnMachineA"
    {
      Execute-Sometasks -ParameterA $task.ParameterA
      break
    }
  }
}
```

Listing 4.5: Executing structure of `Run.ps1`

4.3.1 Unit Tests

In order to provide PowerShell scripts which also meet a certain level of stability and quality, the testing framework *Pester*[5] is used. Pester provides a framework for running unit tests from within PowerShell to test PowerShell scripts. Pester consists of a set of functions that exposes a testing domain specific language (DSL)[6] for isolating, running, evaluating, and reporting the results of PowerShell commands.

In order to run the test locally/on a build machine, the particular modules need to be installed on the system. Listing 4.6 shows how to install Pester.

```
Install-PackageProvider Nuget
Set-PSRepository -Name PSGallery -InstallationPolicy Trusted
Install-Module -Name Pester
Install-Module PSScriptAnalyzer
```

Listing 4.6: PowerShell - Pester installation

The output format of the test results can be specified using `-OutputFormat NUnitXml`. NUnit result files can be processed by TFS and a test report shown as part of the build. Additionally, the script coverage can also be measured. Static code analysis[7] is also supported. `PSScriptAnalyzer` checks the Windows PowerShell code against a set of best practice rules indentified by the PowerShell team and the community. `PSScriptAnalyzer` generates diagnostic results (errors and

[5]As a precondition to use it, PowerShell 5.0 is needed. https://www.microsoft.com/en-us/download/details.aspx?id=50395

[6]For a more detailed description see https://github.com/pester/Pester

[7]PowerShell Script Analyzer: Static Code analysis for Windows PowerShell scripts and modules, https://blogs.msdn.microsoft.com/powershell/2015/02/24/powershell-script-analyzer-static-code-analysis-for-windows-powershell-scripts-modules

warnings) to inform users about potential code vulnerabilities and suggests possible improvements. The Pester unit tests are wrapped around the `PSScriptAnalyzer`. The build can be configured to fail in case the analyser finds issues within the scripts.

4.3.2 Implemented Scripting Functions

This section lists all PowerShell functions from the common framework. Actually, there are far more and the documentation will continue to grow while working on updating existing functions.

Export-Dump: Oracle dump export (Dump.ps1)

In addition to the test reports, the database dump is quite a convenient thing to store for running upgrade scenarios with higher software versions. A database dump can be exported using `expdp` which is part of the Oracle Client installation. The PowerShell script in Listing 4.7 can be used to export the schema of a particular database (the script file is located at `<OneiroiRoot>\AVL\Common\Database\Dump.ps1`).

```
function Export-Dump
{
    param
    (
        [parameter(Mandatory=$true)]
        [object]$SystemDbObject,

        [parameter(Mandatory=$true)]
        [string]$Schemas,

        [parameter(Mandatory=$true)]
        [string]$DumpFile
    )

    $pumpDir = Get-PumpDir $SystemDbObject
    $dumpFileName = [System.IO.Path]::GetFileName($DumpFile)
    $dumpFilePath = [System.IO.Path]::Combine($pumpDir[1],$dumpFileName)

    $expdpArgument = "schemas=$Schemas directory=DATA_PUMP_DIR dumpfile=$dumpFileName"
    $(&expdp $SystemDbObject.ConnectionString `"$expdpArgument`") 2> $null
    $exportFileName = $pumpDir[1]+"export.log"
    if(Test-Path $exportFileName)
    {
        Log (Get-Content -Path $exportFileName)
    }
    Copy-Item -Path $dumpFilePath -Destination $DumpFile -Force
}
```

Listing 4.7: Function Export-Dump

5. Dependency Management

“Well, it depends."

— Peter Scheir, Lacrosse Apprentice

In the beginning of 2014, AVL introduced[13] Artifactory [14] - an enterprise software solution for binary repository management. Artifactory was implemented as a tool to manage binary components (primarily DLLs and libraries, but also executables or MSIs) which are aimed at being used by several products within AVL. These components are frequently referred to as *Platform components*. Additionally, Artifactory can also be used as a central repository for all binary components used and generated during the build process.

Binary repositories are similar to source code repositories with respect to the fact that files are managed, archived, and versioned inside the repository itself. A key difference however, is that binary repository management solutions are optimised for large binary files, whereas source code management tools are optimised for large amounts of relatively small text files that change frequently. And whereas state-of-the-art source code management tools provide for a tight integration with development tools (e.g. Visual Studio), binary repository management tools are strongly integrated with build automation tools (TFS[1], Jenkins[2], Team City[3]): build results can be stored and meta-data generated during the build process (e.g. source code revision, build date, build number, user who triggered the build, component version, component name, licenses, etc.) can be attached to the binary. Table 5.1 (inspired by [15]) compares the characteristics of source code and binary repositories. The main drivers behind the need for dedicated binary repository management solutions like Artifactory are the management of binary dependencies and build automation.

[1]https://www.visualstudio.com/tfs/
[2]https://jenkins.io/index.html
[3]https://www.jetbrains.com/teamcity/

Source code management	Binary artifact management
Diffing, branching, tagging	None of these (within the development process proper)
Frequent deletes/overwrites	Rare deletes/overwrites
Small files	Large files
Minimal file-specific metadata	Lots of file-specific metadata
Changing, project-specific dependencies	Relatively static, cross-project dependencies

Table 5.1: Source code vs. binary artifact management

5.1 Binary Dependencies

Delivery of components in binary form (as opposed to delivery as source code) has been popular in software development for decades. Providers of components favour binary components because they do not have to provide the source code. Consumers of components are satisfied as they do not have to worry about building the components from source code, and they get a piece of functionality that is immediately ready to use.

Several teams have switched or are switching to providing their components in binary form. The reasons for this are similar to the reasons stated above:

1. Stronger encapsulation of the components; project-specific branching of components is restricted,
2. the build process is divided; a build can be naturally distributed on several machines and
3. the consuming project requires knowledge about building the components.

Also see [15] for a discussion of this topic.

5.1.1 Traceability

When moving to binary dependencies, traceability needs to be ensured for maintenance purposes: on one hand, the exact version of a component that was used during the creation of a complex piece of software needs to be documented. On the other hand, a binary component that provides functionality needs to document from which source code revision it was built. As part of the configuration management of a software product, all information needed to ensure traceability must be managed properly. In the context of binary repositories, this information is stored as meta-data of the components that are stored in the repository.

5.1.2 Automation

If the meta-data described above is stored in a form that is machine readable (i.e. clearly defined syntax and semantics), it lays the ground for build automation. Based on explicit meta-data of binary components (identifier, version, etc.), it is possible to define on which exact version of an external component a piece of software depends. A build system that can interpret this information is then able to resolve dependencies on external components automatically; based on the requirements of the depending software and the meta-data of the component, the appropriate version of a binary can be identified and linked.

In the world of Java-based software development, the concept of *Binary Dependency Manage-*

ment [16][17] and its tight integration into the build process, is very well established and supported by tools such as Maven[4], Gradle[5], and Ivy[6].

With the release of NuGet[7] in 2010, the Microsoft C++/.NET camp is rapidly catching up regarding this topic. NuGet is developed by Microsoft and tightly integrated into the Visual Studio line of products. The initial version of NuGet enabled binary dependency management for the .NET platform. In mid 2013, a version of NuGet was presented that supports native code. A detailed presentation on the benefits of using NuGet and binary dependencies can be found in [18].

5.2 Key Benefits

In brief, Artifactory can be viewed as a file system with a powerful application layer on top that provides ways for detailed management of the meta-data of artifacts and a means for strong automation. Artifactory's key features include:

- All interaction with the repository can be performed programmatically via an API (search, reading and writing of artifacts, and meta-data)
- Meta-data can be attached directly onto artifacts
- The repository offers advanced search over different properties of the components (version, name, organisation, meta-data, etc.)
- Artifactory handles replication to different locations out of the box
- The repository software guides the naming style of components and entering meta-data
- Artifactory allows extending its functionality via plug-ins (custom checks of meta-data and workflows can be implemented in this way)
- Artifactory can be used like a Windows network drive via WebDAV
- Users can subscribe to folders and artifacts to get notified via email when changes occur (e.g. when a new binary is published)

The selection of Artifactory as a tool for binary repository management was triggered by a project called *Platform Repository*. The project aimed at providing a uniform way to deliver Platform components in AVL development projects. In addition to the original use case of Artifactory as a place to uniformly store and manage binary deliveries of Platform components, a second, more advanced use case arose during the evaluation phase: as intended by its developers, Artifactory can also be used as part of a build automation system, as the place in which builds are stored, and as the repository from which binary dependencies are retrieved during build time (see Figure 5.1, inspired from [19]).

The key benefit of using Artifactory for this purpose is the automation of the whole process: after building a component, it can be stored in the repository and meta-data can be attached to it. Based on this meta-data, the artifact can be later retrieved and used to build a product depending on the component. For example, a build process can retrieve (based on the meta-data) a binary component of a certain version that was flagged as *approved* by the QA team after their test of the

[4]https://http://maven.apache.org/
[5]https://gradle.org/
[6]https://ant.apache.org/ivy/
[7]http://nuget.codeplex.com/

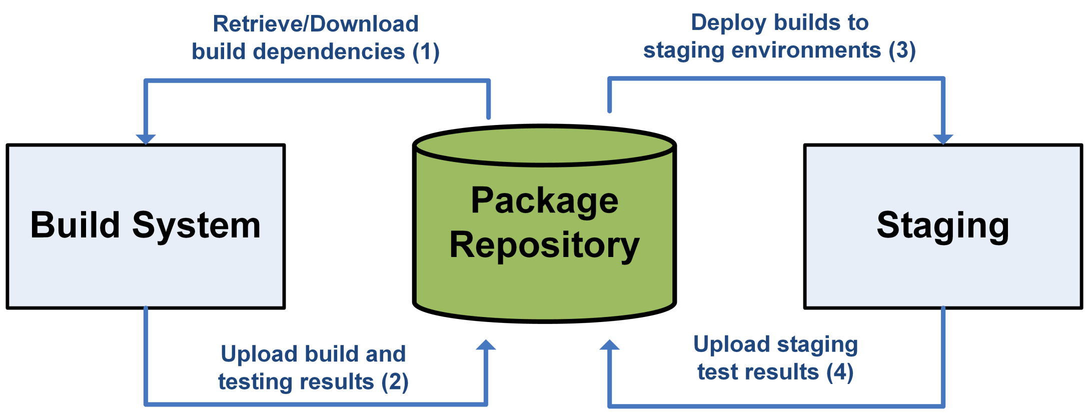

Retrieve/Download build dependencies (1)
Deploy builds to staging environments (3)
Build System
Package Repository
Staging
Upload build and testing results (2)
Upload staging test results (4)

Figure 5.1: Usage of Artifactory by AVL

component. What's especially interesting is Artifactory's capability to replicate the contents of a repository to other locations. This will be employed to synchronise the component repository between the Austrian and the Indian development teams so that it is ensured that everybody builds upon the same version of a component.

A logical next step would then be to deploy artifacts using the NuGet package format (see above). It offers advanced resolution mechanisms of dependencies to the build tool (e.g. retrieve at least a certain version of a certain library). Also, transitive dependencies (dependencies that have dependencies, cf. [16] [17]) can be resolved automatically using this approach.

5.3 Build Integration

A major improvement to the old build system discussed in Chapter 3 is that the dependency management is part of source control. The whole build is wrapped around two actions:

- **Download References:** In the current implementation there is a single batch file which contains all vendor and inter-product dependencies. As part of a pre-step of the build process, all dependencies are downloaded and provided within the build folder. All application projects within the build refer to this build folder for their reference resolution. In Listing 5.1, a small working example is illustrated. A specific version of a vendor component, package name, and download path are set. The download operation `DownloadArtifactoryPackage.bat` uses `Curl.exe`[8]. The download does not require authentication since Artifactory is configured to allow anonymous downloads. As part of the download, the package is extracted to a local directory specified by *OracleManagedDataAccessLocalInterfacesFolder*. In this example, the binaries are copied to the `"%TargetDir%"` from where they can be referenced.

[8]"Curl is used in command lines or scripts to transfer data.", https://curl.haxx.se/download.html

```
set  ArtifactoryRoot =  https://artifactory   . avl . com/artifactory

echo ### OracleManagedDataAccess Download ######################################
set  OracleManagedDataAccessVersion=12.1.2400
set  OracleManagedDataAccessDownloadPath=
%ArtifactoryRoot%/nuget−vendor/Oracle/Oracle/%OracleManagedDataAccessVersion%/
set  OracleManagedDataAccessPackageName=oracle.manageddataaccess.%OracleManagedDataAccessVersion%.nupkg
set  OracleManagedDataAccessLocalInterfacesFolder=%TargetDir%_Artifactory\OracleManagedDataAccessInterfaces\
echo %OracleManagedDataAccessDownloadPath%%OracleManagedDataAccessPackageName% >> %IntegratedArtifactsFile%
call  DownloadArtifactoryPackage.bat  "%ProjectDir%" "%TargetDir%" "%OracleManagedDataAccessDownloadPath%" "
%OracleManagedDataAccessPackageName%" "%OracleManagedDataAccessLocalInterfacesFolder%"
xcopy "%OracleManagedDataAccessLocalInterfacesFolder%lib\net40\∗.∗"          "%TargetDir%" /Y
```

Listing 5.1: Example download of Oracle Managed Data Access references

- **Build:** The software is built.
- **Upload Interfaces and Build Results:** After a successful build, the build artifacts are also generated in a batch file. The build artifacts consist of the whole source and build output, several interface packages, release packages, and documentation. Listing 5.2 shows an example upload of the source code artifact. The source code is zipped using *7za.exe*[9]. For upload, *Curl* is used. In contrast to the download operation, for upload, user authentication is necessary. The user `s13d01` was created for build automation and has the rights to upload to Artifactory. This procedure is replicated for other artifacts as well.

```
set  Product=SANTORIN
set  /p  Version=< "%TargetDir%Version.txt"
set  /p  VersionMajor=< "%TargetDir%VersionMajor.txt"
set  /p  VersionMinor=< "%TargetDir%VersionMinor.txt"
set  /p  VersionBuild=< "%TargetDir%VersionBuild.txt"
set  /p  VersionRevision=< "%TargetDir%VersionRevision.txt"

set  ArtifactoryUploadUser=s13d01
set  ArtifactoryUploadUserPassword=NF4d$1cQ
set  ArtifactoryRoot =  https://artifactory   . avl . com/artifactory

REM ######################################################################
REM Zipping Source
set  SourceTempDir=%SolutionDir%..\
for  /F  "delims="  %%F IN ("%SourceTempDir%") DO SET "SourceTempDir=%%~fF"
echo %SourceTempDir%

"%TargetDir%7za.exe" a −tzip  "%TargetDir%_Delivery\%TFSSourcePackageName%" "%SourceTempDir%∗" −xr!Bin −xr!
CopyFinishedBuild.log −xr!PostBuildStep.opensdf −xr!PostBuildStep.sdf

"%TargetDir%curl.exe"  −−user %ArtifactoryUploadUser%:%ArtifactoryUploadUserPassword% −−upload−file "
%TargetDir%_Delivery\%TFSSourcePackageName%"
%ArtifactoryRoot%/simple/repo−snapshot/AVL/%Product%/%VersionMajor%.%VersionMinor%/%VersionBuild%.
%VersionRevision%/%TFSSourcePackageName% −−insecure
```

Listing 5.2: Example upload of Source Code as an artifact

[9]"7-Zip is an open source file archiver with a high compression ratio.", http://www.7-zip.org/download.html

6. Documentation

“Documentation, when it is good, it is very, very good; and when it is bad, it is better than nothing."

— Rajeev Kumar Tyagi, Every day is a new day, let’s do it

In software projects proper documentation is often one of the first topics which get cut due to time constraints. Writing and maintaining documentation is most of the time considered to be extra unplanned effort. The term *documentation* refers to the written text that is produced as part of the software development project[20]. Documentation can be split into several subtypes:

- **Architecture and Design documents:** This kind of documentation provides the overall information about the software projects. Initially, is is generated at the beginning of the software project but to be useful it must be reviewed and continuously updated during the development.
- **Technical documents:** The actual documentation of the code itself in respect to design patterns, interfaces, and APIs. The best practice approach is to keep this kind of documentation inside the source code and use tools (e.g. Doxygen[1]) to automatically parse the source code in order to build up a reference handbook.
- **End-User Manuals:** These manuals are meant for the real end-user, the system administrators, trainers and trainees, and support staff. This kind of documentation has usually nothing to do with the source code or other technical insights. It is written in a behavioural way in order to give guidance to the usage of the software.

[1]“Doxygen is the de facto standard tool for generating documentation from annotated C++ sources, but it also supports other popular programming languages such as C, Objective-C, C#, PHP, Java, Python, IDL (Corba, Microsoft, and UNO/OpenOffice flavours), Fortran, VHDL, Tcl, and to some extent D.”, http://www.stack.nl/ dimitri/doxygen/

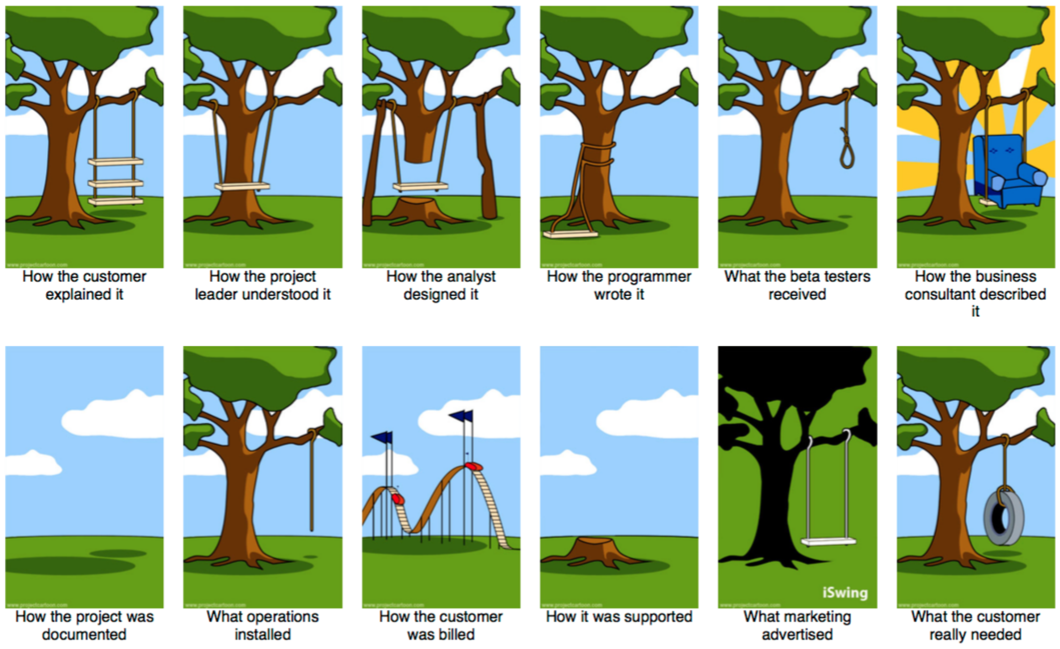

How the customer explained it
How the project leader understood it
How the analyst designed it
How the programmer wrote it
What the beta testers received
How the business consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
iSwing
What marketing advertised
What the customer really needed

Figure 6.1: How the project was documented

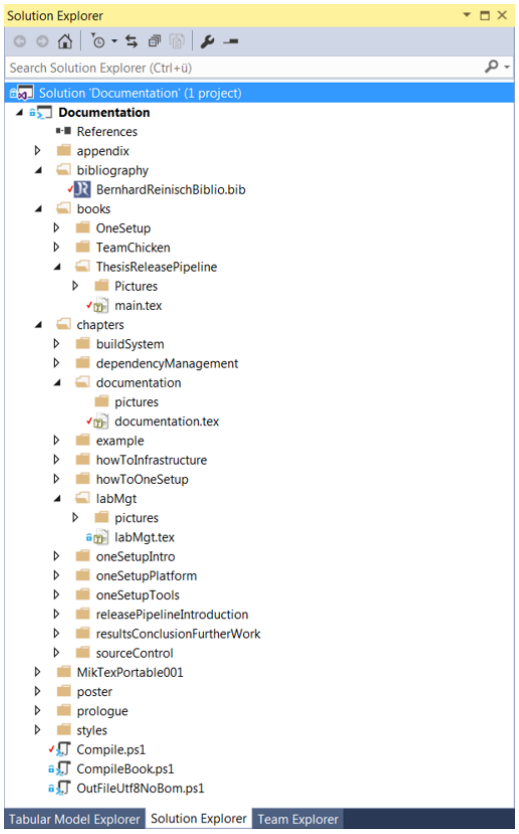
In AVL, regarding documentation, particularly the end-user manuals, a separate department is responsible. Resources in the documentation department are heavily dependent on the input from the development teams in order to create proper documentation. The development teams prepare the systems, create screen-shots and assist in writing the text itself. On the other hand, the documentation department is responsible for ensuring the documentation follows the style guides and are also providing translations. As part of the *Agile Transformation*, the restructuring/relocation of the documentation resources was not carried out. This lead to the situation that the development teams sped up but were not able to actually close features without having documentation done. **The teams were not able to update the documentation by themselves** due to the tooling setup (licensing cost). Therefore the documentation was removed from the *Definition of Done (DoD)*, and it was expected that the documentation resources will eventually catch up. What actually happened can be seen in Figure 6.1 (taken from [21]). It might seem a bit exaggerated but serious problems arose due the fact that the documentation could not keep up with the speed of the development.

6.1 Collaborative Writing

To solve the problem that only designated people were allowed to create documentation, a collaborative writing model was needed. The teams needed to be enabled to actively contribute. As part of the internal OneSetup project, the documentation was created using LaTex[2] for typesetting and MikTeX Portable[3] for compilation. The main advantage of LaTex is that the text and the style of a

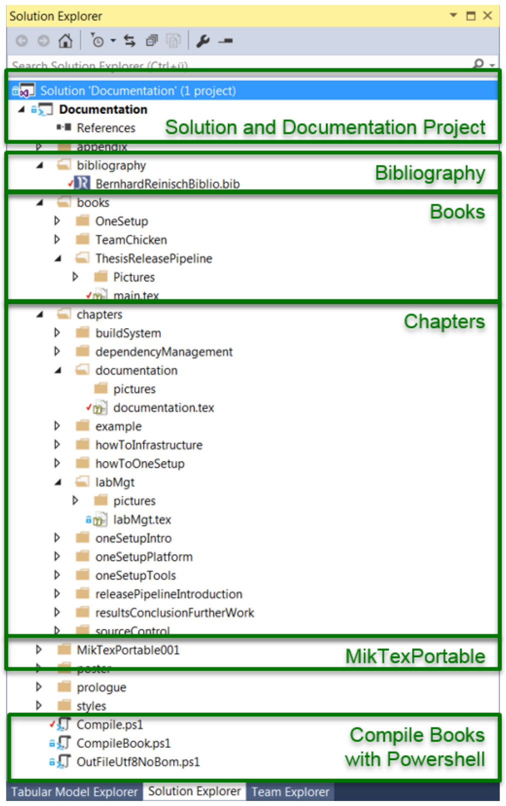
[2]"LaTex is a high-quality typesetting system; it includes features designed for the production of technical and scientific documentation. LaTex is the de facto standard for the communication and publication of scientific documents. LaTex is available as free software.", https://www.latex-project.org/

[3]"MiKTeX (pronounced mick-tech) is an up-to-date implementation of TeX/LaTex and related programs for Windows (all current variants).", https://miktex.org/portable



```
Solution Explorer
Search Solution Explorer (Ctrl+ü)
Solution 'Documentation' (1 project)
  Documentation
    References
    appendix
    bibliography
      BernhardReinischBiblio.bib
    books
      OneSetup
      TeamChicken
      ThesisReleasePipeline
        Pictures
        main.tex
    chapters
      buildSystem
      dependencyManagement
      documentation
        pictures
        documentation.tex
      example
      howToInfrastructure
      howToOneSetup
      labMgt
        pictures
        labMgt.tex
      oneSetupIntro
      oneSetupPlatform
      oneSetupTools
      releasePipelineIntroduction
      resultsConclusionFurtherWork
      sourceControl
    MikTexPortable001
    poster
    prologue
    styles
    Compile.ps1
    CompileBook.ps1
    OutFileUtf8NoBom.ps1
Tabular Model Explorer | Solution Explorer | Team Explorer
```

Figure 6.2: Documentation Solution



```
Solution Explorer
Search Solution Explorer (Ctrl+ü)
Solution 'Documentation' (1 project)
  Documentation
    References
    appendix
    bibliography
      BernhardReinischBiblio.bib
    books
      OneSetup
      TeamChicken
      ThesisReleasePipeline
        Pictures
        main.tex
    chapters
      buildSystem
      dependencyManagement
      documentation
        pictures
        documentation.tex
      example
      howToInfrastructure
      howToOneSetup
      labMgt
        pictures
        labMgt.tex
      oneSetupIntro
      oneSetupPlatform
      oneSetupTools
      releasePipelineIntroduction
      resultsConclusionFurtherWork
      sourceControl
    MikTexPortable001
    poster
    prologue
    styles
    Compile.ps1
    CompileBook.ps1
    OutFileUtf8NoBom.ps1
Tabular Model Explorer | Solution Explorer | Team Explorer
```

Figure 6.3: Documentation Solution - Areas

document is strictly separated from each other. The written text is really text-based and therefore can be nicely checked-in to source control and is furthermore mergeable over branches. This is a key advantage over all other documentation tools used at AVL.

As part of the source code, a Visual Studio solution was created to incorporate the documentation creation into the build process. Figures 6.2 and 6.3 show the structure:

- **Solution and Documentation Project:** This is the actual solution and contains one PowerShell project. All LaTex files are located within the project.
- **Bibliography:** The folder to place different bibliographies which may be used in various books.
- **Books:** The book definitions are located in this folder. Every folder represents one product. One product can have several books and every LaTex file represents the definition of a book.
- **Chapters:** The smallest unit are chapters. Chapters are written on a specific topic and have all their references in the particular chapter folder. Chapters can be re-used over various books.
- **MikTeX Portable:** The MikTeX toolset is located within this folder. No installation is needed.
- **Compile Books with PowerShell:** Three PowerShell scripts are needed for compiling the books. The main file is `Compile.ps1` (the other two are helper scripts). This file needs to be

opened and run. All books defined within the `Compile.ps1` are compiled.

