Automated Dependency Updates for Dart Projects

BACHELOR THESIS

Johann Schramm

Submitted on 21 June 2021

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Supervisor:
Georg Schwarz, M. Sc.
Prof. Dr. Dirk Riehle, M.B.A.

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 21 June 2021

License

Erlangen, 21 June 2021

Abstract

It is important to keep track of new dependency versions, in order to receive security fixes and new features. Similar to other programming languages with dependency mechanisms, libraries and frameworks for the Dart programming languages get distributed as packages using its package manager Pub.

This thesis shows the steps needed to implement an automatic dependency update tool for the Dart programming language. The implemented tool is an extension to the Dependabot project, an application which automatically scans source code repositories of various programming languages for outdated dependencies and creates updates in the form of "Pull Requests" for them.

Finally, this thesis evaluates the impact an automatic dependency update tool could have for the Dart programming language ecosystem and its community by analyzing its performance on the 1020 most starred GitHub Dart repositories. This showed that the tool was able to successfully process over 85% of the detected dependencies. Around 40% of these detected dependencies were up to date, and the tool was able to update the remaining 60% of dependencies to the latest version.

Contents

1 Introduction

It is common for modern applications to depend on multiple third party packages. These dependencies add components, helper libraries or entire frameworks to the project. In most cases, they are developed and maintained by members of the programming language community. Developers can publish and download these packages, to and from their respective language registries.

Nowadays, nearly every popular programming language has an established form of package management. Popular package managers for example include, npm for the JavaScript ecosystem (npm, Inc., 2021a) or pip for the Python ecosystem (PyPA, 2020).

This is also the case for the Dart programming language (The Dart project authors, 2021b) and its package manager pub (The Dart project authors, 2021e). The language is known for its popular Flutter framework (The Dart project authors, 2021a), a UI toolkit for developing high performance and cross-platform applications on mobile and desktop devices, as well as the web platform (The Flutter authors, 2021a).

As Flutter and the Dart language itself are evolving rapidly, with new major versions of the Dart language the Flutter framework releasing regularly (Google Developers, 2021). The community has to adapt and keep up with that velocity and publish new packages version to benefit from new language features like null safety (Thomsen, 2021).

As projects get more and more complex, and the number of included packages increases, it gets harder to track dependency updates. This is important, because newer versions can deliver new features to aid developer and user experience, or fix critical security issues.

Numerous tools exist to automate the update process. One of the more known ones is Dependabot (Dependabot Ltd, 2019b), which regularly creates "version update pull requests" for outdated dependency specifications in repositories. The Dependabot tool is open source and can be used with major collaborative source code platforms like GitHub or GitLab.

Since June 2020, Dependabot is integrated directly into the GitHub system itself (Mullans, 2020) and provides the service to every user free of charge. Currently, the Dependabot tool does not support the Dart language, as there exists no official language module. By implementing such a module, more than 400000 Dart repositories on GitHub (GitHub Search API, 2021) would be able to enable Dependabot and receive automatic dependency updates.

This thesis describes the steps needed to monitor dependency updates and extend the Dependabot tool with a Dart language module, as well as the impact automatic dependency updates could have on the Dart language community.

2 Requirements

2.1 Functional Requirements

This section describes the functional and non-functional requirements for this thesis. Later in section 6.2 it will be evaluated if the requirements were met.

Fetch dependency specifications

Available dependency specification files should be fetched from the repository. These files define each required dependency as well as its version, or possible version range.

Unify dependency specifications

Fetched dependency metadata should be parsed into a common representation, which simplifies further access and work with the data.

Update dependency specifications

Identified dependencies should each be checked for updates or security patches, depending on given version constraints. If updates exist, the dependency declarations in the original specification file will be patched with the new dependency versions.

Provide upgrade pull-requests

A version upgrade for each outdated dependency should be provided as a separate pull request. The user can then easily accept or reject these pull requests.

Repository-based Configurable

The system should be configurable using a configuration file located in the target repository.

2.2 Non-Functional Requirements

Multiple Source Platform Compatible

The implemented system should be compatible with different source code platforms like GitHub or GitLab.

Working Standalone

The implemented module should be usable standalone without any other already existing language modules.

Provide Upstream Pull-Request

A pull request for the official Dependabot repository which adds the implemented module should be provided.

3 Fundamentals

3.1 Dependency management

To simplify complex patterns and speed up development, most modern programming projects integrate external code. These pieces of external code are called dependencies, as the program depends on them at compile- or runtime.

For most developers it is standard practice to install and use external dependencies, or also known as packages, in their projects. Around this practice entire ecosystems and tools were developed.

One of the most known dependency management systems and the largest package registry in the world might be **npm**, the node package manager (npm, Inc., 2021a). It allows developers to easily install JavaScript modules, while keeping track of project-wide installed dependencies and their installed version inside a project specification file called **package.json** (npm, Inc., 2021b).

These dependencies are usually hosted in a public registry where developers can upload their own packages for the community or download ones published by other developers. Because of this community driven interaction, most dependencies regularly receive improvements in the form of updates.

3.2 Dart and its package manager Pub

The Dart programming language is developed by Google (The Dart project authors, 2021c). The Dart language is standardized as the standard ECMA-408 (TC52, 2015). Programs written in Dart can be easily transpiled into JavaScript programs, however Dart also supports just-in-time compilation as well as ahead-of-time compilation to run Dart programs natively on mobile and desktop devices (The Dart project authors, 2019a).

In the recent years the Dart programming language started to gain traction again with the release of the Flutter toolkit, which enables developers to easily write

cross-platform mobile, web and desktop applications using Dart. The Flutter toolkit is also developed by Google (The Flutter authors, 2021a).

Similar to other programming languages, Dart has its own dependency management system. The central part of this ecosystem is the pub package manager, which consists of the pub tool, as well as the online repository (The Dart project authors, 2021e). Users can discover packages and import them from the platform to use in their projects. Users can also publish the packages they created themselves.

According to the **pubspec.yaml** documentation (The Dart project authors, 2021f), every Pub package must consist of at least one **pubspec.yaml** file, like the one seen in listing 3.1. This file specifies the metadata for the current library or application. The metadata must include the package name, current version, Dart runtime version and description. Additionally, the file can optionally contain the required dependencies, a homepage or repository URL.

The pub tool can then use these **pubspec.yaml** files to install the required dependencies for a project.

```
name: dart-basic
description: A basic Dart project

publish_to: 'none'
version: 1.0.0+1

environment:
  sdk: ">=2.7.0 <3.0.0"

dependencies:
  http: ^0.13.1
```

Listing 3.1: A basic pubspec.yaml file.

Every declared dependencies need to specify a source, currently the Pub package manager supports the following four source formats (The Dart project authors, 2020c).

- **Hosted packages**: This is the most common way to specify a dependency, by just providing the dependency name and a version constraint. Using this format, pub will fetch the dependencies by default from the *pub.dev* platform. If available, developers can reference their own package registry to pull the packages from.
- **Git packages**: Instead of providing a version constraint a package can also be referenced using a Git URL. The packages to be pulled can be further

specified with a Git ref and the actual path where the packages are located inside the repository.

- **Path packages**: Local packages can be referenced using an absolute or relative path in the package specification.
- **SDK**: Used for referencing packages which are bundled with another SDK. Currently, this only applies for the Flutter SDK.

A common type of version declaration is needed to detect and interpret changes in package version. Dart and pub use a modified version of SemVer 2.0.0-rc1 (Preston-Werner, 2011-2013) called *pub_semver* (The Dart project authors, 2019b) to define, prioritize and sort versions.

According to this specification, versions are separated into *major.minor.patch* additionally they can have a prerelease suffix, a build number suffix, or both. The different parts of the version number define compatibility between other versions, for example a higher *major* version indicates backwards incompatible changes between the previous versions.

When specifying a Pub package inside a **pubspec.yaml** a specific, or a range of versions can be required. The package manager will then select a compatible version, taking the defined constraint and other dependencies into account. Valid constraints are exact versions, a range between two versions or none at all. Version ranges which allow all backward compatible versions according to SemVer can be defined using *caret syntax* (The Dart project authors, 2020c).

The installed dependencies might also rely on further dependencies which will have to be installed as well. As each dependency can require a different version of a particular dependency, Pub uses a version constraints mechanism to choose a compatible version of a dependency if multiple different dependencies depend on it. This mechanism works by trying to combine the different version requirement ranges and finding a version which is in every range and is only possible due to the rules of SemVer (The Dart project authors, 2020d).

Once the dependency graph is resolved, the pub tool generates a **pubspec.lock** file which contains the exact version of every installed dependency and transitive dependency. If this file is shared with other users, their pub tool can replicate the original dependency environment to ensure that the application works in the same way as it does for the original author.

3.3 Dependabot

Outdated dependencies are a common problem in open source projects and software projects in general. Developers who don't use the latest version of their

dependencies may miss out on new features and put themselves and their projects to potential security risks as newer versions might have fixed vulnerabilities. Although most package managers provide some functionality to upgrade the installed dependencies, available updates are often missed by the user. This can happen especially in older projects which are maintained only sporadically.

The Dependabot project (Dependabot Ltd, 2019b) solves this problem by creating an application that automatically scans source code repositories for outdated dependencies and notifying the maintainers by creating pull requests, which update these dependencies.

