CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Efficient and secure document rendering from multiple similar untrusted sources
Student:	Mikuláš Poul
Supervisor:	Ing. Miroslav Hrončok
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2018/19

Instructions

Currently, the content of naucse.python.cz, a website with learning materials, is rendered into a static website from a single git repository. This is problematic, because only trusted users can add or modify the content. Expand the system to enable content rendering from untrusted forked repositories.
Implement a tool that renders static HTML fragments from multiple git repositories containing Python code and source materials (such as Markdown). Assume that the code and most of the source materials are typically identical across repositories. The tool must cache rendered fragments across repositories to avoid duplicate work. Assume that the code in the repositories is untrusted. The tool must provide proper isolation to prevent exploits by malicious users.
Integrate the tool into naucse.python.cz according to upstream issue n. 175 [1]. The code should be well structured, written in Python, commented in English, well tested and released under the terms of the MIT license.

References

1. VIKTORIN, Petr. *Render courses from forked repositories: Issue #175* [online]. 2017 [cit. 2017-07-20]. Dostupný z WWW: < https://github.com/pyvec/naucse.python.cz/issues/175 >.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague September 7, 2017

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Efficient and secure document rendering from multiple similar untrusted sources

Mikuláš Poul

Department of Software Engineering

Supervisor: Ing. Miroslav Hrončok

15th of May 2018

Acknowledgements

First and foremost I would like to thank my supervisor Miro. Without his advice and constructive criticism, this thesis would not have come to a successful end.

To Petr, thank you for the thoughtful review of my code on both Arca and Naucse. To Mikey, thank you for the advice on the structure of Arca's documentation.

My great gratitude also goes to Red Hat, which kindly offered me an internship to work on this project. To my colleagues at SparkTECH, thank you for being patient with me while I worked on this thesis.

And finally, I would like to thank my friends Charlie, Naty, Hilda and Hannah for helping me to find grammatical and stylistic errors in the text.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 15th of May 2018 .....

Czech Technical University in Prague

Faculty of Information Technology

Citation of this thesis

POUL, Mikuláš. *Efficient and secure document rendering from multiple similar untrusted sources.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020. Available also from: ⟨`https://bachelor-thesis.mikulaspoul.cz/BP_Mikulas_Poul_2018.pdf`⟩.

Abstract

Previously, only trusted maintainers could modify the content of *Nauč se Python!*, a project for educational content deployed to the web. The goal of this thesis was to allow for some of the content to be rendered from forks of the base Git repository, but safely and efficiently. That was accomplished by creating a tool which can run Python code in various levels of isolation and cache the results. This tool was then integrated into the *Nauč se Python!* project, building parts of the website in an isolated environment, either in Docker containers or in virtual machines managed by Vagrant, and sharing appropriate content fragments across repositories.

Keywords sandboxing, process isolation, content caching, Python, Git, Docker, Vagrant

Abstrakt

Dříve mohli upravovat obsah projektu pro vzdělávací materiály *Nauč se Python!* jen důvěryhodní správci. Cílem této práce bylo umožnit vykreslovat některý obsah z forků hlavního gitového repozitáře, nicméně ale bezpečně a efektivně. Toho bylo dosáhnuto implementací nástroje, který dokáže spustit kód v Pythonu pod nastavitelnou úrovní izolace a uchovávat výsledky dluhodobě v mezipaměti. Tento nástroj byl následně integrován do projektu *Nauč se Python!*, kde vykresluje části obsahu v izolovaném prostředí, buď v Docker kontejnerech nebo na virtuálním stroji pomocí Vagrantu. Integrace také umožňuje sdílení částí obsahu napříč repozitáři.

Klíčová slova sandboxing, izolace procesů, cachování obsahu, Python, Git, Docker, Vagrant

Contents

List of Code examples

List of Figures

Introduction

Nauč se Python, Czech for *Learn Python*, is a project maintained by the Czech Python community that serves as a hub for open-source teaching materials, mostly about Python. These materials are used in courses organised by the community in the Czech Republic, for both beginners and more advanced programmers. The main output of the project, the website naucse.python.cz, contains not only the materials themselves but also metadata about the courses, like the schedule, dates and times or the location.

The contents of materials, associated files, metadata about the courses and the rendering code are stored in a single Git (version control system) repository. A small group of maintainers takes care of the repository, but they usually are not the primary organisers of the courses.

Only these maintainers can modify the content directly and everybody else has to use Pull Requests, a GitHub feature, to submit changes. The maintainers have to approve any changes, resulting in the first issue with the current situation – last-minute changes are almost impossible because the review of the changes can take even several days.

Another issue, which is only getting worse with passing time, is that any changes must be compatible with all current and past courses. That is rather impractical when the materials need to be updated considerably or if only a specific course has to be changed.

The current situation is clearly not satisfactory. Unfortunately, conventional solutions to this kind of problem are not ideal for this specific project and that is how this thesis came to be.

I chose this thesis topic because it combines several fields in which I am both professionally and personally interested. The whole code is in Python, the whole project is open-source and the tools required to implement the thesis also have to be open-source. The solution lays out quite a lot of challenges, requiring smart and creative solutions. Most importantly, the final result will help actual people with their real problems and contribute to an excellent project helping the community provide quality teaching materials and organise great courses for the public.

This thesis will first explain in depth how the project works, the issues with the current situation and what solutions are available. I will then analyse the assigned solution and define the requirements.

The assigned solution utilizes the functionality of Git and GitHub, the host of the base repository of the project. GitHub allows users to make a *fork* of a repository, a complete clone that belongs to them with all the privileges. The project will use these forks as a source of parts of the website – of specific courses. The users will be able to make almost any change to the contents of the course in their fork, including the rendering code, but any changes will only apply to their course.

The first thing the solution requires is a tool that can render HTML fragments from Git repositories using Python. Since the code will not be reviewed by the maintainers, the tool also must sufficiently isolate the runtime, so the main process is safe from leaks or exploits. As the chapter about research into existing tools will reveal in more detail, there is not an existing tool suitable for the job. Therefore, I will have to develop one.

I will write the tool using the latest Python to utilize all the latest language features and make it highly configurable. The primary isolation provider will be Docker containers and virtual machines, however, the tool will also enable rendering HTML without isolation. I will place great emphasis on the effectiveness of the tool, speeding up cloning repositories as much as possible, caching results to prevent duplicate work, and implementing other enhancement that will enable quick integration of this tool.

Then the tool will be integrated into the project. In addition to allowing courses from the forks, the assignment of this thesis requires this be done efficiently, caching fragments of the content across repositories sharing the same rendering code. The integration also has to catch any errors occurring in the forks to not break the whole site and display a warning text instead.

Finally, the changes made in the forks have to be propagated to the main website in the most automated fashion possible. Webhooks will be used for this purpose, triggering automatic rebuilds of the website once the underlying sources change in the forks.

All the written code and all the decisions made will try to respect standard Python practices and follow the Zen of Python. All the code will be made publicly available under open-source licenses. While not strictly utilizing test driven development, the tests will be first-class citizens of the projects. The individual parts of this thesis will practice the Continuous Integration principles.

Chapter 1

Analysis

Nauč se Python, Czech for *Learn Python*, is a project maintained by the Czech Python community that serves as a hub for open source teaching materials. The main output of the project is the website `http://naucse.python.cz/`. While the project could be deployed to multiple locations, it is practically a singleton; there is only one production instance. *Naucse* will be used to refer to the project, the web application and the only production instance on naucse.python.cz. For the purpose of this thesis, all of them are the same thing.

This chapter will describe how Naucse currently works, how it is rendered and the workflow for adding and updating content. Then, I will outline the issues with the current situation and describe possible solutions, including the one assigned for this thesis. Finally, I will define the functional and non-functional requirements set forth by the assignment of this thesis.

1.1 Current situation

The materials in Naucse are organised in two major ways. First, there are *canonical courses*, curated collections of materials separated into lessons and individual parts of lessons. These canonical courses are written to make sense on their own without any context and are accessible to anybody on the website.

Then there are *runs*. Runs are specific real-life courses or workshops bound to some date and place. These runs can be based on canonical courses, like the 2017/2018 *MI-PYT (Advanced Python)* class at FIT CTU [1]. Runs can also be compiled of just some materials used in the canonical courses or even other materials which are not

a part of the canonical courses. An example of a run compiled this way would be the *Asteroids workshop* at InstallFest 2018 [2].

Both canonical courses and runs will be referenced in the text under a common term *courses*.

The code, the contents of materials and the definitions of courses are all in a single Git repository on GitHub [3]. The website is always rendered from the last version of the repository, as can be seen in figure 1.1. The repository is primarily maintained by two people – these two maintainers sometimes attend or organise runs, but usually they are not the primary organisers.

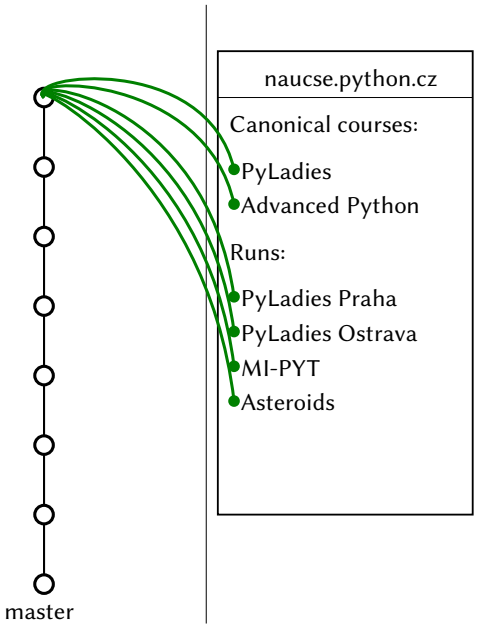

Figure 1.1: Illustration of rendering a website from the last version of a repository

1.1.1 Structure of the application

The core of Naucse is a `Flask` [4] application – `Flask` is a Python microframework for web applications. The `elsa` [5] package is then used to turn this dynamic application into static Hypertext Markup Language (HTML) pages. The package calls it *freezing* the application and it works by going through the application, finding all links within the application and creating a directory structure with HTML files matching the URLs of application. The package also implements updating the content on *GitHub Pages*. `Flask` and `elsa` and other packages are declared as requirements in `requirements.txt`.

As mentioned, the whole application is in a single Git repository. The structure of the repository is shown in directory structure 1.1. The code is contained in the `naucse` folder, the content in `courses`, `lessons`, `licenses` and `runs`.

The content consists of files of multiple types, while the folder names act as identifiers.