```
Param
(
    [Parameter(Mandatory=$false)] [string]$productVersion = "2.1.17074.3",
    [Parameter(Mandatory=$false)] [string]$productMetaVersion = "AVL OneSetup 2 R1.1",
    [Parameter(Mandatory=$false)] [string]$outputDirectory = "..\bin\releaseU\documentation\"
)
Set-Location $psscriptroot

If (Test-Path $outputDirectory) {
    Remove-Item $outputDirectory -Recurse
}

$OneSetup = @( 'OneSetupDeveloperGuideBook',
               '".\books\OneSetup\DevGuide.tex"',
               '".\books\OneSetup"',
               '".\styles"',
               '".\prologue\generic"',
               '".\chapters\oneSetupIntro"',
               '".\chapters\oneSetupPlatform"',
               '".\chapters\oneSetupTools"',
               '".\bibliography"')

."$PSScriptRoot\CompileBook.ps1"
."$PSScriptRoot\OutFileUtf8NoBom.ps1"

## OneSetup Developer Guide Book
(Get-Content ".\prologue\generic\titlepage.tex" -Encoding UTF8).replace('[PLACEHOLDER_PRODUCT]', $productMetaVersion).replace('[PLACEHOLDER_BRAND_PROMISE]', 'Experience the spirit of open source').replace('[PLACEHOLDER_DOCUMENT_NAME]', 'Developer Guide').replace('[PLACEHOLDER_BELOW_DOCUMENT_NAME]', 'Build: ' + $productVersion) | Out-FileUtf8NoBom ".\prologue\generic\oneSetupDevGuideTitlepage.tex"
(Get-Content ".\prologue\generic\prologue.tex").replace('[PLACEHOLDER_TITLEPAGE]', 'oneSetupDevGuideTitlepage') | Out-FileUtf8NoBom ".\prologue\generic\oneSetupDevGuidePrologue.tex"

CompileBook -productVersion $productVersion
  -productMetaVersion $productMetaVersion
  -outputDirectory $outputDirectory
  -bookDefinition $OneSetup
  -quiet $true
```

Listing 6.1: LaTex Book Compile `Compile.ps1`

In order to understand how the `Compile.ps1` needs to be adapted/extended, an example for one book is shown in Listing 6.1. This example shows how the *OneSetup Developer Guide* is compiled:

- **Input Parameters:** The documentation is versioned with the same version number as the regular build. Therefore, the version information needs to be provided (`$productVersion`, `$productMetaVersion`). The resulting PDF files are created in
  the output directory (`$outputDirectory`).
- **Book definition:** The `$OneSetup` list provides all the information for the compile process. The first entry, `OneSetupDeveloperGuideBook` is the *job name* which is also the name of the resulting PDF file. The second entry, `.\books\OneSetup\DevGuide.tex`, points to the root LaTex file of the book. All other paths are include paths to the LaTex files which are

directly referenced in the root LaTex files.

- **Customising generic template:** The OneSetup books use a generic template as part of their prologue. The lines beginning with `Get-Content` are modifying the generic prologue in respect to versioning, book name, subtitle, and images on the title page.
- **CompileBook.ps1:** Compiles one particular book and takes the version information and the book definition as input.

In the OneSetup project, all documentation is generated with this documentation framework. OneSetup lives from the contribution of various developers and the immediate documentation of the developed features (no additional documentation resources are available). From the feedback and based on the results, integrating documentation in a collaborative way is a big step forward in respect to the velocity and quality of delivered software.

7. Infrastructure: How To ...

“Changing Stuff and Seeing What Happens."

— Severin Kann, C/C++ Tutor

This chapter provides a collection of use-cases applicable to daily work. They are more oriented to achieve a goal than to provide a good explanation.

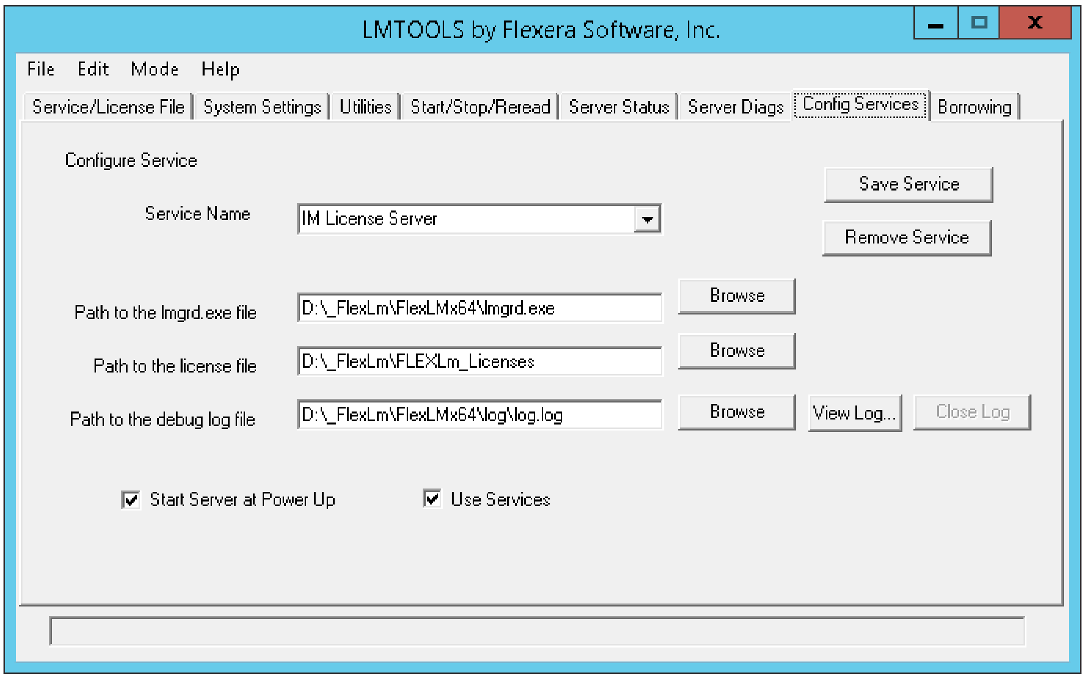
7.1 ... setup a Build Machine

To setup a Build Machine for the AVL products *Santorin* and *TFMS*, a *Windows Server 2012 R2* machine is needed with the following minimum configuration:

- 8 CPU Cores
- 8 GB RAM
- 100 GB C-drive
- 500 GB D-drive

The build user is **avl01\s18f30**. The **pbuild** user which is often used within AVL should NOT be used. The following software needs to be installed on a build machine:

- Visual Studio 2010 Premium
- Visual Studio 2010 PS1
- Visual Studio 2015 Update 3 or newer
- Infragistics 2011.1
- PerpetuumSoft Sharpshooter 7.3.0
- NorthWood GoDiagram
- NUnit Visual Studio Test Adapter


LMTOOLS by Flexera Software, Inc.
File Edit Mode Help
Service/License File | System Settings | Utilities | Start/Stop/Reread | Server Status | Server Diags | Config Services | Borrowing
Save Service
Remove Service
Browse
Browse
Browse
View Log...
Close Log

Configure Service

Service Name: IM License Server

Path to the lmgrd.exe file: D:_FlexLm\FlexLMx64\lmgrd.exe

Path to the license file: D:_FlexLm\FLEXLm_Licenses

Path to the debug log file: D:_FlexLm\FlexLMx64\log\log.log

☑ Start Server at Power Up ☑ Use Services

Figure 7.1: License Server configuration for multiple license files

- vSphere PowerCLI 6.5 or newer
- PowerShell 5.0 or newer
- Silverlight SDK 4.0
- Java JDK 8 Update 77 or newer
- MVC4
- ActivePerl 5.24.1.2402 or newer

All installers can be found on the Dropfolder:
`\\avl01.avlcorp.lan\ATGRZ\misc\TFS_Dropfolder\_HowToSetUpABuildMachine`.

7.2 ... setup a License Server

For licensing AVL software, a solution of Flexera Software[1] is used. As part of this software component, a license server is also provided. For all installations in the Lab Management, one license server is installed which is located on `\\birdisland`. The installer of the license server can be found at `\\avl01\nbuild\PumaOpen\V153\Vendor\FlexLMx64`. An installation guide (AT1729E10_LicServInstall) is also provided. To install, follow the instructions found in *Chapter 3 Installation of the License Server on WindowsServer 2008 R2 (64 Bit)*.

It is also possible to provide a folder which contains multiple license files. Instead of providing a path to a single license file, enter the path to the folder containing all license files as shown in Figure 7.1.

Auto generated license files are provided by license@avl.com and have the `.dat` file ending. In order to get them working with the license server, they need to be renamed to files with the ending

[1] FlexLM, http://www.flexerasoftware.de

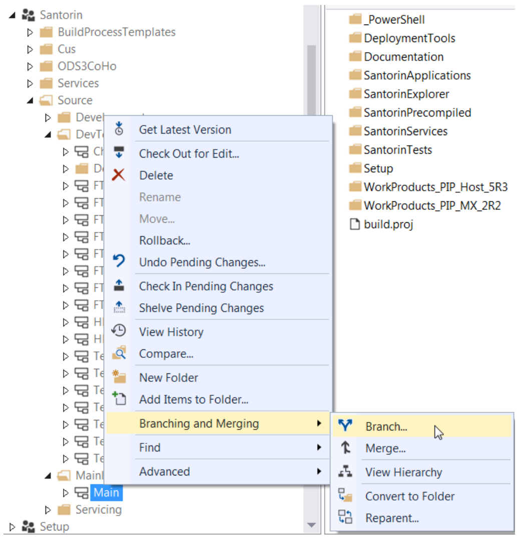
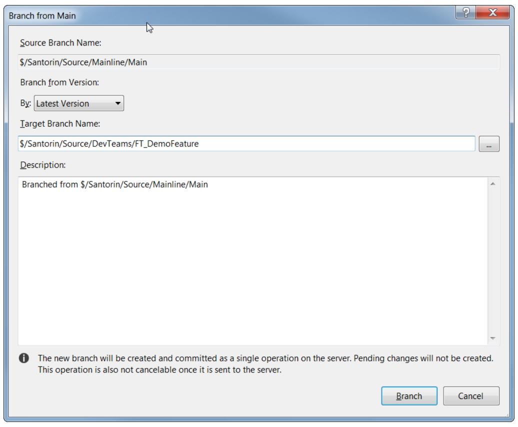


Figure 7.2: Create feature branch from Santorin Main


Branch from Main

Source Branch Name:
$/Santorin/Source/Mainline/Main

Branch from Version:
By: Latest Version

Target Branch Name:
$/Santorin/Source/DevTeams/FT_DemoFeature

Description:
Branched from $/Santorin/Source/Mainline/Main

The new branch will be created and committed as a single operation on the server. Pending changes will not be created. This operation is also not cancelable once it is sent to the server.

Figure 7.3: *Branch from <Branch>* dialogue

`.lic`. Only then they can be loaded processed properly by the license server.

7.3 ... build a Feature Branch

In order to create a feature branch, the following information is needed:

- Parent trunk branch.
- Feature name.
- Build needed?

  If yes: Which check-in policy is needed?
- Lab environment needed?

  If yes: Which rollout policy is needed?

The following naming convention is used:

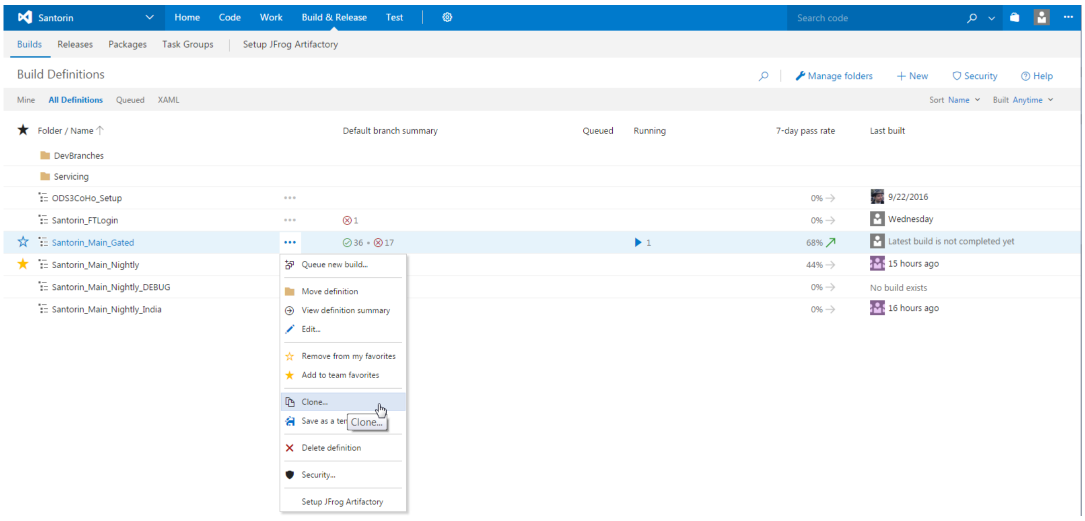
- Branch: FT_<Featurename>
- Build Definition

  From Main: <Product>_FT_<Featurename>

  From Servicing: <Product>_Svc_V<Version>_FT_<Featurename>
- Release Definition: follows the same as build definition.

Once the information is provided, the feature branch can be created:

1. Open the context menu of the parent branch (see Figure 7.2, in this case, Santorin `Main`) and select `Branching and Merging` and `Branch...`.
2. The *Branch from <Branch>* dialogue (see Figure 7.3) will pop-up. Provide the target branch name. The feature branch should be located within the Folder `DevTeams`.
3. The source code branch is now created.
4. To create the build definition, navigate to the `Builds - All Definitions` section of the Team Project. The folder structure for the build definitions is similarly organised as within the source control. Clone the build definition of the parent branch (see Figure 7.4) and edit



Folder / Name	Default branch summary	Queued	Running	7-day pass rate	Last built
DevBranches					
Servicing					
ODS3CoHo_Setup				0%	9/22/2016
Santorin_FTLogin	⊗ 1			0%	Wednesday
☆ Santorin_Main_Gated	✓ 36 • ⊗ 17		▶ 1	68%	Latest build is not completed yet
★ Santorin_Main_Nightly				44%	15 hours ago
Santorin_Main_Nightly_DEBUG				0%	No build exists
Santorin_Main_Nightly_India				0%	16 hours ago

Figure 7.4: Clone the build definition for the feature branch

the `Server Path` repository tab, as shown in Figure 7.5, and the `Triggers`, as shown in Figure 7.6, accordingly.

5. As a final step, save the build definition using the naming convention shown in Figure 7.7.

7.4 ... build a Servicing Branch

On the `Main` branch the version *SANTORIN 5 R3.0* is currently in development. In order to create this first release, `Main` becomes *SANTORIN 5 R3.1* and the servicing branches then need to be created. In addition, also a build and lab environment are needed for *SANTORIN 5 R3.0*.

7.4.1 Branching

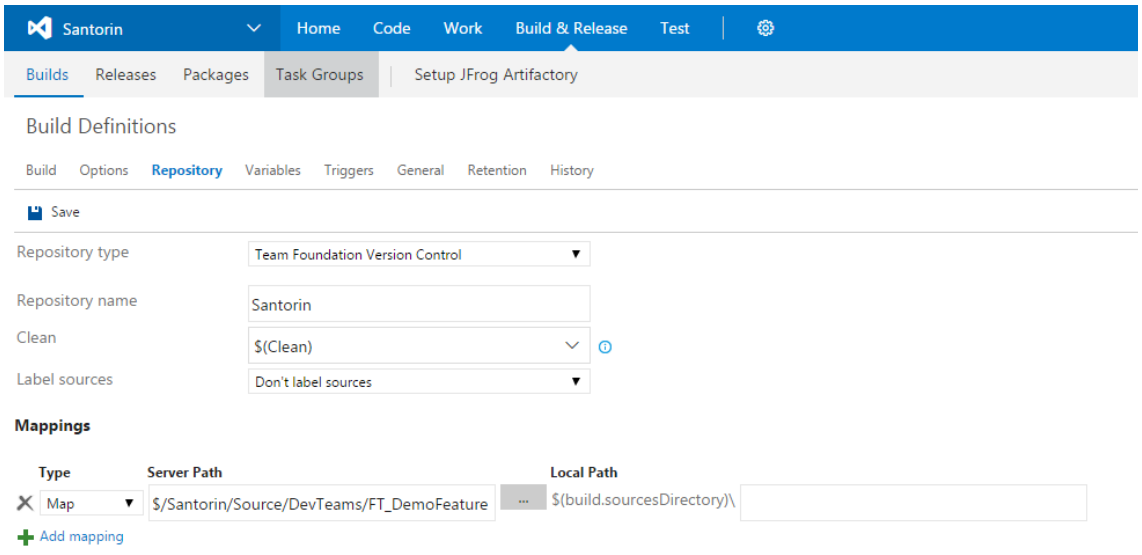
An additional branch `5R3` is needed and it becomes a placeholder until `Main` becomes R4. From then on, the `5R3` branch will become the maintenance branch for *SANTORIN 5 R3.x*. The two sub-branches, `/Servicing/5R3.x/5R3` and `/Servicing/5R3.x/5R3.0` also need to be created.

1. In order to create the Folder *5R3.x*, the parent folder `Servicing` needs to be mapped. Create the branch as shown in Figure 7.8 and repeat for the `/Servicing/5R3.x/5R3.0` branch.
2. After both branches are complete, the branch hierarchy should appear as shown in Figure 7.9 and the folder structure should appear as shown in Figure 7.10.

7.4.2 Build

To create the build definitions for the newly created branches:

1. Clone the build definition `Santorin_Main_Nightly`.
2. Adapt the `Server Path` in the repository tab (as shown in Figure 7.5, e.g. `$/Santorin/Source/Servicing/5R3.x/5R3.0`) and the `Triggers` (as shown in Figure 7.6) accordingly .
3. Adapt the build number format in the General tab. The default value is `5.53.$(Year:yy)$(DayOfYear)$(rev:.r)`. In this particular case, the build number



Repository type: Team Foundation Version Control

Repository name: Santorin

Clean: $(Clean)

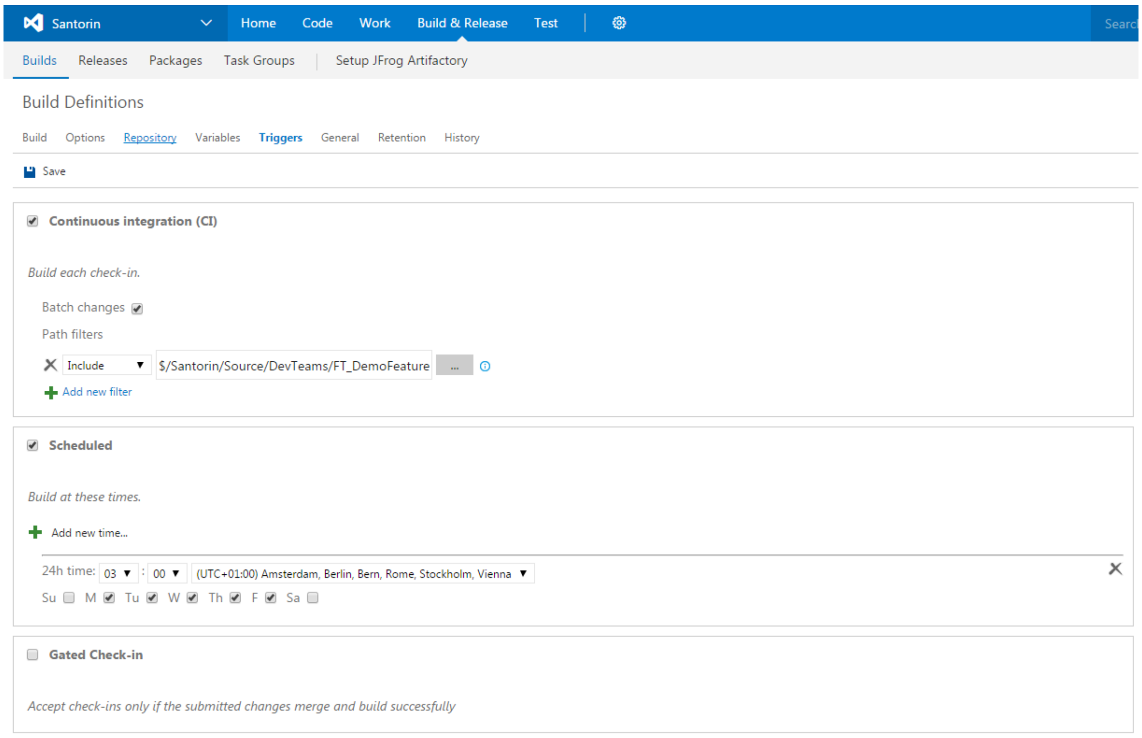
Label sources: Don't label sources

Mappings

Type	Server Path	Local Path
Map	$/Santorin/Source/DevTeams/FT_DemoFeature	$(build.sourcesDirectory)\

Add mapping

Figure 7.5: Adapt the Server Path



☑ Continuous integration (CI)

Build each check-in.

Batch changes ☑

Path filters

Include $/Santorin/Source/DevTeams/FT_DemoFeature

Add new filter

☑ Scheduled

Build at these times.

Add new time...

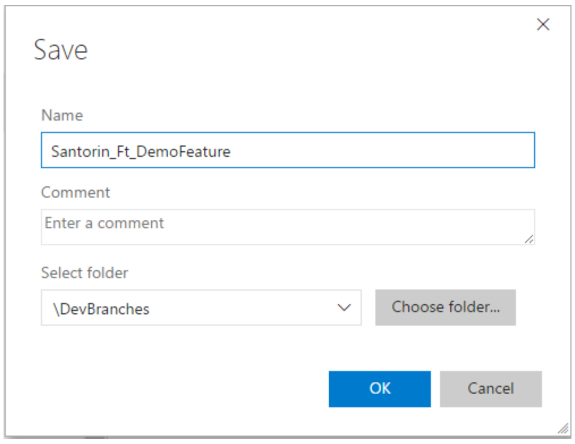
24h time: 03 : 00 (UTC+01:00) Amsterdam, Berlin, Bern, Rome, Stockholm, Vienna

Su ☐ M ☑ Tu ☑ W ☑ Th ☑ F ☑ Sa ☐

☐ Gated Check-in

Accept check-ins only if the submitted changes merge and build successfully

Figure 7.6: Adapt the Triggers

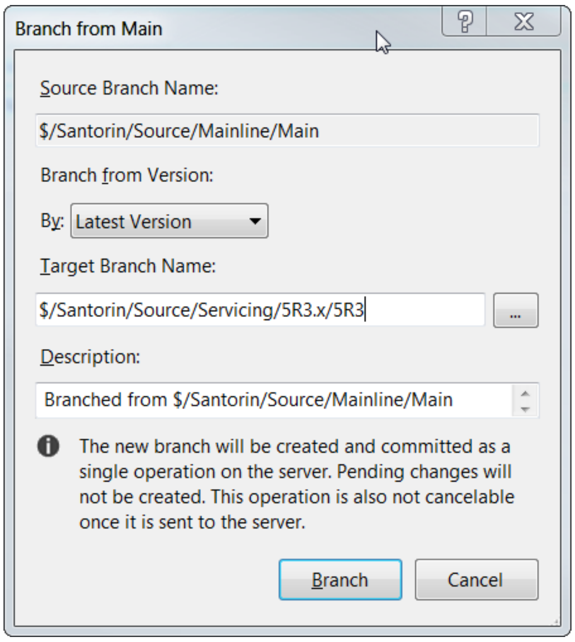


Name: Santorin_Ft_DemoFeature

Comment: Enter a comment

Select folder: \DevBranches

Figure 7.7: Saving the cloned build definition



Source Branch Name: $/Santorin/Source/Mainline/Main

Branch from Version:

By: Latest Version

Target Branch Name: $/Santorin/Source/Servicing/5R3.x/5R3

Description: Branched from $/Santorin/Source/Mainline/Main

The new branch will be created and committed as a single operation on the server. Pending changes will not be created. This operation is also not cancelable once it is sent to the server.

Figure 7.8: Creation of the servicing branch 5R3

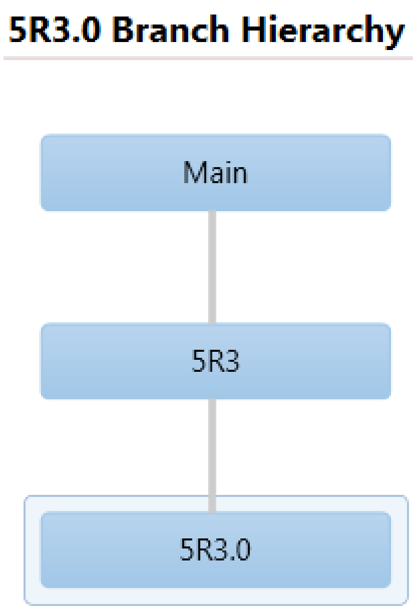


Figure 7.9: Servicing Branch Hierarchy

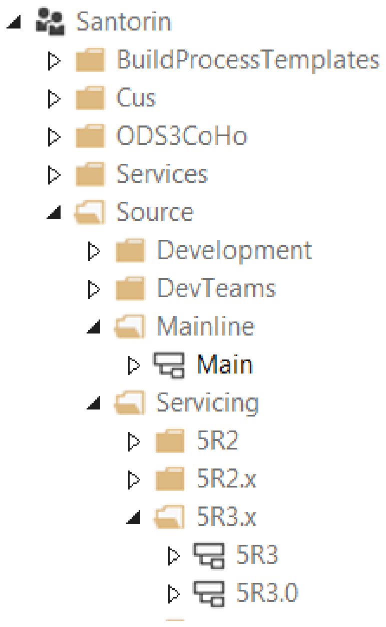


Figure 7.10: Servicing Branch Folder

needs to be adapted as described in Section 3.5 (e.g. to `5.53.17085.$(DayOfYear)$(rev:r)`).

7.4.3 Lab Machine Creation

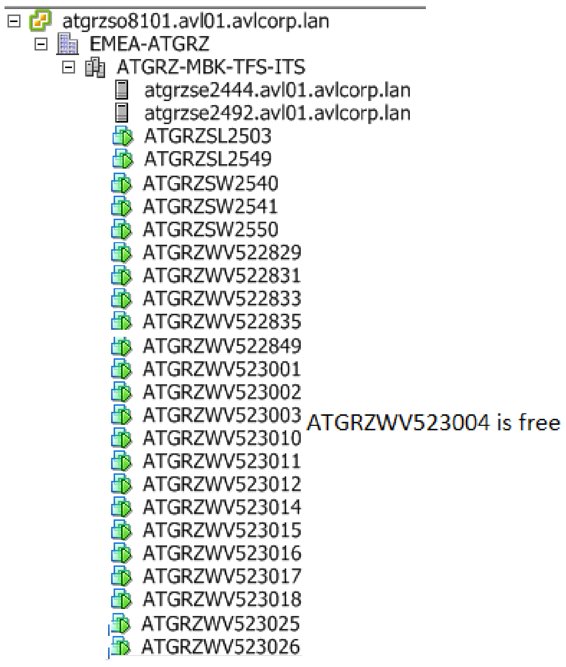
For the `/Servicing/5R3.x/5R3.0` branch, only one *Windows Server 2012 R2* machine is needed. To create the new virtual machine, a machine name needs to be chosen. To achieve this, within the vSphere Client, navigate to *Hosts and Clusters* View as shown in Figure 7.11. All machine names are listed in alphabetical order, and in this example, the next free machine name is `ATGRZWV523004`.

Within the OneSetup project there is a build definition which creates new virtual machines. Navigate to the build definition *All Definitions/New-Machines*, queue a new build (as shown in Figure 7.12), and set the variable `New-Win2012` to the available machine name, `ATGRZWV523004` (this step takes about 30 minutes). **Machine names have to be in capital letters**.

7.4.4 Release Definition

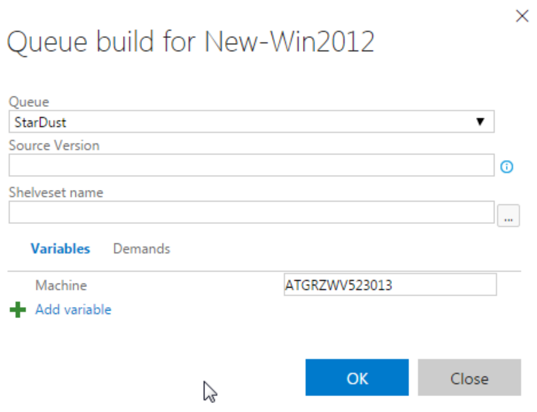
In general, one of the latest servicing Release Definitions should be taken as a template for new release definitions:

1. Clone a release definition, e.g. `Santorin_Svc_V5R2.1`, as shown in Figure 7.13.
2. Adapt the variables (as shown in Figure 7.14 and Figure 7.15) for the Development stage.
3. Link the created release definition to the artifacts of the created build definition as shown in Figure 7.16.
4. Update the Continuous Deployment trigger to match the artifact source as shown in Figure 7.17.
5. Finally, save the Release Definition with the same name as the source build definition. In our, case `Santorin_Svc_V5R3.0`.