The application currently supports dependency updates for a set of package managers, in the context of Dependabot called language ecosystems, including popular ones like *bundler* for Ruby, *cargo* for Rust, *pip* for Python, *npm* for JavaScript, and other languages (GitHub, 2021a).

The Dependabot project is mainly written in Ruby (Ruby Community, 2021a) and split into two parts, the core application and a client (or script) which uses the modules implemented in the core to perform the actual update process. The core project is open source, with the source code available on GitHub in the **dependabot-core** repository for the core (Dependabot Ltd, 2021b), and a sample client in the form of a simple script in the **dependabot-script** repository (Dependabot Ltd, 2021e). The **dependabot-core** repository is licensed under *The Prosperity Public License 2.0.0*, a license that encourages free use and modification and discourages direct commercial use of the software (License Zero, 2018). The **dependabot-script** repository is licensed under the *MIT License*.

At the time this thesis was written, no new language ecosystems got accepted into **dependabot-core** due to internal lack of capacity and in-house expertise. However, new ecosystems can still be developed and maintained as forks of **dependabot-core** by the community (Dependabot Ltd, 2021h).

In early 2019 Dependabot has been acquired by GitHub, since then the application has been integrated directly into the GitHub platform, with additional features like automatic security issue fixing pull requests (Dependabot Ltd, 2019a).

4 Architecture

This chapter describes the architecture of dependency update workflows and a Dart dependency update tool in the form of an extension to the Dependabot project.

4.1 Dependabot

The Dependabot project consists of multiple Ruby modules, these modules can be split into two categories, common modules and language modules. Each of these modules are their own Ruby package. In the case of the Ruby ecosystem, packages are called *Ruby Gem* or simply *Gem* (Silva, Gonçalo and Ruby Community, 2021).

All these Dependabot Ruby Gems are published to the **rubygems.org** repository (Dependabot Ltd, 2021d), a hosting platform for community Ruby Gems (Ruby Community, 2021b). The Dependabot also specifies and publishes **dependabot-omnibus**, a Ruby Gem package (Dependabot Ltd, 2021g), which depends on all available Dependabot Gems and allows easy installation and use of the entire Dependabot source code (Dependabot Ltd, 2021c). To use the Dependabot modules in an update task, a script can depend on and install these Gems.

In this thesis, the **dependabot-core** project is used as the base for the implementation. It was chosen because the core modules already provide a lot of relevant functionality for the use case of implementing a Dart package dependency updater and to the large potential user base the Dependabot application provides, due to it being already part of the popular GitHub platform. Additionally, the project includes already implemented API interfaces for major source code providers like GitHub and GitLab, resulting in a more flexible application that is automatically available to a larger user group.

This section further describes the general workflow of a dependency update and the Dependabot update workflow and the architecture of the separate classes needed to build a new language ecosystem module.

4.1.1 General Dependency Update Workflow

The process of updating a dependency in a project which uses a package manager, can be simplified into four basic steps as described in the Dependabot architecture documentation (Dependabot Ltd, 2021c) and in figure 4.1.

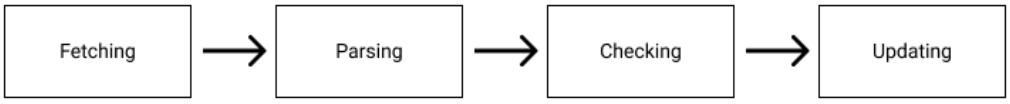

Figure 4.1: A basic dependency update workflow.

- **Fetching**: Collecting all required files for the update step. These might include specification files or lockfiles.
- **Parsing**: Extracting the installed dependencies from the previously fetched files.
- **Checking**: Fetching the latest versions of a or the detected dependencies and checking if a newer version is available.
- **Updating**: Patching the fetched files with the newly detected dependency updates.

It is common to directly apply the file updates after the process is complete, or send them as a patch to a remote repository for further review.

4.1.2 Dart Dependency Update Workflow

The Dart dependency update workflow is similar to the theoretical dependency update workflow. This pattern can even be applied to the manual workflow a user would perform to update a dependency, which would look like the following.

- **Fetching**: The user opens the relevant *pubspec.yaml* file locally.
- **Parsing**: The user chooses the dependencies from the *pubspec.yaml* that are to be updated.
- **Checking**: The user either looks at the official source of the package to find the newest version, or uses the *pub outdated* tool to get a list of updates for all installed versions.
- **Updating**: The user chooses the version he wants to use and replaces the requirement for that dependency in the *pubspec.yaml* file. Finally, the user runs the *pub upgrade* command to update the requirements in the *pubspec.lock* file.

Alternatively the *pub upgrade* command with the *–major-versions* argument can be used to simplify this process and automatically use the latest resolvable version. This however might still not update all available dependencies to the latest version, as the latest version might be incompatible due to version requirement conflicts with other installed dependencies (The Dart project authors, 2021d).

To easily automate the entire dependency update workflow, the described manual steps have to be further broken down into smaller instructions and potential issues that can occur have to be identified.

4.1.3 Dependabot Update Workflow

The actual, automatic dependency update workflow steps implemented by a Dependabot script is similar to the one described in section 4.1.1. It contains the same steps as the basic once, however adds additional steps like creating a pull request and selecting the file source (Dependabot Ltd, 2018).

- **Describing**: Specifying the dependency file source provider, repository, directory and branch.
- **Fetching**: Collecting all required files for the update step. These might include specification files or lockfiles.
- **Parsing**: Extracting the installed dependencies from the previously fetched files.
- **Checking**: Fetching the latest versions of a or the detected dependencies and checking if a newer version is available.
- **Updating**: Patching the fetched files with the newly detected dependency updates.
- **Submitting**: Creating a pull request, merge request or other equivalent with the updated dependency files on the specified source platform.

The Dependabot common modules of the **dependabot-core** project contain the core logic and functionality. They are the modules that represent and simplify the steps a dependency update workflow uses. Additionally, the common module provides helper functions for common use cases like interacting with code written in the language ecosystem modules native language or the file system. Each language module then extend the functionality of the single workflow steps from the common modules to adjust to the programming languages unique way of processing dependency updates (Dependabot Ltd, 2021c). These language ecosystems modules can also be described as plugins for the host Dependabot application, in this case the core module acts as host for the plugin ecosystem.

To support a new language ecosystem, extensions for a specific set of the step

modules, as well as some additional Dependabot specific components, have to be implemented. The Dependabot project provides a basic guideline detailing the implementation steps needed to implement a new language ecosystem module which can be seen in appendix B.

The final workflow can then use all the previously described components to scan the repository and publish pull requests for the outdated dependencies. The usual way to implement a Dependabot update workflow is to link the outputs of the previous components to the inputs of the next component.

4.1.4 Dependabot Workflow Steps

This section describes the existing modules for the steps (seen in section 4.1.3) necessary for a dependency update workflow in the context of the Dependabot. The Interfaces for these classes can be seen in appendix A.

Source

Instances of the *Source* class are data classes and the common representation of the location of source code files for project. This file is used to pass the reference to the required platform and repository to the other steps like the *fetching* step.

The data an instance of this class is created with includes for example the source code platform, the name of the repository on that platform, the git commit and branch that should be fetched and the directory the relevant files for the update process are located relatively.

Currently, the Dependabot supports *GitHub*, *GitLab*, *Bitbucket*, *Azure DevOps Repos* and *AWS CodeCommit* as one of the four source providers.

File fetcher

Using the previously defined *Source* reference, and if necessary the credentials to access a private or restricted remote repository, the required specification files can be fetched from the remote repository.

The file fetcher does not return plain files but instead puts the file content and the metadata into an own instance of the Dependabot *DependencyFile* class. The fetched files and the current HEAD commit can then be retrieved using the *files* and *commit* methods.

File parser

This step extracts the relevant dependencies from the fetched specification files. To easily access the data in further steps, the file parser extracts the identified

dependencies and converts them into the common Dependabot *Dependency* instance representation, which is also used in further steps.

The *Dependency* data class can contain, for example the name and version of a dependency, as well as the current version requirements it specifies. In the *Dependency* class implementation the requirements field is an array that can contain one or more hash data structures with information about the requirement. The specific implementations of the file parser can specify themselves what requirement information is necessary for the current update workflow.

Finally, an array containing every detected dependency can be retrieved from the *parse* instance method.

Update checker

The update checker step determines the most recent available and resolvable versions for a given dependency. It has to be instantiated with a Dependabot *Dependency* instance, most likely fetched from the previous file parser step.

For that dependency it was created with, the update checker provides a wide range of methods. Most important are probably the *up_to_date?* method, a method that checks if a dependency is already at the latest version, the *can_update?* a method that checks if the dependencies is actually allowed to update, and the *updated_dependencies* method which returns a modified Dependabot *Dependency* instance with the updated version constraints.

These methods depend on the individual language module extensions of the update checker to add methods, which fetch the available versions of the current dependency, using the specific APIs of the language's package registry.

The update checker supports three *unlocking* modes "all", "own" and "none", each regarding if the version constraints of one or more dependencies are allowed to be updated.

In a update workflow this update checker has to be initialized once for every *Dependency* that should be updated, afterwards the *up_to_date?* and *can_update?* methods can be called to check if a dependency can actually be updated and finally the *updated_dependencies* method to receive the data for the next step.