- YAML Ain't Markup Language (YAML) [6] files are used for metadata like titles, descriptions, dates and times and for defining the plan of courses.
- `Markdown` [7] files or `Jupyter Notebooks` [8] are used for the texts of materials.
- The lessons can also contain static content, such as images or audio files.

```
naucse.python.cz
├── courses/
│   ├── <canonical course identifier>
│   │   └── info.yml ..... metadata and plan of sessions and lesson
│   └── info.yml ..... display order of canonical courses
├── lessons/
│   ├── <collection>/<identifier>/ ..... folder for a specific lesson
│   │   ├── static/ ..... (optional) folder for related static files
│   │   ├── index.(md|ipynb) ..... the main page of the lesson
│   │   ├── info.yml ..... metadata (name, licence etc.)
│   │   └── *.(md|ipynb) ..... subpages of the lesson
├── licenses/
│   ├── <license identifier>/
│   │   └── info.yml ..... information about the license
├── naucse/ ..... the code rendering the project
├── runs/
│   ├── <year>/<run identifier>/
│   │   └── info.yml ..... metadata, times and plan of sessions and lessons
├── test_naucse/ ..... tests for the rendering code
├── .travis.yml ..... definition file for Travis CI
├── requirements.txt ..... dependencies for the app
└── test_requirements.txt ..... dependencies for tests
```

Directory structure 1.1: Structure of the Naucse repository

1.1.2 Local development

The application can be launched locally to preview changes. The app requires Python versions 3.5 or 3.6 and installation is quite straightforward (shown in code example 1.1). This preview is available in two modes, "serve" (shown in code example 1.2) and "freeze" (shown in code example 1.3). Mode "serve" creates a web server which renders individual pages ad hoc when requested – local changes are visible immediately. Mode "freeze" builds the entire site as if it was going to be deployed to production and creates a web server which serves the already rendered static HTML pages.

```
git clone https://github.com/pyvec/naucse.python.cz
cd naucse.python.cz
pip install -r requirements.txt
```

Code example 1.1: Installing Naucse locally

```
PYTHONPATH=. python -m naucse serve
```

Code example 1.2: Running Naucse locally in mode "serve"

```
PYTHONPATH=. python -m naucse freeze --serve
```

Code example 1.3: Running Naucse locally in mode "freeze"

1.1.3 Content deployment

Naucse uses the Continuous Integration (CI) and Continuous Delivery (CD) tool *Travis CI* [9] for automatic deployment to *GitHub Pages* [10]. *GitHub Pages* is a service by *GitHub, Inc.* for deployment of static HTML pages. Once an update is pushed to the base branch of the Git repository (`master`), a new build is triggered on *Travis CI*. The build launches tests and if they pass, a new version of the content is frozen and updated on *GitHub Pages*.

Apart from running tests after the base branch is updated, *Travis CI* runs tests in Pull Requests. Pull Requests are a GitHub feature for proposing changes in the main repository, the difference being the frozen content is not updated on *GitHub Pages*.

1.1.4 Current workflow for modifying content

Naucse utilizes GitHub Pull Requests for proposals of what should be modified. Pull Requests are a GitHub feature which allows GitHub users to submit code which they think should be changed in the base branch. This code can be and usually is accompanied by a text explaining the changes and the reasoning behind them. Anyone can comment on these pull requests, but only the maintainers can approve the changes and merge them. Naucse has two core maintainers and usually at least one has to approve the change before it is merged. Even the maintainers use pull requests to submit changes – the other one then has to approve the change.

There is a wide range of changes which can be proposed with a wide range of consequences. The core principle is that any changes to lessons or the rendering code affect all courses. Each pull request, therefore, has to be evaluated individually to determine whether the effect of the changes is destructive in some way.

The workflow is following:

1. The user creates a fork (will be explained further in detail) of the base repository to their own GitHub account and installs it locally.
2. The user makes the appropriate changes and pushes them to their own repository.
3. The user creates a pull request with the changes and submits it for review. *Travis CI* launches tests and if they pass it also freezes the content as an integration test. But it does not deploy the changes on *GitHub Pages*.
4. The maintainers review the changes and merge them if everything is in order, otherwise asking for modifications. *Travis CI* build is triggered when changes are merged and the website is updated.

1.2 Issues with the current situation

There are several issues with the current situation and the workflow of updating the content, which will be described in this section.

1.2.1 Changes in content affect every course and run

This issue comes from the fact that all the materials are in the same repository, so runs just select which materials they want to use. This has obvious benefits: when an error or a typo is fixed in the materials it is automatically fixed in all the courses. On the other hand, this limits customisation of material for runs – when making

changes to lessons one must be considered if the change will make sense in all courses.

1.2.2 Major changes break old runs

This issue is an escalation of the previous one. The current situation makes it practically impossible to substantially rework old materials without updating all the old runs. The issue is most prominent when merging or splitting individual material fragments. When splitting materials the old runs must be updated with the change. When merging materials this becomes harder because the old runs might have not even included all the merged parts.

This problem is only getting worse. Currently there are only about 20 ongoing or past runs, which is still manageable and the old runs could be updated, but at the current pace of 10 runs a year, the old runs will get out of hand.

1.2.3 Old runs should not change

The runs are not removed from Naucse once they have occurred, but are kept there so people who attended the course or the workshop can return to the materials and use them as a cheat sheet or for inspiration. However, because of the two previous issues, in time the materials slide away from the state they were in when the run ended. This should not be the case; runs are specific real-life events, and the materials for those runs should stay the same forever, except for error fixes. This issue is also described in the upstream issues no. 214 [11].

1.2.4 Unfinished materials should not be merged

The aim of the project is to provide quality materials which can be reused by others with the potential to organise runs with these materials. The impact of this aim is that only finished and polished materials are usually merged to the repository. This complicates things for runs which run on experimental or completely new materials, which need to be submitted early for review by the maintainers.

1.2.5 Last minute changes are complicated

This is an issue for organisers of runs; any problems with their material and their runs often cannot be fixed quickly since the pull requests have to wait for approval. While minor changes like typos are usually merged quickly, bigger changes can take days to be reviewed and approved.

1.3 Possible alternative solutions to the issues

This section will explain solutions to the issues described in the previous section other than the solution outlined in the assignment of this thesis, that one is described in the next section. This will hopefully explain why the assigned solution is really necessary.

1.3.1 Material duplication

One solution to the issue of backwards compatibility and breaking changes would be duplicating the materials. After a run would finish, the materials could be frozen at the last version by making a duplicate version of all their materials. All the materials would require new identifiers to not clash with the originals and the identifiers would have to be updated in the run metadata and in the materials themselves since they are cross-referenced. The runs would further be rendered from this duplicate version and changes in the upstream version would not have an effect on the past runs.

This solution is not ideal. It does not solve all issues and is inefficient, storage and computational wise. The history of the changes in the materials would be effectively lost for the duplicates and in general, it breaks the Don't Repeat Yourself (DRY) principle [12].

1.3.2 Relaxing standards

The issue of backwards compatibility, breaking changes and unfinished materials could be solved by relaxing standards. If past runs were declared as unimportant, changes in materials would be easier since backwards compatibility would not have to be considered. Similarly, unfinished materials could be allowed in the repository so runs can be organised with them.

This solution is also not ideal. To facilitate good experiences for the users of the website, old content should still work as expected – in the form they were used to. And while past runs could be deleted from the website, there are usually multiple ongoing runs which would still be affected by changes.

1.3.3 Extended privileges

The issue with last minute changes being complicated could be solved by granting merge privileges to the organisers of the runs. That would enable them to make last minute changes to their runs.

This solution would solve that particular issue, but it would create even more other issues. With merge privileges, an organiser could cause a lot of damage to the project, from disrupting the contents of other runs to disabling rendering completely. This solution is therefore not ideal.

1.3.4 Self-hosting

The whole project is open-source, so an organiser could decide to host the entire website on their own. This solution would solve the issues, but it is not ideal. The solution requires far more technical skills for the organisers than the solution from the assignment. There is also an issue with the domains; the default domains that *GitHub Pages* provides are not very representative and not everybody has their own domains. Finally, sometimes an organiser wants to change something in the run after it already started and suddenly the attendees would have to be notified of the change, creating a lot of confusion.

1.4 The assigned solution

This section will describe the solution selected by the maintainers of the project, the subject of this thesis. The assignment is available at the beginning of this thesis and is described in detail in the upstream issue no. 175 [13].

The core of the solution is to render parts of the website (courses) from other sources than the base branch of the base Git repository, from *arbitrary branches*. Arbitrary branch is a term I will be using in the rest of the text which stands for both a branch of the base repository or a branch in a *fork*. A fork is a copy of a Git repository, that is fully owned by the person that forked it, with all privileges.

Instead of merging the full `info.yml` file (as described in section 1.1.1) to the repository, a file containing only a link to the arbitrary branch, consisting of an URL to the repository and a branch, will be merged. When the `Flask` application will be asked to render this course, it will clone the arbitrary branch and use its contents to display the materials.

Figure 1.2 shows a diagram to help illustrate the principle. The left part is a graph of arbitrary branches, with the left-most path that only goes up being the `master` branch – the base one. The circles symbolize individual commits, the offshoots to the right are the branches. The right part symbolizes the website. The blue and green lines indicate what part of the website is rendered from which commit, blue indicating rendering from arbitrary branches, green from the base branch.

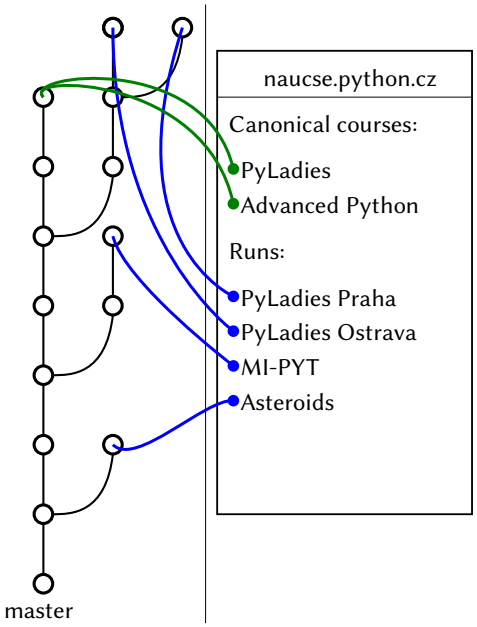

Figure 1.2: Illustration of rendering parts of a website from arbitrary branches

Once the solution is complete, all the issues with the current situation outlined in section 1.2 will have a solution:

1.2.1 The organisers will be able to make changes to the content and its rendering as they wish in their forks and it will only affect their run, the canonical courses and content rendered from the base repository will remain unaffected.

1.2.2 Since runs will be rendered from an arbitrary branch, changes will only affect the canonical courses and will not break the runs. The changes then can be merged to ongoing runs, if the organisers wish so.

1.2.3 Finished runs can be moved to an arbitrary branch and if the maintainers of that repository do not merge any new changes, the contents will remain frozen.

1.2.4 Runs relying on unfinished materials can be rendered from an arbitrary branch, the base branch will remain clean.

1.2.5 Changes made in runs rendered from forks will not be subject to a review from the maintainers, the organisers will be able to make changes when they want. The only thing subject to review will be the link to the arbitrary branch.

1.4.1 Integration

The assignment and the upstream issue no. 175 [13] also lay out some instructions for the integration of this solution into Naucse.

First, a tool must be found or implemented that enables rendering HTML fragments from Git repositories. This tool has to be written in Python, in a version that is compatible with the version used now in Naucse, meaning Python 3.5 or 3.6. The tool must provide proper isolation of the runtime – the code in forks is untrusted, it is not reviewed by the maintainers. The tool has to be able to install requirements using `pip` from a requirements file [14]. Further, it should be documented, tested and released as open source under the terms of the Massachusetts Institute of Technology (MIT) license.

The tool has to be then integrated into Naucse, enabling rendering of parts of the page based just on a link to a Git repository and a branch within it. According to the issue forks will usually share code and will most likely share a large part of the content as well, the integration has to cache individual fragments across arbitrary branches that share the same rendering code to prevent duplicate work. Errors in rendering in forks must be handled in such a way as to not break the deployment of the entire website. Instead, a warning should be displayed and if the page is available in the base branch, it should be displayed instead.

A part of the materials defined in Naucse are images like illustrations or diagrams and other files, like audio samples or prepared scripts. These files are embedded or linked in the HTML generated by freezing. The tool has to be able to retrieve these files from arbitrary branches since the files therein can be changed as well.

When running locally, Naucse should behave as-is and no extra non-Python dependencies should be required.

Deployment of the page should not change in any way, the website should still be deployed using *Travis CI* to *GitHub Pages*. A new build on *Travis CI* is only triggered when an update is pushed to the `master` branch, so with the integration a new build must be triggered after each update in the forks.

The tool will be often launched on arbitrary branches that have not changed, so the tool must cache the results to prevent repetitive work. This is different from the caching of individual fragments described above, this is caching the entire calls in the arbitrary branches based on the state of the branch. This cache must work on *Travis CI* and must be persistent.