```
atgrzso8101.avl01.avlcorp.lan
  EMEA-ATGRZ
    ATGRZ-MBK-TFS-ITS
      atgrzse2444.avl01.avlcorp.lan
      atgrzse2492.avl01.avlcorp.lan
      ATGRZSL2503
      ATGRZSL2549
      ATGRZSW2540
      ATGRZSW2541
      ATGRZSW2550
      ATGRZWV522829
      ATGRZWV522831
      ATGRZWV522833
      ATGRZWV522835
      ATGRZWV522849
      ATGRZWV523001
      ATGRZWV523002
      ATGRZWV523003 ATGRZWV523004 is free
      ATGRZWV523010
      ATGRZWV523011
      ATGRZWV523012
      ATGRZWV523014
      ATGRZWV523015
      ATGRZWV523016
      ATGRZWV523017
      ATGRZWV523018
      ATGRZWV523025
      ATGRZWV523026
```

Figure 7.11: Find an available machine name



Queue build for New-Win2012

Queue: StarDust

Source Version:

Shelveset name:

Variables | Demands

Machine	ATGRZWV523013

Figure 7.12: Queue build in order to create a new virtual machine

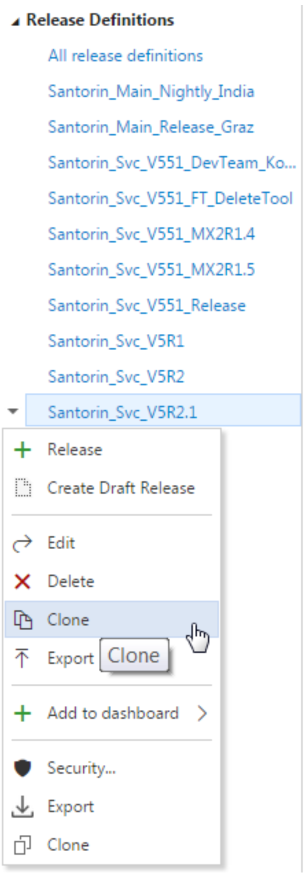

Release Definitions
All release definitions
Santorin_Main_Nightly_India
Santorin_Main_Release_Graz
Santorin_Svc_V551_DevTeam_Ko...
Santorin_Svc_V551_FT_DeleteTool
Santorin_Svc_V551_MX2R1.4
Santorin_Svc_V551_MX2R1.5
Santorin_Svc_V551_Release
Santorin_Svc_V5R1
Santorin_Svc_V5R2
Santorin_Svc_V5R2.1
Release
Create Draft Release
Edit
Delete
Clone
Export
Clone
Add to dashboard
Security...
Export
Clone

Figure 7.13: Clone the latest servicing Release Definition

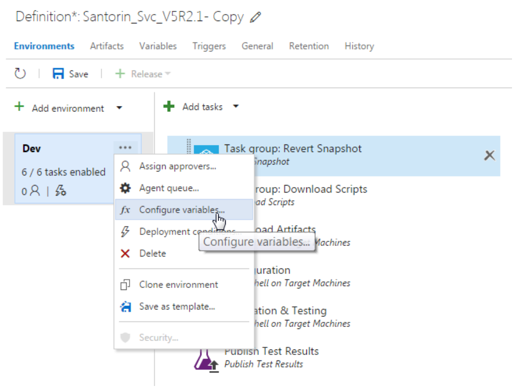
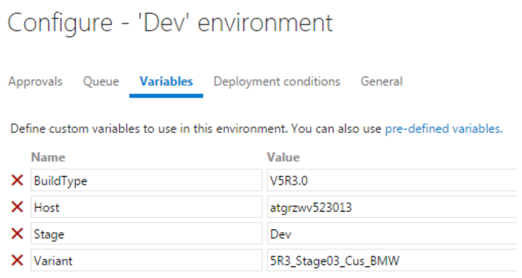


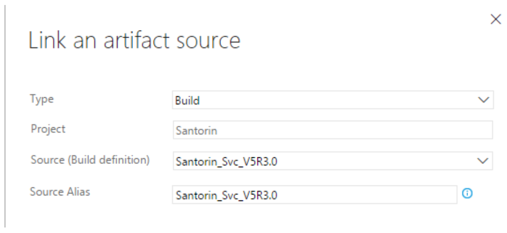
Figure 7.14: Open `Configure Variables` on Release Definition


Configure - 'Dev' environment

Define custom variables to use in this environment. You can also use pre-defined variables.

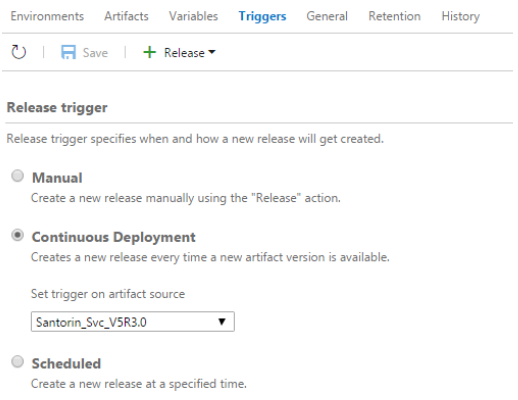
	Name	Value
✗	BuildType	V5R3.0
✗	Host	atgrzwv523013
✗	Stage	Dev
✗	Variant	5R3_Stage03_Cus_BMW

Figure 7.15: Configure variables on Release Definition


Link an artifact source

Type: Build
Project: Santorin
Source (Build definition): Santorin_Svc_V5R3.0
Source Alias: Santorin_Svc_V5R3.0

Figure 7.16: Remove/Renew the artifact source to the new build definition



Release trigger

Release trigger specifies when and how a new release will get created.

Manual
Create a new release manually using the "Release" action.

[✓ Continuous Deployment]
Creates a new release every time a new artifact version is available.

Set trigger on artifact source: Santorin_Svc_V5R3.0

Scheduled
Create a new release at a specified time.

Figure 7.17: Update the release definition trigger

7.5 ... create a Virtual Machine

Virtual machines in the AVL infrastructure should follow the following naming pattern:

- ATGRZWV522xxx -> Win2008
- ATGRZWV523xxx -> Win2012
- ATGRZWV524xxx -> Win7
- ATGRZWV525xxx -> Win10
- ATGRZWV526xxx -> Win2016
- ATGRZWV527xxx -> Build Machines

In addition to creating virtual machines within the vSphere client, where every step needs to be applied manually, the VMWare PowerCLI can also be used. As a precondition, the vCenter Server Connection, the Data Center, the Template Name, the Cluster Name, the Datastore Name, the guest operating system customisation specification, and the machine name is needed. The guest operating system customisation specification can be found with the vSphere Client (Management -> Customisation Specifications Manager). It describes how to customise the virtual machine in respect to computer name, license options, domain joins, and so on.

7.5.1 via TFS Build Tasks

One way to create machines from a template is to use the TFS build tasks for VMWare Resource Deployment. A build agent with an installed VMWare PowerCLI is needed. Five steps are required:

1. Power off the virtual machine (it has to be powered off in order to be deletable)
2. Delete the virtual machine
3. Deploy the virtual machine using a template
4. Reboot the virtual machine to apply group policies
5. Take a snapshot of the virtual machine (to be used in the lab environment in the *revert to snapshot* tasks)

7.5.2 via PowerShell Script

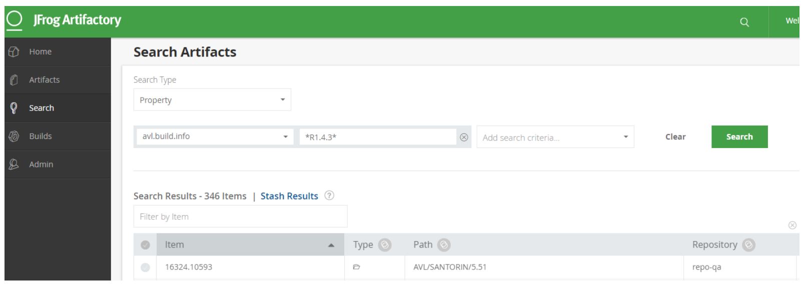
The other way is to use a PowerShell script to perform these tasks. In Listing 7.1, the PowerCLI is used to communicate to the VMWare server.

```
Connect-VIServer atgrzso8101 -Credential $mycreds -Force
$targetCluster = Get-Cluster -Name $clusterName
$sourceVmTemplate = Get-Template -Name $sourceVmTemplateName
$sourceCustomSpec = Get-OSCustomizationSpec -Name $sourceCustomSpecName
New-VM -Name $nextVmName -Template $sourceVmTemplate -ResourcePool $targetCluster
      -OSCustomizationSpec $sourceCustomSpec
$myVm = Get-VM -Name $nextVmName
Start-VM -VM $myVm
```

Listing 7.1: Create new build virtual machine

7.6 ... find artifacts in Artifactory

The *Product Versioning* is part of the meta-data of all software releases within Artifactory. This meta-data can be used to search for particular software versions right from the search pane as shown in Figure 7.18.



Search Artifacts

Search Type: Property

avl.build.info | *R1.4.3* | Add search criteria...

Search Results - 346 Items | Stash Results

Filter by Item

Item	Type	Path	Repository
16324.10593	🗀	AVL/SANTORIN/5.51	repo-qa

Figure 7.18: Search Artifactory using meta data

II

OneSetup Deployment Framework

8. OneSetup Introduction

"MSI is wearing a tight corset. Keep your installer simple and MSI will do the maintenance job for you, ignore MSI's rules and you'll finish up in hell."

— Johannes J. Klinger, Telling ghost stories to installer newbies

As part of software development, the question *"How can the software be delivered to the customer?"* is part of the whole development process. There are quite a lot of competing installer frameworks on the market. All of them have to deal with standard problems/questions such as:

- *Per-User* or *Per-Machine* installation needed?
- What happens if an error occurs in the middle of the setup?
- How to deal with folder redirection and/or roaming profiles?
- Which processor platform is the target platform: x86, x64, AnyCPU?
- Different folder visibilities on 32-bit, 64-bit operating systems.
- How to deal with patches and upgrades?
- Are *Launch Conditions* met? What about conditional installations?
- ...

One approach to deal with these concerns is *Windows Installer (MSI)*.

8.1 Windows Installer Concepts

Windows Installer (MSI) started around 1999 (with Microsoft Office 2000). Microsoft wanted to provide a standardised way for installation processes on their Windows operating system. Windows Installer consists of three main components:

- Msiexec.exe[1] (see command line options[2])
- Windows Installer API[3]
- Windows Installer SDK[4]

By using MSI as part of the deployment strategy, the operating system can keep track of the consistency and versioning of an application. Even repair operations can be initiated on faulty programs. Figure 8.1 (inspired by [22]) shows the sequence of an installation.

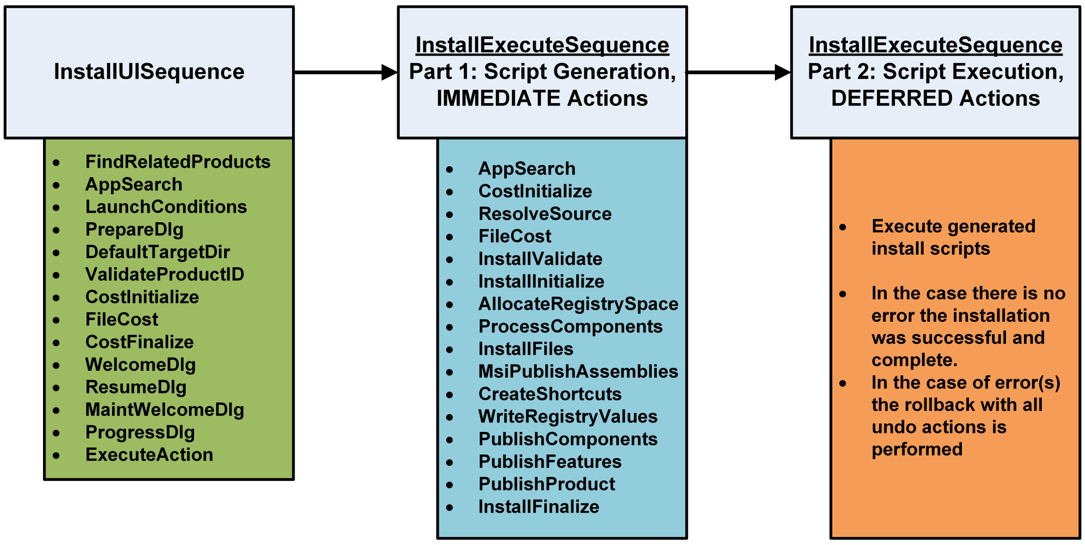


Figure 8.1: MSI Installation Sequence

An installation (which also includes uninstallations/updates/etc.) always runs through the same events. The three main sequences are:

- InstallationUISequence: This is the user interaction mode. Here the user enters all necessary information required for the installation.
- InstallExecuteSequence (Immediate): Here all preparation is done for the actual installation itself.
- InstallExecuteSequence (Deferred): All scripts and changes on the system are applied. In case of errors, a full rollback can be performed.

To round up this basic introduction to MSI, the major constructs within an MSI package are illustrated in Figure 8.2 (inspired by [22]).

An MSI package is structured in a hierarchical way. A product is always deployed as part of a package which consists of a set of features, each of which consists of components which again contain the actual *Installables* (e.g. files, shortcuts, registry keys, ...). Essentially, an MSI package is simply a database which contains a description of the product along with its installation sequence (database tables like *Components, Condition, Custom Action, Dialog, ...*). All files are part of a data stream. The proper database viewer for MSI packages is *Orca*[5].

[1]http://msdn.microsoft.com/en-us/library/aa372024(v=VS.85).aspx
[2]http://technet.microsoft.com/en-us/library/cc759262(WS.10).aspx
[3]http://msdn.microsoft.com/en-us/library/aa372860(v=vs.85).aspx
[4]http://msdn.microsoft.com/en-us/library/aa370834(v=vs.85).aspx
[5]https://msdn.microsoft.com/en-us/library/windows/desktop/aa370557(v=vs.85).aspx

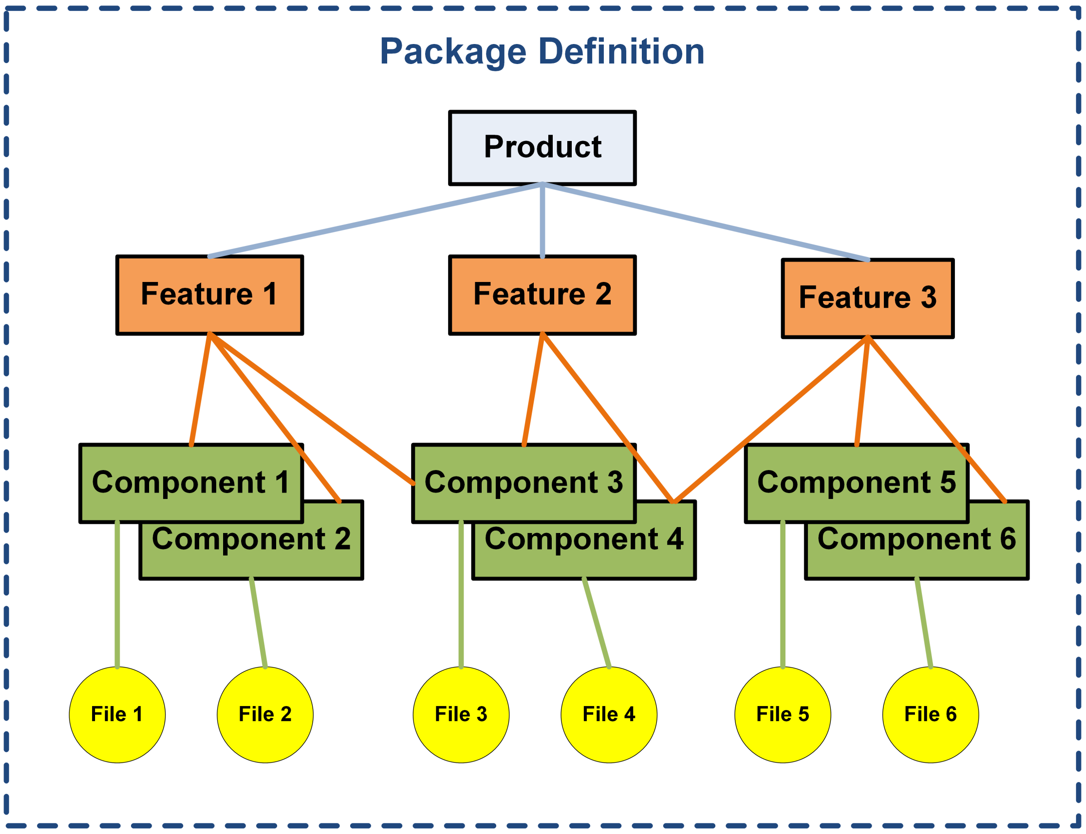


Figure 8.2: Major constructs of an MSI package

8.2 Windows Installer XML

As already discussed in Chapter 8.1, Microsoft came up with the Windows Installer service in 1999. The Windows Installer service is able to execute MSI package installations. However, Microsoft did not come up with an authoring tool to create MSI packages. They defined the standard but left the challenge of providing development environments for authoring MSI packages up to the software development industry.

Rob Mensching[6], who was an employee at Microsoft at this time and was part of the team which developed the Windows Installer service, saw this missing authoring tool as a failure of Microsoft. He believed Microsoft also needed to provide an authoring tool in order to ensure that MSI packages are created in the proper way. Therefore, he started the *Windows Installer XML (WiX)* project in the years 1999-2000 as an open source project, and published it under the *Common Public License (CPL)*. In 2004, Microsoft began to officially fund the project and it was actually one of the very few open source projects of Microsoft.

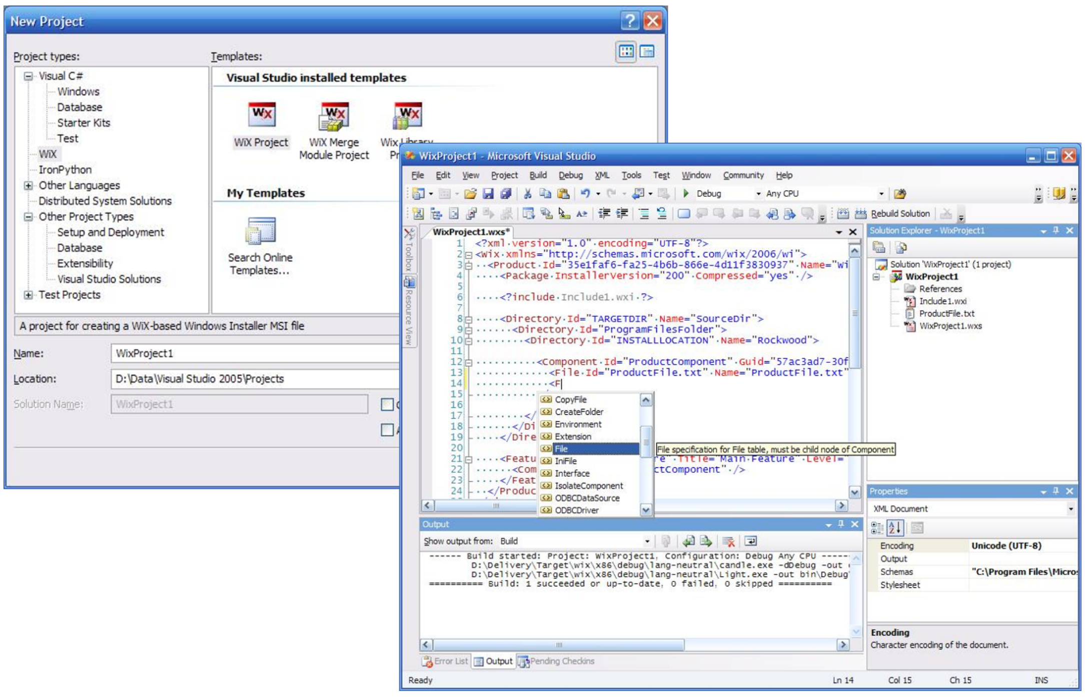
Over the years, the WiX-Toolset grew in popularity. The key factors for its success and also its main advantages over other commercial Windows Installer developer tools were:

- **XML-based:** The source code is XML-based and can be easily edited with any text editor, although proper IDEs als provide auto-complete functionality. The XML-based files are also perfect to be used within source control and can be merged very easily (support of branching

[6]http://robmensching.com/

is given).

- **Compatibility:** Since the founder, Rob Mensching, was also part of the development team of Windows Installer, WiX itself attaches very nicely to the Windows Installer API. The produced installation packages follow the same strict rules.
- **Integration into build automation:** The WiX-Toolset consists of a set of command line tools which integrate very well into build and continuous integration environments.
- **Votive[7] / Visual Studio Integration / MSBuild:** With Votive, the WiX-Toolset integrates itself into Microsoft's development environment, Visual Studio. This integration provides WiX project templates within Visual Studio (see Figure 8.3, taken from [23] under CPL). The integration into Visual Studio hides the complexity of the command line tools.



A project for creating a WiX-based Windows Installer MSI file

Name: WixProject1

Location: D:\Data\Visual Studio 2005\Projects

Solution Name: WixProject1

WixProject1.wxs*

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <Product Id="35e1faf6-fa25-4b6b-866e-4d11f3830937" Name="Wi
    <Package InstallerVersion="200" Compressed="yes" />

    <?include Include1.wxi ?>

    <Directory Id="TARGETDIR" Name="SourceDir">
      <Directory Id="ProgramFilesFolder">
        <Directory Id="INSTALLLOCATION" Name="Rockwood">

          <Component Id="ProductComponent" Guid="57ac3ad7-30f
            <File Id="ProductFile.txt" Name="ProductFile.txt"
            <F
          
          
        </
      </Di
    </Dire

    <Featu                re" Title="Main Feature" Level=
      <Com                ctComponent" />
    </Feat
  </Produc
```

CopyFile
CreateFolder
Environment
Extension
File
IniFile
Interface
IsolateComponent
ODBCDataSource
ODBCDriver

File specification for File table, must be child node of Component

Show output from: Build

```
------ Build started: Project: WixProject1, Configuration: Debug Any CPU -----
        D:\Delivery\Target\wix\x86\debug\lang-neutral\candle.exe -dDebug -out
        D:\Delivery\Target\wix\x86\debug\lang-neutral\Light.exe -out bin\Debug
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
```

Encoding	Unicode (UTF-8)
Output	
Schemas	"C:\Program Files\Micro
Stylesheet	

Encoding
Character encoding of the document.

Figure 8.3: Visual Studio Integration with Votive

- **Bootstrapper/Chainer:** Since 2012, with **Burn**, the WiX-Toolset provides a powerful bootstrapper engine (see Section 8.2.2).
- **No licensing costs:** WiX is free.
- **Team empowering:** Installation tasks can be done on team level. The teams can take over responsibility for their components and update the deployment by themselves (e.g. file lists, registry, COM component IDs, etc.).