File updater

This step updates the version constraints of one or more dependencies in a dependency specification files. A file updater instance has to be created with a list of one or more *Dependency* instances and an array containing, most of the time the previously fetched, *DependencyFile* instances. The file updater uses the original *DependencyFile* instances, creates a duplicate of that instance and modifies

the content of the file directly using the implementation in a language ecosystem module extension.

A workflow can then receive the updated *DependencyFile* instances using the *updated_ dependency_ files* method.

Metadata Finder

Although this class isn't used directly in a dependency update workflow most of the time, it is still used by the Dependabot *PullRequestCreator*.

Based on a source code URL a language ecosystem module extension provides, the metadata finder can gather metadata information about the dependency it was instantiated with like, git commits, releases, changelog and more. This metadata is then accessible using the respective functions of that instance.

Pull Request Creator

This last step creates a pull requests with the updated files. The creator has to be instantiated with one or more *Dependency* and *DependencyFile* instances, the source repository, the base commit, if necessary a set of credentials and optionally a flag if the pull request should have the language label assigned.

Using the *create* method of the instance, the pull request against the base commit and with the updated files, will be created for the repository at the source provider. Additionally, the Dependabot *PullRequestCreator* uses the *MetadataFinder* to fill the pull request description with metadata like the dependencies commit history and changelog.

4.1.5 Docker

The Docker documentation (Docker, 2021) describes Docker as a platform and application for developing and running container based applications. It allows users to define application environments in the form of images. These images can then be used to start containers, a secure and efficient way of executing the predefined images.

The **dependabot-core** project uses Docker to assemble standardized images of the **dependabot-core** application environment. Currently, three different images exist, all three can be built using Docker image specification files, called *Dockerfiles*, located inside the **dependabot-core** repository.

- **Dockerfile**: The main Dockerfile, which installs all dependencies needed to run the Dependabot common module, as well as all the native helper applications needed for other language ecosystem modules. This image

doesn't include the actual Dependabot language modules, and is not able to perform the dependency update process by itself.

- **Dockerfile.ci**: This docker image is based on the core image and includes the **dependabot-core** source. This image is intended to be used inside the official continuous integration (CI) workflows.
- **Dockerfile.development**: This docker image is also based on the core image and includes the **dependabot-core** source as well as some adjustment to improve the debugging experience. This image is intended for debugging and testing new features locally.

4.2 Pub module

Language ecosystem modules are the main method of adding support for new programming languages and their dependency management systems to the Dependabot project (Dependabot Ltd, 2021c). They represent the unique update flow for a specific dependency manager. Each language ecosystem module is a separate Ruby module that depends on the common framework of the Dependabot. Additionally, language modules can be partially written in other programming languages to simplify steps of the update process. The language module extends the core components described in section 4.1.4, these core components handle most of the required IO for the language module.

Each component that is to be extended needs to overwrite a set of public and private class functions. These have to be implemented, otherwise the components can't interact with each other, as the functions in the base classes often depend on these extended functions.

When initializing the module, the classes have to register their own implementations for the package manager they implement. Through this, the core module can then fetch the required classes for the update process of a specific package manager. In a workflow the implementation can then be retrieved by using the *for_package_manager(package_manager)* getter, as seen in listing 4.1.

```
<file_fetcher.rb>
Dependabot::FileFetchers.
    register("pub", Dependabot::Pub::FileFetcher)

<workflow.rb>
# Fetch File Fetcher
fetcher = Dependabot::FileFetchers.
    for_package_manager("pub").new(
        source: source,
        credentials: credentials,
    )
```

Listing 4.1: Registering a FileFetcher for the "pub" package manager and retrieving it later.

This section further describes the architecture decisions made for the Pub language ecosystem module.

Version and Requirement

The version and requirement classes represent the Pub version and requirement rules defined in the **pub_semver** specification (The Dart project authors, 2019b) and the Pub dependencies documentation (The Dart project authors, 2020c).

The two possible architecture approaches that could be pursued to implement the version and requirement representations are:

- **Reimplement in Ruby**: The version and requirement classes would be reimplemented in Ruby according to the specifications. This could easier lead to potential errors and bugs if the specification is not followed closely or implemented in the wrong way.
- **Delegate to native code**: A wrapper around the official **pub_semver** implementation could be used to quickly verify version and requirements. However, working with the native code would add additional complexity, as a lot of data has to be passed between the Ruby class and the native Dart implementation.

For this work the approach to reimplement the specification in Ruby was chosen, to keep as much of the implementation in Ruby as possible. To mitigate the issue of potential programming errors a huge amount of unit tests for the version and requirement classes had to be planned and implemented.

File fetcher

The Pub file fetcher simply extends the Dependabot file fetcher and searches for the required files in the given directory. In this case for the Dart programming language these are the **pubspec.yaml** and optionally the **pubspec.lock**.

File parser

The Pub file parser extends the Dependabot file parser. It receives the fetched **pubspec** files and parses the detected dependencies from the **pubspec.yaml** directly into a set Dependabot dependency instances.

Update checker

The Pub update checker will fetch the latest version information for the dependencies it was created with from the **pub** API. The implemented tool is designed to only operates using the "own" unlocking mode, meaning that a dependency update can only update its own requirements.

Similar as it was the case for the version and requirements classes, the update checker could be implemented partially with native code, especially the *pub outdated* command (The Dart project authors, 2020a). However, to keep it simple, as there is only one network call for each dependency necessary, the update checker was implemented in Ruby.

File updater

The Pub file updater will apply the updated requirement information of one dependency to the **pubspec.yaml** and generate a new **pubspec.lock** file.

This step is partially implemented in Ruby and partially uses a native helper to update all files. The **pubspec.yaml** file will be edited using a regex on the content of the file directly, while to update the **pubspec.lock** file the updater will write the updated **pubspec.yaml** and the old **pubspec.lock** files to a temporary directory where the *pub upgrade* (The Dart project authors, 2021d) command will be executed.

Metadata Finder

The Pub metadata finder also simply extends the Dependabot metadata finder and performs an API request to lookup the associated source code repository of the dependency.

4.3 Pub update script

Finally, a separate Ruby script, similar to the **dependabot-script** (Dependabot Ltd, 2021e), can then use the old and newly implemented modules of the **dependabot-core** project to connect the dependency update workflow steps and perform the actual dependency update for the specified source.

In the case for the Pub module, this will work by depending on the implemented Pub module and the Dependabot common module. The script itself will then fetch the dependency update step implementations for the *pub* package manager. The input and outputs of the different steps are then chained together to fetch the files, parse the dependencies, update them and create pull requests with the updated files.

5 Implementation

This chapter addresses the concrete implementation steps needed to build an application which automatically updates dart package specifications. As described in chapter 4 this implementation will be a new language ecosystem module for the existing Dependabot application.

5.1 Dependabot pub module

In the concrete case of implementing a Dependabot language module, the **dependabot-core** common module **README.md** files provide a basic implementation guideline for writing new language modules, as for example the guide for a file fetcher (Dependabot Ltd, 2019c). The modules implemented in this thesis are mainly implemented in the **Ruby** programming language.

Each module extends the common Dependabot classes or other Ruby classes and overwrites different public and private functions, as seen in Appendix B.

The final implementation will be able to update the most common form of required dependencies in the Dart ecosystem, which are hosted dependencies on the *pub.dev* registry, to the latest available stable version, each single dependency with have an own pull request. Projects with path dependencies and conflicting resolvability won't be supported.

5.1.1 Pub Module Setup

To develop the Pub language ecosystem module, the official **dependabot-core** repository was forked on GitHub. The implementation of the module was done publicly on the *wip/pub* git branch.

In an effort to keep the implementation of the Dependabot Dart language ecosystem module clean and similar to the other ecosystem modules, a separate folder with a new Ruby gem that contains the implementation was created.

Pub Gem

The Ruby Gem for the Pub module is in a separate folder, at the root of the repository next to the other modules and the common modules. The folder of the Gem is named **pub** and consist of the following files and directories:

- **lib/dependabot/**: The directory that contains the Dart language ecosystem module implementation Ruby files.
- **script/ci-test**: A shell script which executes the tests for the implementation during the CI pipeline run.
- **spec/dependabot/**: The directory that contains the Ruby test files for the Dart language ecosystem module implementation.
- **spec/fixtures/**: The directory that contains the data of HTTP calls and sample **pubspec.yaml** files to support the test files.
- **.rubocop.yml**: A configuration file for the **RuboCop** Ruby linter that is used in the **dependabot-core** project.
- **dependabot-pub.gemspec**: The specification file for the Ruby Gem that defines metadata for the Pub language ecosystem module.
- **Gemfile**: The specification file for Bundler that defines the Ruby Gems required for the implementation.

Additionally, a reference to the new Pub Gem had to be created in the specification of the **dependabot-omnibus** Gem.

Pub Entrypoint

As seen in listing 5.1, the Pub module has an entry point named **pub.rb**, this file requires all implemented Pub module implementation files and registers the *pub* package manager for the *PullRequestLabeler* class with the name and color of the label as well as for the *Dependency* class.

```
require "dependabot/pub/file_fetcher"
[...]
require "dependabot/pub/version"

require "dependabot/pull_request_creator/labeler"
Dependabot::PullRequestCreator::Labeler.
  register_label_details(
    "pub", name: "pub", colour: "0175C2"
  )

require "dependabot/dependency"
```

```
Dependabot::Dependency.
  register_production_check("pub", ->(_) { true })
```

Listing 5.1: The entrypoint for the Pub module.