1.5 Requirements

This section will summarize the functional and non-functional requirements for the tool for rendering HTML fragments from Git repositories and for its integration into Naucse, as described in the previous section.

1.5.1 Naucse

Functional requirements:

NF1 Naucse has to allow rendering courses from arbitrary branches based on merged metadata.
NF2 Naucse has to cache individual content fragments between all courses that share the identical rendering code.
NF3 Naucse has to be rendered and deployed on each update in arbitrary branches, by triggering a *Travis CI* build. This triggering should be as automatic as possible.
NF4 Naucse has to handle errors in arbitrary branches, showing a warning with a description of the problem. If the content exists in the base branch of the base repository, it should be rendered as a replacement.

Non-functional requirements:

Naucse is an established project which is only being extended in this thesis, the code extending the functionality has to work within the existing structure. Furthermore, the code has to be written under the terms of the existing open-source license.

NN1 The code has to be well structured.
NN2 The code has to be written in Python.
NN3 The code has to be commented in English.
NN4 The code has to be released under the MIT license.
NN5 The code has to be well tested.
NN6 When running Naucse locally, there cannot be non-Python dependencies and forks should be ignored, unless specified otherwise.
NN7 The workflow of using the extension has to be described in a Czech meta-course on Naucse.

1.5.2 The tool

Functional requirements:

AF1 The tool has to be able to render static HTML fragments.
AF2 The tool has to be able to render the fragments from code contained in Git repositories.
AF3 The tool has to be able to install requirements using `pip` from requirements files.
AF4 The tool has to be able to cache entire results of calls based on the state of the repository.
AF5 The tool has to be able to retrieve files from the repositories.
AF6 The tool has to isolate the runtime to prevent exploits.
AF7 The tool has to be runnable even without non-Python dependencies, even at the cost of less isolation.

Non-functional requirements:

AN1 The code has to be well structured.
AN2 The tool has to be written in Python 3.5 or 3.6.
AN3 The code has to be commented in English.
AN4 The tool has to be open source.
AN5 The tool has to be well tested.

Chapter 2

Research

This chapter contains the research into existing tools for rendering HTML fragments using Python with runtime isolation. The tools are judged on the count of requirements described in section 1.5.

2.1 process_isolation package

`process_isolation` [15] is a Python 2 package that can create a wrapper around Python modules enabling interacting with them as per usual, but running their code in a subprocess. The original intent of the package was to test unstable C modules, which can fail completely without throwing a catchable exception. The scope was then extended with a possibility to put the process in a chroot jail and therefore serving as a tool for running untrusted code in an isolated environment.

The usage of the package without chroot is shown in code example 2.2 – it imports code example 2.1. Interacting with the imported module is the same as if it was imported regularly, but the call is actually in a subprocess. This is illustrated by printing the Process ID (PID). When the `os.getpid` function is called a different PID is printed as opposed to when `untrusted.getpid` is called. The `untrusted.ls_root` function call then shows that the process has read access to the root directory.

```
import os

def ls_root():
    return os.listdir('/')

def getpid():
    return os.getpid()
```

Code example 2.1: The `untrusted.py` file for `process_isolation` examples

```
import os
from process_isolation import import_isolated

untrusted = import_isolated("untrusted")

# prints e.g.: 7646
print(os.getpid())

# prints e.g.: 7694
print(untrusted.getpid())

# prints: ['lost+found', 'tmp', 'mnt', 'boot', 'srv', 'usr', 'opt', 'sbin', 'etc',
#          'dev', 'root', 'run', 'proc', 'media', 'sys', 'home', 'var', 'lib',
#          'lib64', 'bin']
print(untrusted.ls_root())
```

Code example 2.2: Example usage of `process_isolation` without chroot

The usage of the package with the chroot jail is shown in code example 2.3. This example prints only the `some_folder` folder in its `untrusted.ls_root` call.

```
import os
import sys
import process_isolation

context = process_isolation.default_context()
context.ensure_started()

try:
    # Install the chroot
    os.mkdir('/tmp/chroot_jail')
    os.mkdir('/tmp/chroot_jail/some_folder')

    context.client.call(os.chroot, '/tmp/chroot_jail')
except OSError:
    print('This script must be run with superuser priveleges or '
          'the chroot folder already exist.')
    sys.exit(1)

# the module can be now safely imported
untrusted = context.load_module('untrusted', path=['.'])
print(untrusted.ls_root())  # prints: ['some_folder']

# clean up
os.rmdir('/tmp/chroot_jail/some_folder')
os.rmdir('/tmp/chroot_jail')
```

Code example 2.3: Example usage of `process_isolation` with chroot

A benefit of the package is the possibility to run entire modules with the package, which can import further packages. That means the whole Naucse `Flask` app could be imported and interacted with. The downside is that the package does not match requirement AN2, it only runs in Python 2 and because of that is not compatible with Naucse. More importantly, chroot jail does not provide proper isolation. As Joshua Bressers says in *Red Hat Security Blog* entry "Is chroot a security feature?" [16] "*[...] it is not really a security feature, it is closer to what we would call a hardening*

feature. It might slow down an attacker, but in most situations it is not going to stop them."

Conclusion: the package `process_isolation` cannot be used since it does not sufficiently isolate runtime and it is not compatible with the Python used to run Naucse.

2.2 RestrictedPython package

As the name suggests, the `RestrictedPython` [17] package aims to provide a restricted and safe subset of the Python language. It is managed by the *Zope Foundation* [18] and it is primarily used in the *Plone* [19] content management system, built on top of the `Zope` [20] application server.

It works by whitelisting specific operations (the whitelist is managed by the package itself) during the compilation of the code on the level of Abstract Syntax Tree (AST) objects and then running the compiled code in a restricted environment. The restricted environment consists of safe builtins and can be expanded by explicitly whitelisting further builtins or packages [17]. The code example 2.4 shows example usage. This is the minimal setup required to print anything.

```
from RestrictedPython import compile_restricted_exec, PrintCollector

hello_world = """
import platform

print("Hello, World!")
print(platform.python_version())
"""

safe_globals = {"_print_": PrintCollector, '_getattr_': getattr}

byte_code = compile_restricted_exec(hello_world)

exec(byte_code.code, safe_globals)

# prints:
# Hello, World!
# 3.6.5
print(safe_globals["_print"]())
```

Code example 2.4: A Hello World example in `RestrictedPython`

This package could be configured in such a way that rendering HTML fragments would be possible, however, its integration into Naucse would be complicated. Naucse uses multiple different packages for many purposes and an extensive whitelist would have to be compiled and maintained while adding new features – and all of the packages would have to be carefully reviewed if they are safe to use by

untrusted code. Furthermore, the restrictions put on the code by `RestrictedPython` are far too big to write reasonable programs. For example, the `yield` keyword is prohibited from code (code example 2.5) – a feature of Python which is used extensively in Naucse. Even the current *Plone* documentation says "*RestrictedPython was bad idea and mostly causes headache. Avoid through-the-web Zope scripts if possible*" [21].

```
from RestrictedPython import compile_restricted_exec
function_with_yield = """
def func():
    for i in 10:
        yield i

print(list(func()))
"""

byte_code = compile_restricted_exec(function_with_yield)

# prints: ('Line 4: Yield statements are not allowed.',)
print(byte_code.errors)
```

Code example 2.5: Restrictions in `RestrictedPython`

Conclusion: the `RestrictedPython` package cannot be used since the subset of Python it uses is not compatible with Naucse and the configuration would be far too complicated to manage. Above that, even the system which uses `RestrictedPython` discourages its usage.

2.3 pysandbox package

`pysandbox` [22] is a Python sandbox. It puts highly configurable limitations on what the executed code can do. The code is executed in a subprocess, so even system limitations can be placed on the program, such as maximum execution time or memory usage.

The default limitations are very strict – for example, file system access is forbidden completely and all imports are forbidden, but individual features can be whitelisted. File access can be enabled and limited to only specific paths. Imports can be enabled as well, even from external modules. Alternatively, only specific objects can be whitelisted.

Even if the fact that only Python 2.5 and Python 2.6 are supported was ignored, the author himself declared the project to be broken by design in an email [23] to the *Python-Dev* email conference – meaning that it would be unwise to use this sandbox.

Conclusion: `pysandbox` cannot be used since the Python it uses is not compatible with Naucse and because its author declared it broken.

2.4 PyPy sandbox

The `PyPy` [24] sandbox [25] implements a different approach to sandboxing than `pysandbox`. Instead of limiting language features it replaces calls to external modules with stubs that communicate with the parent process which decides if the call is safe [26].

This sandbox, unfortunately, cannot be used. Firstly, the PyPy site itself says that the sandbox is only a prototype and it warns that even extensions modules like `time` do not work [26]. Secondly, it is a `PyPy` sandbox, meaning `PyPy` is used to run the code and although `PyPy` is mostly compatible with Python code [27], there could be problems. According to the documentation, the sandbox is only tested with PyPy 2 and it might not work with PyPy 3. To run Naucse in PyPy, the PyPy 3 version would have to be used, because that one is compatible with Python 3, while PyPy 2 is only compatible with Python 2.7. So even if Naucse ran on PyPy, the sandbox could not be used.

Conclusion: the `PyPy` sandbox cannot be used since the Python it uses is not compatible with Naucse and because, according to its own documentation, it is not finished.

2.5 codejail package

`codejail` [28] is sandboxing package which relies on `AppArmor` [29] for its security. `AppArmor` provides Mandatory Access Control, a security method where all actions must be explicitly allowed by a policy.

To use the package, a Python virtual environment [30] must be installed and an `AppArmor` profile must be manually configured for the environment. The profile limits what files and resources the Python executable can access. The package then provides a Python class which can execute scripts in the virtual environment – these scripts can be entire files. The class further enables to provide more read-only files which might be required by the script.

This package is unsuitable for the thesis because of `AppArmor`. `AppArmor` is only available for some Linux distributions (for example Fedora is missing, from the more prominent distributions) [31]. That, unfortunately, means `codejail` cannot be used

generally on Linux and cannot be used on Windows at all. A further problem is that even on Linux distributions that support `AppArmor`, this tool requires complicated manual configuration and it doesn't work out of the box.

Conclusion: `codejail` cannot be used since `AppArmor` is not generally available and it requires a lot of manual configuration.

2.6 docker-python-sandbox package

`docker-python-sandbox` [32] is a `NodeJS` [33] sandbox, which sandboxes Python code using Docker [34] containers. This package can be dismissed immediately due to requirement AN2 (the tool must be written in Python) because it is in fact written in JavaScript. Nevertheless, it is an interesting example of what can be achieved using different technologies.

The package maintains a pool of running containers ready for immediate usage, which saves on boot up time. Once the container is used, it is shut down and another one is booted up to take its place in the pool. A diagram of how the package works is shown in figure 2.1.