8.2.1 MSI Package Creation

As already discussed, WiX integrates nicely into Visual Studio. In order to understand how the whole framework interacts, Figure 8.4 (taken from [23] under CPL) shows an overview of all the tools within WiX and their place in the tool chain.

[7] Votive is part of the WiX-Toolset which allows you to easily create WiX projects, edit WiX files using IntelliSense, and compile/link projects within the Visual Studio IDE. http://wixtoolset.org/documentation/manual/v3/votive/

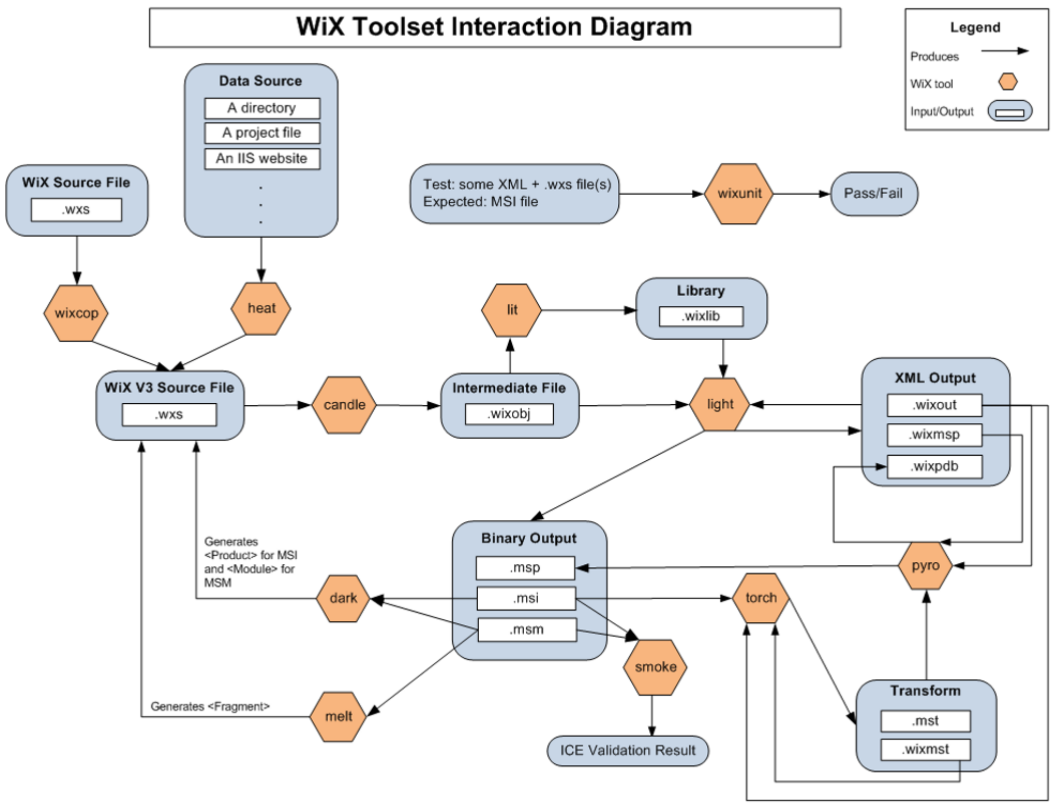


Figure 8.4: Command line tools in WiX

For a simple MSI package creation, only the following four tools are needed:

- **Heat:** With Heat, file lists can be created based on an already existing directory structure. The created .wxs files can be used as part of projects.
- **Candle:** The counterpart to a compiler. In this step, the source is compiled into object files (.wixobj) for later reuse. These object files contain all symbols and references to the original source files.
- **Lit:** A library creation tool. It takes compiled object files and creates .wixlib files. These libs can then be reused by various other projects.
- **Light:** The counterpart to a linker which takes the object files (.wixobj) as input and possibly also libraries (.wixlib). Here the package is being generated (e.g. MSI package).

This part of the WiX-Toolset is rather well documented and due to the nature of the XML-based source code, reference implementations or guidelines can also be found on the internet. As part of OneSetup, some best practice implementations are also provided (see Section 9.8).

8.2.2 Bootstrapper Application (BA)

The original design of Windows Installer included the idea that with a single MSI package all components of an application could be installed as part of the installation. This also included vendor components and other prerequisite packages. Therefore, features and merge modules are part of Windows Installer. Although merge modules are a very powerful asset, they do not fulfil the needs of all variations of vendor components. As part of Rob Menschings blog entry[8], he also stated that the limitations are far more serious when it comes to real professional installers. So the idea was born to create a bootstrapper to provide a state of the art User Interface (UI) for installation procedures which contain one or more deployment packages. The main features of a Burn-based Bootstrapper Application are:

- **User Experience (UX):** Since Windows Installer was created back in 1998-2000, the User Interface of MSI packages is already a bit old fashioned. Burn provides the possibility to implement a custom User Interface tailored to the needs of an application installation.
- **Elevation:** Nearly all bootstrapper applications immediately prompt the user to get access elevation on a client machine. Burn provides functionality to first analyse a client machine and then make an installation plan. Once this plan is set and the user wants to install, the request for elevation is prompted (*User Access Control (UAC)*). This request is only made once and granted for all further installation packages.
- **Progress:** Burn provides proper messaging to the bootstrapper application and, consequently, also to the User Interface, about the progress and status of the installation.
- **Download functionality:** Modern install packages are normally quite small when they are downloaded from a software provider. Only once the user starts the setup procedure, does the installer check the system status and downloads the packages which are needed for the installation process. Burn also offers some built-in functionality to download the needed packages required by an installation from the internet.

Building a custom Bootstrapper Application

So the general idea is that there are several installer packages which should be installed as part of a product installation. These packages are packed into a single .exe package. The user has a nice User Experience for the whole installation procedure by having a rich state-of-the-art User Interface, proper progress information, and by not being interrupted by other events (since all install packages are installed silently in the background). In Figure 8.5, an overview of a Bundle-based installer is shown. In the middle, the actual Bundle installer is shown along with its components. A Bundle installer contains the actual bootstrapper application and all install packages.

When the user initially starts the Bundle installer, the WiX native (C++) application is loaded. This application is the underlying engine which actually interacts with the Windows Installer API. The engine itself performs some launch condition checks. Once these checks are passed, the WiX engine loads the custom bootstrapper. From this point on, the communication between engine and bootstrapper is done via events which are provided by the `BootstrapperApplication` base class. The logical flow (with the corresponding events) is shown in Figure 8.6 (inspired by [24]).

For the minimum working example (e.g. a successful installation), the following steps are needed:

1. Startup: Initial registration of the custom Bootstrapper Application to the events of the `Engine`. Data structures need to be initialised in order to use them in the event-based coding.
2. `Engine.Detect`: The purpose of the detect phase is to determine the current state of the

[8]http://robmensching.com/blog/posts/2009/7/14/lets-talk-about-burn/

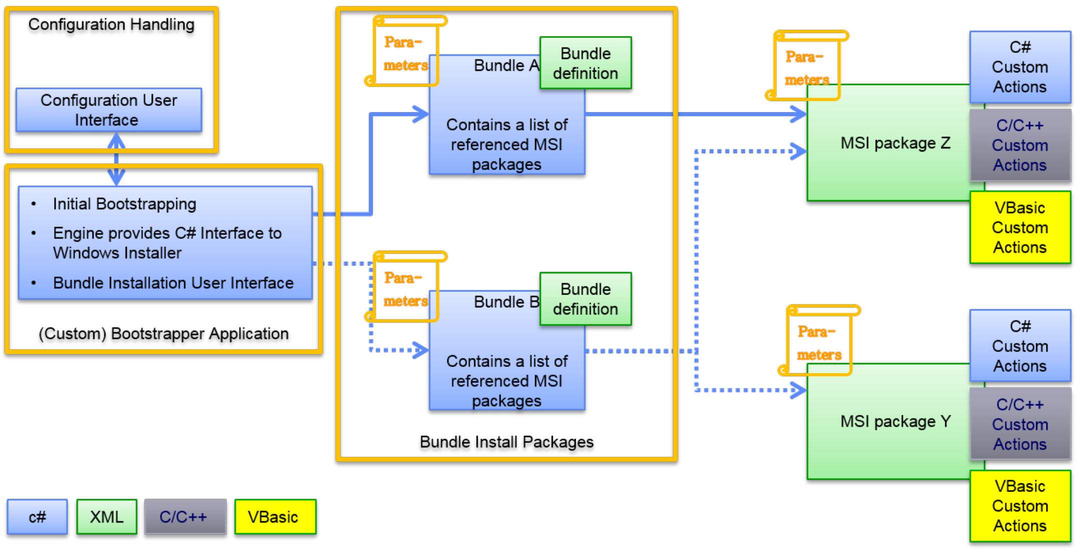


Figure 8.5: Burn-based Bundle installer overview

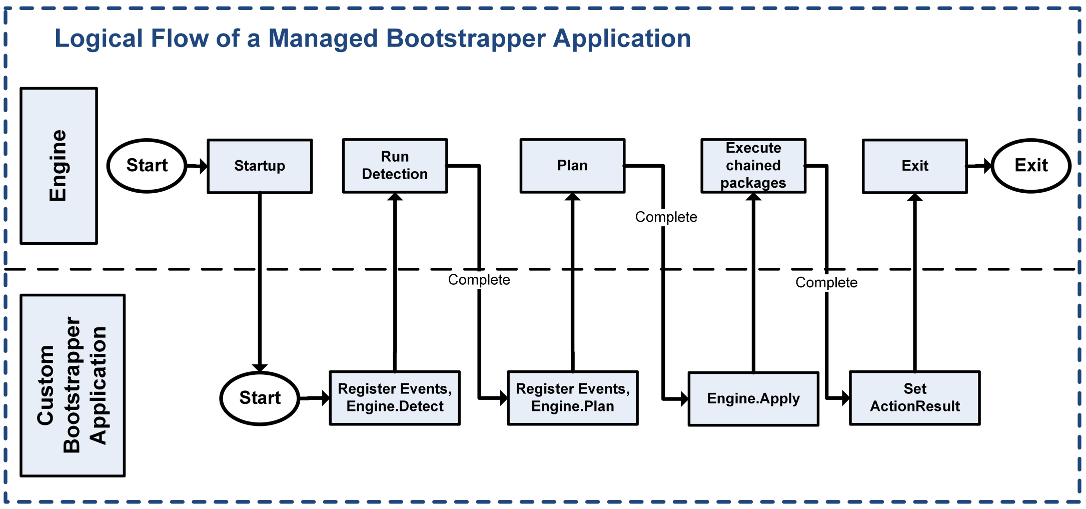

Logical Flow of a Managed Bootstrapper Application
Engine
Start
Startup
Run Detection
Plan
Complete
Execute chained packages
Exit
Exit
Custom Bootstrapper Application
Start
Register Events, Engine.Detect
Complete
Register Events, Engine.Plan
Engine.Apply
Complete
Set ActionResult

Figure 8.6: Bootstrapper Application logical event flow

machine. The engine provides, by using the Windows Installer API, the current state of install packages within the Bundle. The information provided helps to define the actions which need to be taken. Listing 8.1 shows all events which are fired within `Engine.Detect`.

```
//Events related to the Engine.Detect method
event EventHandler<DetectBeginEventArgs> DetectBegin;
event EventHandler<DetectPriorBundleEventArgs> DetectPriorBundle;
event EventHandler<DetectRelatedBundleEventArgs> DetectRelatedBundle;
event EventHandler<DetectPackageBeginEventArgs> DetectPackageBegin;
event EventHandler<DetectRelatedMsiPackageEventArgs> DetectRelatedMsiPackage;
event EventHandler<DetectTargetMsiPackageEventArgs> DetectTargetMsiPackage;
event EventHandler<DetectMsiFeatureEventArgs> DetectMsiFeature;
event EventHandler<DetectPackageCompleteEventArgs> DetectPackageComplete;
event EventHandler<DetectCompleteEventArgs> DetectComplete;
```

Listing 8.1: Events related to the `Engine.Detect` method

3. `Engine.Plan`: As an outcome of `Engine.Detect`, all states of the installer packages are known. The requested future state now needs to be set. This can be done via user interaction or in a predefined way. Listing 8.2 shows all events within the planning phase.

```
//Events related to the Engine.Plan method
event EventHandler<PlanBeginEventArgs> PlanBegin;
event EventHandler<PlanRelatedBundleEventArgs> PlanRelatedBundle;
event EventHandler<PlanPackageBeginEventArgs> PlanPackageBegin;
event EventHandler<PlanTargetMsiPackageEventArgs> PlanTargetMsiPackage;
event EventHandler<PlanMsiFeatureEventArgs> PlanMsiFeature;
event EventHandler<PlanPackageCompleteEventArgs> PlanPackageComplete;
event EventHandler<PlanCompleteEventArgs> PlanComplete;
```

Listing 8.2: Events related to the `Engine.Plan` method

4. `Engine.Apply`: After detecting and defining the future state of packages, the actual work begins. All packages are processed in the pre-defined order from the Bundle definition. The events shown in Listing 8.3 are there to send feedback / progress information to the Bootstrapper Application. By utilising a progress bar, the information can also be presented to the user.

```
//Events related to the Engine.Apply method
event EventHandler<ApplyBeginEventArgs> ApplyBegin;
event EventHandler<ApplyCompleteEventArgs> ApplyComplete;

event EventHandler<RegisterBeginEventArgs> RegisterBegin;
event EventHandler<RegisterCompleteEventArgs> RegisterComplete;
event EventHandler<UnregisterBeginEventArgs> UnregisterBegin;
event EventHandler<UnregisterCompleteEventArgs> UnregisterComplete;
```

Listing 8.3: Events related to the `Engine.Apply` method

5. `Engine.Exit`: Finally, after applying the whole plan (successful, failed, or even cancelled by the user), to shutdown the Bootstrapper Application and exit the installation process, the `Engine.Exit` is called.

This was just a short introduction to the behavioural concept of a custom bootstrapper application. For the creation of the `Avl.OneSetup.BootstrapperApplication`, code examples, and guidelines were quite difficult to find. Therefore, the Bootstrapper Application of the WiX installer itself was used as a starting point. In the meantime, good literature has become available. The books [25] and [26] of Nick Ramirez, who is also a developer for the WiX-Toolset, are a personal recommendation. Also, the blog of John M. Wright [24] provides a very good starting point.

9. OneSetup Platform

“A deployment engineer had a problem, she decided to use WiX, now she has a whole Bundle."

— Klemens Wallner, Coding Happy Hour(s): Friday Afternoon

9.1 Configuration Handling

Deployment packages should be as simple as possible to install. Every choice a user has to make is a possible source of errors. It also assumes that the user has the proper knowledge to make a decision. Therefore, the authors of deployment packages tend not to give a lot of choices to users. Most of the time, the only configuration which needs to be provided is the installation directory. This goal is certainly true for office-side applications where installation processes need to be bullet-proof. For server software, this is not always the case. Of course, the installation process still needs to be bullet-proof, but for server software often configuration parameters are also needed (e.g. database connections, firewall settings, port selections, etc.).

In Figure 8.5, a general overview of a Burn-based installation was shown. To illustrate the configuration needed for the particular packages, the *contract* symbol is shown next to them. The contract is not standardised (this is something which is part of OneSetup and how it handles configurations). A meta contract/configuration file is shown in Listing 9.1.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<AVLOneSetupConfiguration>
  <!-- for which package is this definition for -->
  <Package>
    <!--
    PackageType: -
    Allowed values are:
    UpgradeableConfigFileReadyMSI|StandardMSI|Upgradeable-Bundle|Standard-Bundle
    -->
    <PackageType/>
    <!--
    InstallPackageInContext: Defines if the package should be installed in context of another
    package
    Allowed values are:
    0|1 (false|true)
    -->
    <InstallPackageInContext EngineVariableName="InstallPackage_OneSetup" Value="1"/>

    <UserSettings>
      <!-- Parameters are shown to the user -->
      <Parameter>
        <!-- Parameters can be forwarded to other packages -->
        <ForwardParameterValueTo/>
        <!-- For StandardMSI packages -->
        <EngineVariable />
        <!--
        ConditionalParameters which are only shown when a
        specific value of the parent parameter is choosen
        -->
        <ConditionalParameters>
          <Parameter/>
        </ConditionalParameters>
      </Parameter>
    </UserSettings>
    <CustomerPredefinition>
      <!-- Parameters which are not shown to the user -->
      <Parameter>
        <!-- Parameters can be forwarded to other packages -->
        <ForwardParameterValueTo/>
        <!-- For StandardMSI packages -->
        <EngineVariable />
      </Parameter>
    </CustomerPredefinition>
  </Package>
</AVLOneSetupConfiguration>
```

Listing 9.1: Meta configuration file

- ***Package:***
  Attribute:Name: This is the name of the package. This needs to be the same as the defined *product name* for MSI or *bundle name* for Bundle packages/installers. **Example**: see Listing 9.2

```
<Package Name="AVL OneSetup Example">
</Package>
```

Listing 9.2: Node `<Package>` - Example

- ***PackageType:***
  Attribute:Value: Four different parameters are allowed.
  Allowed Values:
  UpgradeableConfigFileReadyMSI: These types of MSI packages are built using the OneSetup framework. They are able to read all their properties from a `OneSetupConfiguration.zip` file whose path is provided by the `CONFIG_FILE` property. A newer version of the MSI package can upgrade itself without additional information. For a detailed description, see Section 9.9.1.
  StandardMSI: These are standard MSI packages. They get all their properties by providing them in the calling method.
  Upgradeable-Bundle: These types of Bundles can upgrade with a *Single-Button-Upgrade* functionality.
  Standard-Bundle: These types of Bundles can only be upgraded by first uninstalling the old and then installing the new bundle. **Example**: see Listing 9.3.

```
<PackageType Value="UpgradeableConfigFileReadyMSI" />
```

Listing 9.3: Node `<PackageType>` - Example

- ***InstallPackageInContext:***
  Attribute:EngineVariableName: This attribute is directly related to the install condition of a package definition (see Listing 9.4) in WiX. `InstallPackageOneSetup` is the variable name within the package.
  Attribute:Value: The variable `InstallPackageOneSetup` will be substituted within the package with `Attribute:Value`. In the case where the evaluation resolves to `1=1->true` the package will be installed.

```
<MsiPackage Vital="yes" EnableFeatureSelection="yes" DisplayInternalUI="no" Visible="yes" Name="
data\AVL OneSetup Example.msi" SourceFile="\$(var.TargetDir)en-us\AVL OneSetup Example.msi" Cache=
"yes" InstallCondition="InstallPackage_OneSetup = 1 OR OneSetupInstalled" >
  <MsiProperty Name="CONFIG_FILE" Value="[OneSetupXMLConfigFile]"/>
</MsiPackage>
```

Listing 9.4: MSI Package Definition - Example

Example: see Listing 9.5.

```
<InstallPackageInContext EngineVariableName="InstallPackage_OneSetup" Value="1"/>
```

Listing 9.5: Node `<InstallPackageInContext>` - Example

- ***UserSettings:*** This node has no attribute. It is a grouping element. All parameters within this node are **visible** to the user.
- ***CustomerPredefinition:*** This node has no attribute. It is a grouping element. All parameters

within this node are **not visible** to the user. Installation packages are reused in different products. Depending on the products, parameters can be pre-defined differently.

- ***Parameter:*** This is by far the most powerful node in terms of attributes and implementation. Every product that builds their install packages using the OneSetup framework can implement their own parameter implementations and be dynamically loaded (see Section 9.1.1).
  Attribute:InternalParameterName: Within the configuration file this is the unique ID of the parameter. No duplicate names within a package are allowed. On the MSI level this is the property name within the MSI and will be set via the custom action (see Section 9.9.1). *Mandatory*.
  Attribute:Value: Represents the actual parameter value. *Mandatory*.
  Attribute:HumanReadableName: This is the parameter text presented to the user. No multi-language support is implemented. *Mandatory*.
  Attribute:Description: This is the description presented to the user. No multi-language support is implemented. *Mandatory*.
  Attribute:ParameterType: The parameter type defines which specific parameter implementation is used (see Section 9.1.1). In case no `ParameterType` is defined, the default `ParameterType String` is used.
  Attribute:ParameterStyle: Besides the functional definition, the style of the parameter can also be defined.
  Allowed Values:
  Directory: This parameter field in the configurator (see Section 9.3) adds a directory chooser to the parameter field.
  File: This parameter field in the configurator adds a file chooser to the parameter field.
  DropDown: This parameter field in the configurator is a drop down box.
  Password: This parameter field in the configurator is a password field which hides the input of the user.
  NotSpecified: The style for `String` is used.
  Attribute:AllowedParameterValues: A list of allowed parameter values separated by "`|`" can be predefined. This is most commonly used for predefined drop down boxes, e.g. for language selection. These values can also be set inside the implemented parameter.
  Attribute:InputParameter: In case the selected parameters depend on the input of other parameters. A list of parameters can be specified in pairs
  `<LogicalNameForImplementedParameter>=`
  `<InternalParameterNameOfNeededOtherParameter>` separated by "`;`".
  Attribute:InputValue: A list of predefined `<Name:Value>` pairs separated by "`;`". Used for different range limitations of install packages integrated in different products.
- ***ForwardParameterValueTo:*** Parent parameter values can be forwarded to other package definitions.
  Attribute:Package: Defines the target package. *Mandatory*.
  Attribute:ParameterName: Defines the parameter name within the target package. *Mandatory*.

Attribute:WithSuffix: A special suffix can be added to the value. In the case of the install directory parameter, a special subfolder is defined in this way.
Example: see Listing 9.6.

```
<Parameter InternalParameterName="INSTALLDIR" Value="d:\AVL\SANTORIN QMS Web Services{
Instance}" HumanReadableName="Installation directory" Description="Base Install Directory
for the Host Web Services." ParameterType="Directory" >
  <ForwardParameterValueTo Package="AVL SANTORIN Base Web Services" ParameterName="
  INSTALLDIR" WithSuffix="\Base Web Services" />
</Parameter>
```

Listing 9.6: Node <ForwardParameterValueTo> - Example

- ***EngineVariable:*** This parameter forwards the parent value directly to a Burn engine variable. This is needed for packages that do not understand the OneSetup configuration zip file.
  Attribute:Name: This is the name of the engine variable to be set.
  Example: see Listing 9.7 within the package definition of Listing 9.8.

```
<Parameter InternalParameterName="INSTALLDIR" Value="d:\AVL\Cobra Runtime" >
  <EngineVariable Name="INSTALLDIR_CobraRuntime" />
</Parameter>
<Parameter InternalParameterName="INSTALLLEVEL" Value="1" >
  <EngineVariable Name="INSTALLLEVEL_CobraRuntime" />
</Parameter>
```

Listing 9.7: Node <EngineVariable> - Example

```
<MsiPackage Vital="yes" EnableFeatureSelection="yes" DisplayInternalUI="no" Visible="yes"
Name="data\CobraRuntime.msi" SourceFile="\$(var.TargetDir)en-us\CobraRuntime.msi"
ForcePerMachine="yes" Cache="yes" InstallCondition="InstallPackage_CobraRuntime = 1">
  <!-- UserSettings-->
  <MsiProperty Name="INSTALLDIR" Value="[INSTALLDIR_CobraRuntime]"/>
  <MsiProperty Name="INSTALLLEVEL" Value="[INSTALLLEVEL_CobraRuntime]"/>
</MsiPackage>
```

Listing 9.8: MSI Package Definition with Engine Variables - Example

- ***ConditionalParameters:*** Parameters underneath this node are conditional parameters and are only active and visible to the user in case the value of the parent parameter meets the condition.
  Attribute:ActiveIfParentParameterValueEquals: The value of this attribute needs to be the same as the parent value to make the parameters visible to the user.
  Example: see Listing 9.9

```
<Parameter InternalParameterName="LICENSING_MODEL" Value="" HumanReadableName="Licensing Model"
Description="Define the Licensing model." ParameterType="DropDown" AllowedParameterValues="License
 Server|License File" >
   <ConditionalParameters ActiveIfParentParameterValueEquals="License Server" >
      <Parameter/>
   </ConditionalParameters>
</Parameter>
```

Listing 9.9: Node `<ConditionalParameters>` - Example

An example contract/configuration file used in OneSetup is shown in Listing 9.10. The example is taken from the *OneSetup Example MSI* project from Chapter 9.8.