Docker

Because the Pub language module file updater, as seen in section 5.1.6, depends on the **pub** tool included in the Flutter SDK, the main **dependabot-core** Dockerfile had to be adjusted to additionally install the Flutter SDK.

```
# Install Flutter SDK
RUN git clone --branch stable $GIT_PATH /opt/flutter
ENV PATH="$PATH:/opt/flutter/bin"
RUN flutter precache
```

Listing 5.2: Installing Flutter in the dependabot-core **Dockerfile**

This clones the latest stable version of the Flutter SDK, which includes the Dart SDK, from the Flutter GitHub repository into the */opt/flutter* directory (The Flutter authors, 2021b). The flutter binaries will now be put into the list of executables by appending */opt/flutter/bin* to the PATH variable. The final step is precaching Flutter specific development binaries, as these are required to execute the *pub upgrade* command.

5.1.2 Version and Requirement

In contrast to the other classes that have to be implemented, the version and requirement classes extend their respective implementations from the Ruby::Gems module, instead of a class from the Dependabot module.

The version and requirement classes are essential for most of the other dependency update step classes that have to be implemented, as they contain the logic and rules of Dart dependency versions and requirement structure and operations.

The implemented Dart and Pub version and requirement classes follow the rules defined in the *pub_semver* documentation (The Dart project authors, 2019b), which in turn is based on Semantic Versioning 2.0.0-rc.1 (Preston-Werner, 2011-2013).

Version

In the case of the version class, a regex was added to accept the format of pub version numbers like the one seen in listing 5.3, as it differs from the one implemented by default in the *Gem::Version* class.

```
# major.minor.patch-prerelease+buildnumber

1.2.3-test+1
```

Listing 5.3: The format of a Pub version number.

The default sort operator was also overwritten with a custom sorting method which operates according to the *pub_semver* specification. Additionally, as required by the specification a priority order sorting function was added, which gives prerelease versions a lower priority than any other version. This priority order sorting function is used when selecting the latest version for an update. The *pub_semver* specification states that this is because prerelease versions, also referred to as unstable versions, should be opt-in for the user, by using a constraint that specifically allows a prerelease version (The Dart project authors, 2019b).

Requirement

In the case of the requirement class, the available *Gem::Requirement* operators and logic were overwritten with the requirement operators, `< > <= >= ^` and *any*, that are available in the Dart ecosystem (The Dart project authors, 2020c). Additionally, the requirements parser had to be adjusted to allow Dart requirements and parse them correctly.

As seen in listing 5.4, a *satisfied_by?* method was implemented as well, which returns true if a given version satisfies the requirements of that current instance.

```
req = Dependabot::Pub::Requirement.new("^1.0.0")

req.satisfied_by?(Dependabot::Pub::Version.new("1.0.0"))
  # --> True
req.satisfied_by?(Dependabot::Pub::Version.new("1.1.0"))
  # --> True
req.satisfied_by?(Dependabot::Pub::Version.new("2.0.0"))
  # --> False
```

Listing 5.4: The format of a Pub version number.

5.1.3 File fetcher

The file fetcher implementation for the pub module fetches two relevant files from the target repository.

- **pubspec.yaml**: The main specification of all metadata and dependencies
- **pubspec.lock**: Additional detailed machine generated specification of all current direct and transitive dependencies

The file fetcher base class provides multiple functions to easily load the file directly from the source repository.

According to the specification the three methods *.required_files_in?*, *.required_files_message* and *#fetch_files* have to be implemented. The *.required_files_** methods are basic helpers to simplify error handling.

As it can be seen in listing 5.5, the method *#fetch_files* calls the actual file fetching methods for the two files described above. If the **pubspec.yaml** file can not be found the implementation will abort and raise an error. The **pubspec.lock** file on the other hand is optional, and will only be fetched if it is available in the repository.

```
def fetch_files
  files = []
  files << pubspec if pubspec

  if files.empty?
    raise Dependabot::DependencyFileNotFound,
          Pathname
            .new(File.join(directory, "pubspec.yaml"))
            .cleanpath.to_path
  end

  files << pubspec_lock if pubspec_lock
  files
end

def pubspec
  @pubspec ||= fetch_file_if_present("pubspec.yaml")
end

def pubspec_lock
  @pubspec_lock ||= fetch_file_if_present("pubspec.lock")
end
```

Listing 5.5: The fetch_files implementation.

To read the actual files from the specified source code repository, the *FileFetchers::Base* class defines the *fetch_file_if_present(filename)* method.

5.1.4 File parser

The file parser implementation reads the fetched dependency files from the previous file fetcher step and creates a requirements data structure for each detected

dependency.

The method *parse* creates and returns a *DependencySet* instance containing all parsed *Dependency* instances. This method reads every value under the **dependency** and **dev_dependency** keys and extracts the name and requirement string to fill into the custom requirements data structure. This data structure can then be used by further modules to perform the actual update process.

At first the file parser has to parse the **pubspec.yaml** specification file, this is done in the method seen in listing 5.6. As the file is in the **.yaml** format, the Ruby YAML module can be used to convert the file content into a Ruby Hash, a map data structure which allows easy access to the Key-Value structure of the YAML file. Inside the **pubspec.yaml** file the dependencies are located under the *dependencies* and *dev_dependencies* keys, in the context of this method these different keys were called *group*. Once the current group is fetched from the hash, every **pub.dev** hosted dependency gets selected and mapped into another hash with name, requirement and group information. Git and path dependencies are ignored as they are not assigned any version constraints in the **pubspec.yaml** file. Finally, the mapped result will be returned for further processing in the *parse* method.

```
def dependency_strings_from_yaml(yaml, group)
  data = YAML.safe_load(yaml.content, aliases: true)

  return [] if data.nil?

  dependencies = data.fetch(group, {})

  return [] if dependencies.nil?

  dependencies.
    select { |_, value| value.is_a?(String) }.
    map { |key, val|
          { name: key, requires: val, group: group }
        }
rescue Psych::SyntaxError
  raise Dependabot::DependencyFileNotParseable, yaml.name
end
```

Listing 5.6: The method which extracts the dependency declaration strings from the pubspec.yaml file.

As it can be seen in listing 5.7, the main pubspec parsing method then calls the previous mentioned **pubspec.yaml** parsing method with the *dependencies* and *dev_dependencies* as group argument. The results will then be combined and

flattened, as it is irrelevant for the further steps under which key the dependencies were originally defined. Finally, the method will iterate over each identified hosted dependency and create a new Dependabot *Dependency* with the parsed information, that gets added to the dependency set.

```
def pubspec_file_dependencies(file)
  set = DependencySet.new

  dependencies = dependency_strings_from_yaml(file,
    "dependencies")
  dev_dependencies = dependency_strings_from_yaml(file,
    "dev_dependencies")
  all = [dependencies, dev_dependencies].flatten

  all.each do |dependency|
    set << Dependency.new(
      name: dependency[:name],
      version: nil,
      package_manager: "pub",
      requirements: [{
        requirement: dependency[:requires],
        groups: [dependency[:group]],
        source: nil,
        file: "pubspec.yaml"
      }]
    )
  end

  set
rescue Gem::Requirement::BadRequirementError
  raise Dependabot::DependencyFileNotParseable, file.name
end

def exact_version?(req)
  Dependabot::Pub::Requirement.new(req).exact?
end
```

Listing 5.7: The creation process for a Dependabot::Dependency in the file parser.

The Dependency instance then contains the dependency name, if available the exact version, used package manager and one requirement containing the specified requirement string, the dependency group and original filename.

As the **pubspec.lock** file is only relevant for the lockfile updating process in the

file updater step, it will not be parsed in the file parser step, regardless if it is available or not.

5.1.5 Update checker

The main task of the update checker implementation is to fetch and create updated requirements for a dependency. Every parsed dependency is assigned a unique update checker instance.

According to the specification the update checker has to implement multiple methods. These are *latest_version*, *latest_resolvable_version*, *updated_requirements* and *latest_resolvable_version_with_no_unlock*.

Additionally, the methods *latest_version_resolvable_with_full_unlock?* and *updated_dependencies_after_full_unlock* can be implemented, these are optional and not required to build a functional dependency updater for single dependencies, and as such were not implemented in this thesis.

These methods depend on external version information about the specific dependency. To fetch the latest version information of a dependency, the update checker uses the *pub.dartlang.org* API. Although it is only partially officially documented (The Dart project authors, 2020b), unofficial community projects like *pub_api_client* (Farias, 2021) document all known endpoints and features.

Due to simplicity, the essential API endpoint the update checker uses is the /packages/{*name*}.json endpoint. As it can be seen in listing 5.8, the endpoint returns the basic metadata of an on *pub.dev* hosted dependency called *name*. This metadata includes the dependency name, the uploaders and all published versions sorted ascending.

```
GET https://pub.dartlang.org/packages/pub_semver.json

{
  "name": "pub_semver",
  "uploaders": [],
  "versions": [
    "2.0.0-nullsafety.0","1.0.0","1.1.0","1.2.0","1.2.1",
    [...]
    "1.4.1","1.4.2","1.4.3","1.4.4","2.0.0"
  ]
}
```

Listing 5.8: The pub.dev API response for pub_semver package.