Setting up the package and running an example is straightforward, as the code example 2.6 shows. This script, when launched, starts an idle container waiting for instructions. Then a Python 3 script is sent which prints the Python version and a "Hello, world!" message.

```
let Sandbox = require('docker-python-sandbox');
const poolSize = 1;
let mySandbox = new Sandbox({poolSize});

mySandbox.initialize(err => {
  if (err) throw new Error(`Unable to initialize the sandbox: ${err}`);

  const code = `import platform
print(platform.python_version())
print("Hello, world!")`;
  const timeoutMs = 2 * 1000;

  mySandbox.run({code: code, timeoutMs: timeoutMs, v3: true}, (err, result) => {
    if (err) throw new Error(`Unable to run the code in the sandbox: ${err}`);

    console.log(result.stdout);
    console.log(result.stderr);

    process.exit()
  })
});
```

Code example 2.6: A Hello World example in `docker-python-sandbox` (modified code from [32])

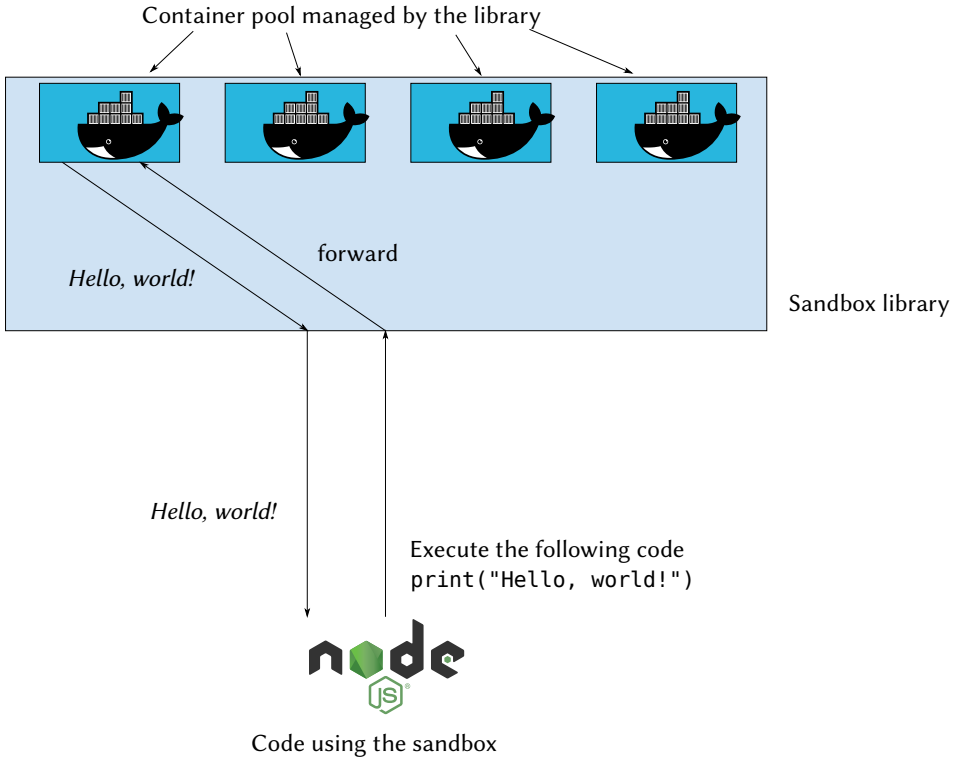

Figure 2.1: Diagram of the `docker-python-sandbox` package (modified image from [32])

Conclusion: `docker-python-sandbox` cannot be used since it is not written in Python.

2.7 Custom tool

None of the existing tools satisfies the requirements set forth by the assignment, a new tool to accomplish the goals will have to be implemented before starting the integration of rendering content from forks.

Chapter 3

Tool implementation

The research did not reveal any suitable tool for rendering HTML fragments from Git repositories, one has to be implemented instead. This chapter will describe the design of the tool, its isolation and the implementation details.

I decided to name the tool `Arca`, the Latin word for "*a place for keeping any thing, a chest, box*" [35]. I also decided not to limit the tool to only rendering HTML fragments, but to allow for any generic Python callable (anything that can be called in Python, functions, static methods, classes) to be launched from the repositories. This should help the reusability of the tool, so it can also be used by other projects, not just by Naucse.

3.1 Design

This section will describe the design of the tool and its parts. The design is dictated by the requirements and is influenced by SOLID design principles [36] and the Zen of Python [37].

One of the primary goals of the tool is to isolate environment of the runtime to prevent exploits. To provide proper isolation external dependencies are required (more in section 3.2), but those non-Python dependencies are forbidden by the requirement NN6. This means the tool has to be able to run without the dependencies, but still provide the same functionality without isolation. Because of this, the Strategy design pattern [38] is used to implement different strategies for running the callables. This enables using a Python-only strategy when running Naucse locally and using a different strategy for deploying from *Travis CI*. It also helps the usability of the tool by other people – enabling quick bootstrapping to test

that the tool matches their requirements and setting up proper isolation later. The tool will also be much more extensible this way. Changing the isolation will not be a matter of rewriting the entire tool but implementing another strategy instead.

The original Strategy pattern is usually used for simpler algorithms, I felt that the name does not quite match what I'm doing here, so I am calling the different strategies *backends* instead. This more matches the complexity of the individual backends, which have their own settings and some even require extra system dependencies.

The Unified Modeling Language (UML) class diagram of the classes used in the tool in displayed in figure 3.1. The class `Arca` serves as the main Application Programming Interface (API) for the tool. It handles the Git repositories, caching and delegates the execution of the callables to backends. The class also initializes a `Settings` instance, which handles the configuration of the instance and of backends. The design of the configuration is described in detail in section 3.1.1.

Once `Arca` is configured, the selected backend is initialized (or set, if passed already initialized). Backends are always a subclass of `BaseBackend` – an abstract class that defines the interface for individual strategies and provides some common functionality. The single purpose of backends is to run the callables it gets from the `Arca` class.

The `Task` class serves for encapsulating the definition of the callables and arguments its arguments. Its instances are immutable, meaning the definition cannot be changed after it is first set.

Finally, the `Result` class serves as a wrapper around the result of the callable.

3.1.1 Configuration design

The design of the configuration has two goals. First is to make the configuration intuitive for first-time users, to make things how a Python programmer would expect them to be. This means passing configuration options straight to constructors of the objects when creating them. The second is overriding of the configuration with environment variables. This makes the integration of the tool smooth on CI tools – the code does not have to know where it is running, it just uses the configuration provided.

These two goals are accomplished by the two ways the tool can be configured.

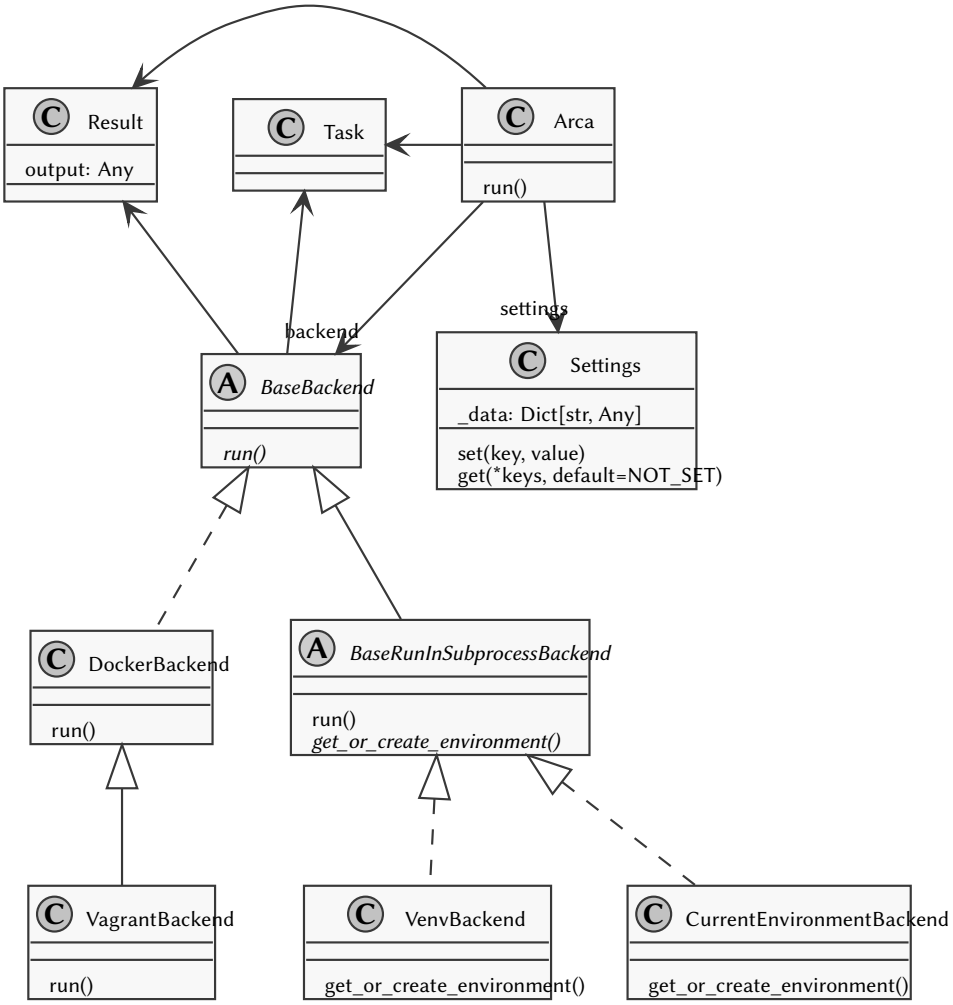

Figure 3.1: UML class diagram of classes used in Arca

Explicit constructor options

The first way to configure the tool is passing the values for the options to constructors of individual classes. This way is shown in code example 3.1.

```
from arca import Arca, DockerBackend

Arca(base_dir="/tmp/arca",
     backend=DockerBackend(python_version="3.6.5"))
```

Code example 3.1: Configuring Arca explicitly

Settings

The second way uses a settings dictionary which also automatically takes in environment variables. The dictionary requires keys in all capitals with a prefix – to match the standard format of environment variables and to prevent name collisions. The usage in shown in code example 3.2.

```
from arca import Arca

Arca(settings={
    "ARCA_BASE_DIR": "/tmp/arca",
    "ARCA_BACKEND": "arca.DockerBackend",
    "ARCA_BACKEND_PYTHON_VERSION": "3.6.5"
})
```

Code example 3.2: Configuring Arca using settings

To further make the settings even more granular, the settings for backends have two forms. One sets up the value for all backends which share the configuration value (with prefix `ARCA_BACKEND`) and one sets up the value only for a specific backend (with prefix `ARCA_<NAME>_BACKEND`, where `<NAME>` is replaced with the name of the backend).

This dichotomy sort of goes against the "*There should be one-- and preferably only one --obvious way to do it*" recommendation in the Zen of Python [37]. There is a *single obvious* was to do it, it is just not the only one. I chose to use a second recommendation from the Zen, "*Although practicality beats purity*", because this tool will be used on *Travis CI*, it is really practical to enable configuring using environment variables out of the box.

3.1.2 Caching design

The requirement AF4 states that the tool has to be able to cache the entire results of calls based on the state of the repository. On the recommendation of the maintainers of Naucse, I decided to use the `dogpile.cache` [39] package for caching in Arca. This package provides a generic API to caching backends and allows Arca to not be dependant on any specific caching system.

The usage of the package relies on two major components – region and backend. The region is the "frontend", the interface for applications. The backend is configured within the region and it is the specific caching mechanism used.

The package provides backends for major cache systems – caching in memory, in files, in `memcached` or `Redis` and also a `null` backend which does not actually store anything [40].

Using the `null` backend as the default, `Arca` instances always have a region configured, which is used in its `run` method. The method always checks if the result is in the cache before delegating the task to a backend.

The base for successful caching is the key under which the values are stored, in this case, the key has to uniquely identify the components used in it. The key Arca uses for caching is comprised out of:

- the target repository Uniform Resource Locator (URL),
- the target branch name,
- the commit hash of the current state,
- the hash of the `Task` instance (described in section 3.3.8).

The repository URL and branch name have to be included in case the callable is analysing in some way where it is running – multiple repositories or branches can share a commit with the exact same hash. If the result of the callable depended on the repository/branch and information about the repository and branch was excluded from the key, an incorrect result could be returned.

3.2 Isolation

As mentioned in the analysis, the tool has to isolate the runtime to prevent exploits. The isolation the tool uses is described in this section.