```
<?xml version="1.0" encoding="utf-8" ?>
<AVLOneSetupConfiguration>
  <Package Name="AVL OneSetup Example">
    <PackageType Value="UpgradeableConfigFileReadyMSI" />
    <InstallPackageInContext EngineVariableName="InstallPackage_OneSetup" Value="1"/>
    <UserSettings>
      <Parameter InternalParameterName="LICENSE_SERVER" Value="" HumanReadableName="License Server"
       Description="" />
      <Parameter InternalParameterName="LICENSE_FILE" Value="" HumanReadableName="License File"
      Description="" />
      <Parameter InternalParameterName="INSTALLDIR" Value="d:\AVL\OneSetup" HumanReadableName="
      Installation Directory" Description="" ParameterType="Directory" />
    </UserSettings>
    <CustomerPredefinition>
    </CustomerPredefinition>
  </Package>
</AVLOneSetupConfiguration>
```

Listing 9.10: Example Configuration File

9.1.1 Implementing Parameters

The basic idea of the configuration framework is that applications which are using the OneSetup framework can use the same User Interface for configuration and also reuse existing parameters wherever possible. In cases where the parameter implementations are not sufficient, consumers can contribute to the OneSetup source and enhance particular, or even introduce new, parameters. There is also the possibility to implement parameters on the application side and dynamically load these parameters. This can be achieved by using the factory design pattern which produces parameters as shown in Figure 9.1.

To implement its own parameter factory, the `IParameterFactory` interface needs to be implemented. The derived `abstract class AParameterFactory` implements some basic functionality such as XML parsing. During runtime, all co-located DLLs (also subdirectories) will be searched for possible interface implementations of `IParameterFactory`. To reduce the search effort, only files which follow the naming pattern `Avl.OneSetup.ParameterFactory.*.dll` are considered.

The `abstract Parameter` class already includes XML file parsing and other useful functionality. For an actual parameter implementation like `ParameterString`, three methods need to be

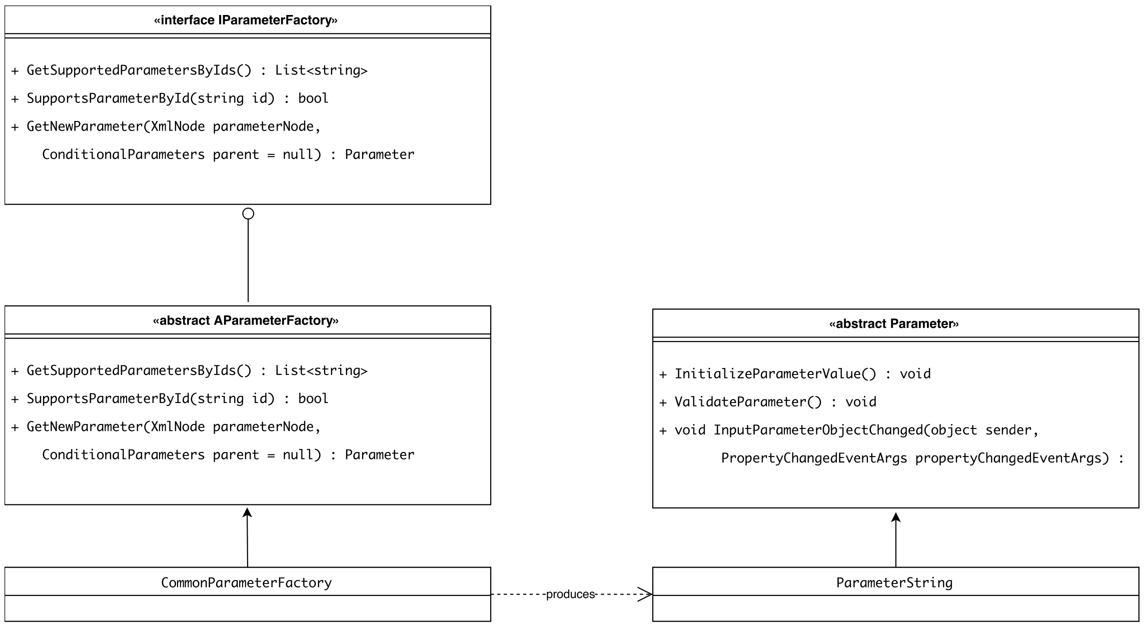

«interface IParameterFactory»
+ GetSupportedParametersByIds() : List<string>
+ SupportsParameterById(string id) : bool
+ GetNewParameter(XmlNode parameterNode,
ConditionalParameters parent = null) : Parameter
«abstract AParameterFactory»
+ GetSupportedParametersByIds() : List<string>
+ SupportsParameterById(string id) : bool
+ GetNewParameter(XmlNode parameterNode,
ConditionalParameters parent = null) : Parameter
«abstract Parameter»
+ InitializeParameterValue() : void
+ ValidateParameter() : void
+ void InputParameterObjectChanged(object sender,
PropertyChangedEventArgs propertyChangedEventArgs) :
CommonParameterFactory
produces
ParameterString

Figure 9.1: Configuration Handling Factory Pattern

overwritten:

- **InitializeParameterValue():** This method is only called once during the construction/initialisation of the parameter. No other `InputParameter` is valid yet. Within initialisation the parameter value is set/modified based on, for example, machine specific settings.
- **ValidateParameter():** This method is called at the latest by the end of the configuration phase. During the parameter validation, the `ParameterValidation` enum needs to be correctly set. There is also a string message for the tooltip of the configurator (can also be used for error messages). For the `ParameterValidation` enum, three states are allowed:

  `ParameterValidation.Successful`: Identifies that the parameter validation was successful and no errors occured.

  `ParameterValidation.Warning`: Identifies that something went wrong during the validation but is not severe enough that an installation would fail.

  `ParameterValidation.Failed`: The parameter validation failed. A Proper error message should be in the tooltip. The user gets an error message.
- **InputParameterObjectChanged(object, PropertyChangedEventArgs):** This event is fired when the parameter itself has changed and/or a parameter on which the current parameter depends on (e.g. via `Attribute:InputParameter`) has changed.

As part of the parameter configuration, the correct `ParameterType` has to be set. As a final step, the parameter factory and its dependencies need to be added to the **ParameterFactoryFiles.wxs** shown in Listing 9.25 in Section 9.4.

9.2 Configuration Handling: Supported Parameters

As part of the common OneSetup framework, a lot of parameters are already implemented to start with. In the following enumeration, all implemented parameters are listed and can be reused:

- **ParameterType:** `Directory`
  ParameterStyle: `Directory`
  Description: Needs to be a logical drive. Environment variables are expanded, e.g. `%ProgramFiles(x86)%`. Directory browser available in the user interface.
  Example: see Listing 9.11

```
<Parameter InternalParameterName="INSTALLDIR" Value="%ProgramFiles(x86)%\AVL\TFMS Client{
Instance}" HumanReadableName="Installation Directory" Description="Defines the installation
directory." ParameterType="Directory" ParameterStyle="Directory"/>
```

Listing 9.11: Parameter `<Directory>` - Example

- **ParameterType:** `File`
  ParameterStyle: `File`
  Description: Path to a file. File needs to exist. File browser available in the user interface.
  Example: see Listing 9.12

```
<Parameter InternalParameterName="CUS_RULE_ENGINE" Value="" HumanReadableName="Custom Rule
Engine File" Description="Filename (including full path) to custom rule engine file (
otherwise leave it empty or default value)." ParameterType="File" ParameterStyle="File"/>
```

Listing 9.12: Parameter `<File>` - Example

- **ParameterType:** `DomainUserName`
  Description: A domain user specified with the pattern `<domainname>\<username>` is checked for existence.
  Example: see Listing 9.13

```
<Parameter InternalParameterName="COND02_DB_SERVICES_USERNAME" Value="" HumanReadableName="
Santorin Services Account Username" Description="Domain Username" ParameterType="
DomainUserName"/>
```

Listing 9.13: Parameter `<DomainUserName>` - Example

- **ParameterType:** `DomainUserPassword`
  ParameterStyle: `Password`
  InputParameter: `DomainUserName` connected to the `DomainUserName` parameter.
  Description: Together with the `DomainUserName` parameter, the credentials of a domain user are checked. In case both parameters validate successfully, the user exists in the domain, and the password was correctly provided.
  Example: see Listing 9.14

```
<Parameter InternalParameterName="COND02_DB_SERVICES_PASSWORD" Value="" HumanReadableName="
Santorin Services Account Password" Description="Domain User Password" ParameterType="
DomainUserPassword" ParameterStyle="Password" InputParameter="DomainUserName=
COND02_DB_SERVICES_USERNAME" >
```

Listing 9.14: Parameter `<DomainUserPassword>` - Example

- **ParameterType:** `DriveDropDown`
  ParameterStyle: `DropDown`
  Description: A drop down box of all logical drives on the machine is provided. The `AllowedParameterValues` are filled during initialisation.
  Example: see Listing 9.15

```
<Parameter InternalParameterName="DB_DRIVE" Value="" HumanReadableName="Database Drive"
Description="Database drive destination." ParameterType="DriveDropDown" ParameterStyle="
DropDown"/>
```

Listing 9.15: Parameter `<DriveDropDown>` - Example

- **ParameterType:** `DropDown`
  ParameterStyle: `DropDown`
  AllowedParameterValues: A list of items separated by "`|`".
  Description: Generic drop down implementation with predefined values.
  Example: see Listing 9.16

```
<Parameter InternalParameterName="LANGUAGE" Value="ENG" HumanReadableName="Language"
Description="Application language. Valid Values are GER/ENG/FRE." ParameterType="DropDown"
ParameterStyle="DropDown" AllowedParameterValues="ENG|GER|FRE" >
```

Listing 9.16: Parameter `<DropDown>` - Example

- **ParameterType:** `Integer`
  InputValue: A list of pre-defined `<name><value>` pairs separated by "`;`". Key names are `MIN` and `MAX`.
  Description: The input value needs to be an integer. In case `MIN` or `MAX` are defined, the entered integer needs to be within the range `MIN <= value <= MAX`.
  Example: see Listing 9.17

```
<Parameter InternalParameterName="DB_TOTAL_SIZE" Value="5000" HumanReadableName="Database
Size [MB]" Description="Database size in megabytes, min. 2500, max. 99999/free disk space."
ParameterType="Integer" InputValue="MIN=2500;MAX=99999">
```

Listing 9.17: Parameter `<Integer>` - Example

- **ParameterType:** `LicenseFile`
  ParameterStyle: `File`
  Description: The provided file needs to exist. The license file needs to be valid in respect to the expiration date.
  Example: see Listing 9.18

```
<Parameter InternalParameterName="LICENSE_FILE" Value="" HumanReadableName="License File"
Description="" ParameterType="LicenseFile" ParameterStyle="File"/>
```

Listing 9.18: Parameter `<LicenseFile>` - Example

- **ParameterType:** `LicenseServer`
  Description: A license server follows the following naming pattern: `[port]@[machinename]` (where the port can be omitted). Multiple license servers can be specified in a list separated by `;`. The `[machinename]` is pinged as part of the validation.
  Example: see Listing 9.19

```
<Parameter InternalParameterName="LICENSE_SERVER" Value="" HumanReadableName="License Server
" Description="" ParameterType="LicenseServer"/>
```

Listing 9.19: Parameter `<LicenseServer>` - Example

- **ParameterType:** `MachineName`
  ParameterStyle: `DropDown`
  Description: All possible names for the local machine are offered as part of a drop down menu. Therefore, all network interfaces are checked for their *DNSSuffixes* and concatenated with the pure machine name.
  Example: see Listing 9.21

```
<Parameter InternalParameterName="SANTORIN_HOST_NAME" Value="" HumanReadableName="Santorin
Server Name" Description="Name of the local machine" ParameterType="MachineName"
ParameterStyle="DropDown" AllowedParameterValues="" />
```

Listing 9.20: Parameter `<MachineName>` - Example

- **ParameterType:** `Password`
  ParameterStyle: `Password`
  Description: This field provides a hidden mask for user inputs. When the password is entered, it is symmetrically encrypted with the Tiny Encryption Algorithm (TEA) with a fixed key. This allows later decryption within the application. This sort of encryption is mainly used to scramble the password so that it is not easily read. It does not guarantee security in the conventional sense.
  Example: see Listing 9.21

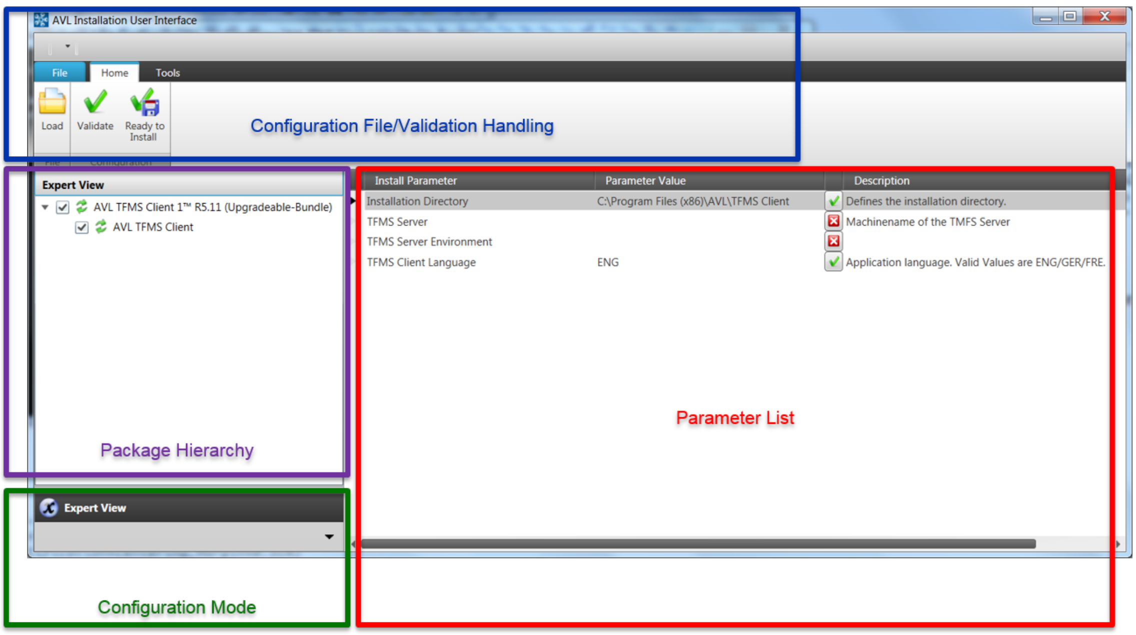
```
<Parameter InternalParameterName="SANTORIN_HOST_NAME" Value="" HumanReadableName="Santorin
Server Name" Description="Name of the local machine" ParameterType="MachineName"
ParameterStyle="DropDown" AllowedParameterValues="" />
```

Listing 9.21: Parameter `<MachineName>` - Example

Due to the amount of already implemented parameters, this is a *Work in Progress* list. Application developers are encouraged to improve and contribute parameter implementations.

9.3 Configurator User Interface

The installation configurator is the User Interface to use for configuration. All parameters defined in the XML configuration files and packed into a zip file (configurator can also deal with single XML/INI files) are shown (see Figure 9.2) in an intuitive way to the user. The configurator is part



Install Parameter	Parameter Value		Description
Installation Directory	C:\Program Files (x86)\AVL\TFMS Client	✓	Defines the installation directory.
TFMS Server		✗	Machinename of the TMFS Server
TFMS Server Environment		✗	
TFMS Client Language	ENG	✓	Application language. Valid Values are ENG/GER/FRE.

Figure 9.2: Configurator User Interface

of the overall concept shown in Figure 8.5 as *Configuration User Interface*. The configurator is also reused as part of the *OneSetup Tools* in order to have a standalone checking tool for commissioning engineers (see Section 10.4).

Four main areas are highlighted in the configuration user interface (see Figure 9.2):

1. **Configuration Mode:** The design allows different configuration modes. The idea is to have different levels of granularity depending on the user . The configurator always starts up in a simple mode with limited configuration possibilities. A power user can then switch to a more advanced view. Currently, only the `Expert View` is implemented.
2. **Package Hierarchy:** The pane on the left shows the package hierarchy. On the top level is always the Bundle package. Underneath, all integrated packages are shown. A user can also define if particular packages should not be installed with the Bundle installation.
3. **Parameter List:** All parameters which are needed for the install are shown in this table. Conditional parameters can also be added.
4. **Configuration File/Validation Handling:** Configuration files can also be stored or previous configurations can be loaded (to ease update processes). Parameters can always be validated by using `Validate`. `Ready to Install` again validates all parameters and finishes and closes the configurator.

As already stated, the configurator currently only supports the `Expert View` as a configuration mode. In addition, the User Interface was originally designed to *just fit the immediate needs*. Therefore, a more user-friendly wizard-based *Non Expert View* was not part of the initial implementation. In any case, together with our user experience expert Sonja Sulzer[1], a new wizard design[2] was created. Figures 9.3 and 9.4 show the next *big design step*. The User Interface is aligned with

[1]Sonja Sulzer is an UX expert within the Concerto product team.
[2]https://confluence.avl.com/its/display/UX/Setup+wizard

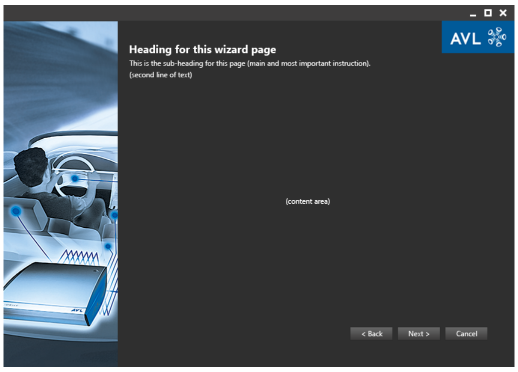

Heading for this wizard page
This is the sub-heading for this page (main and most important instruction).
(second line of text)
AVL
(content area)
< Back
Next >
Cancel

Figure 9.3: Configuration Wizard: At Startup

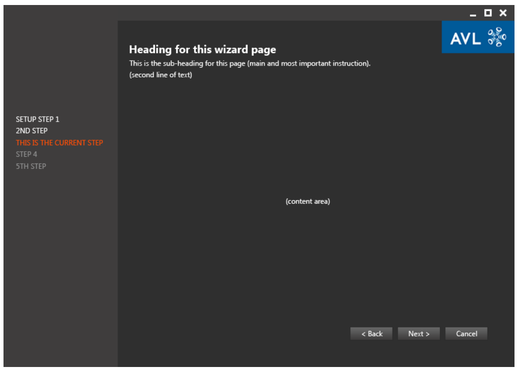


Figure 9.4: Configuration Wizard: Stepping through

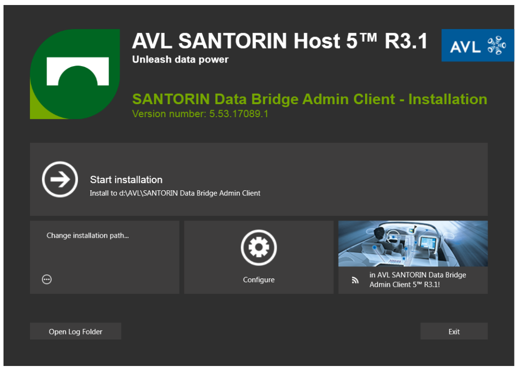

AVL SANTORIN Host 5™ R3.1
Unleash data power
AVL
SANTORIN Data Bridge Admin Client - Installation
Version number: 5.53.17089.1
Start installation
Install to d:\AVL\SANTORIN Data Bridge Admin Client
Change installation path...
Configure
in AVL SANTORIN Data Bridge Admin Client 5™ R3.1!
Open Log Folder
Exit

Figure 9.5: Bootstrapper User Interface: At Startup

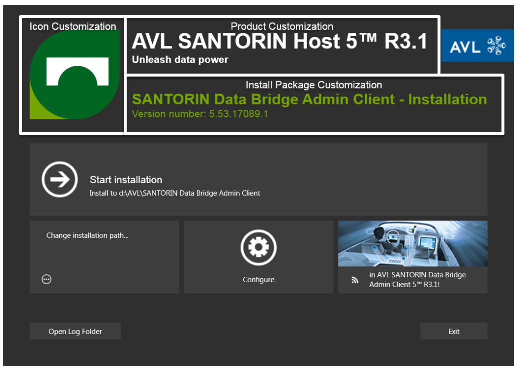

Icon Customization
Product Customization
AVL SANTORIN Host 5™ R3.1
Unleash data power
AVL
Install Package Customization
SANTORIN Data Bridge Admin Client - Installation
Version number: 5.53.17089.1
Start installation
Install to d:\AVL\SANTORIN Data Bridge Admin Client
Change installation path...
Configure
in AVL SANTORIN Data Bridge Admin Client 5™ R3.1!
Open Log Folder
Exit

Figure 9.6: Bootstrapper User Interface: Sections

our corporate design requirements and has a better user experience right from the start. From the configuration handling point of view, no big changes are expected since the package definition just defines, the parameter list, and not their visualisation.

9.4 Bootstrapper User Interface

By starting an installation package as part of the initial start-up, the bootstrapper User Interface (see Figure 9.5) is created and populated. The bootstrapper UI has a parametrised implementation so it can be reused in various products for the application installation bundles.

Figure 9.6 shows the main areas which can be customised. The customisation can be configured as part of the Bundle definition using variables (see Listing 9.22).

```
<!-- UX Settings-->
<Variable Name="ProductBrandPromise" Value="$(var.ProductBrandPromise)" Type="string"
Persisted="yes" />
<Variable Name="Heading" Value="$(var.Heading) $(var.MetaVersion)" Type="string" Persisted="
yes" />
<Variable Name="SubHeading" Value="$(var.SubHeading)" Type="string" Persisted="yes" />
<Variable Name="ProductImage" Value="$(var.ProductImage)" Type="string" Persisted="yes" />
<Variable Name="ProductColor" Value="$(var.ProductColor)" Type="string" Persisted="yes" />
<Variable Name="ConfigureMandatory" Value="true" Type="string" Persisted="yes" />
<Variable Name="ConfigureButtonEnabled" Value="true" Type="string" Persisted="yes" />
<Variable Name="BrowseButtonEnabled" Value="false" Type="string" Persisted="yes" />

<!-- Platform -->
<Variable Name="Platform" Value="x86" Type="string" Persisted="yes" />

<!-- Start Application Button -->
<Variable Name="StartButtonEnabled" Value="false" Type="string" Persisted="yes" />
<!--<Variable Name="RegKeyStartApplication" Value="HKEY_LOCAL_MACHINE\Software\$(var.
ManufacturerName)\$(var.ProductDirectoryName)\ApplicationExecuteable" Type="string" Persisted
="yes" />-->
<!-- Where it the install location stored in case of an upgrade -->
<!--<Variable Name="RegKeyInstallLocation" Value="HKEY_LOCAL_MACHINE\Software\$(var.
ManufacturerName)\$(var.ProductDirectoryName)\InstallLocation" Type="string" Persisted="yes"
/>-->
```

Listing 9.22: Bundle UX Variables

The variables have the following meaning:

- **Product Customisation - ProductBrandPromise:** Marketing brand promise of the product, e.g. *Unleash data power*.
- **Product Customisation - Heading:** Product's brand name with its market release naming `<generation>R<release>.<servicelevel>.<patchlevel>`.
- **Install Package Customisation - SubHeading:** Name of the installer package.
- **Icon Customisation - ProductImage** Every product can use their own product brand icon.
- **Install Package Customisation - ProductColour:** Text colour the installation name. Related to the icon colour.
- **ConfigureMandatory:** Defines if the `Start Installation` button is available immediately after startup. For complex installers, the configuration needs to be run, and only after successful configuration, the `Start Installation` button is enabled.
- **ConfigureButtonEnabled:** For very simple installers where only an installation directory needs to be provided, a configuration UI makes no sense. In this case, even the `Configure` button can be disabled.
- **BrowseButtonEnabled:** Enables the `Change installation path...` button. This is also for very simple installers where no configuration UI is needed.
- **StartButtonEnabled / RegKeyStartApplication:** The start button is not visible in Figure 9.6. The start button allows the starting of an application which was installed right from the bootstrapper UX. The `RegKeyStartApplication` points to a registry key where the start path of the application is provided.
- **RegKeyInstallLocation:** In the current implementation, the configuration is only stored in

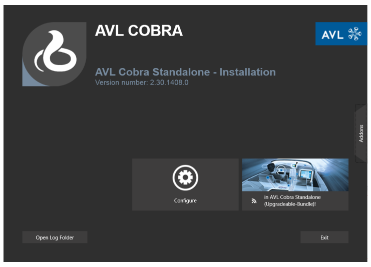

AVL COBRA
AVL
AVL Cobra Standalone - Installation
Version number: 2.30.1408.0
Configure
Open Log Folder
Exit

Figure 9.7: Cobra Installer based on OneSetup

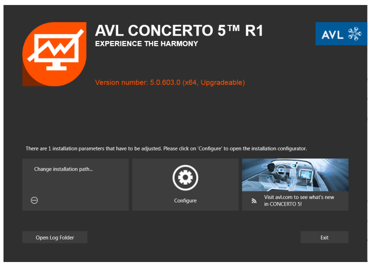

AVL CONCERTO 5™ R1
EXPERIENCE THE HARMONY
AVL
Version number: 5.0.603.0 (x64, Upgradeable)
Change installation path...
Configure
Open Log Folder
Exit

Figure 9.8: Concerto Installer based on OneSetup

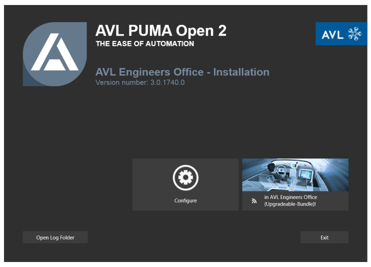


Figure 9.9: Engineers Office Installer based on OneSetup

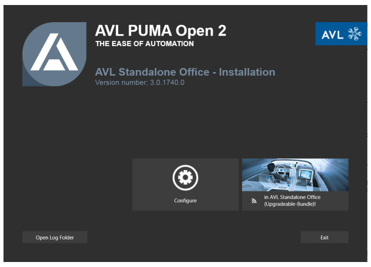


Figure 9.10: PUMA Standalone Office Installer based on OneSetup

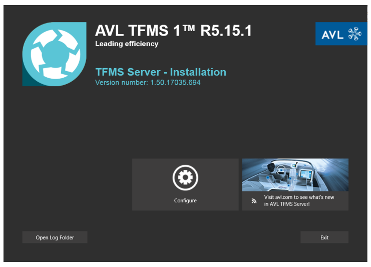


Figure 9.11: TFMS Installer based on OneSetup

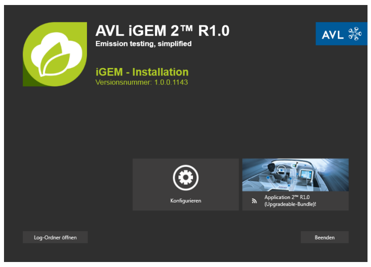


Figure 9.12: iGem Installer based on OneSetup

the Windows registry. In the case of an update, the install directory needs to be read from the Windows registry in order to be presented correctly.

As a result of the modern User Interface, and the possibility to configure it depending on the needs of the product, a lot of products have already adopted the framework. For example, the following figures show example installers within the product lines of Cobra (see Figure 9.7), Concerto (see Figure 9.8), Engineers Office (see Figure 9.9), PUMA Standalone Office (see Figure 9.10), TFMS (see Figure 9.11), iGem (see Figure 9.12), fmi.LAB (see Figure 9.13), inMotion (see Figure 9.14), and SANTORIN (see Figure 9.15).

In addition to the User Interface configuration, the Bootstrapper Application also needs to be extended. Listing 9.23 shows the needed payload groups:

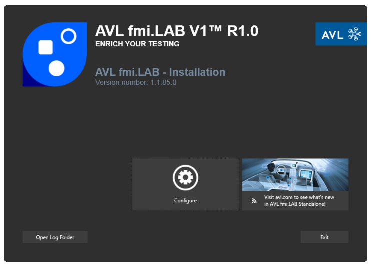


Figure 9.13: fmi.LAB Installer based on OneSetup

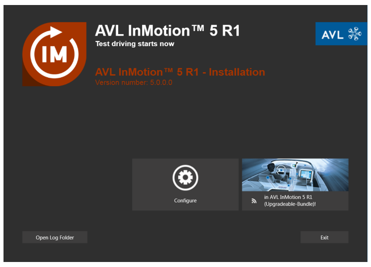


Figure 9.14: inMotion Installer based on OneSetup

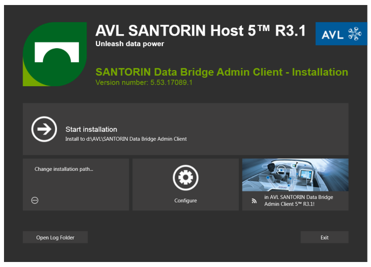

AVL SANTORIN Host 5™ R3.1
Unleash data power
AVL
SANTORIN Data Bridge Admin Client - Installation
Version number: 5.53.17089.1
Start installation
Change installation path...
Configure
Open Log Folder
Exit

Figure 9.15: SANTORIN Installer based on OneSetup

- **PLG_BootstrapperApplication:** This definition and the file list are part of the OneSetup framework and maintained there. It contains all files needed to run the Bootstrapper Application (e.g. bootstrapper DLL with its dependencies, multi-language libraries, images, etc).
- **PLG_Configurator:** Contains the file list for the Configurator (see Section 9.3). The Configurator depends on Infragistics[3] UI Elements, a zipping library, and the VCRedist2010 files.
- **PLG_OneSetupBundleImagesFiles:** A payload list for all customised images which are used. A minimum working example is shown in Listing 9.24.
- **PLG_OneSetupParameterFactoryFiles:** Add the implemented parameter factory (based on Section 9.1.1) to the Bootstrapper Application. A minimum working example is shown in Listing 9.25.