As seen in listing 5.9, a helper method named *all_package_versions* was written, which initiates a HTTP GET request to the previously described API endpoint

with the name of the dependency the update checker was initialized with. If the request is successful, the JSON response will be parsed, cached and returned as a list of available Pub::Version instances, otherwise the method will return an empty list.

```
def all_package_versions
  return @all_versions unless @all_versions.nil?

  res = Excon.get(
    "https://pub.dartlang.org/packages/
#{dependency.name}.json",
    idempotent: true,
    **Dependabot::SharedHelpers.excon_defaults
  )

  return [] unless res.status == 200

  @all_versions = JSON.parse(res.body)["versions"].
    map { |v| Dependabot::Pub::Version.new(v) }
end
```

Listing 5.9: Fetching all available versions of the current dependency.

This method is then used by the *latest_version* and *latest_resolvable_version* methods, which can be seen in listing 5.10. The first method returns the latest, and if possible stable, version of that dependency. The second method returns the latest, also if possible stable, version that matches all the given requirements of that version.

```
def latest_version
  all_package_versions.max { |a, b| a.priority b }
end

def latest_resolvable_version
  all_package_versions.
    select { |ver| pub_requirements.
                     all? { |req| req.satisfied_by?(ver) }
           }.
    max { |a, b| a.priority b }
end
```

Listing 5.10: The *latest_version* and *latest_resolvable_version* implementation.

Both versions pick the latest stable version using priority sorting. As described previously in section 5.1.2, priority sorting is similar to comparison sorting, with

the exception that prerelease versions, also referred to as unstable versions, are always avoided if possible, and therefore less prioritized when sorting.

```
requirement = "^1.0.0"
versions = ["1.0.0", "1.1.0", "2.0.0", "2.1.0-test"]

# latest_version => 2.0.0
# latest_resolvable_version => 1.1.0
```

Listing 5.11: An example for both *latest_version* methods.

Finally, as seen in listing 5.12, the update checker implements the *updated_requirements* method, which updates the requirements of the dependency to the latest found version with *caret syntax*. This requirement constraint was chosen to automatically allow a relatively large range of compatibility with other potential future versions.

```
def updated_requirements
  dependency.requirements.map do |req|
    next req if latest_version.nil?

    new_requirement = "^#{latest_version.version}"
    req.merge(requirement: new_requirement)
  end
end
```

Listing 5.12: The *updated_requirements* method implementation.

The method *latest_resolvable_version_with_no_unlock* is similar to the *latest_resolvable_version*, and simply redirects the method call to the latter one.

Additionally, although not required by the implementation guide, the method *up_to_date?*, which normally acts a short-circuit for already updated dependency, was implemented as well. This is because otherwise dependencies which have an *any* version constraint would be updated in a dependency update workflow, which is not intended because dependencies with an *any* version constraint don't care about the installed version (The Dart project authors, 2020c).

5.1.6 File updater

The file fetcher implementation updates the specification files with the new requirements of a single version. Similar to the update checker, there is one instance for each parsed dependency.

The *updated_dependency_files* is the main function that has to be implemented when writing a Dependabot file updater. This method returns an array of updated file instances, using the updated dependency requirements. The function

checks if the requirement have actually changed and detects the old requirement string in the *pubspec.yaml* using a regex. Once the string is detected, a new dependency string will be built using the *Requirement* implementation class.

```
name = "http"
requirement = "1.0.0"
regex = /(#{name}:)\s*((#{requirement})|('#{requirement}')
  |("#{requirement}"))/m.freeze

# This will result in the following regex

regex = /(http:)\s*((1\.0\.0)|('1\.0\.0')
  |("1\.0\.0"))/m.freeze
```

Listing 5.13: The RegEx to detect the old dependency for the file updater.

If a **pubspec.lock** file was detected by the file fetcher and passed to the file updater, the file updater implementation will also update the **pubspec.lock** file if possible. To achieve this, the file updater passes this task to the native pub tool by executing the 'pub upgrade' command (The Dart project authors, 2021d). The pub tool will then generate a matching new **pubspec.lock** for the file fetcher to read.

As it can be seen in listing 5.14, the method will at first open a new temporary directory. Inside this directory, the **pubspec.yaml** specification file that was updated in the previous step will be written. The method will then run the shell command **flutter pub upgrade [dependency]**, this instructs the Flutter binary to generate a lockfile update for the current dependency while trying to avoid the other dependencies as much as possible. Once the command completes, the method reads the content of the newly generated lockfile and returns an updated file instance.

The upgrade process can fail, one reason might be that the version solver fails to find a version of a transitive dependency that is compatible with every other required dependency, or that a git or path dependency can not be found.

This implementation uses the Flutter SDK to run the **pub upgrade** command. If a project wants to use the Flutter framework, specified by the Flutter **sdk** source inside the dependencies array of the **pubspec.yaml** file (The Dart project authors, 2020c), and therefore depend on the Flutter SDK to successfully run the upgrade command. With only the Dart SDK and without the Flutter SDK to installed on the machine, the **pub upgrade** command can not be executed for such a project. Plain Dart projects which don't use the Flutter SDK on the other hand, work just fine with the **pub** tool bundled with the Flutter SDK.

The method uses the **SharedHelpers** class, a module implemented by Depend-

abot to provide different helper functions for language modules which interact with binaries or code for the respective language they intend to support.

```
def updated_lockfile_for_pubspec_dependency(spec,
    dependency)
  SharedHelpers.in_a_temporary_directory
      (spec.directory) do
    File.write("pubspec.yaml", spec.content)
    File.write("pubspec.lock", lockfile.content)

    SharedHelpers.run_shell_command(
      "flutter␣pub␣upgrade␣#{dependency.name}"
    )

    updated_file(
      file: lockfile,
      content: File.read("pubspec.lock")
    )
  end
end
```

Listing 5.14: The method to update the pubspec.lock file.

The resulting updated **pubspec.yaml** and **pubspec.lock** files can then be used to create a new update pull request.

5.1.7 Metadata Finder

The primary function of the Metadata Finder implementation is to fetch the URL for additional metadata of a source repository of a current dependency, if available. In contrast to the previous modules, the metadata finder is not used by the actual Dependabot Runner script directly. It is however used by the pull request creator to fill pull requests with rich data. This data includes for example the commit history and release information (Dependabot Ltd, 2019f).

According to the pubspec file documentation (The Dart project authors, 2021f), the *repository* field contains a URL to the package's source code repository and the *homepage* field a URL to the package's homepage or source code repository. Although most Pub packages on *pub.dev* are open source, due to the fact that the source code is available publicly on *pub.dev* anyway, a definite reference to a source code repository in packages is not required. This limitation makes it hard to reliably discover the source code repository. To mitigate this issue, the implemented Metadata Finder will first look up the *repository* field for a valid URL and otherwise fall back to the value in the *homepage* field.

```
def look_up_source
  pubspec = package_info["latest"]["pubspec"]
  repo = pubspec["homepage"]
           unless pubspec["homepage"].nil?
  repo = pubspec["repository"]
           unless pubspec["repository"].nil?
  Source.from_url(repo)
end
```

Listing 5.15: The implementation of the metadata look_up_source function.

Package Metadata Statistics

To validate the hypothesis that a valid source code reference can be found in the *homepage* field, the *pubspec.yaml* metadata of all 21111 valid published *pub.dev* packages (as of 6.5.2020) was fetched from the *pub.dev* API and scanned using a custom script. As seen in table 5.1, only a small fraction of 1.54% of packages contained neither the homepage or repository field, and are therefore unavailable to get rich metadata support for pull requests updating them.

Category	Count	Percentage
Total	21111	100.00%
Homepage	19975	94.62%
Repository	3031	14.36%
Both	2220	10.52%
None	325	1.54%

Table 5.1: Amount of packages on pub.dev that include the homepage or repository fields in their pubspec.yaml file.

After running the analysis, it turned out that the assumption that the *homepage* field most of the time contains references to source code repositories is indeed true. As seen in table 5.2, the analysis of the 19975 packages containing a *homepage* field, shows that over 80% of these packages actually lead to the popular source code platform *www.github.com* or *github.com*, with similar platforms like *gitlab.com* and *bitbucket.org* following. Most of the remaining homepages lead either to the actual project website or other source code hosting platforms.

Of the 16426 URLs that lead to *GitHub*, a total of 14550 are in the format of https://github.com/{owner}/{repository}, which is a reference to a source code repository hosted on *GitHub*.

Hostname	Count	Percentage
github.com (www.github.com)	16426	82.23%
gitlab.com	253	1.27%
bitbucket.org	112	0.56%
Other	3184	15.94%

Table 5.2: Most common homepage hostnames of packages on pub.dev that include the homepage field in their pubspec.yaml file.

5.2 Testing the Dependabot pub module implementation

Testing the implementation and therefore insuring the proper functionality of the implemented work, is an important part of software development. The **dependabot-core** modules use unit tests to ensure their implementation actually work the way they are intended. For this project **RSpec** (The RSpec Team, n.d.) testing framework is used to implement and run the unit test. Additionally, the source code files are being checked by the **RuboCop** linter (Batsov, 2012-2020).

5.2.1 Linting

Linters like the **RuboCop** tool (Batsov, 2012-2020) used in the **dependabot-core** project, help developers and maintainers to ensure a consistent code style in their projects. This includes source code style checks like line length and preferred quote type for string literals, as well as basic checks for the program logic.

The files that were implemented in this thesis for the **dependabot-core** Pub module were all scanned using **RuboCop**, the tool finished without errors or warnings.