The research into existing tools (section 2) already rules out some ways to implement the isolation. Sandboxing Python to only run a subset of the language is not advised (`RestrictedPython` in section 2.2 and `pysandbox` in section 2.3). Running processes in a chroot jail is not sufficient (`process_isolation` in section 2.1). `AppArmor` or `SELinux` could be used, but they are not even available for all Linux distributions equally, let alone on macOS or Windows.

Another detail to consider is that the tool has to be able to run on CI tools, which limits what security measures can be used. This rules out `AppArmor` and `SELinux` as well.

Arca provides three tiers of isolation, users can implement their own.

3.2.1 No isolation

Two backends are included with the tool which provide virtually no isolation. They are `CurrentEnvironmentBackend` and `VenvBackend` and they both run Python in a subprocess – they only differ in the Python used. The first uses the same Python executable used to launch Arca, the latter uses a separate Python, from a Python virtual environment [30] with its own requirements.

The purpose of these two backends is to provide a quick way stitch together a proof of concept – they don't provide any actual security. Alternatively, they can be used to programmatically run Python callables from trusted Git repositories.

3.2.2 Isolation through containers

The `DockerBackend` aims to provide isolation through containers on CI tools or in environments where isolation through virtual machines cannot be used. It uses *Docker* [34] containers to isolate the runtime.

The approach of using Docker is much more secure than just running callables in a subprocess, however, the security is not ultimate. As Daniel J Walsh (a security expert currently working on container integration in Red Hat [41]) says in his "Are Docker containers really secure?" [42] article: "*Containers do not contain*" and "*Stop assuming that Docker and the Linux kernel protect you from malware*".

This article (and its second part "Bringing new security features to Docker" [43]) goes to great depth about the issue, but it can be summed up in the fact that the containers talk directly to the kernel. These two articles also lay out some recommendations for running things in Docker – which are incorporated in Arca.

The first recommendation (which is even in the `Docker` documentation on security [44]) being followed is that "*only trusted users should be allowed to control your Docker daemon*" [44]. This recommendation is followed because the programmer using Arca is the *trusted* user, they and only they control the containers.

The second recommendation "*Don't run random Docker images on your system*" [42] is followed by running an image made specifically for the usage by Arca. This image is based on a stable Linux distribution, Debian. By building a custom image for the containers, another recommendation "*Drop privileges as quickly as possible*" [42] and "*Run as non-root whenever possible*" [43] can be followed more easily. The root user is only used to install system dependencies, otherwise, everything is executed

under an unprivileged user which only has access to the Python executable and the data of the repository.

Docker (in its Community Edition which is required for running Arca) is available for most major Linux distributions, on macOS and on Windows [45] and is available for on *Travis CI* [46], the CI tool used to render Naucse.

3.2.3 Isolation through virtual machines

Finally, the third tier of isolation is running Python in a full Virtual Machine (VM), using *Vagrant* [47]. *Vagrant* is "*a tool for building and managing virtual machine environments in a single workflow*" [48], which makes creating and controlling VMs easy. *Vagrant* is built on top of other VM tools which makes it highly configurable and enables it to work on all platforms – Linux, macOS and Windows are all supported [49]. The tool also works with multiple different VM technologies (*Vagrant* calls them *providers*), so the VM can be launched in a variety of ways. *VirtualBox* [50], *VMWare* [51], *libvirt* [52] or *Hyper-V* [53] are available. The VMs can be even launched on Amazon Web Service (AWS) [54].

The `VagrantBackend` uses *Vagrant* to create a VM using the configured provider (*VirtualBox* is used by default) and then uses Docker in the VM to run the Python callables. This is why `VagrantBackend` is a subclass of `DockerBackend` as can be seen in figure 3.1 – the extra functionality is only there for launching the VM. Docker is used inside the VM to serve as a further separation of the environments and to remove the need to handle Python installation inside the VM – Docker usage is supported out of the box in *Vagrant* [55].

This backend is more of a proof-of-concept rather than a ready-to-use backend since the integration to Naucse did not require for it to work efficiently, but it does work and it does provide the extra layer of isolation.

3.3 Implementation

This section describes how the tool has been implemented. The full source code of the tool is included on the enclosed CD or can be viewed on its GitHub at `https://github.com/mikicz/arca`.

3.3.1 The Settings class

The `Settings` class facilitates a large portion of the configuration described in section 3.1.1. It implements a small wrapper around a Python dictionary which is initialized with the values provided by the programmer and by the contents of environment variables. The values from settings can be then retrieved in a simpler format (without the need for the prefix), plus the `get` method also enables passing multiple possible keys to get and the first encountered value is returned. The method can also return a default (if provided) when none of the keys is set. Individual configuration options of objects are lazily evaluated, the values are retrieved from settings only when needed.

3.3.2 The Arca class

As mentioned above, the `Arca` class serves as the main interface for the tool . There are three configuration options for the class:

base_dir The folder where Arca should store all its files, `.arca` by default.

single_pull If each repository/branch should be pulled only once, `False` by default, more in section 3.3.7.

ignore_cache_errors If cache initialization errors should be ignored, `False` by default, more in section 3.3.4.

Apart from arguments matching the names of configuration options (which are used to set the options directly) the constructor has two further arguments. **backend**, which can be also used to set backend directly (more in section 3.3.3), and **settings**, which is used to initialize the `Settings` instance.

3.3.3 Initializing the backend

After the settings are populated, the backend is initialized using the Arca's `get_backend_instance` method. It is shown in code example 3.3. First, if the backend was not provided directly to `Arca`, it's retrieved from settings. Then the backend value goes through conversions to get the instance – the value can be provided as a string or as a callable object (a class or a factory) or as an already initialized instance. Finally, the instance is checked if it is a subclass of the `BaseBackend` (section 3.3.10). This contradicts the typical Python duck typing, where the original class of an object does not matter if all the methods are the same, but I chose to use this check to fail fast, to make sure the instance will have the required methods.

```
def get_backend_instance(self, backend) -> BaseBackend:
    if backend is NOT_SET:
        backend = self.get_setting("backend", "arca.CurrentEnvironmentBackend")

    if isinstance(backend, str):
        backend = load_class(backend)

    if callable(backend):
        backend = backend()

    if not issubclass(type(backend), BaseBackend):
        raise ArcaMisconfigured(f"{type(backend)} is not an subclass of BaseBackend")

    return backend
```

Code example 3.3: `Arca`'s `get_backend_instance` method

After the backend instance is generated,

- it is assigned to the `backend` attribute of the `Arca` instance,
- the backend instance is injected with the `Arca` instance, so it can retrieve values from settings and access helper methods.

3.3.4 Initializating cache

The last thing done in the initialization of the `Arca` instance is setting up the cache. As mentioned in the section 3.1.2 about the design of caching, the `dogpile.cache` package is used.

A `dogpile.cache` region is configured in `Arca` using the `make_region` method. This method configures the region based on settings or returns the default, a region with the `null` backend. The region is configured using two keys, `cache_backend` which is a string representation of the backend entry point (loaded by `dogpile.cache`) and `cache_backend_arguments` which is either JavaScript Object Notation (JSON) string or a Python dictionary with the arguments for the backend.

Once the region is configured, a test write is made to ensure the cache works. If that or any of the region configuration fails, an exception is raised with the details of the problem. This exception can be explicitly silenced using the `Arca` configuration option **ignore_cache_errors**. In that case, a `null` region is used instead, if the configured cache fails.

3.3.5 Running the callables

The primary functionality of the `Arca` class is served by the `run` method. This method clones or updates the target repository, checks if the result is in the cache

and alternatively delegates the launch of the task to the configured backend. A sequence diagram of calling the method is shown in figure 3.2.

The method accepts the following arguments:

repo The URL of the repository.
branch The branch from the repository to use.
task The `Task` instance (more in section 3.3.8) with the definition of what should be called.
depth Optional argument for speeding up cloning, more in section 3.3.7.
reference Optional argument for speeding up cloning, more in section 3.3.7.

First, the **repo** argument is validated, if the URL is something `Arca` can clone (only local repositories or repositories accessible publicly over HTTP/S) – this is done by the `validate_repo_url` method. Then, the **depth** and **reference** arguments are validated (more in section 3.3.7).

If those arguments are all valid the repository is cloned or updated, based on if the repository was cloned previously. The `get_files` method which handles it returns a tuple of two instances, a `Repo` instance and a `Path` instance. The `Repo` class is from the package `gitpython` [56] and is a Python interface for interacting with the cloned Git repository. The `Path` class is from the standard Python library, the instance returned contains the path to where the repository was cloned.

Then the cache key is generated using the `Arca` `cache_key` method. The method is shown in code example 3.4. The key and the reasoning behind it is described in section 3.1.2, the implementation of the task hash is described in section 3.3.8).

```
def cache_key(self, repo: str, branch: str, task: Task, git_repo: Repo) -> str:
    """ Returns the key used for storing results in cache.
    """
    return "{repo}_{branch}_"
           "{hash}_{task}".format(repo=self.repo_id(repo),
                                  branch=branch,
                                  hash=self.current_git_hash(repo, branch, git_repo),
                                  task=task.hash)
```

Code example 3.4: Generating cache key for results

Then the region's `get_or_create` method is used to get the result. This method is passed a cache key and a callable which creates the result, it then does all the work. It checks the cache, creates the value if it is not present, saves the value in the cache and returns it. This process can be seen in figure 3.2.

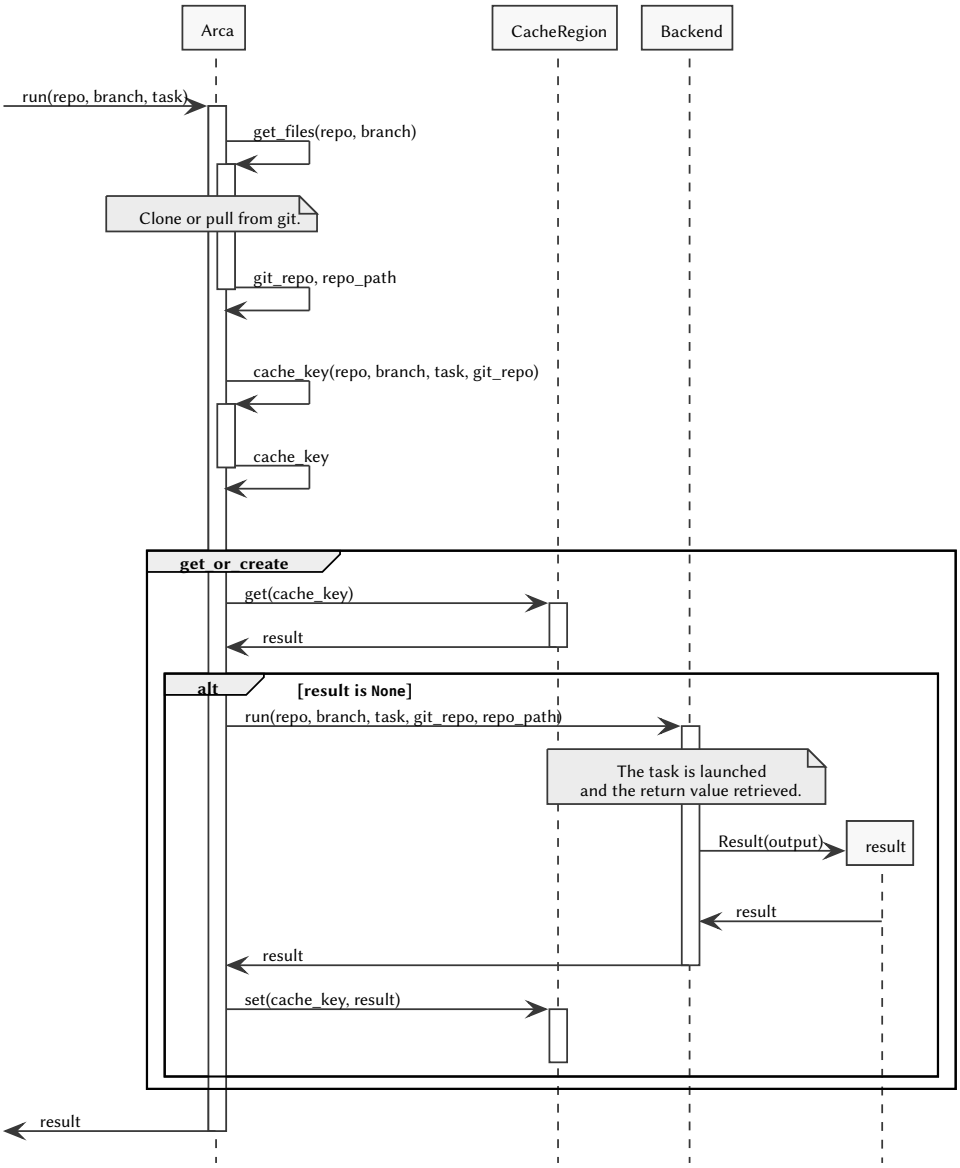

Figure 3.2: UML sequence diagram of the `Arca` `run` method

The value is returned encapsulated in a `Result` instance. More about the `Result` class is in section 3.3.9.