```xml
<BootstrapperApplicationRef Id="ManagedBootstrapperApplicationHost">
    <!-- Tiles Bootstrapper Application -->
    <PayloadGroupRef Id="PLG_BootstrapperApplication" />
    <!-- Configurator for MSI / Bootstrapper Application -->
    <PayloadGroupRef Id="PLG_Configurator"/>
    <!-- Application files / Bootstrapper Application -->
    <PayloadGroupRef Id="PLG_OneSetupBundleImagesFiles"/>
    <PayloadGroupRef Id="PLG_OneSetupParameterFactoryFiles"/>
    <Payload SourceFile="$(var.TargetDir)AVLOneSetupDefaultConfiguration.zip" />
</BootstrapperApplicationRef>
```

Listing 9.23: Customised Bootstrapper Definition

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <Fragment>
    <PayloadGroup Id="PLG_OneSetupBundleImagesFiles">
      <Payload SourceFile="$(var.ProjectDir)images\product_tfms_configuration_tool_256x.png" />
    </PayloadGroup>
  </Fragment>
</Wix>
```

Listing 9.24: PayloadGroup file with Bundle Image files

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <Fragment>
    <PayloadGroup Id="PLG_OneSetupParameterFactoryFiles">
      <!--OneSetup Parameter Factory -->
      <Payload SourceFile="\$(var.TargetDir)Avl.OneSetup.ParameterFactory.OneSetup.dll" />
      <!--OneSetup Parameter Factory Dependencies -->
    </PayloadGroup>
  </Fragment>
</Wix>
```

Listing 9.25: PayloadGroup file with Application Factory

[3]Infragistics UI Controls and Tools, https://www.infragistics.com/

9.5 Bundle Conditions

Bundle conditions are the counterpart of the MSI launch conditions. During its immediate start-up phase, the Burn engine evaluates conditions provided by `bal:Condition` XML nodes. Most of the time, such launch conditions are based on registry searches[4], but they can also be more complicated. In that case they need to be processed as part of the bootstrapper application.

The `AVL_COMPLEX_CONDITION_EVALUATION` launch conditions are complex launch conditions implemented within OneSetup and executed at the bootstrapper startup. Two types are currently implemented in the framework:

- **_MinimumBundleVersion:** This condition blocks the update in case the current Bundle needs a minimum version of the previous Bundle installed on the machine. In case this minimum version is not installed, the software can only be updated via uninstall/install. An example is shown in Listing 9.26.

```
<bal:Condition Message="Upgrade path was intentionally terminated with this version. Please uninstall the old version and install with this installer." >
  <![CDATA[AVL_COMPLEX_CONDITION_EVALUATION_MinimumBundleVersion >= 1.50.15249.0 ]]>
</bal:Condition>
```

Listing 9.26: Bundle Launch Condition

- **_MinimumMsiVersion_<packageName>.msi:** This condition targets an integrated MSI package of the Bundle installation chain. In case the already installed MSI package does not have a minimum version the Bundle installation is blocked. An example is shown in Listing 9.27.

```
<bal:Condition Message="'AVL SANTORIN Base Windows Services' are installed in a too small version (at least version 5.50.14300.0 is necessary). Please uninstall them first manually - the Bundle will reinstall them with higher version." >
  <![CDATA[
  AVL_COMPLEX_CONDITION_EVALUATION_MinimumMsiVersion_AVL_SANTORIN_Base_Windows_Services.msi
  >= 5.50.14300.0 ]]>
</bal:Condition>
```

Listing 9.27: Bundle MSI Launch Condition

9.6 Bundle Command line Switches

One of the fundamental requirements is that the installation packages need to install silently. Therefore, the whole installation process can also be triggered by command line. Table 9.1 lists all the available switches.

9.7 Prerequisites Packages

Most of the time, an application's installation Bundle does not consist only of its application install packages. Applications might have dependencies to vendor packages. As part of the

[4] How To: Block Bootstrapper Installation Based on Registry Key, http://wixtoolset.org/documentation/manual/v3/howtos/

Switch	Description
-q, -quiet, -s, -silent	Performs a silent install. No user interface is shown and the installation runs in the background. The main use-case is for the rollout.
-passive	Same behaviour as the silent install but the progress bar is shown during the installation.
-norestart	In case any packages requests a reboot the request is supressed.
-forcerestart	Restarts the machine after a successful installation. This might be necessary in case an integrated installer does not request a reboot but actually does need one.
-l, -log	Defines the path and file name of the log file. Default is the bundle name with a timestamp.
-uninstall	Uninstalls the bundle.
-repair	Repairs or, if not already installed, installs the bundle.
-configFile <pathToConfigFile>	Provides the (relative or absolute) path to the OneSetup configuration file.

Table 9.1: Bundle command line switches

OneSetup framework, a growing set of vendor packages is provided. The advantage to this is that once the installation logic (detect conditions, install conditions, etc.) is written, every new application does not need to think about its implementation. Additionally, in the case where applications need additional vendor packages, the application development team is encouraged to add this new vendor package to the OneSetup framework so that others might also benefit from this contribution. The OneSetup prerequisite packages follow the naming convention `Avl.Common.Package.Group.<SpecificPrerequisitePackage>.wixlib`.

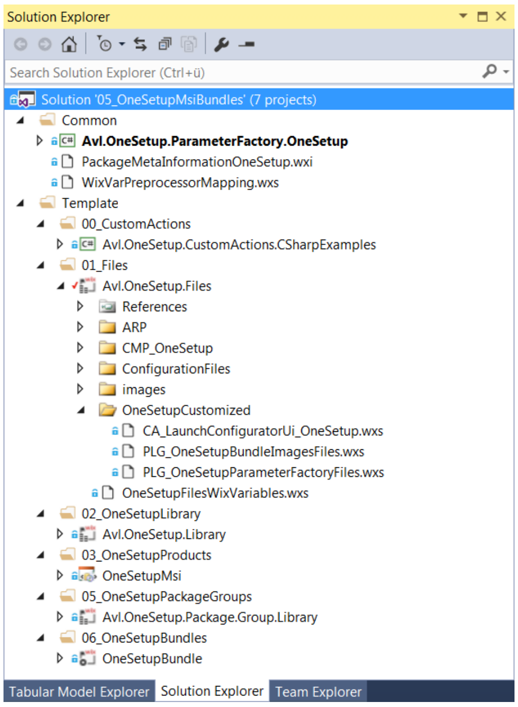
Listing 9.28 gives an example of how predefined prerequisite packages can be used as part of a Bundle chain. Only a specific `PackageGroupId` needs to be referenced. An overview of all supported prerequisite packages is shown in Table 9.2.

```
<Chain>
  <PackageGroupRef Id="Netfx4Full" /> <!-- This is also the prerequisite for the Installer -->
  <PackageGroupRef Id="SingleButtonFailureDump"/>
  <RollbackBoundary />
  <PackageGroupRef Id=" ... applicationPackages ... " />
</Chain>
```

Listing 9.28: Using predefined prerequisite packages in the Bundle chain

9.8 Example Solution

WiX is known to have a flat learning curve. To compensate for this, and in order to provide a good starting point, as part of the OneSetup framework a *best practice example* is provided for creating an application installer package. The `OneSetup-<version>-ExampleInstallation.zip` is provided as part of the OneSetup delivery. In order to use the package, the developer needs to have **Visual Studio 2013/2015** and **WiX Toolset**



```
Solution Explorer
Search Solution Explorer (Ctrl+ü)
Solution '05_OneSetupMsiBundles' (7 projects)
  Common
    Avl.OneSetup.ParameterFactory.OneSetup
    PackageMetaInformationOneSetup.wxi
    WixVarPreprocessorMapping.wxs
  Template
    00_CustomActions
      Avl.OneSetup.CustomActions.CSharpExamples
    01_Files
      Avl.OneSetup.Files
        References
        ARP
        CMP_OneSetup
        ConfigurationFiles
        images
        OneSetupCustomized
          CA_LaunchConfiguratorUi_OneSetup.wxs
          PLG_OneSetupBundleImagesFiles.wxs
          PLG_OneSetupParameterFactoryFiles.wxs
        OneSetupFilesWixVariables.wxs
    02_OneSetupLibrary
      Avl.OneSetup.Library
    03_OneSetupProducts
      OneSetupMsi
    05_OneSetupPackageGroups
      Avl.OneSetup.Package.Group.Library
    06_OneSetupBundles
      OneSetupBundle
Tabular Model Explorer | Solution Explorer | Team Explorer
```

Figure 9.16: Example Solution

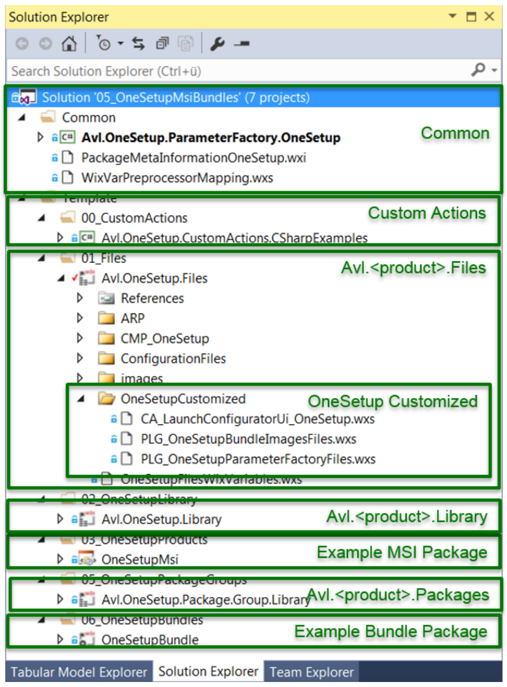


Figure 9.17: Example Solution - Areas

v3.9 R2 installed. It can be started with the `05_OneSetupMsiBundles.sln` solution.

The example solution is shown in Figure 9.16. In Figure 9.17 the main areas are highlighted and are described below:

- **Common:** `Avl.OneSetup.ParameterFactory.<Product>.dll` contains the customised application parameter factory based on the design from Figure 9.1.
  `PackageMetaInformation<Product>.wxi` defines all upgrade GUIDs for all packages used within the solution. `WixVarPreprocessorMapping.wxs`: Passes the version information into WiX variables (which can then be used in the setup authoring).
- **Custom Actions:** All custom action project DLLs are located here within the solution.
- **Avl.<Product>.Files:** This WiX library contains component-wise grouped deployment file lists within .wxs files. All deployment file lists should be kept here. Having them in a separate .wixlib (and not part of the MSI project itself) allows easy reuse over multiple MSI projects. The files within this WiX library are meant to be modified by anybody who works on particular components. The file lists are pure XML files which allow modifications without having deep knowledge about WiX.
- **OneSetupCustomised-Folder:** Every application solution should adapt the three files within this folder. `PLG_OneSetupBundleImagesFiles.wxs` and `PLG_OneSetupParameterFactory.wxs` were already discussed as part of Listings 9.24 and 9.25. For `CA_LaunchConfiguratorUi_OneSetup.wxs` see Section 9.9.1.
- **Avl.<Product>.Library:** This WiX library contains the deployment operations beyond simple file copying (e.g. shortcut creation, Windows service management, ini file modifications,

...). For changes within this library, knowledge about the WiX-Toolset is needed.

- **Example MSI Package:** This is a very simple MSI package project. It contains only a single file for deployment (referenced from `Avl.<product>.Files.wixlib`) but has full functionality in respect to upgradeability, User Interface, and command line functions.
- **Avl.<Product>.Packages:** This package library contains the package definitions of all MSI packages within the solution which can then be reused by multiple Bundle install projects.
- **Example Bundle Package:** Contains the example Bundle project with DotNet, Single Button Failure Dump, and the example MSI package.

9.9 Custom Actions

This Section lists all custom actions from the common OneSetup framework which are ready and meant for reuse within applications. This is a non-complete list. Actually, there are far more custom actions which are not yet fully documented. In this Section only fully documented custom actions are listed.

9.9.1 Avl.OneSetup.CustomActions.LaunchConfiguratorUi

The configurator discussed in Section 9.3 is not only used within the Bootstrapper Application, it is also used as the User Interface part of an MSI installation package. The `Avl.OneSetup.CustomActions.LaunchConfiguratorUi.CA.dll` is actually a zip file which contains the `Avl.OneSetup.CustomActions.LaunchConfiguratorUi.dll` with its dependencies, and a basic OneSetup configuration file.

In order to reuse the `*.LaunchConfiguratorUi.CA.dll` within the customised application installers, the actual application's `AVLOneSetupConfiguration.zip` needs to be *RePacked*. For this purpose, the tool from Section 10.2 can be used. On the other side, as part of the customised MSI packages, the sequencing and launch of the configurator User Interface can be changed by re-implementing the custom action as shown in Listing 9.29. The following things are important:

- Defining of `Avl.OneSetup.CustomActions.LaunchConfiguratorUi.CA.dll` as part of a product binary `Id="OSConf.CA.dll"`.
- Defining a new custom action `Id="LaunchOneSetupConfigurator"` and scheduling this custom action properly (see `InstallUISequence` and `InstallExecuteSequence`).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <Fragment>
    <Property Id="CA_LaunchConfiguratorUi_OneSetup" Value="1"/>

    <Property Id="CONFIG_FILE" Value="NO FILE SET" Secure="yes"/>

    <Binary Id="OSConf.CA.dll" SourceFile="$(var.TargetDir)Avl.OneSetup.CustomActions.
    LaunchConfiguratorUi.CA.dll" />
    <CustomAction Id="LaunchOneSetupConfigurator" BinaryKey="OSConf.CA.dll" DllEntry="
    LaunchConfiguratorUi" Execute="immediate" Return="check"/>

    <InstallUISequence>
      <Custom Action="LaunchOneSetupConfigurator" After="LaunchConditions">(NOT OLD_VERSION_FOUND)
      AND (NOT Installed) AND (UILevel = "5")</Custom>
    </InstallUISequence>
    <InstallExecuteSequence>
      <!-- After LaunchConditions because then we have already checked the other needed products
      installed -->
      <!-- Otherwise we configure and then a LaunchCondition can be fired -->
      <Custom Action="LaunchOneSetupConfigurator" After="LaunchConditions">(NOT OLD_VERSION_FOUND)
      AND (NOT Installed) AND NOT(UILevel = "5")</Custom>
    </InstallExecuteSequence>
  </Fragment>
</Wix>
```

Listing 9.29: Custom Action: Launch Configurator UI OneSetup

PackageGroupId	**Wixlib**	**Description**
DotNetHotfix_Vista_2k8	*.DotNetHotfix.wixlib	Microsoft .NET Framework 3.5 Family Update for Windows XP x64, and Windows Server 2003 x64, Download
JavaRuntimeEnvironment	*.JRE.1.6.0.23.wixlib	Java SE Downloads, Download
KBs	*.KBs.wixlib	Security Update for Windows XP (KB2916036), Download
MSXML40	*.MSXML.wixlib	MSXML 4.0 Service Pack 3 (Microsoft XML Core Services), Download
Netfx20	*.Netfx20.wixlib	Microsoft .NET Framework 2.0 Service Pack 2, Download
Netfx35	*.Netfx35Full.wixlib	Microsoft .NET Framework 3.5 Service pack 1 (Full Package), Download
Netfx4Full	*.Netfx4Full.wixlib	Microsoft .NET Framework 4 (Standalone Installer), Download
Netfx452Full	*.Netfx452Full.wixlib	Microsoft .NET Framework 4.5.2 (Offline Installer) for Windows Vista SP2, Windows 7 SP1, Windows 8, Windows 8.1, Windows Server 2008 SP2, Windows Server 2008 R2 SP1, Windows Server 2012 and Windows Server 2012 R2, Download
Netfx462Full	*.Netfx462Full.wixlib	Microsoft .NET Framework 4.6.2 (Offline Installer) for Windows 7 SP1, Windows 8.1, Windows Server 2008 R2 SP1, Windows Server 2012 and Windows Server 2012 R2, Download
NUnit2.5.10.11092	*.NUnit2.5.10.11092.wixlib	Nunit testing framework, Download
NUnit3.4.1	*.NUnit3.4.1.wixlib	NUnit testing framework, Download
SingleButtonFailure-Dump	*.SingleButtonFailure-Dump.wixlib	AVL Single Button Failure Dump Install Package
VCRedist2005	*.VCRedist2005.wixlib	Microsoft Visual C++ 2005 Redistributable Package, Download
VCRedist2008	*.VCRedist2008.wixlib	Microsoft Visual C++ 2008 Redistributable Package, Download
VCRedist2010	*.VCRedist2010.wixlib	Microsoft Visual C++ 2010 Redistributable Package, Download
VCRedist2013	*.VCRedist2013.wixlib	Visual C++ Redistributable Packages for Visual Studio 2013, Download
VCRedist2015	*.VCRedist2015.wixlib	Visual C++ Redistributable for Visual Studio 2015, Download
VS2013TestAgent	*.VS2013.TestAgent.wixlib	Agents for Microsoft Visual Studio 2013, Download
WindowsInstaller45	*.WindowsIn-staller45.wixlib	Windows Installer 4.5 Redistributable, Download

Table 9.2: Available prerequisite packages

10. OneSetup Tools

“Quotes are smarter than the author ... but not this time."

— Jadranko Lucic, The Croatian Connection

The OneSetup framework offers a set of tools which are either related to the framework usage itself or were developed as a by-product for the deployment. These by-products are mainly for the use of commissioning engineers and operations staff for the preparation, execution, and maintenance of customer installations. The OneSetup Tools always follow the same namespace and naming convention: `Avl.OneSetup.Tools.<Toolname>.exe`.

10.1 Tool: *.CreateProductVersionIncludeWxi.exe

The purpose of the `CreateProductVersionIncludeWxi` tool is to provide functionality in order to have better integration in build automation in respect to versioning and version stamping. All deployment packages should also have proper version stamping. Of course, the version number is different with every build. Therefore, within WiX projects, a version number can only be used as a variable which changes with every build. This variable needs to be set somewhere. In context of WiX, this can be done by creating an include file which contains the variable definition with properly generated values.

Therefore, the tool takes a DLL as an input and reads the version information imprinted on it. With this information the following files are created:

- A *WiX Include File (.wxi)* and
- Simple text files named `Version.txt`, `VersionMajor.txt`, `VersionMinor.txt`,

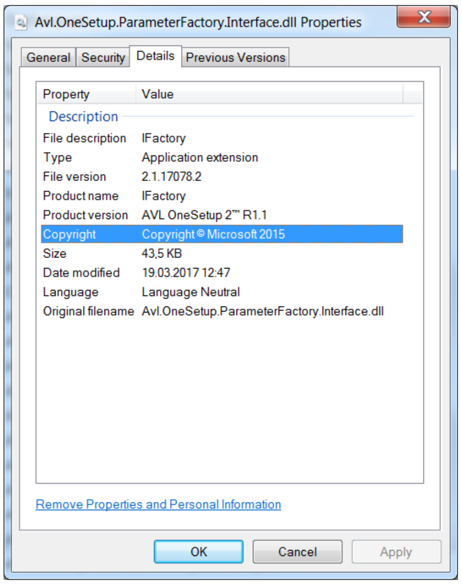
`VersionBuild.txt`, `VersionRevision.txt`, `VersionMeta.txt`, and `VersionMetaSecond.txt`.

Listing 10.1 shows the command line call.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

D:\temp>Avl.OneSetup.Tools.CreateProductVersionIncludeWxi.exe
  -versionFile "Avl.OneSetup.ParameterFactory.Interface.dll"
    -out "OneSetup.wxi"

Input Parameters:
-versionFile: Avl.OneSetup.ParameterFactory.Interface.dll
-out: OneSetup.wxi
Creating product version.wxi file.
Creating file version.txt files.

D:\temp>
```

Listing 10.1: Command line call for `CreateProductVersionIncludeWxi.exe`

The content of the generated files depends on the properties of the provided input DLL. The property tab of `Avl.OneSetup.ParameterFactory.Interface.dll` is shown in Figure 10.1. The information needed is taken from the *File version* and *Product version* entries.


Avl.OneSetup.ParameterFactory.Interface.dll Properties
General
Security
Details
Previous Versions
Remove Properties and Personal Information
OK
Cancel
Apply

Property	Value
Description	
File description	IFactory
Type	Application extension
File version	2.1.17078.2
Product name	IFactory
Product version	AVL OneSetup 2™ R1.1
Copyright	Copyright © Microsoft 2015
Size	43,5 KB
Date modified	19.03.2017 12:47
Language	Language Neutral
Original filename	Avl.OneSetup.ParameterFactory.Interface.dll

Figure 10.1: Properties Tab of a provided DLL

In Listing 10.2, the content of the created `OneSetup.wxi` is shown. This include file can now be used within the deployment projects. Within WiX, the variables `ProductVersion`, `MajorVersion`, `etc.` are defined and also populated with values.

```xml
<?xml version="1.0" encoding="utf-8"?>
<Include>
  <!-- Test Comment -->
  <?define ProductVersion = "2.1.17078.2" ?>
  <?define MajorVersion   = "2" ?>
  <?define MinorVersion   = "1" ?>
  <?define BuildVersion   = "17078" ?>
  <?define RevisionVersion = "2" ?>
  <?define MetaVersion    = "2 R1.1" ?>
  <?define MetaVersionSecond = "no second meta version found" ?>
  <?define Comments       = "Version 2.1.17078.2 from build 17078, 19.03.2017 01:52" ?>
</Include>
```

Listing 10.2: Content of created `OneSetup.wxi`

The text files and their contents are shown in Table 10.1. They can be used within batch scripting as shown in Listing 10.3.

Text file name	Text file content
Version.txt	2.1.17078.2
VersionMajor.txt	2
VersionMinor.txt	1
VersionBuild.txt	17078
VersionRevision.txt	2
VersionMeta.txt	2^{TM} R1.1
VersionMetaSecond.txt	no second meta version found

Table 10.1: Content of the created Version*.txt files

```bat
set /p Version=< "Version . txt "
set /p VersionMajor=< "VersionMajor. txt "
set /p VersionMinor=< "VersionMinor. txt "
set /p VersionBuild=< "VersionBuild . txt "
set /p VersionRevision=< "VersionRevision . txt "
set /p VersionMeta=< "VersionMeta. txt "
```

Listing 10.3: Reading Version*.txt files within a batch script

10.2 Tool: *.RePackCustomActionDll.exe

The main purpose of this tool is to assist in the repacking of the OneSetup framework custom action DLL, `Avl.OneSetup.CustomActions.LaunchConfiguratorUi.CA.dll`. This custom action allows deployment packages which are based on the OneSetup framework to use the same *Configurator User Interface* for their MSI packages (for more information see Section 9.9.1).

The `*LaunchConfiguratorUi.CA.dll` is actually not a real DLL. It is a compressed archive (Cabinet Format[1]) containing the real `Avl.OneSetup.CustomActions.LaunchConfiguratorUi.dll` and all its dependencies. In order to reuse this custom action, the configuration files and parameter factory files of the consuming product must be packed into the `*LaunchConfiguratorUi.CA.dll`. Therefore, the original

[1]https://msdn.microsoft.com/en-us/library/bb267310.aspx

`CA.dll` needs to be unpacked to a temporary folder, additional files are then added to this folder, and the folder is zipped again containing the additional files.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

D:\temp>Avl.OneSetup.Tools.RePackCustomActionDll.exe -Help
Adding a single file :
-CustomActionDll              <FullPathFilename|RelativePathFilename>
-AddFile                      <FullPathFilename|RelativePathFilename>

Adding files based on a PayloadGroup Definition:
-CustomActionDll              <FullPathFilename|RelativePathFilename>
-WixPayloadGroupFile          <FullPathFilename|RelativePathFilename>
-WixVariableNameValuePairs    <VariableName=VariableValue>(;<VariableName=VariableValue>)*
D:\temp>
```

Listing 10.4: `*.RePackCustomActionDll.exe` functions

The tool offers an interface as shown in Listing 10.4. The two main functions are:

- Adding a single file. In the OneSetup framework it is used to add the `AVLOneSetupDefaultConfiguration.zip` and shown in Listing 10.5.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

D:\temp>Avl.OneSetup.Tools.RePackCustomActionDll.exe
  -CustomActionDll "Avl.OneSetup.CustomActions.LaunchConfiguratorUi.CA.dll"
  -AddFile "AVLOneSetupDefaultConfiguration.zip"
Repacking successful.