5.2.2 Unit tests

Every implementation file of the Pub module has a corresponding unit test file was created and implemented. The tests strive to reach the highest possible line coverage, meaning source code lines that were passed during the test runs, and highest possible branch coverage, meaning conditional branches in the source code that were passed during the test runs. As seen in table 5.3, nearly 100% of lines and branches are covered by the implemented unit test.

Filename	Lines	Cov.	%	Branches	Cov.	%
pub.rb	12	11	91.67%	0	0	100%
pub/file_fetcher.rb	22	22	100%	6	6	100%
pub/file_parser.rb	38	38	100%	8	6	75%
pub/file_updater.rb	50	50	100%	10	8	80%
pub/metadata_finder.rb	16	16	100%	6	6	100%
pub/requirement.rb	51	51	100%	22	22	100%
pub/update_checker.rb	36	35	97.22%	8	8	100%
pub/version.rb	82	82	100%	48	47	97.92%
Total	307	305	99.35%	108	103	95.37%

Table 5.3: The line and branch coverage of the individual Pub module implementation files.

To support the unit tests multiple fixtures were defined that contain sample *pubspec.yaml* files or resemble basic network requests to several APIs like the *GitHub* API or the *pub.dev* API.

For nearly every function a corresponding unit test was written to confirm that the targeted behavior is correctly implemented.

As an example, the tests for the version and requirement implementations from section 5.1.2 assert that the rules of the **pub_semver** (The Dart project authors, 2019b) specification.

Tests for the file fetcher implemented in section 5.1.3 control the correct execution of network requests for a source and assert that the available required files in scenarios with different directory contents are returned.

The tests for the file updater, described in section 5.1.6, assert that the implementation actually properly updates the **pubspec.yaml** and **pubspec.lock** specification files for a set of different requirements and calls the *flutter pub upgrade* command with the correct arguments.

5.2.3 Test automation

The original **dependabot-core** repository uses GitHub Actions Workflows (GitHub, 2021c) as an automated CI environment (Dependabot Ltd, 2021f). The workflow *ci.yml* automatically runs a set of quality assurance tests on the branch. Currently, these are configured to run for every branch that matches "main", "actions/**" and "wip/**" as well as for every branch that is currently part of an open pull request.

The Dependabot Pub module was developed publicly as a fork of the main

dependabot-core repository. To enable the CI functionality the branch "wip/pub" was chosen for the development of the pub module.

The workflow uses a matrix strategy to run the test suite simultaneously for every implemented package manager module. It then builts the Dependabot Core and Dependabot CI docker images. Once the CI image is built, the workflow will spin up the container image and run the *ci-test* script inside the script directory of the current package manager module. This *ci-test* script, as it can be seen in listing 5.16, will install the current dependencies of the ruby implementation using the *bundle install* command. Once the dependencies are fully installed, the **RuboCop** linting tool will lint the Pub module codebase for any style errors or bad patterns. The **RuboCop** linting rules for the module are located in the *.rubocop.yml* file, which in turn inherits from the root *.rubocop.yml* intended for all Dependabot modules. Finally, the unit tests in the *spec* directory of the current module will be run using the **rspec** tool.

```
#!/bin/sh

bundle install
bundle exec rubocop .
bundle exec rspec spec
```

Listing 5.16: The implemented CI-test script file for the Pub module.

If no linting errors are found and the unit tests run successfully the current matrix will be marked as completed. To avoid accidentally breaking other package manager modules, the entire workflow run will only be marked as completed if all matrix runs complete successfully.

5.3 Dependabot Pub Runner

To test and actually use the implementation in real projects a Dependabot pub runner was implemented. The runner uses the public API of the different classes needed to complete the update process.

In this project the runner is a basic Ruby script which chains the separate classes together. To allow easy access for external projects the runner was published as a GitHub Dockerfile Action.

Every time the action is executed, a docker container based on a simplified dependabot-core image (see section 5.1.1), and the source of the Dependabot pub implementation will be built. Afterwards the container entry point will run the basic Ruby update script implementation which will scan the specified pubspec.yaml file for outdated dependencies and create a new pull request for each outdated dependency in the defined repository.

The action can be further configured to adjust for a specific project configuration, the configuration options can be seen in table 5.4.

Input	Description	Default
token	Personnal access token used to modify the repository.	github.token
project	The repository one wants to create pull requests for.	github.repository
path	The path of the pubspec.yaml.	"/"

Table 5.4: Available configuration options for the GitHub Action Dependabot pub runner.

If a Dart project hosted on GitHub should use this implemented Dependabot Pub Runner GitHub Action in a workflow, a YAML file has to be created inside the *.github/actions/* directory. An example configuration can be seen in listing 5.17, which represents a scheduled workflow (GitHub, 2021b) which runs every day at 6AM UTC, on an Ubuntu Linux host runner and executes the update script for the pubspec specification files in the *frontend* directory. The created pull requests can then be merged into the repository. It is still recommended using a CI pipeline in the project, which automatically test the compatibility of the current code with the new version to prevent accidentally merging an incompatible dependency version.

```
name: Dependabot Pub

on:
  schedule:
    - cron: '0 6 * * *'

jobs:
  pub:
    name: Dependabot Pub
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Update
        uses: JohannSchramm/dependabot-pub-runner@main
        with:
          path: /frontend
```

Listing 5.17: A GitHub Workflow configuration file for the .

6 Evaluation

6.1 Experiment

To test the pub language ecosystem module developed in the previous chapters in a real world scenario, an experiment to validate the implementation and to gather stats about the Dart/Pub ecosystem was set up.

The test sample chosen for this test are the 1020, publicly available GitHub repositories, which list **Dart** as their primary programming language and have the highest amount of GitHub stars for that category. The key metrics that were recorded and later analyzed are:

- **Stability** of the implementation on different projects
- **Quantity** of outdated dependencies that were identified

Building the test sample

The GitHub Search API (GitHub, 2021e) can be queried for the relevant test samples using the GET request seen in listing 6.1. Each response will return 30 results, to gather the full amount of results required for the test sample, the request has to be performed 34 times, every time with a modified page parameter, as the API will only return the first 1020 results for this type of query. The language query parameter set **Dart** as the required primary programming language for the search results. This request automatically sorts by the amount of "GitHub Stars" descending.

```
https://api.github.com/search/repositories?q=language:Dart
    &page=1
```

Once all requests are done, the results can be merged into a single array to simplify further processing.

To collect the actual raw test data, the Dependabot Pub Runner implemented in section 5.3 was slightly modified to run the dependency update process in

sequence on every of the previously collected projects. The modification to the runner were the replacement of the *File Fetcher* and the removal of the *Pull Request Creator*.

The file fetcher had to be replaced, as it normally uses the API of the source repository hosting platform to fetch directories and file content, which has in the case of the GitHub API, a rate limit of just 60 requests per hour for unauthenticated requests and 5000 per hour for authenticated requests (GitHub, 2021d). As the file fetcher implemented in section 5.1.3 uses at least one to three API requests, one for the directory listing and another one or two for the *pubspec.lock* and *pubspec.yaml* files, per directory with a *pubspec.yaml* file, the application can quickly reach the unauthenticated rate limit when performing bulk requests for more than 1000 repositories containing sometimes more than 50 *pubspec.yaml* files. As an alternative the script performs a *git clone* of the source code repository into a temporary directory, from which the replacement *File Fetcher* will read the required files.

Instead of creating pull requests, the script will record if the update process was successful and if so, save the detected and updated dependencies as well as the exact changes made to the **pubspec.yaml** and **pubspec.lock** to a new file. Additional the script will record metadata about the current HEAD commit, and the date the analysis was performed.

Analyzing the test data

To analyze the test data two scripts were implemented using the JavaScript programming language. The first one merges the distributed test results into a single **results.json** output file. The second script then uses this combined **results.json** file to extract statistics about the quality and performance of the implemented Dart dependency updater.

Results

As seen in table 6.1, during the evaluation a total of 1020 project were downloaded and analyzed, which worked for all except one project which could not be completely analyzed due to private git dependencies. In these successfully fetched projects, a total of 3702 **pubspec.yaml** files were found and analyzed.

Category	Count	Percentage
Total projects analyzed	**1020**	100%
... successfully	1019	> 99.9%
... unsuccessfully	1	< 0.1%
Total pubspec.yaml files analyzed	**3702**	100%

Table 6.1: A summary of the projects analyzed for the evaluation.

As seen in table 6.2, in these 3702 **pubspec.yaml** files a collected total of 17670 project dependencies were found. Of these dependencies 1543 are unique, in the sense of that they can occur in multiple projects, but every one of the 17670 dependency is part of the unique dependencies set.

The implementation tried to update every of the 17670 listed dependencies to the latest versions. The update was successful for a large majority of 15368 of the listed dependencies, with 9445 of them being updated to the latest version by the tool, as 5923 of them are already up to date. The update failed for 2302 or around 13.03% of all listed dependencies.

Category	Count	Percentage
Total dependencies	**17670**	100%
... update failed	2302	13.03%
... update successful	15368	86.97%
..... update done	9445	61.46%
..... already up to date	5923	38.54%
Total unique dependencies	1543	100%

Table 6.2: A summary of the dependencies analyzed for the evaluation.