The `run` method indicates errors by raising exceptions. All the exceptions raised by `Arca` and its backends are subclassed from the `ArcaException` exception. If cloning or pulling of the branch fails, `PullError` is raised. If the callable cannot be imported or it raises an exception, `BuildError` is raised with the details of the problem. If the task takes too long and exceeds the configured timeout, `BuildTimeoutError` is

raised. Individual backends also raise their own exceptions, which are described in the sections about them.

3.3.6 Retrieving static files

According to the requirement AF5 the tool has to be able to retrieve files from the repositories. The `static_filename` method serves this purpose. The arguments are very similar to the `run` method, only **task** is replaced with **relative_path**. This argument is for the relative path to the file inside the repository.

Once the validation of **repo**, **depth** and **reference** is completed, the repository is cloned or updated in the same way as it was in the `run` method. Then an absolute path to where the static file is stored inside the **base_dir** directory is returned if the file exists. Otherwise, the `FileNotFoundError` exception is raised.

The absolute path is created by combining the paths to the repository and to the file like this: `result = repo_path / relative_path`. This is the standard functionality of the `Path` class and works on all platforms.

3.3.7 Cloning and updating repositories

Cloning and pulling of repositories is handled by the `get_files` method. Apart from the **repo** and **branch** arguments it also accepts two arguments mentioned before.

Depth argument

depth determines how deep the copy of the repository should go. The default is 1 since only the last version of the contents is important to the output of a callable. Setting the argument to `None` results in a clone with a complete history. This argument is only effective when the repository has not been cloned yet. The **depth** value is validated by the `validate_depth` method which checks that the value is either `None` or a positive integer (negative depth does not make sense).

Reference argument

Cloning some repositories can take a long time – for example, if they contain big files or they have a large history (and **depth** can't be used). If those repositories are forks of another repository which is already cloned to the computer, the **reference** argument can be set to a path to that repository and the repository is cloned using Git's `--reference-if-able` and `--dissociate` switches. The first switch

tries to take any common objects from the reference repository as opposed to downloading them from the remote repository. The second switch makes sure that the repository is made independent after the clone and that a deletion of the reference repository does not affect the clone. The **reference** value is validated by the `validate_reference` method which checks the value is either `None` or that it can be converted to a `Path` instance.

Actual cloning and updating

The Python package `gitpython` [56] is used in Arca to interact with Git – it provides a Pythonic wrapper around the command line tool. The initial clone is made using the static `Repo.clone_from` method. This method returns `Repo` instance for interacting with Git in that directory. If the repository was already cloned, the `Repo.init` static method is used to get the instance and then an update is performed. The repository is updated using fetch/reset approach – fetch updates from remote branch and then the current version is forcefully updated using reset. The `Repo` could simply perform a pull but that approach breaks when the remote branch is rebased.

The repositories are cloned to a `repos` folder in the directory defined by `Arca` configuration option **base_dir**. In that directory, there is a folder for each individual repository with a name generated by the `repo_id` method. This folder contains all the individual branches pulled from that repository. This approach has its benefits and drawbacks – the drawback is that it takes up more space and cloning can take longer. However, the benefits outweigh the drawbacks. Once the branch is cloned it doesn't have to be interacted with until update is requested which makes the feature described further much easier. To offset the increase in cloning time the reference functionality is used with already existing branches (if the reference is not provided by the user). This means that only the updates in that specific branch are downloaded compared to what was already cloned.

Single pull functionality

For some purposes, it does not make sense to update the repository every time before running a callable – for efficiency sake and for consistency sake. It could ruin a build of a website if the repository was updated in the middle of building it – the result would not be consistent. For this the **single_pull** configuration option is available. When set to `True`, `get_files` will clone or update each branch for each repository only once per initialization of the instance. This is done by saving the commit hash to an attribute of the instance `_current_hashes`. This dictionary is

checked on each call of `get_files` to see if the branch was already pulled. This pulling status can be reset by calling the `pull_again` method.

3.3.8 The Task class

The `Task` class serves for defining what should be launched in the isolated environment – what Python object should be called, where it is located in the repository, what arguments should be passed to the callable and the time limit for the execution. This could be defined directly in arguments of the `run` method, but it already has quite a few different arguments. This approach also enables the definition to be reused in multiple different repositories but defined only once and saves on some execution time by caching computed arguments in the instance.

The class is implemented to produce immutable objects – new instance has to be created for every new task. This is due to conversions being done in the `__init__` method and to enable easy caching of the task hash (for result caching) and of the definition JSON.

The constructor has four arguments:

entry_point The string representation of the callable in the repository. The string representation is validated by the package `entrypoints` [57], which is inspired by the `setuptools` `entry_points` [58] [59].

timeout A positive integer with the time limit for the executing in seconds. The default is 5 seconds.

args Positional arguments for the callable, strictly as a keyword argument for the constructor. It is optional and can be any iterable [60].

kwargs Keyword arguments for the callable, also strictly as a keyword argument for the constructor. It is also optional and it can be anything that is convertible to a dictionary.

The constructor validates the definition. The first validation is creating an `EntryPoint` instance from the string representation – `EntryPoint` is the main class of the package `entrypoints`. The second validation is a check that the provided value is not just a module but an actual object in some module. This is done by checking if the attribute `object_name` of the `EntryPoint` instance is not `None` – if an object was passed in the string representation, this attribute contains its name, otherwise, it is `None`. The timeout **timeout** is also validated, by converting it to `int` and checking the value is positive.

Then the arguments are checked. The positional arguments, if provided, are converted to a list. The keyword arguments, if provided, are converted to a dictionary and that dictionary is checked if all the keys are strings – only strings can be used for argument names. Finally, the JSON definition for the runner (more further) is generated to check if all arguments can be serialized. The JSON is not needed until much later in the process, but it doesn't make any sense to continue with the instance being invalid.

Hash

For the purposes of caching a unique hash of the `Task` instance is required to serve in the hash key. The generation of the hash is shown in code example 3.5. This code example only includes the three properties of the `Task` class which are involved. All three of these properties are cached using the `cached_property` decorator from the `cached-property` package [61], meaning the calculation is only done once per initialization – this saves some computing time.

The first property used is `serialized`. This property returns a dictionary with all the information specifying the Task – the callable used, the arguments and also the current version of Arca. The dictionary is then used by `json` – this property returns a string with a serialized JSON. Finally, the `hash` property returns a SHA256 [62] hash of the JSON.

```
class Task:
    ...

    @cached_property
    def hash(self):
        return hashlib.sha256(bytes(self.json, "utf-8")).hexdigest()

    @cached_property
    def json(self):
        return json.dumps(self.serialized)

    @cached_property
    def serialized(self):
        import arca
        return {
            "version": arca.__version__,
            "entry_point": {
                "module_name": self._entry_point.module_name,
                "object_name": self._entry_point.object_name
            },
            "args": self._args,
            "kwargs": self._kwargs
        }
```

Code example 3.5: Generating hash for `Task` instances

3.3.9 Runner and Result

All the backends were made to run on the same principle. While for example the `CurrentEnvironmentBackend` could be implemented in a different manner, by manipulating with the Python `sys.path` and importing the callables directly, a common approach enables more code overlap and simplifies code maintenance.

A Python script (runner) loads a JSON definition of task (generated by a `Task` instance) and prints out the returned value from the callable serialized in a JSON format. If the runner fails to load or execute the definition another JSON is printed with the information on what failed. The JSON output is then read by backend and deserialized and passed to the `Result` class. The `Result` `__init__` method checks if the runner succeeded and raises a `BuildError` exception if not. Otherwise, the `output` of the `Result` instance is populated with the returned value.

Runner

The runner script is common for all the current backends. It is implemented in the file `_runner.py` – the path to it is defined in the `RUNNER` attribute of the `BaseBackend` class, so it can be overridden by the subclasses.

The runner script includes mechanisms for loading the definition of the task from JSON, for validating the definition, for importing the callable and finally for executing it. It always prints a JSON result, even when exceptions occur. The dictionary printed can have four different keys:

success Always present, indicates if there as an error or not.
reason A specification of the kind of error which occurred – during loading of the definition, during the import of the callable or during the execution.
error The traceback of the exception raised if there was an error.
result The returned value from the callable if there were no errors.

3.3.10 BaseBackend

The `BaseBackend` is an abstract class that defines the interface individual backends have to follow – the `run` abstract method. The class also implements handling of configuration, declares some common configuration options and implements some functions used by all the backends.

The configuration is handled by multiple methods. The first being `__init__`. This method takes all the keyword arguments provided and overrides the configuration options of the class, the options which are not overridden are lazily evaluated once

needed. The `inject_arca` method is implemented, which is called once the backend is set in an `Arca` instance. This method sets the `_arca` attribute for the instance and calls the `validate_configuration` method – this method does nothing in this class, but can be overridden to validate configuration.

`BaseBackend` defines three configuration options which are common for all the backends:

requirements_location This configuration sets the location of a `requirements.txt` file in the repositories. This file is used to install packages using `pip`, the Python package manager. The default is simply `requirements.txt`, but it can also be set to `None` to indicated that requirements should be ignored completely.

requirements_timeout This configuration sets how long the installation of requirements can maximally take, in seconds. Must be set to an integer or something that can be converted to one and the default is 5 minutes.

cwd This configuration sets the relative path inside the repository where the tasks should be launched, the working directory. The default is the root of the repository.

Finally, three helper methods are implemented in this class, two related to the file with requirements and one to the task.

`get_requirements_file` returns an absolute `Path` for the requirements file based on the configuration option **requirements_location** in relation to the argument **path**. If the file does not exist inside of **path** or if the configuration option is disabled, `None` is returned instead.

`get_requirements_hash` reads the file provided in its argument **requirements_file** and returns the SHA256 hash of the contents.

`serialized_task` returns the filename and the contents of the JSON definition of the `Task` instance provided in the argument **task**.

3.3.11 BaseRunInSubprocessBackend

The `BaseRunInSubprocessBackend` is an abstract subclass of the class `BaseBackend` which implements the `run` method to launch the task in a subprocess (using the Python `subprocess` module). The Python used to launch the task is determined by the `get_or_create_environment` abstract method which must be implemented in the subclasses of this class.