D:\temp>
```

Listing 10.5: RePacking a single file

- Adding files defined in a payload group (within a .wxs file). The content of such an example payload group file can be seen in Listing 10.6.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <Fragment>
    <PayloadGroup Id="PLG_SantorinParameterFactoryFiles">
      <!--Santorin.Common Parameter Factory -->
      <Payload SourceFile="$(var.TargetDir)Avl.OneSetup.ParameterFactory.Santorin.Common.dll"
      />

      <!--Santorin Parameter Factory Dependencies -->
      <Payload SourceFile="$(var.TargetDir)aods511.dll" />
      <Payload SourceFile="$(var.TargetDir)IIOPChannel.dll" />
      <Payload SourceFile="$(var.TargetDir)SantorinHelper.dll" />
      <Payload SourceFile="$(var.TargetDir)regclnt.exe" />
      <Payload SourceFile="$(var.TargetDir)AsamHelper.dll" />
      <Payload SourceFile="$(var.TargetDir)LoggingHelper.dll" />
      <Payload SourceFile="$(var.TargetDir)Logging.dll" />
      <Payload SourceFile="$(var.TargetDir)SantorinRegistryInterface.dll" />
      <Payload SourceFile="$(var.TargetDir)SAPI.dll" />
      <Payload SourceFile="$(var.TargetDir)TnsNamesEditor.exe" />
      <Payload SourceFile="$(var.TargetDir)TnsNamesUtility.dll" />
    </PayloadGroup>
  </Fragment>
</Wix>
```

Listing 10.6: Content of created example payload group file

All files within this `PayloadGroup` need to be packed into the `*LaunchConfiguratorUi.CA.dll`. Therefore the command line help is shown in Listing 10.7.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

D:\temp>Avl.OneSetup.Tools.RePackCustomActionDll.exe
 -CustomActionDll "Avl.OneSetup.CustomActions.LaunchConfiguratorUi.CA.dll"
 -WixPayloadGroupFile ".\SANTORIN\Files\OneSetupCustomized\PLG_SantorinParameterFactoryFiles.wxs"
 -WixVariableNameValuePairs "TargetDir;D:\temp\."
Repacking successful.

D:\temp>
```

Listing 10.7: RePacking a payload group

As a result, all files are RePacked into `*LaunchConfiguratorUi.CA.dll`. This can be double-checked by unzipping `*LaunchConfiguratorUi.CA.dll` with any standard zipping tool like *Winzip* or *7-Zip*.

10.3 Tool: *.CopyPayloadGroupFiles.exe

The `*.CopyPayloadGroupFiles.exe` tool provides similar functionality to the `*.RePackCustomActionDll.exe` (see Section 10.2) with respect to processing payload group files. This tool copies all files specified within such a payload group file to a defined location. The interface of `*.CopyPayloadGroupFiles.exe` is shown in Listing 10.8.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

D:\temp>Avl.OneSetup.Tools.CopyPayloadGroupFiles.exe -Help
-WixPayloadGroupFile            <FullPathFilename| RelativePathFilename>
-WixVariableNameValuePairs      <VariableName=VariableValue>(;<VariableName=VariableValue>)*
-DestinationFolder              <FullPath>| RelativePath>

D:\temp>
```

Listing 10.8: `*.CopyPayloadGroupFiles.exe` interface

In the example usage shown in Listing 10.9, the same payload group file (see Listing 10.6) is used.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

D:\temp>Avl.OneSetup.Tools.CopyPayloadGroupFiles.exe
  -WixPayloadGroupFile ".\SANTORIN\Files\OneSetupCustomized\PLG_SantorinParameterFactoryFiles.wxs"
  -WixVariableNameValuePairs "TargetDir;D:\TFS\SANT_Main\Bin\ReleaseU\."
  -DestinationFolder "d:\temp\test\."
Copying file <D:\TFS\SANT_Main\Bin\ReleaseU\.\Avl.OneSetup.ParameterFactory.Santorin.Common.dll> to
 <d:\temp\test\.\Avl.OneSetup.ParameterFactory.Santorin.Common.dll>.
Copying file <D:\TFS\SANT_Main\Bin\ReleaseU\.\aods511.dll> to
 <d:\temp\test\.\aods511.dll>.
---- snipped (all file copies are listed) ---
Copying successful.
```

Listing 10.9: `*.CopyPayloadGroupFiles.exe` example

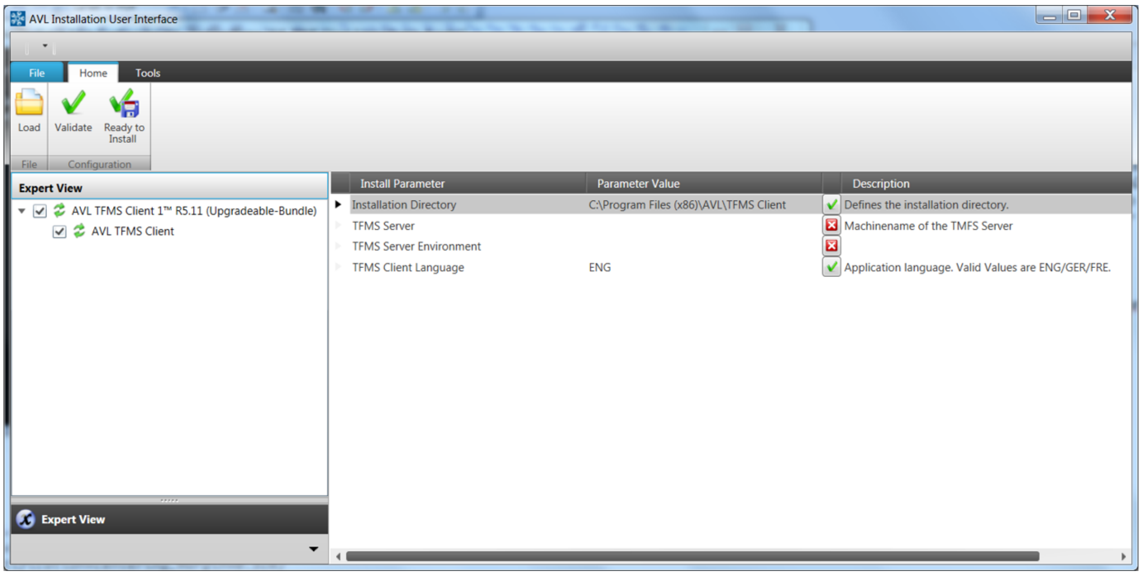
This functionality is used as part of the runtime generation for the OneSetup configurator discussed in Section 9.3.

10.4 Tool: *.Configurator.exe

The *AVL Installation User Interface* discussed in Section 9.3 is embedded in a reusable DLL. The `Avl.OneSetup.Tools.Configurator.exe` provides a command line interface for:

1. **Launching *AVL Installation User Interface* for a particular package definition.** This is used for commissioning and pre-installation checks. The tool can be sent to a customer. Since the *AVL Installation User Interface* actually contains all the logic for the deployment parameter checks, the customer themselves can run the tool as a pre-installation check for the installer. In case of invalid parameter validation, additional preparation actions can be taken before the commissioning engineer is sent to the customer. The goal is to enable the personnel on the customer side to do the commissioning by themselves. Therefore, the parameter validation needs to be very robust. In Listing 10.10, the command line for launching the user interface is shown. In this example, the product configuration for *AVL TFMS Client (Upgradeable-Bundle)* is launched, as shown in Figure 10.2.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

d:\temp\>Avl.OneSetup.Tools.Configurator.exe
```



Expert View

- ☑ AVL TFMS Client 1™ R5.11 (Upgradeable-Bundle)
  - ☑ AVL TFMS Client

Install Parameter	Parameter Value	Description
Installation Directory	C:\Program Files (x86)\AVL\TFMS Client	✓ Defines the installation directory.
TFMS Server		✗ Machinename of the TMFS Server
TFMS Server Environment		✗
TFMS Client Language	ENG	✓ Application language. Valid Values are ENG/GER/FRE.

Figure 10.2: Configurator “AVL Installation User Interface” (launched with listing 10.10)

```
 −packageName "AVL TFMS Client (Upgradeable−Bundle)"

Passed  arguments:  −packageName AVL TFMS Client (Upgradeable−Bundle)
Launching AVLOneSetupConfigurator.
 Starting   AVLOneSetupConfigurator for package <AVL TFMS Client (Upgradeable−Bundle)>.

d:\temp\>
```

Listing 10.10: Launching Configurator for a particular package

The user can now configure the particular deployment package and test the parameter validation. The created deployment configuration can also be saved and reused for rollout purposes.

2. **Changing configuration values without user interface for automated deployment / roll-out.** To change parameter values, some knowledge about the internal structure of the OneSetup configuration zip is needed (see Section 9.1). Based on this, all packages have unique internal parameter names. Therefore, a particular parameter value is uniquely specified by `PackageName` and `InternalParameterName`. In Listing 10.11, an example for changing the installation directory of the *AVL TFMS Client (Upgradeable-Bundle)* package is shown.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

d:\temp\>Avl.OneSetup.Tools.Configurator.exe
 − sourceConfigurationFile .\AVLOneSetupDefaultConfiguration.zip
 − destinationConfigurationFile .\TFMSConfiguration.zip
 −packageName "AVL TFMS Client (Upgradeable−Bundle)"
 −internalParameterName "INSTALLDIR"
 −setParameterValueTo "d:\Temp\AVL\TFMS Client"

Passed arguments:
 − sourceConfigurationFile .\AVLOneSetupDefaultConfiguration.zip
 − destinationConfigurationFile .\TFMSConfiguration.zip
 −packageName AVL TFMS Client (Upgradeable−Bundle)
 −internalParameterName INSTALLDIR
 −setParameterValueTo d:\Temp\AVL\TFMS Client
 ConfigurationFile is not rooted <.\AVLOneSetupDefaultConfiguration.zip>.
 ConfigurationFile is now rooted <D:\TFS\Setup_Main\Bin\ReleaseU\AVLOneSetupDefaultConfiguration.zip>.
 ConfigurationFile is not rooted <.\TFMSConfiguration.zip>.
 ConfigurationFile is now rooted <D:\TFS\Setup_Main\Bin\ReleaseU\TFMSConfiguration.zip>.
Launching AVLOneSetupConfigurator.
Changing <INSTALLDIR> to value <d:\Temp\AVL\TFMS Client> for
 package <AVL TFMS Client (Upgradeable−Bundle)>.
 Successfull .

d:\temp\>
```

Listing 10.11: Changing configuration values without user interface

10.5 Tool: *.IISHelper.exe

The IISHelper tool is a command line tool which provides functionality for the installation and configuration of the Internet Information Services (IIS)[27]. The tool provides three main functions:

- **Install/Check IIS:** IIS consists of a set of various features. Some features are recommended or needed for AVL products. The tool provides functionality to install/check already installed feature sets and can also add and remove features.
- **SSL and Self-Signed-Certificate:** To enable SSL on IIS, a 443 port binding and a certificate is needed. The port binding (including a self-signed-certificate) can be automatically added with this feature.
- **Production-Maintenance-Toggle:** During customer maintenance phases, it is necessary to completely shut off the IIS from external requests. Therefore, the `IIS-IPSecurity` feature of IIS is used. The tool allows the turning on and off of outside communication by toggling between `Production` and `Maintenance` modes.

All IISHelper functions can be listed with the **-Help** or **-?** switches as shown in Listing 10.12.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

d:\temp\>Avl.OneSetup.Tools.IisHelper.exe -Help
Avl.OneSetup.Tools.IisHelper.exe (64 bit)

OPTIONS:

-Cert :
    Generates and installs self signed IIS certificate .

-InstallAVLDefault :
     Installs all recommended IIS features.

- InstallFeatures FEATURES :
     Installs IIS features .
         FEATURES : List of feature names (case-sensitive !).
             Example: Avl.OneSetup.Tools.IisHelper.exe - InstallFeatures "IIS - IPSecurity"
             Example: Avl.OneSetup.Tools.IisHelper.exe - InstallFeatures "IIS - IPSecurity" "IIS-WebServerRole"

-GetEnabledFeatures :
    Shows all enabled IIS features .

------ snipped here -------
```

Listing 10.12: IISHelper functions

10.5.1 Install/Check IIS

IIS on Windows can be installed by using either of the following the User Interface options:

- **Control Panel:** Windows Desktop operating systems like Windows 7
- **Server Manager:** Windows Server operating systems like Windows 2012
- or by using the **Deployment Image Servicing and Management (DISM)** (see [28])

DISM provides a command line driven possibility to install features on Windows operating systems. *IISHelper* wraps this DISM functionality and provides a tailored, lightweight interface for AVL purposes.

In `Avl.OneSetup.Tools.IisHelper.exe.config` config file, the IIS features are predefined. The current AVL default IIS features are:
`IIS-IPSecurity`, `IIS-WebServerRole`, `IIS-WebServerManagementTools`, `IIS-ManagementConsole`, `IIS-WebServer`, `IIS-ApplicationDevelopment`, `IIS-NetFxExtensibility`, `IIS-ASPNET`, `IIS-ISAPIExtensions`, `IIS-ISAPIFilter`, `IIS-CommonHttpFeatures`, `IIS-DefaultDocument`, `IIS-DirectoryBrowsing`, `IIS-HttpErrors`, `IIS-StaticContent`, `IIS-HealthAndDiagnostics`, `IIS-HttpLogging`, `IIS-RequestMonitor`, `IIS-Performance`, `IIS-HttpCompressionStatic`, `IIS-Security`, `IIS-RequestFiltering`, `WAS-WindowsActivationService`, `WAS-ProcessModel`, `WAS-NetFxEnvironment`, and `WAS-ConfigurationAPI`. By modifying the .config file, the default feature set can be changed.

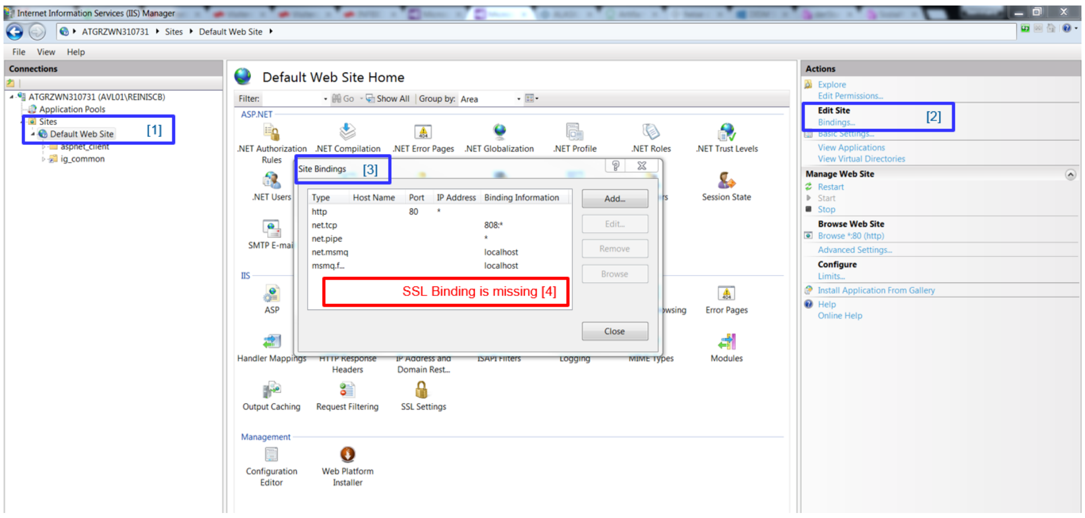
The command line switches, descriptions, and examples are shown in Listing 10.13.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

d:\temp\>Avl.OneSetup.Tools.IisHelper.exe -Help
Avl.OneSetup.Tools.IisHelper.exe (64 bit)

OPTIONS:

-InstallAVLDefault :
      Installs all recommended IIS features.

- InstallFeatures FEATURES:
      Installs IIS features.
          FEATURES: List of feature names (case-sensitive!).
               Example: Avl.OneSetup.Tools.IisHelper.exe - InstallFeatures "IIS - IPSecurity"
               Example: Avl.OneSetup.Tools.IisHelper.exe - InstallFeatures "IIS - IPSecurity" "IIS-WebServerRole"

-GetEnabledFeatures :
     Shows all enabled IIS features.

-GetAllFeatures :
     Shows all IIS features with state.

-CheckAVLDefault :
     Checks if all recommended AVL IIS features are already installed.
```

Listing 10.13: IISHelper `Install/Check IIS` functions

10.5.2 SSL and Self-Signed-Certificate

In order to run SSL secured communication, an SSL binding on port 443 is necessary for the website where the applications are running in IIS. Figure 10.3 shows the `DefaultWebSite` where port 80 is bound, but not port 443.



Type	Host Name	Port	IP Address	Binding Information
http		80	*	
net.tcp				808:*
net.pipe				*
net.msmq				localhost
msmq.f...				localhost

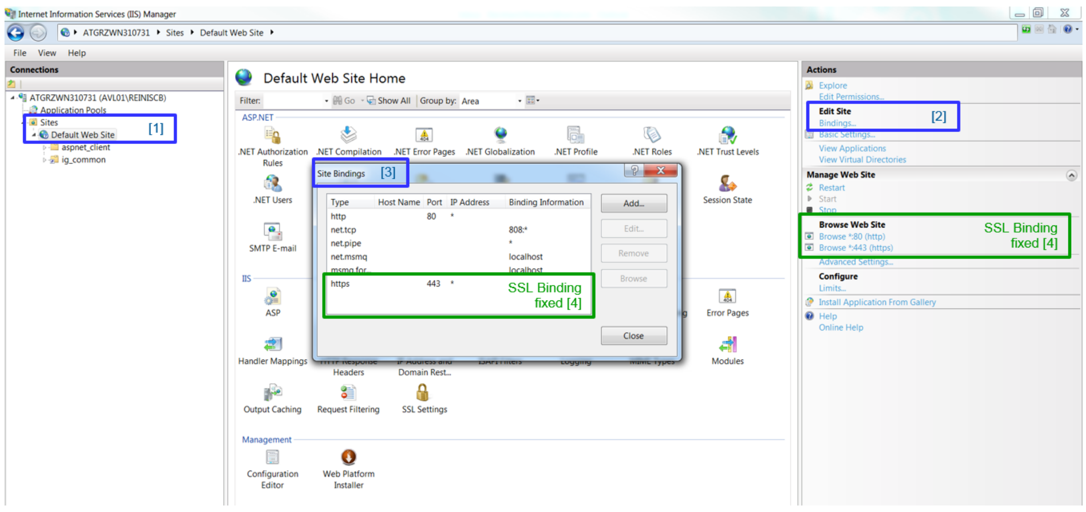
Figure 10.3: IIS with missing SSL binding

In order to create the SSL binding, IISHelper supports the `-Cert` switch. As shown in Listing 10.14, this function checks all bindings on the `DefaultWebSite`. If an SSL binding already exists, this binding is removed and a new binding is created.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

d:\temp\>Avl.OneSetup.Tools.IisHelper.exe -Cert
Checking site: DefaultWebSite
 Iterating through bindings.
Checking binding: <http>.
Checking binding: <net.tcp>.
Checking binding: <net.pipe>.
Checking binding: <net.msmq>.
Checking binding: <msmq.formatname>.
Checking binding: <https>.
Https binding found.
Removing https binding.
All https bindings are removed.
 self signed certificate created.
 Certificate added to store.
Adding https binding for: Default Web Site
Calling: netsh http add sslcert ipport=0.0.0.0:443
   certhash=BFD2CC88625E15189DF88509056B3110FA8F07AD
   appid={c0298694-cad5-4971-8fdf-5a07d9a095eb}

SSL Certificate successfully added

d:\temp\>
```

Listing 10.14: IISHelper adds port binding and certificate

A new binding also needs a certificate. A self-signed-certificate is created (as described here[2] and here[3]) and added to the *Trusted Root Authority Store* of the operating system. As a last step, an update of the certificate-to-endpoint mappings are necessary using *netsh.exe*[4].



Type	Host Name	Port	IP Address	Binding Information
http		80	*	
net.tcp				808:*
net.pipe				*
net.msmq				localhost
msmq.for...				localhost
https		443	*	

Figure 10.4: IIS with fixed SSL binding

Figure 10.4 shows the result of this function. The SSL binding is created and properly initialised.

[2]see http://stackoverflow.com/questions/13806299/how-to-create-a-self-signed-certificate-using-c
[3]see https://msdn.microsoft.com/en-us/library/windows/desktop/aa377124(v=vs.85).aspx
[4]see http://serverfault.com/questions/460090/binding-ssl-certificate-reset-in-win-2012

10.5.3 Production-Maintenance-Toggle

During maintenance phases of IIS Web Service based applications, there is often the need to shut off all outside requests to the application. Therefore, IISHelper provides the switches `-Maintenance` and `-Production`. These switches rely on the `IP Address and Domain Restrictions (IIS-IPSecurity)` feature of IIS. In case the feature is missing, it can be installed as shown in Section 10.5.1.

Figure 10.5 and Figure 10.6 show how to access the `IP Address and Domain Restrictions` via the IIS User Interface. In production mode, IIS is configured to serve all outside requests as shown in Figure 10.6.

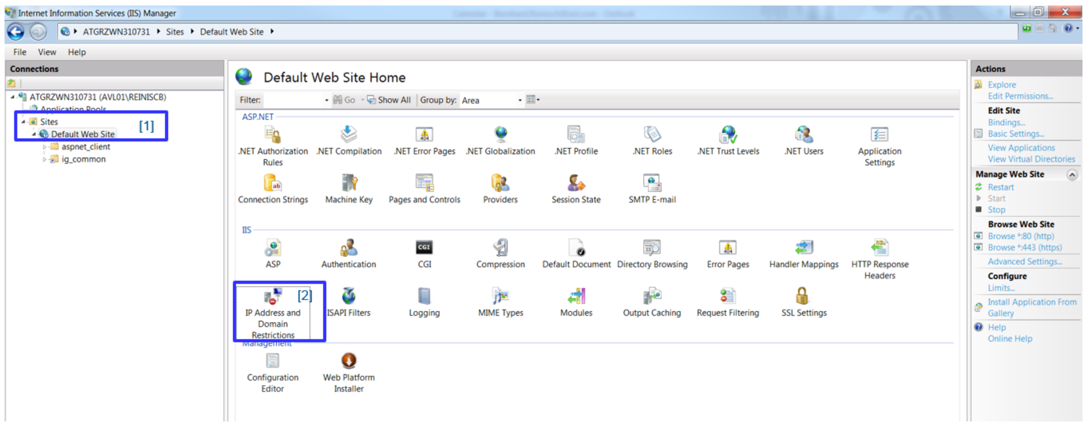
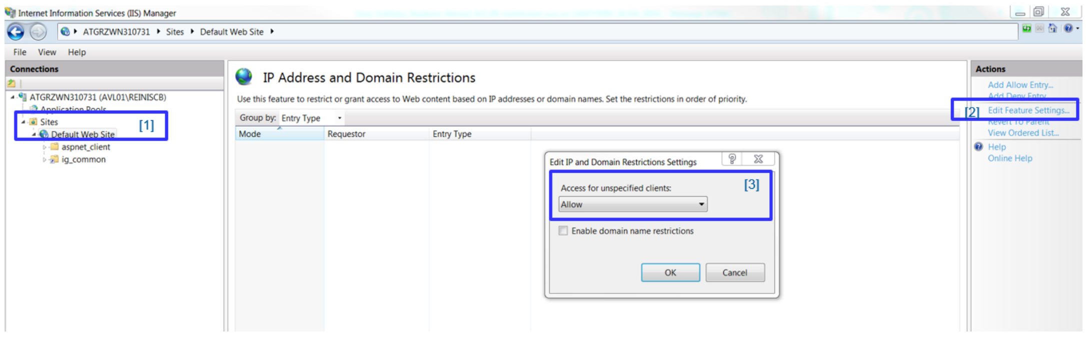


Figure 10.5: IIS IP address and domain restrictions



Use this feature to restrict or grant access to Web content based on IP addresses or domain names. Set the restrictions in order of priority.

Access for unspecified clients: Allow

☐ Enable domain name restrictions

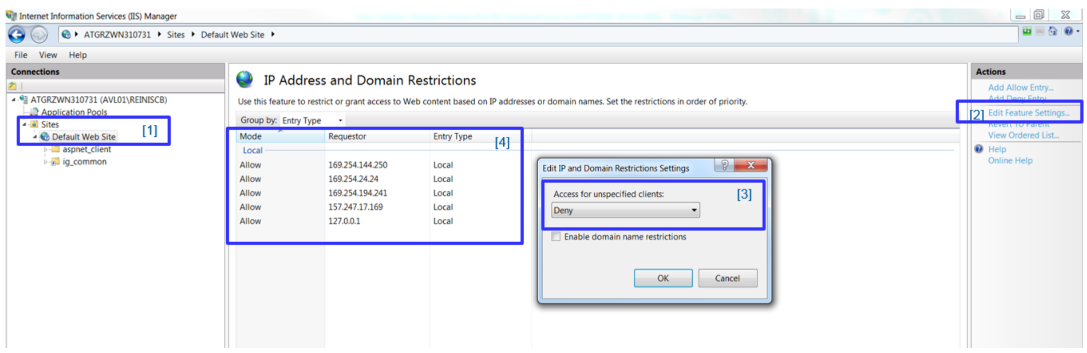
Figure 10.6: IIS IP Address and Domain Restrictions in Production Mode

To bring IIS into maintenance mode, the `-Maintenance` switch can be used as shown in Listing 10.15. This operation turns off all communication from unspecified clients and only allows the communication from local network adapters.

```
Microsoft  Windows [Version 6.1.7601]
Copyright  (c)  2009 Microsoft  Corporation .  All   rights   reserved .

d :\temp\>Avl.OneSetup.Tools. IisHelper .exe  —Maintenance
Allow:   169.254.144.250
Allow:   169.254.24.24
Allow:   169.254.194.241
Allow:   157.247.17.169
Allow:    127.0.0.1
Access  for   unspecified   clients   denied .

d :\temp\>
```

Listing 10.15: IISHelper toggle into Maintenance Mode

Figure 10.7 shows the `IP Address and Domain Restrictions` after IIS is brought into maintenance mode.