Possible reasons for the observed dependency update failures could be:

- **Not resolvable due to version constraints**: The **pubspec.lock** file could not be updated for this dependency, as updating the version of this dependency would create a conflict with a transitive requirement of another installed dependency. To fix this issue the conflicting dependencies would have to be updated at the same time. This however is not a guarantee to solve the conflict as newer versions of the updated dependencies might still be incompatible. This is currently not supported by the implementation.
- **Path dependencies**: The **pubspec.lock** file could not be updated for this dependency, as the **pubspec.yaml** contains references to path dependencies which could not be located. This is currently not supported by the implementation.

- **Invalid SDK constraints**: The **pubspec.lock** file could not be updated for this dependency, as the **pubspec.yaml** contains invalid SDK constraints. This in the **pubspec.yaml** has to be fixed manually by the maintainer of the project.
- **Unable to fetch external git dependency**: The **pubspec.lock** file could not be updated for this dependency, as the **pubspec.yaml** contains references to git dependencies that could not be downloaded.
- **Other errors**: The two others errors are one instance where a version could not be parsed correctly and another instance where the *pub* tool exited with the exit code *-9*.

Category	Count	Percentage
Total Exceptions	**2302**	100%
... Not resolvable due to version constraints	1298	56.38%
... Path dependencies	554	24.07%
... Invalid SDK constraints	429	18.64%
... Unable to fetch external git dependency	19	0.82%
... Other errors	2	0.09%

Table 6.3: A summary of the errors and exceptions that occurred during the evaluation test run.

The implementation is stable and able to update the dependencies of most projects. The evaluation showed that a huge amount of dependencies of the top 1020 most started Dart GitHub repos don't use the latest available version and are able to be updated.

6.2 Requirements

This section reviews the functional and non-functional requirements defined in chapter 2.

The experiment seen in section 6.1 showed that the implemented Dart dependency application can update the majority of the most popular Dart and Flutter projects.

6.2.1 Functional Requirements

Fetch dependency specifications

> The available dependency specification files should be fetched from the repository. These files define each required dependency as well as its version, or possible version range.

The file fetcher implemented in section 5.1.3 fetches both files that are relevant to complete the Dart dependency update process.

Unify dependency specifications

> The fetched dependency metadata should be parsed into a common representation, which simplifies further access and work with the data.

The file parser implemented in section 5.1.4 creates a common representation in the form of a collection of Dependabot *Dependency* instances from the dependency specification file. These instances were then used to complete the further steps.

Update dependency specifications

> The identified dependencies should each be checked for updates or security patches, depending on given version constraints. If updates exist, the dependency declarations in the original specification file will be patched with the new dependency versions.

The update checker and file parser implemented in section 5.1.5 and section 5.1.6 respectively, search and replace the old version in the specification file with the latest stable one from the *pub.dev* hosting site.

Provide upgrade pull-requests

> A version upgrade for each outdated dependency should be provided as a separate pull request. The user can then easily accept or reject these pull requests.

The Pub Dependabot Runner implemented in section 5.3 uses the updated files and the already existing Dependabot pull request creator to create pull request for the source code repository.

Repository-based Configurable

> The system should be configurable using a configuration file located in the target repository.

The Pub Dependabot Runner implemented in section 5.3 has to be configured as a GitHub Action workflow using a file in the target repository. This workflow can then pass additional configuration parameters to the runner.

6.2.2 Non-Functional Requirements

Multiple Source Platform Compatible

> The implemented system should be compatible with different source code platforms like GitHub or GitLab.

As seen in section 4.1.4, due to the fact that the implemented module is an extension of Dependabot, it automatically is compatible with all source code platform sources supported by Dependabot. This includes fetching the files and sending pull request or the other platforms equivalent.

Working Standalone

> The implemented module should be usable standalone without any other already existing language modules.

The Pub Dependabot Runner implemented in section 5.3 is independent of all the other language modules. It only installs itself and the Dependabot common module as the only dependency.

Provide Upstream Pull-Request

> A pull request for the official Dependabot repository which adds the implemented module should be provided.

During the time this thesis was written, the **dependabot-core** repository updated the contribution guidelines to mention that currently additional language ecosystem modules cannot be accepted, due to lack of internal capacity and in-house expertise. Although in the meantime, other community members started to work on a similar Dart language module implementation which was proposed as a pull request to the GitHub repository of **dependabot-core**.

As an alternative the contribution guidelines recommended maintaining new language ecosystem modules as a separate fork and provide a Dependabot runner, like the one implemented in section 5.3. As this is the only requirement with external dependencies, only the alternative solution was fulfilled. Both the fork and the runner for the pub language ecosystem are publicly available on GitHub.

6.2.3 Results

As presented in this section, all set requirements are fulfilled. The implemented Dart dependency update tool is working and although not the way originally intended, available for the public to use in their projects.

7 Conclusion

Keeping dependencies updated is a necessary but time-consuming task. However, projects like the Dependabot have allowed developers to easily track new version of dependencies and keep them updated.

This thesis explained the architecture of the Dependabot Core module ecosystem and showed the steps necessary for implementing automatic dependency updates as a new extension to the project in form of a language ecosystem module for the Dart programming language.

For a total of over 17000 installed direct dependencies in 1019 of the most starred Dart GitHub projects, the implementation was able to successfully analyze over 15000 of them. Of these successful runs, about 60% were updated to a newer version, while only about 40% were already at the latest version.

Around 15% of the dependency runs failed due to exceptions during the execution. This was most of the time due to invalid specification files and missing support for further dependency source formats. In the future the automatic dependency update application can be further improved by supporting multiple dependency updates over different dependencies in one pull request, which would allow dependencies to be updated that are currently ignored due to unresolvable version constraints. And path dependencies sources by building temporary path structure for these dependencies.

Additionally, the hosted packages metadata that was analyzed revealed that a huge amount of projects, while according to the official documentation allowed, still use the homepage metadata field to refer to the project source code repository instead of the made for that use available repository metadata field. In the future, the tool could be updated to send recommendations for the project specification files as well. Or specifically in the case of the homepage metadata field, the problem could be fixed directly at the source, by updating the Pub package publishing tool to send a hint to the uploaders if they use the wrong metadata field.

The evaluation showed that a huge amount of dependencies were able to be

updated automatically, which helps developers to keep their projects updated and secure. The application implemented and described in this work, is publicly available and easy to set up for everyone to use in their projects. In the future a Pub module could even be implemented into the Dependabot by default.

Appendices

A Dependabot Steps Modules Public API

Relevant public API parts of the different step modules implemented in the **dependabot-core** common modules, needed for the Pub update script implementation. This API documentation is based on the source code of the different classes and the documentation of the **dependabot-core** common module (Dependabot Ltd, 2021a).

Source

Type	Name	Description
Input	provider	The name of the source code provider for this source. Possible options include for example "github", "gitlab", "bitbucket" and "azure".
Input	repo	The name of the target repository.
Input	directory	The directory the relevant files for the workflow are located in.
Input	branch	The target branch of the git repository.
Input	commit	The target commit of the git repository.

File fetcher

Type	Name	Description
Input	source	A Dependabot::Source reference to the project source code repository.
Input	credentials	An array of credentials to access the source code repositories and platforms.
Method	files	An array of Dependabot::File with the relevant files for the current package manager.
Method	commit	The HEAD commit.

File parser

Type	Name	Description
Input	dependency_files	An array of Dependabot::File with the relevant files for the current package manager.
Input	source	A Dependabot::Source reference to the project source code repository.
Input	credentials	An array of credentials to access the source code repositories and platforms.
Method	parse	An array containing the parsed Dependabot::Dependency objects.

Update checker

Type	Name	Description
Input	dependencies	The Dependabot::Dependency instances the Update Checker Instance should be created for.
Input	dependency_files	An array of Dependabot::File instances with the relevant files for the current project.
Input	credentials	An array of credentials to access the source code repositories and platforms.
Method	up_to_date?	Returns if the current dependencies are up to date.
Method	can_update?(level)	Returns if the current dependencies can be updated when unlocking different constraint levels.
Method	updated_dependencies(level)	Returns an array of updated Dependabot::Dependency instances for different unlocking constraint levels.

The constraint level are also referred to as **requirements_to_update**. Possible constraint levels are **:all**, **:own** and **:none**. In the implemented runner only **:own** is used.

File updater

Type	Name	Description
Input	dependencies	An array of Dependabot::Dependency the File Updater Instance should be created for.
Input	dependency_files	An array of Dependabot::File with the relevant files for the current package manager.
Input	credentials	An array of credentials to access the source code repositories and platforms.
Method	updated_dependency_files	An array of modified Dependabot::File instances containing the data of the provided dependencies.

Pull Request Creator

Type	Name	Description
Input	source	A Dependabot::Source reference to the project source code repository.
Input	base_commit	The parent commit for the pull request.
Input	dependencies	An array of updated Dependabot::Dependency instances.
Input	files	An array of modified Dependabot::File instances containing the data of the provided dependencies.
Input	credentials	An array of credentials to access the source code repositories and platforms.
Input	label_language	If the pull request should have a language label.
Method	create	Creates the pull request on the source platform.

B Dependabot Language Ecosystem Module Implementation API

This describes the functions that have to be implemented when writing a language ecosystem module. This API documentation is based on the documentation of the **dependabot-core** common module (Dependabot Ltd, 2021a).

File fetcher

.required_files_in?	Checks if the required filenames for the update process are in an array.
.required_files_message	Static error message if required files are not present.
#fetch_files	Fetch and select the required files from the source.