After the Python executable path is determined by the new abstract method, the file with the JSON definition of the task is created using the `serialized_task` method. These two paths are then used by the `subprocess.Popen` class to launch the runner script as can be seen in code example 3.6. The instance `process` is used to read the output of the subprocess, which is passed to a new `Result` instance. The `Result` instance deserializes the output from JSON and determines if the task was successful, it not, a `BuildError` exception is raised. The working directory of the subprocess is determined by the **cwd** configuration option, combined with the path where the target repository is stored. The `communicate` method handles the timeout of the task, raising an exception if the execution takes too long.

```
process = subprocess.Popen([python_path,
                            str(self.RUNNER),
                            str(task_path.resolve())],
                           stdout=subprocess.PIPE,
                           stderr=subprocess.PIPE,
                           cwd=cwd)

out_stream, err_stream = process.communicate(timeout=task.timeout)
out_output = out_stream.decode("utf-8")

return Result(out_output)
```

Code example 3.6: Launching a subprocess in `BaseRunInSubprocessBackend`

3.3.12 CurrentEnvironmentBackend

`CurrentEnvironmentBackend` is a subclass of `BaseRunInSubprocessBackend`. It uses the Python executable used to launch Arca to run the tasks – its implementation of `get_or_create_environment` always returns `sys.executable`, which contains the path to the current Python executable.

The backend can also handle the requirements in the repositories, based on the configuration options this class defines:

requirements_strategy Which strategy the backend should take to differences in requirements of the current environment and of the repository. Can be either keys from the enum class `RequirementsStrategy` or their values. Individual strategies are described further.

current_environment_requirements Defines the path to the requirements of the current repository, can be set to `None` if there are none.

The bases of all the strategies are the differences in requirements, more specifically, the extra requirements of the repository. Only the extra requirements from the

repositories are considered because if a repository, let us say Naucse, was launching its own old fork, but there was new functionality implemented since the branching that required some new packages, there would be a difference in each repository. But only considering extra requirements from the fork, only the changes made in the forks are checked.

The difference is determined by creating a set of lines from both files (if they exist, an empty set is used otherwise) and subtracting the set of requirements of the current environment from the set of requirements from the repository.

The strategies are:

raise Raises an exception if there are extra requirements (the default value). Based on the principle "*Errors should never pass silently*" from the Zen of Python [37].

ignore Ignores all extra requirements. Based on the principle "*Unless explicitly silenced*" from the Zen of Python [37].

install_extra All the extra requirements from the repositories are installed.

Before returning the Python executable in the `get_or_create_environment` method, `handle_requirements` is called first to check the differences in requirements. When the ignore strategy is configured, the method ends right away. When there are extra requirements, either an exception is thrown or the `install_requirements` method is called, which install the extra requirements using `pip` in a subprocess. The timeout of installing requirements is handled in this method, by passing the **timeout** argument to the `communicate` method of the created process, just like in launching of tasks in `BaseRunInSubprocessBackend`.

3.3.13 VenvBackend

`VenvBackend` is also a subclass of `BaseRunInSubprocessBackend`. This backend using Python virtual environments [30] (further also as virtualenvs) executables to launch tasks. The virtual environments are shared between repositories which share the identical requirements, meaning that one repository could disrupt the functionality of the virtualenv for other repositories. This backend claims no actual isolation or being secure, so the fact that different repositories can affect a common virtualenv is not significant.

The implementation of the `get_or_create_environment` method in this class calls the `get_or_create_venv` method, which returns the path to the virtualenv for the current repository, and returns the path to the Python executable in that virtualenv. The `get_or_create_venv` method generates a name for the requirement

(either SHA256 hash of the requirements file or `no_requirements_file`), checks if it already exists and creates it if it does not.

The virtual environments are created using the `EnvBuilder` class from Python's `venv` module. The requirements are installed using `pip` in a subprocess, just like in `CurrentEnvironmentBackend`, but only if the virtualenv does not exist already. The timeout of installing requirements is also handled in the call to the `communicate` method, just like in launching of tasks in the `BaseRunInSubprocessBackend` class.

3.3.14 DockerBackend

`DockerBackend` is a subclass of `BaseBackend` and utilizes *Docker* [34] *containers* to provide isolation around Python.

Containers are "*lightweight and portable encapsulations of an environment in which to run applications*" [63]. Containers are isolated from the operating system, however, they do share the kernel and system resources. From the point of the application inside the container, it seems like it is running on a full-fledged computer.

Containers are started based on *images*, "templates" for containers. An image is a file, basically an immutable snapshot of a container, which can be distributed across computers. Images are not modified when a container started based on it modifies something, the change only affects the container itself, not even other containers launched from the same image.

The process of running a task using this backend is approximately this:

1. Checking that Docker can be accessed.
2. Determining the name of the container.
3. Checking if the container with that name is running, starting it if it does not. After a container is started, the repository and the runner script is copied inside it.
4. Copying the task definition file to the container.
5. Launching the runner script in the container with the location of the definition file as an argument.
6. Stopping the container, if **keep_container_running** (more about that later) is not set.
7. Returning the output or raising an exception if anything failed.

The backend uses the `docker-py` [64] package to communicate with Docker. This package provides a Python interface for all the operations required by the backend. The check that Docker is accessible is made through the `check_docker_access` method. The method creates a client (an object from the package for communication) for the backend's instance and the calls its `ping` method, which raises an exception if Docker cannot be accessed. Docker needs to be accessible by the user that is launching Arca, meaning they either have to be root or have sufficient permissions.

The name of the container serves a similar purpose to the cache key in Arca's `run` method. To enable effective reuse of containers, the container name has to identify for what it is supposed to be used. The name is determined by the `get_container_name` method, which creates a name based on the URL of the repository, the target branch name and the current commit.

Checking if a container with the generated name is running is done using the `container_running` method, starting one using `start_container`. Before starting a container, an image must be selected. That is done using the `get_image_for_repo` method – the images themselves are described further. When an image is selected, a container is started using the client, with the correct working directory. The client returns a `Container` instance from the package, which is then used for communicating with it. Then the repository and the runner script are copied over to the container, using the `Container`'s `put_archive` method. The files have to be copied over in a tar archive – the archives are created using Python's `tarfile` module.

The task definition JSON is copied over to the container using the same methods.

Tasks are launched in the container using the container's `exec_run` method. Unfortunately, it does not provide a timeout option, so the limit is enforced by the Linux timeout command [65]. The method returns a status code of the command and the output, the code can be checked if it is the 124 timeout returns (or 143 for Alpine). The output is otherwise returned in a `Result` instance.

The container is stopped in a `finally` block if **keep_container_running** is not set. Otherwise, the used container is added to a set of containers used by the instance, so they can be stopped later by calling the `stop_containers` method.

Images

Containers can be started by this backend from two basic types of base images, either an image configured directly by the programmer or the default, custom built image made specifically for Arca. The base image is then extended by installing extra dependencies or Python requirements. The timeout of installing requirements is enforced, just like the timeout of tasks, by the Linux `timeout` command.

A specific image to use can be configured using the **inherit_image** option. The backend does not check anything about this image, so the programmer has to know what they are doing. The image has to have a Python and `pip` executable that can be launched by the default user. Setting a specific image disables two other configuration options, **python_version** and **apt_dependencies**. The first specifies what Python version should be used since the image is determined by the programmer, the option cannot be enforced. The second specifies what extra dependencies should be installed since the backend cannot know which system the image is running, it cannot determine how to install dependencies.

The default image is based on Debian Stretch [66], in the slim version – a cut-down version, which is smaller and does not include things not needed in containers. In the first version of Arca, Alpine [67] was used to save space, however, it was replaced with Debian because Alpine does not support installing wheels [68]. Wheels will be mentioned later in the section about releasing Arca, but practically it means that all python packages have to be built or compiled during installation, which takes a lot of time.

The default image is built in several stages to maximise reusability:

1. a base image with all the backend's dependencies installed,
2. an image with Python installed,
3. optionally an image with the extra dependencies (set by **apt_dependencies**)
4. and finally an image with the requirements installed.

The `pyenv` [69] tool is used to manage Python. The slim version of Debian does not have a Python installed by default, plus I wanted to have precise control of what Python version is installed. `pyenv` enables installing specific versions of CPython including the patch version, and even other implementations of Python, like PyPy or Anaconda. The specific Python version used can be configured using the **python_version** option, by default the version of the Python used to launch Arca is selected.

Following recommendations mentioned in section 3.2.2, the root user is only used to installing system dependencies, otherwise the `arca` user is used – for installing Python, for installing requirements and for running Python.

To save time the first two stages are pre-built and pushed to a Docker *registry*, from where they can be pulled by anyone. A Docker registry is a server where images can be pushed and published for usage by other people. The registry is available at `https://hub.docker.com/r/mikicz/arca/`.

The images are pushed there automatically using the CI tool *Travis CI*, which is described in detail later. On pushes to the master branch, after all tests pass, a script is launched which lists all version of Python that are installable by `pyenv` and supported by Arca, checks if those versions are already pushed to the registry and builds images for those which are not. The backend can be configured to ignore the images on the registry by the **disable_pull** option, forcing the backend to build all images locally. This is used mostly for testing.

By default, the stages with extra dependencies installed and requirements have to be built locally. The option **use_registry_name** can be set to use a registry for these images, so they do not have to be rebuilt repeatedly on different computers or in different builds on CI tools. When an image is needed, the backend then first checks the repository if the image has already been built and pulls it. After a new image is built, it is pushed to the registry to save for further usage. To push to a registry, being logged in to docker is required, but that has to be handled by the programmer.

Push is disabled when the option **registry_pull_only** is set. This is needed for pull requests – when running CI tools on pull requests, authentication tokens and other secret keys are not available to prevent them leaking [70], so login to docker is not available. But if the images were built elsewhere, for example in the `master` branch, they can still be reused in pull requests.

Configuration options

Following is the summary of configuration options available for the backend:

use_registry_name The registry where images should be pulled from and pushed to, `None` by default.

registry_pull_only Only allow pulls from the registry, `False` by default.

keep_container_running Prevents containers from being stopped right away after a task is finished, to save time on booting up containers, `False` by default.

disable_pull Disables pulling of the default base images from the registry, `False` by default.

inherit_image Defines what image that should be used instead of the default base image, `None` by default.

python_version The specific Python version that should be used, `None` by default, which sets the version of the Python used to launch Arca.

apt_dependencies A list of system dependencies that will be installed using `apt-get`, `None` by default.

3.3.15 VagrantBackend

`VagrantBackend` is a subclass of `DockerBackend`. It utilizes *Vagrant* to launch a full Virtual Machine to isolate the environment.

The backend is a subclass of `DockerBackend` because Docker is used inside the VM to facilitate further separation of data and environments. It also eradicates the need to install Python directly to the VM, meaning it does not matter what operating system is used inside the VM, as long as Docker is supported.

As stated in the section 3.2.3, this backend is a proof-of-concept, since it was not needed in the Naucse integration. The biggest problem to solve to make it ready is how the images with Python get to the VM. The easiest solution in relation to the backend's subclassing of `DockerBackend` was to build the images in the current environment, push them to a registry and then pull them in the VM. This obviously is quite inefficient, because to get the image to another location inside the same computer it has to make a two-way trip to the cloud and back.

The process of running a task using this backend is approximately this:

1. Checking that the VM is running, launching it otherwise.
2. Checking that Docker can be accessed.
3. Making sure that the Docker image for the certain requirements exists, building and pushing to the registry it otherwise.
4. Determining the name of the container.
5. Connecting to the VM, pulling the image and executing the task.
6. Shutting down the VM if **keep_vm_running** is not set.
7. Deleting the whole VM if **destroy** is set.

The `python-vagrant` [71] package is used to manage the VM. One virtual machine is used per `Arca`'s **base_dir**, so practically for each project. Vagrant uses a file called `Vagrantfile` for the definition of a VM – the backend creates this file based on the

configuration, and `python-vagrant` handles the rest, launching the VM, stopping it and destroying it.

Vagrant enables sharing files with the VM using the option *synced_folder* [72], the backend configures the VM to share two locations. The location of the `Vagrantfile`, where the runner script and task definitions are stored, and the folder containing cloned repositories, so they can be copied to the containers running inside the VM.