Internet Information Services (IIS) Manager

ATGRZWN310731 ▸ Sites ▸ Default Web Site ▸

IP Address and Domain Restrictions

Use this feature to restrict or grant access to Web content based on IP addresses or domain names. Set the restrictions in order of priority.

Group by: Entry Type

Mode	Requestor	Entry Type
Local		
Allow	169.254.144.250	Local
Allow	169.254.24.24	Local
Allow	169.254.194.241	Local
Allow	157.247.17.169	Local
Allow	127.0.0.1	Local

Edit IP and Domain Restrictions Settings

Access for unspecified clients: Deny

☐ Enable domain name restrictions

Figure 10.7: IIS IP Address and Domain Restrictions in Maintenance Mode

Once the maintenance phase is over, IIS can be brought into production mode by using the `-Production` switch. Listing 10.16 shows this operation. The access of all unspecified clients is allowed again and local exceptions are once again removed. All other exceptions which might have been added stay untouched from this operation. As a result, IIS is in production mode once again as shown in Figure 10.6.

```
Microsoft  Windows [Version 6.1.7601]
Copyright  (c)  2009 Microsoft  Corporation .  All   rights   reserved .

d :\temp\>Avl.OneSetup.Tools. IisHelper .exe  —Production
Access  for   unspecified   clients   allowed .
Delete :   169.254.144.250
Delete :   169.254.24.24
Delete :   169.254.194.241
Delete :   157.247.17.169
Delete :    127.0.0.1

d :\temp\>
```

Listing 10.16: IISHelper toggle into Production Mode

10.6 Tool: *.JiraHelper.exe

The `Avl.OneSetup.Tools.JiraHelper.exe` tool provides a command line interface to Jira[5] for:

- **Checking all defects, stories, and features** for incorrectly formated *Integration/Verification Build Numbers*.
- **Selecting all defects, stories, and features** within a particular window of *Integration/Verification Build Numbers*.
- **Finding all defects, and features** which are not linked to a Jira item called Software Release (SWR).

Listing 10.17 shows the interface of `Avl.OneSetup.Tools.JiraHelper.exe`.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

d:\temp\>Avl.OneSetup.Tools.JiraHelper.exe -Help
Example Usage:
-project JiraProjectName
-affectedVersion JiraAffectedVersionField

-operation <operation>
   GetWrongBuildNumbersCount
   GetWrongBuildNumbersList
-buildNumberFormat <Major>.<Minor>.<Build>.<Revision>

-operation <operation>
   GetIssuesInWindowCount
     GetIssuesInWindowList
-from <Major>.<Minor>.<Build>.<Revision>
-to <Major>.<Minor>.<Build>.<Revision>

-operation <operation>
   NotLinkedToSwr
-from <Major>.<Minor>.<Build>.<Revision>
-to <Major>.<Minor>.<Build>.<Revision>
-swr JiraIssueKey

d:\temp\>
```

Listing 10.17: JiraHelper command line interface

For all operations, a particular Jira project (**-project**) and an affected version (**-affectedVersion**) are needed. The Jira project is the overall product name within Jira e.g. `SANTORIN, TFMS, PUMAOpen, etc`. Within a project are several releases which can be identified via the affected version attribute, e.g. `SANTORIN_V552, SANTORIN_5R3, etc`. The project/affected version attributes specify a specific release. Within such a release, all version stamping is following the same pattern **<Major>.<Minor>.<Build>.<Revision>** where `Major` and `Minor` are the same over the whole release. `Build` and `Revision` at least must exist.

The operations **-GetWrongBuildNumbersCount** and **-GetWrongBuildNumbersList** show an overview / detailed list of all items within the project which are not following the version numbering pattern defined with the **-buildNumberFormat** switch. Since various people are

[5]Jira is a proprietary issue tracking product, developed by Atlassian, https://www.atlassian.com/software/jira

involved in the software development process, it might happen that somebody incorrectly enters a build number. However, these build numbers are essential for bug and feature tracking purposes.

Therefore, at least as part of the release creation, a double-check of all build numbers should be performed. Listing 10.18 shows an example of the **-GetWrongBuildNumbersCount** switch functionality. For a more detailed overview, the **-GetWrongBuildNumbersList** switch can be used.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

d:\temp\>Avl.OneSetup.Tools.JiraHelper.exe −project SANTORIN −affectedVersion Santorin_5_R3
 −operation GetWrongBuildNumbersCount −buildNumberFormat 5.53.16245.0
Defect with wrong 'Integration Build Number' (count: 0):
Defect with wrong 'Verification Build Number' (count: 2):
Story with wrong 'Integration Build Number' (count: 34):
Story with wrong 'Verification Build Number' (count: 34):
Feature with wrong 'Integration Build Number' (count: 2):
Feature with wrong 'Verification Build Number' (count: 2):

d:\temp\>
```

Listing 10.18: JiraHelper `-GetWrongBuildNumbersCount` switch

The operations **-GetIssuesInWindow** and *-NotLinkedToSwr* are part of the release process checks. With the switch *-GetIssuesInWindow*, all items within a build number window can be listed. For example, assume that the software has reached the desired content and a certain level of quality has been reached. The build number of the software which will be released is `5.53.17080.1`. The last service release had the build number `5.53.16250.12`. Therefore, all bugfixes and features which were resolved/integrated in the window from `5.53.16250.12` to `5.53.17080.1` is the additional content of the current release `5.53.17080.1`. An example of the *-GetIssuesInWindow* switch is shown in Listing 10.19.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

d:\temp\>Avl.OneSetup.Tools.JiraHelper.exe −project SANTORIN
 −affectedVersion Santorin_5_R3 −operation GetIssuesInWindowCount
 −from 5.53.0.0 −to 5.53.17079.1
Defect integrated BUT NOT closed in Window (count: 4):
Defect integrated and closed in Window (count: 72):
Story integrated BUT NOT closed in Window (count: 0):
Story integrated and closed in Window (count: 0):
Feature integrated BUT NOT closed in Window (count: 0):
Feature integrated and closed in Window (count: 0):

d:\temp\>
```

Listing 10.19: JiraHelper `-GetIssuesInWindowCount` switch

The third operation **-NotLinkedToSwr** is also a crosschecking functionality. In the release process, all bugfixes and features within a release need to be linked to a Jira item named Software Release (SWR). This Jira item is the entry point for the management to check the content of a release. Therefore, the *-NotLinkedToSwr* switch helps to identify all Jira items which are not correctly linked. An example of this operation is shown in Listing 10.20.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

d:\temp\>Avl.OneSetup.Tools.JiraHelper.exe -project SANTORIN
 -affectedVersion Santorin_5_R3
 -operation NotLinkedToSwr
 -from 5.53.0.0 -to 5.53.17079.1
 -swr SANTORIN-20348
Defect integrated BUT not linked (count: 1):
SANTORIN-20120
Defect closed BUT not linked (count: 2):
SANTORIN-18250
SANTORIN-19320
Feature integrated BUT not linked (count: 0):
Feature closed BUT not linked (count: 0):

d:\temp\>
```

Listing 10.20: JiraHelper `-NotLinkedToSwr` switch

11. OneSetup: How To ...

“Ok, it’s fixed now. It will be available in the next build."

— Nadeem Ahamd Siddiqui, Quick Action Required for Healing

This chapter provides a collection of use cases used throughout daily work. They are more oriented to achieve a goal than to provide good explanations.

11.1 ... add a SANTORIN Environment Service

Installing a Windows service as part of an MSI installation is a standard feature of WiX. Various XML elements are available for the installation[1], startup/shut down[2], error[3], and firewall[4] handling of particular Windows services. Various examples can be found on the internet. However this section deals in particular with the service installation of a SANTORIN environment Windows service. As an example, the newly created *ETL Loader* service is described in Listing 11.1. The main building blocks of a SANTORIN environment service are:

- The **service file** embedded in the `File` element.
- The **firewall exception** (`fire:FirewallException`) for allowing all inbound and outbound communication ports for this file.
- The **service install** (`ServiceInstall`) element which deals with the service creation.

[1]ServiceInstall element: http://wixtoolset.org/documentation/manual/v3/xsd/wix/serviceinstall.html
[2]ServiceControl element: http://wixtoolset.org/documentation/manual/v3/xsd/wix/servicecontrol.html
[3]ServiceConfig element: http://wixtoolset.org/documentation/manual/v3/xsd/util/serviceconfig.html
[4]FirewallException element: http://wixtoolset.org/documentation/manual/v3/xsd/firewall/firewallexception.html

- The **service dependency** (`ServiceDependency`) element for adding dependencies. In this case, the dependency to the `AVLStopEnvironment_[SANTORIN_ENVIRONMENT]` needs to be set.
- The **error handling** (`util:ServiceConfig`) element defines what happens in the case of an error. To avoid endless service reboot loops (maybe even endless machine reboot loops) caused by erroneous services, the environment services are configured to take no action after the third attempt.
- The **service control** (`ServiceControl`) element starts/stops the service a part of the installation.

```
<Component Id="CMP_EtlLoader" Guid="*">
  <File Id="fil094F8916C5044401AA899065B56D97B7" KeyPath="yes" Source="!(wix.SantorinExplorerTargetDir)\ETLLoader.exe">
      <fire:FirewallException
      Id="FirExc_EtlLoader" Name="AVL [SANTORIN_ENVIRONMENT] ETL Loader"
      Description="no description"
      Profile="all" Scope="any" IgnoreFailure="yes"/>
  </File>

  <ServiceInstall
    Id="SI_EtlLoader" Name="AVLEtlLoader_[SANTORIN_ENVIRONMENT]"
    Start="demand" Type="ownProcess" ErrorControl="normal" Vital="no" Interactive="no"
    DisplayName="AVL [SANTORIN_ENVIRONMENT] ETL Loader"
        Description="ETL Loader service, transfers measurement to MX host"
    Arguments="-env &quot;[SANTORIN_ENVIRONMENT]&quot; -sysRoot &quot;[INSTALLDIR]\&quot; -service 1"
    Account="[DB_SERVICES_USERNAME]" Password="[DB_SERVICES_PASSWORD_DECRYPTED]">

        <ServiceDependency Id="AVLStopEnvironment_[SANTORIN_ENVIRONMENT]"/>

        <util:ServiceConfig ServiceName="AVLEtlLoader_[SANTORIN_ENVIRONMENT]"
                        FirstFailureActionType="restart"
                        SecondFailureActionType="restart"
                        ThirdFailureActionType="none"/>
  </ServiceInstall>
  <ServiceControl
    Id="SC_EtlLoader" Name="AVLEtlLoader_[SANTORIN_ENVIRONMENT]" Remove="both" Stop="both" Wait="yes" />
</Component>
```

Listing 11.1: *ETL Loader* service definition

The service definition is created within the `Avl.Santorin.Library` project. Within this project, the proper `AVLStartEnvironment` service dependency also needs to be adapted. In the case of the *ETL Loader*, a service dependency is needed for the *SANTORIN MX Host*. Listing 11.2 shows the additional service dependency to the *AVL Start Environment Service* added to the *ETL Loader* service.

```
<Component Id="CMP_StartEnvironmentServices_SantorinMxHost" Guid="*">
  <File Id="fil_StartControlServiceFile_SantorinMxHost" KeyPath="yes"
        Source="!(wix.SantorinExplorerTargetDir)\ControlServiceStart.exe" />
    <ServiceInstall ... > <!-- Attributes of ServiceInstall snipped -->

            <ServiceDependency Id="AVLDispatcher_[SANTORIN_ENVIRONMENT]"/>
            <ServiceDependency Id="AVLEMailDistributor_[SANTORIN_ENVIRONMENT]"/>
            <ServiceDependency Id="AVLFileStorageServer_[SANTORIN_ENVIRONMENT]"/>
            <ServiceDependency Id="AVLExtractor_[SANTORIN_ENVIRONMENT]"/>
            <ServiceDependency Id="AVLJQMgr_[SANTORIN_ENVIRONMENT]"/>
            <ServiceDependency Id="AVLSantorinServer_[SANTORIN_ENVIRONMENT]"/>
            <ServiceDependency Id="AVLEtlLoader_[SANTORIN_ENVIRONMENT]"/>
    </ServiceInstall>
</Component>
```

Listing 11.2: Service dependency to *ETL Loader*

The final step is to add the newly created service to the correct install package. All host related MSI packages have the same shared source file name `SantorinHost.wxs`. Within the WiX file are sections related to the particular applications (Host, MX, QMS, PUMA). The created service needs to be added as part of the `AVL SANTORIN MX*` install packages. Therefore, as shown in Listing 11.3, the *Component Group* is referenced within the `AVL SANTORIN MX*` section.

```
<?if $(var.TargetName) = "AVL SANTORIN MX" Or
     $(var.TargetName) = "AVL SANTORIN MX 2nd Env" Or
     $(var.TargetName) = "AVL SANTORIN MX 3rd Env" ?>
             <!-- snipped content -->

        <!--Environment specific services-->
        <ComponentGroupRef Id="CmpGroupAVLStartEnvironmentSantorinMxHostServiceFiles"/>
        <ComponentGroupRef Id="CmpGroupAVLEMailDistributorServiceFiles" />
        <ComponentGroupRef Id="CmpGroupAVLJobQueueManagerServiceFiles"/>
        <ComponentGroupRef Id="CmpGroupAVLEtlLoaderServiceFiles"/>
<?endif ?>
```

Listing 11.3: Referencing the created *ETL Loader* ComponentGroup

III

Epilogue

12. Results / Conclusion / Further Work

"Don't count pages, count finished chapters."

— Elisabeth Jöbstl, Pilot User

12.1 Results

At the beginning of this year, in comparison to 2011, the whole AVL software development process was different. The planning was previously done in large work packages which took about half a year to go through the entire process. The development tools were also set up for these long-running work package cycles. Code complete dates, integration phases, and system tests where planned in advance and designed to last several days or weeks. Although this process looks rock solid in generating a yearly stream of value, internally a lot of friction emerged during transition phases. System integration always took quite a while and it wasn't rare that the developed solutions did not integrate seamlessly into the whole product. Rework of implementations based on system test results was nearly impossible, and was often moved to the next work package. The current implementation (with minor changes) was the solution that needed to be shipped. Even without the Agile Transformation movement beginning in 2013, there was the need to improve the development process to reduce the friction and improve the quality of software shipments.

Accompanied with other (process) actions (not covered in this thesis), the tools right at the core of the development process were replaced with a state-of-the-art tooling landscape, **driven by the idea to enable** the developers/testers, and later also **the teams, to get their work done by themselves**:

- **Source Control:** All the source code was migrated from Harvest to TFS using the source control module Team Foundation Version Control (TFVC). This migration was done part-wise, and was by far the most unpredictable part of the whole work. This uncertainty had its root cause in the original tooling setup. A lot of build actions / build hacks were no longer applicable to the new tooling landscape. Although this was a rather crucial part, the problems that occurred during the migration were not covered by this thesis since most of the time, the root causes where technical debts (e.g. no Visual Basic support any more). The most interesting parts of the new source control system are the branching concepts, which were introduced along with it. A proper branching model was introduced in order to support the release planning of particular software products. Training and coaching was provided to the teams during the transition. Chapter 2 is also meant to be a reference branching guide for the teams.

- **Build System:** The proprietary Perl-based build system was substituted with build system modules provided as part of TFS. At first, this XAML-based build system was introduced in 2013, and again replaced by the new Build vNext build system released with TFS 2015. As with the source control, the initial migration was the hardest part. To get from the proprietary build system to a quasi-industry standard build system was challenging, e.g. a lot of build steps existed and tools relied on hardcoded paths that were defined within the old build system. Once this big step was taken, the replacement of the XAML build system with the Build vNext build system was quite easy. Getting the software buildable in the Build vNext environment was no challenge at all. It is a good example of how fast particular modules/tools can be exchanged, when the original implementation sticks to industry standards (with little or no customisation). The current build infrastructure relies on a pool of distributed build machines located on different continents which are centrally managed. The scaling and maintenance, once the whole system proved its reliability, was handed over to the central tool service department which also freed resources within the development teams.
- **Dependency Management:** The old build system relied on recompiling source code of other product interfaces in order to reference them in its own product build. This was additional overhead and the knowledge how other product interfaces are built was needed. With the introduction of Artifactory, references were not rebuilt any more, but rather interfaces were provided in binary form by the products. This reduced the complexity of the build system and made the interface management more transparent.
- **Deployment:** As part of the system integration, the installation packages were created using InstallShield. To overcome the limitations of InstallShield, in respect to the high licensing costs and the missing *merge capability* of InstallShield source files, the tool was replaced with WiX. At its core, the WiX-Toolset is strictly XML-based and therefore perfectly fits the needs to be part of a source control system with branching/merging capabilities. In addition, the XML-based source enables the teams to maintain the deployment of their particular components by themselves.

  As part of this work, the OneSetup framework was created in order to emphasise the capabilities of WiX and to create reusable libraries to be consumed by multiple install packages. Although the OneSetup framework is already used in several products, organisational-wise, the development and maintenance relies on the voluntarily contribution of developers within the products.

- **Documentation:** As part of the OneSetup framework, LaTex was also introduced as a documentation tool. Since OneSetup lives *the spirit of open source*, no resources from the documentation department are available. Therefore, all contributors write their documentation as part of the feature development and add it to the source code as part of their check-in. Because of this, the documentation is also always up to date. This positive effect has been recognised and the current documentation process is now under discussion.

The three main building blocks, *Source Control, Build System,* and *Deployment*, were replaced as part of this work. This replacement was already a large step forward towards Continuous Integration. Teams are enabled to do their feature development isolated on team branches, and can create and trigger builds to verify the whole system by using the adapted deployment packages. Even in the old work package oriented process, these improvements would allow to verify smaller increments, more often, and adapt/steer the implementation at an early stage. For Agile software development, the fast implementation cycles are a key factor to success. As part of the Agile Transformation, the need of having an even more automated toolchain was obvious. Therefore, the workflow was extended with:

- **Lab Management:** This module extends the whole infrastructure and was also not part of the original infrastructure. The ability to have physical/virtual machines attached to the whole build process is a big step towards Continuous Delivery. Currently, over fifty virtual machines are used within various testing stages. The TFS build steps handle the machine setup in order to provide clean snapshots. The actual product installation and testing is handled by the PowerShell-based scripts of Oneiroi. A full integration test cycle is run at least nightly to provide fast feedback.

12.2 Conclusions

First and foremost, and this might actually be very hard to swallow for a technician, to really master the Agile Transformation, the single most important thing is the **cultural change**. At the beginning of this work, it was my personal belief, that by providing the right tools, showing alternative workflows, and thus enabling teams to become more powerful, changes will eventually occur and the transition will happen by itself. I was a strong believer, and still am, that technicians/scientists have an intrinsic motivation for advancement and a curiosity for new things. However, I had to learn that an organisation needs to support these self-learning mechanisms. The book *How Google Works* [29] was a good companion during my work and illustrates a lot of good working patterns which are worth learning from.

On the technical side, this thesis is a conclusion of nearly six years of my work at AVL (with a lot of invested leisure time). Unfortunately, not all topics were elaborated on the level of detail as I would have wished (simply due to time constraints). From the tooling point of view, there was, after a decade of not improving the tool landscape, the urgent need and necessity to change the infrastructure. The Agile Transformation made those limitations even more visible.

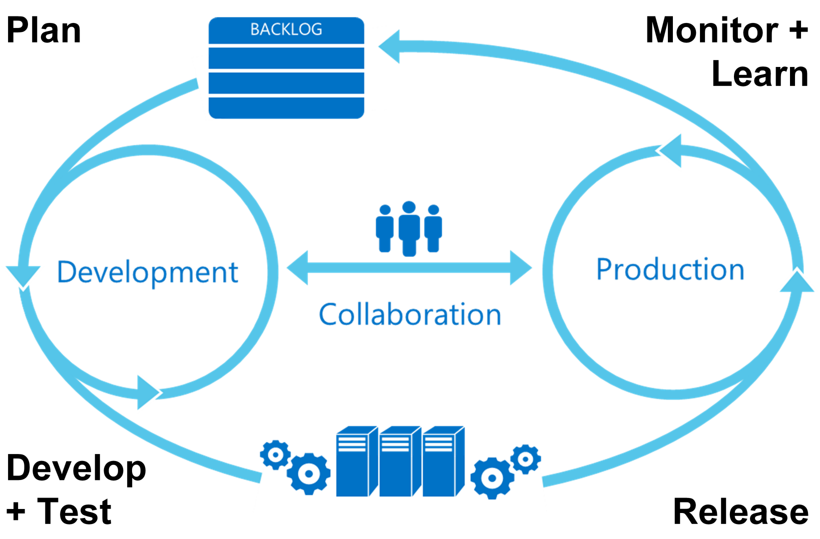


Figure 12.1: Modern DevOps Cycle

12.3 Further work

The *build system* was changed twice during the time of this work. This example illustrates quite well that infrastructural topics can never reach a state of *Done*. Even worse, the technological advancements make it necessary to keep up with the pace of the industry, e.g. in order to provide the latest operating systems, additional security testing environments, or adding new deployment technologies.

Currently, the possibility to extend the current *Release Pipeline* into the cloud is being discussed. The current *Lab Management* needs to be extended in order to deploy not only to virtual machines in-house, but also to machines located in the *Microsoft Azure Cloud*. Since the current infrastructure relies heavily on TFS, there are already build steps available to manage cloud machines in-house. Additionally, during the development of the PowerShell scripting solution Oneiroi, cloud support was always considered as a possible target.

Besides cloud support, the concept of a *Modern DevOps Cycle*, as shown in Figure 12.1 (inspired by [30]), is part of the game. Installation and maintenance, which is currently done by commissioning engineers on-site becomes closer to the development. Commissioning will become a real operations team, which runs the products within the cloud. Incidents need to be handled by operations/development teams together. Performance data can be gathered directly and used to continuously improve the product.

Bibliography

[1] D. Leffingwell, *Agile Software Requirements*. Addison-Wesley, 2010.

[2] L. Brader, R. Leibovitz, and J. L. S. Teruel, *Building a Release Pipeline with Team Foundation Server 2012*. Thought Works, 2013.

[3] L. Brader, H. Hilliker, and A. C. Wills, *Testing For Continuous Delivery With Visual Studio 2012*. Thought Works, 2012.

[4] J. Humble and D. Farley, *Continuous Delivery, Reliable Software Releases through Build, Test and Deployment Automation*. Addison-Wesley, 2010.

[5] C. Technologies, "Ca harvest software change manager." `https://www.ca.com/us/products/ca-harvest-software-change-manager.html`, 2017.

[6] Wikipedia, "Ca harvest software change manager." `https://en.wikipedia.org/wiki/CA_Harvest_Software_Change_Manager`, 2017.

[7] Wikipedia, "Merge (version control)." `http://en.wikipedia.org/wiki/Merge_(revision_control)`, 2017.

[8] Microsoft, "Agent pools and queues.." `https://www.visualstudio.com/en-us/docs/build/concepts/agents/pools-queues`, 2017.

[9] C. Caum, "Puppet blog, continuous delivery vs. continuous deployment: What's the diff?." `https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff`, 2013.

[10] Puppet, *Continuous Delivery: What It Is and How to Get Started*. White Paper, Puppet, 2013.

[11] Microsoft, "Release management, continuous deployment for your applications - ship faster, ship often." `https://www.visualstudio.com/team-services/release-management`, 2017.

[12] Microsoft, "Release definition in release management." `https://www.visualstudio.com/en-us/docs/build/concepts/definitions/release`, 2017.

[13] P. Scheir and B. Reinisch, "Binary repository management with artifactory," *ITS-I Info Letter*, 2014.

[14] JFrog, "Jfrog artifactory, the world's most advanced repository manager.." `https://www.jfrog.com/open-source/#os-top/`, 2017.

[15] C. Sanchez, "Binary repository management - patterns for performance, security, and traceability." `http://refcardz.dzone.com/refcardz/binary-repository-management`, 2017.

[16] Maven, "Introduction to the dependency mechanism." `http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html`, 2017.

[17] Gradle, "Dependency management." `http://www.gradle.org/docs/current/userguide/dependency_management.html`, 2017.

[18] B. Sadogursky, "Dependency management with .net - doing it right." `http://dotnet.dzone.com/articles/dependency-management-net`, 2017.

[19] E. Minick, "Package repositories: The unsung heroes of configuration and release management." `http://de.slideshare.net/Urbancode/package-repositories-the-unsung-heroes-of-configuration-and-release-management`, 2017.

[20] S. Talens-Oliag, "Agile documentation tools," *V Jornades Programari Lliure*, 2006.

[21] E. Minick, "How projects really works." `http://www.projectcartoon.com/`, 2017.

[22] A. Davis, "Beginners guide to windows installer xml (wix)." Powerpoint presentation, March 2011.

[23] Mensching, "Wix toolset, the most powerful set of tools available to create your windows installation experience.." `http://wixtoolset.org/`, 2017.

[24] J. M. Wright, "Writing your own .net-based installer with wix." `http://www.wrightfully.com/part-1-of-writing-your-own-net-based-installer-with-wix-overview`, 2017.

[25] N. Ramirez, *WiX 3.6: A Developer's Guide to Windows Installer XML.* Packt Publishing Ltd, 2012.

[26] N. Ramirez, *WiX Cookbook*. Packt Publishing Ltd, 2015.

[27] Microsoft, "Internet information server (iis) for windows.." `https://www.iis.net/`, 2017.

[28] Microsoft, "Deployment image servicing and management dism." `https://msdn.microsoft.com/en-us/windows/hardware/commercialize/manufacture/desktop/dism-image-management-command-line-options-s14#`, 2017.

[29] E. Schmidt and J. Rosenberg, *How Google Works*. Grand Central Publishing, 2014.

[30] B. Keller, "Modern application lifecycle management and devops." Powerpoint presentation, October 2014.