(Dependabot Ltd, 2019c)

File parser

#parse	Return an array of parsed dependency instances.
#check_required_files	Raise an error unless all required files are present.

(Dependabot Ltd, 2019d)

Update checker

#latest_version	The latest available version for the dependency.
#latest_resolvable_version	The latest version that allows the full dependency set to resolve.
#latest_resolvable_version_with_no_unlock	The latest version within the current constraints that allows the full dependency set to resolve.
#updated_requirements	Updated set of requirements that should replace the current requirement in the manifest file.
#latest_version_resolvable_with_full_unlock?	Only needed for multi dependency updates.
#updated_dependencies_after_full_unlock	Only needed for multi dependency updates.

(Dependabot Ltd, 2019g)

File updater

.updated_files_regex	An array of regexes matching the filenames the updater will update.
#updated_dependency_files	An array of updated DependencyFile instances.

(Dependabot Ltd, 2019e)

Metadata Finder

#look_up_source	Return the source instance for the current dependency.

(Dependabot Ltd, 2019f)

References

Batsov, B. (2012-2020). RuboCop. website. Retrieved June 20, 2021, from https://rubocop.org/

Dependabot Ltd. (2018). update-script.rb. source code. Retrieved June 21, 2021, from https://github.com/dependabot/dependabot-script/blob/main/update-script.rb

Dependabot Ltd. (2019a). Dependabot is joining GitHub. homepage. Retrieved June 20, 2021, from https://dependabot.com/blog/hello-github/

Dependabot Ltd. (2019b). Dependabot Ltd. homepage. Retrieved June 19, 2021, from https://dependabot.com/

Dependabot Ltd. (2019c). File fetchers. documentation. Retrieved June 21, 2021, from https://github.com/dependabot/dependabot-core/blob/main/common/lib/dependabot/file_fetchers/README.md

Dependabot Ltd. (2019d). File parsers. documentation. Retrieved June 21, 2021, from https://github.com/dependabot/dependabot-core/blob/main/common/lib/dependabot/file_parsers/README.md

Dependabot Ltd. (2019e). File updaters. documentation. Retrieved June 21, 2021, from https://github.com/dependabot/dependabot-core/blob/main/common/lib/dependabot/file_updaters/README.md

Dependabot Ltd. (2019f). Metadata finders. documentation. Retrieved June 21, 2021, from https://github.com/dependabot/dependabot-core/blob/main/common/lib/dependabot/metadata_finders/README.md

Dependabot Ltd. (2019g). Update checkers. documentation. Retrieved June 21, 2021, from https://github.com/dependabot/dependabot-core/blob/main/common/lib/dependabot/update_checkers/README.md

Dependabot Ltd. (2021a). Dependabot Common. repository. Retrieved June 21, 2021, from https://github.com/dependabot/dependabot-core/tree/main/common/lib/dependabot

Dependabot Ltd. (2021b). Dependabot Core. repository. Retrieved June 19, 2021, from https://github.com/dependabot/dependabot-core

Dependabot Ltd. (2021c). Dependabot Core. documentation. Retrieved June 21, 2021, from https://github.com/dependabot/dependabot-core#architecture

Dependabot Ltd. (2021d). Dependabot Ltd. rubygems profile. Retrieved June 20, 2021, from https://rubygems.org/profiles/dependabot

Dependabot Ltd. (2021e). Dependabot Update Script. repository. Retrieved June 20, 2021, from https://github.com/dependabot/dependabot-script

Dependabot Ltd. (2021f). dependabot-core workflows. repository content. Retrieved June 20, 2021, from https://github.com/dependabot/dependabot-core/tree/main/.github/workflows

Dependabot Ltd. (2021g). dependabot-omnibus. Retrieved June 20, 2021, from https://rubygems.org/gems/dependabot-omnibus

Dependabot Ltd. (2021h). Feedback and contributions to Dependabot. project documentation. Retrieved June 20, 2021, from https://github.com/dependabot/dependabot-core/blob/main/CONTRIBUTING.md

Docker. (2021). Docker Overview. project documentation. Docker. Retrieved June 17, 2021, from https://docs.docker.com/get-started/overview/

Farias, L. (2021). pub_api_client. project readme. Retrieved June 15, 2021, from https://github.com/leoafarias/pub_api_client/blob/main/README.md

GitHub. (2021a). Configuration options for dependency updates. documentation. Retrieved June 20, 2021, from https://docs.github.com/en/code-security/supply-chain-security/keeping-your-dependencies-updated-automatically/configuration-options-for-dependency-updates

GitHub. (2021b). Events that trigger workflows. documentation. Retrieved June 20, 2021, from https://docs.github.com/en/actions/reference/events-that-trigger-workflows

GitHub. (2021c). Introduction to GitHub Actions. documentation. Retrieved June 20, 2021, from https://docs.github.com/en/actions/learn-github-actions/introduction-to-github-actions

GitHub. (2021d). Resources in the REST API. documentation. Retrieved June 20, 2021, from https://docs.github.com/en/rest/overview/resources-in-the-rest-api

GitHub. (2021e). Search. documentation. Retrieved June 20, 2021, from https://docs.github.com/en/rest/reference/search

GitHub Search API. (2021). Public GitHub Repositories using the Dart language. api response. Retrieved June 19, 2021, from https://api.github.com/search/repositories?q=language:Dart

Google Developers. (2021). Announcing Flutter 2. blog. Retrieved June 20, 2021, from https://developers.googleblog.com/2021/03/announcing-flutter-2.html

License Zero. (2018). The Prosperity Public License 2.0.0. homepage. Retrieved June 20, 2021, from https://prosperitylicense.com/versions/2.0.0.html

Mullans, A. (2020). Keep all your packages up to date with dependabot. GitHub Blog. Retrieved June 19, 2021, from https://github.blog/2020-06-01-keep-all-your-packages-up-to-date-with-dependabot/

npm, Inc. (2021a). Npm. homepage. Retrieved June 19, 2021, from https://www.npmjs.com/

npm, Inc. (2021b). Using npm packages in your projects. documentation. Retrieved June 21, 2021, from https://docs.npmjs.com/using-npm-packages-in-your-projects

Preston-Werner, T. (2011-2013). Semantic Versioning 2.0.0-rc.1. homepage. Retrieved June 15, 2021, from https://semver.org/spec/v2.0.0-rc.1.html

PyPA. (2020). pip. homepage. Retrieved June 20, 2021, from https://pip.pypa.io/en/stable/

Ruby Community. (2021a). Ruby. homepage. Retrieved June 20, 2021, from https://www.ruby-lang.org/en/

Ruby Community. (2021b). RubyGems. homepage. Retrieved June 20, 2021, from https://rubygems.org/?locale=en

Silva, Gonçalo and Ruby Community. (2021). STRUCTURE OF A GEM. documentation. Retrieved June 20, 2021, from https://guides.rubygems.org/what-is-a-gem/

TC52. (2015). ECMA-408. ecma specification. Retrieved June 19, 2021, from https://www.ecma-international.org/publications-and-standards/standards/ecma-408/

The Dart project authors. (2019a). Dart FAQ. homepage. Retrieved June 19, 2021, from https://dart.dev/faq

The Dart project authors. (2019b). pub_semver README.md. project readme. Retrieved June 15, 2021, from https://github.com/dart-lang/pub_semver/blob/master/README.md

The Dart project authors. (2020a). dart pub outdated. documentation. Retrieved June 21, 2021, from https://dart.dev/tools/pub/cmd/pub-outdated

The Dart project authors. (2020b). Hosted pub repository specification version 2. project documentation. Retrieved June 16, 2021, from https://github.com/dart-lang/pub/blob/master/doc/repository-spec-v2.md

The Dart project authors. (2020c). Package dependencies. dart.dev documentation. Retrieved June 20, 2021, from https://dart.dev/tools/pub/dependencies

The Dart project authors. (2020d). Package versioning. dart.dev documentation. Retrieved June 20, 2021, from https://dart.dev/tools/pub/versioning

The Dart project authors. (2021a). Dart overview. documentation. Retrieved June 20, 2021, from https://dart.dev/overview

The Dart project authors. (2021b). Dart programming language. homepage. Google. Retrieved June 19, 2021, from https://dart.dev/

The Dart project authors. (2021c). Dart programming language specification 5th edition. homepage. Retrieved June 19, 2021, from https://dart.dev/guides/language/specifications/DartLangSpec-v2.10.pdf

The Dart project authors. (2021d). dart pub upgrade. documentation. Retrieved June 21, 2021, from https://dart.dev/tools/pub/cmd/pub-upgrade

The Dart project authors. (2021e). pub.dev. homepage. Google. Retrieved June 19, 2021, from https://pub.dev/

The Dart project authors. (2021f). The pubspec file. dart.dev documentation. Retrieved June 15, 2021, from https://dart.dev/tools/pub/pubspec#supported-fields

The Flutter authors. (2021a). Flutter. homepage. Google. Retrieved June 19, 2021, from https://flutter.dev/

The Flutter authors. (2021b). Flutter. repository. Retrieved June 21, 2021, from https://github.com/flutter/flutter

The RSpec Team. (n.d.). RSpec. homepage. Retrieved June 19, 2021, from https://rspec.info/

Thomsen, M. (2021). Announcing Dart 2.12. medium article. Retrieved June 20, 2021, from https://medium.com/dartlang/announcing-dart-2-12-499a6e689c87