Checking that Docker can be accessed, that the image exists and determining the container name is all done using methods from `DockerBackend`.

The `fabric3` [73] (a port of `fabric` to Python 3) package is used for communicating with the VM over Secure Shell (SSH). `farbic3` provides an API for executing commands over SSH, using *tasks* (different from `Arca`'s tasks). The `fabric3` task is defined in the `fabric_task` property. The task pulls the image from the registry, starts up the container if it is not already running and then executes the tasks. Timeout is implemented in this backend using an argument of the `fabric3`'s function used to run the commands over SSH.

By default, after the task is finished executing the VM is shut down. Similarly to `DockerBackend`, this backend provides an option to keep the VM running to save time on booting it up before each task and adds a `stop_vm` method which stops the VM. There are two different ways of shutting down the VM, one being only stopping it and the other is stopping it and then deleting it completely. By default, it is only stopped, but the backend can be configured using the option **destroy** to delete it.

Configuration options

The backend inherits options of `DockerBackend`, with **keep_containers_running** being `True` by default.

box The environment that should be used, very similiar to images in Docker. `ailispaw/barge` [74] is used by default, a lightweight OS built specifically to run Docker containers. Any box can be used as long as it has Docker installed with version greater than or equal to 1.8, or not installed at all – Vagrant can install it [55].

provider The underlying technology used to start the VM, `virtualbox` by default.

quiet Sets how verbose the logs should be, `True` by default.

keep_vm_running As described above, should the VM be kept running until explicitly stopped? `False` by default.

destroy Should the VM be deleted once stopped? `False` by default.

3.3.16 Logging

To relay information on what is going on inside Arca, standard library Python logging [75] is used. The logging is done through a Python *logger* named `arca`. This logger is not configured to store the logs anywhere, but projects using the tool can set up the logger to store logs using whatever handler they wish.

Most of the logs is where individual files are stored, the generated hashes of some stuff, what had to be downloaded or built locally. Debug outputs usually include the status codes of processes and their raw outputs. The exceptions raised during launches of tasks are logged under the level `exception`.

3.3.17 Other miscellaneous functions

The `arca.utils` module contains a couple of miscellaneous functions that can be used by the programmer using the package. All of them are shortcuts to functionality of the `Repo` class from package `gitpython`. The usage of some of these functions is described later in chapter 4.

The first is `is_dirty` with argument **repo**. The function returns `True` if the local repository represented by **repo** has been modified or if there are some untracked files in it.

The second function is `get_last_commit_modifying_files` with an argument **repo**. The function accepts an unlimited number of positional arguments, paths relative to the root of the repository to files or folders. This function returns the hash of the newest commit which modified one of these files or folders in the repository.

The third function is `get_hash_for_file` with arguments **repo** and **path**. As the name suggests, the function returns the Git hash of the file in the repository, specifically the tree hash for folders and the blob hash for files.

3.4 Testing and Continuous Integration

One of the requirements for the tool states it has to be well tested. Arca is tested using unit and integrations tests with the help of `pytest` [76]. `pytest` is used

because it makes it easy to create effective tests and provides some extra utils not included in the standard Python testing modules.

The tests are defined in the `tests` folder in the base of the repository. This directory contains multiple different files that contain individual tests.

The file `conftest.py` defines `pytest` fixtures, resources that can be used in all other tests. This file is loaded automatically by `pytest` [77]. There are two fixtures defined in the file, `temp_repo_func` which creates a temporary repository containing a file `test_file.py` for launching tasks and `temp_repo_static` containing a text file for testing of retrieving static files. The repositories are created in a temporary directory using the Python module `tempfile` and are deleted once they are no longer needed.

The file `common.py` defines constants used in tests and the contents of files in test repositories. The rest of the files contains individual tests. These tests are written in the form of functions which test individual parts of Arca and their integration in the whole tool. Some of these functions are parameterized, meaning that they are launched multiple times by `pytest` with different arguments [78] – to test more cases and different configuration options.

Following is a list of individual files with the description of the test they contain:

test_arca_class.py Contains tests for functionality around cloning and pulling repositories, validation of arguments and of requesting paths for static files.

test_backends.py Contains parameterized tests for the basic functionality of backends (except for the Vagrant one, which is tested separately). The tests check tasks are performed and that they return a correct output, that the backends can handle requirements and that the backends can handle unicode correctly and that they did not install the requirements to the current environment. The test also checks that an exception is raised in case the timeout is exceeded, both for requirements and execution of tasks.

test_cache.py Contains tests for the initialization of cache and its functionality.

test_current_environment.py Contains tests for individual strategies for installing requirements.

test_docker.py Contains tests for individual configuration options in the Docker backend.

test_result.py Contains tests of validation in `Result`.

test_runner.py Contains tests of the runner, loading definition file and execution, unicode handling.

test_settings.py Contains tests of the `Settings` class and its integration into `Arca`.

test_single_pull.py Contains tests for the `single_pull` configuration option and the pull efficiency in general.

test_task.py Contains tests of the validation in `Task` and its serialization.

test_utils.py Contains tests of the functions described in section 3.3.17.

test_vagrant.py Contains tests of validation in Vagrant and the execution in the backend. This backend is not included in the `test_backends.py` tests, because the tests would take too much time, a shortened version is used.

Arca also uses two tools to analyse code statically – without any code execution. The first is `flake8` [79] which checks that Arca's code is compliant with PEP8 [80], the recommended style guide for writing Python code (with the max line length extended to 120 from the original 80). The second tool used is `mypy` [81], a static type checker for Python. The type check is optional, it is more like an early warning system.

Arca uses *Travis CI* as its CI tool, I selected it to match the tool used in Naucse. Builds are triggered for all updates in the `master` branch and for all pull requests. The builds launch the tests including the static analysis. All tests for the `VenvBackend` and `VagrantBackend` are skipped. *Vagrant* is not supported on *Travis CI* [82] and virtual environments are broken. The details of why virtualenvs do not work properly are described in the issue #8589 I reported to *Travis CI*, available at `https://github.com/travis-ci/travis-ci/issues/8589`.

3.5 Documentation

There are several levels of documentation for this tool and its code.

3.5.1 Docstrings

First, there are docstrings – documentation of individual classes, methods, functions and attributes. These, with the help of typing annotations, help programmers to determine what each Python object does, what is it used for and what arguments it uses. The docstrings can be displayed either directly in the source code, using the `help` function in an interpreter or they are also included in the full documentation. The docstrings are written in `reStructuredText` [83], the standard format for writing docstrings in Python [84].

3.5.2 README

The second level of documentation is a README file located in the root of the repository. This file is automatically displayed by GitHub when the repository is visited and it is also used on Python Packaging Index (PyPI).

This file contains the basic description of the tool, installation instructions, a quick example and links to the full documentation. README is also written in `reStructuredText`, this format allows for the README to be processed into a nice HTML output both on GitHub and PyPI.

3.5.3 Full documentation

The final level is a full documentation, with a quick start and proper guide to using and configuring the tool. The documentation also includes an API Reference, an overview of classes and their methods with docstrings, linked with documentations of packages used in the tool.

This documentation is also written in `reStructuredText` and uses `Sphinx` [85] to create the output. The primary output is HTML, but the tool can also export the documentation in Portable Document Format (PDF) or in ePub or other formats [86].

The entire documentation is in the `docs` folder. The primary file of the documentation is `conf.py`, originally generated by `Sphinx`, which contains basic metadata and the configuration. The file defines the name of the documentation, the author, the format the documentation is written in, the skin used to render the HTML and other minor details which are mostly generated automatically. The folder also includes a `Makefile`, which provides some shortcuts for building the documentation. The rest of the folder are `rst` files with the contents of the documentation.

Distributing the documentation

Documentation, meaning the one written in Sphinx, is not usually distributed with Python packages, it must be distributed separately. There are two major ways of distributing the build documentation (converted into HTML), one being self-hosting and the other using a dedicated service that hosts documentation primarily. The documentation site created by `Sphinx` is a completely static HTML page, so self-hosting it is not hard, the trouble comes when the documentation is updated. Since a part of the documentation is generated automatically from the source code (the API reference) it changes regularly, when self-hosting, a manual update would be

required after each update. The dedicated services solve this problem by updating the documentation after each update of the repository.

Arca uses *Read The Docs* [87] for documentation hosting – the documentation is available at `https://arca.readthedocs.io/`. *Read The Docs* is made specifically for hosting projects that use `Sphinx`, the service automatically builds the documentation once a new commit is pushed to the configured branch.

There was one small issue while setting up the documentation on *Read The Docs*. Arca is written in Python 3.6, which introduced a couple of breaking changes to the syntax, so the builder had to be configured using `.readthedocs.yml` to use Python 3.6 [88].

3.6 Releasing

There are multiple ways of distributing Python code.

One of them is simply publishing the source codes and leaving it to the people wanting to use them to handle the installation, but this will only discourage people from using the code – even with instructions the installation is too much of a hassle.

The recommended way [89] is packaging the code using using `setuptools` [90]. This enables for the code to be installed in different environments using the `pip` or `easy_install` tools.

Once the code is packaged using `setuptools` it can be installed to other environments by referencing the folder where it is on the computer or by referencing the Git repository where the code is stored. Installing from a local folder is still very impractical – distribution of the code still has to be handled, installing from a Git repository is better, but still complicated. Repositories can change ownership or names, can be deleted or else corrupted, plus the command for installing from Git is hard to remember.

I decided to go one step further with the tool, uploading the package to PyPI, the official package repository for Python, under the name `arca`. By uploading the package there, it can be installed simply by running `python -m pip install arca`.

This section describes the process of packaging the tool and uploading the package to PyPI.

3.6.1 Packaging

The code gets packaged using `setuptools` by creating a file (usually called `setup.py`) which contains a call to the `setuptools` function `setup`. This function call only has to specify name, version and packages to include, but can and should contain much more, for example, metadata about the author, description of the package, the license, the dependencies of the package.

Once the `setup` function call is defined, the file can be executed from the command line with commands that specify what action should be performed. Some of the relevant commands:

install Installs the package to the current environment [91].
sdist Creates an archive installable using the `install` command [92].
bdist_wheel Creates an archive that can be installed just by moving the contents to the right place. This archive is called a *wheel* and its installation is much faster because the archive is just unpacked and no code has to be executed. An extra package `wheel` needs to be installed for this command to work [92].

The `sdist` and `bdist_wheel` commands generate the archives in the `dist` folder [92] and are then uploaded to PyPI – as shown further.

Arca's setup.py

Apart from the usual options like name, author, keywords or descriptions, the Arca's `setup.py` also defines some functionality which is not so common in other packages.

The first is optional requirements. The backends `DockerBackend` and `VagrantBackend` require extra requirements (as described in the sections about them), which also have more requirements. For people not using those backends, the extra dependencies only take up space and slow down installation – the extra time can be considerate on slower machines or on CI tools. By using the configuration option `extras_require` [93], the package can define optional requirements under a specific keyword. Arca uses this option for Docker and Vagrant backends as shown in code example 3.7. The package can be then installed with the extras by running `pip install arca[docker]`.

The package `pytest-runner` [94] is used for launching tests. After the package is added to `setup_requires` a new command `pytest` is added to the Command Line Interface (CLI) of `setup.py`. The argument `tests_require` contains all the dependencies for the tests. To use the standard command `test` an alias is configured

```
setup(
    ...,
    extras_require={
        "docker": [
            "docker~=3.2.1",
        ],
        "vagrant": [
            "docker~=3.2.1",
            "python-vagrant",
            "fabric3",
        ],
    },
    ...
)
```

Code example 3.7: Usage of extras_require in Arca's setup.py

for the `pytest` command in `setup.cfg`, the default options for `pytest` are defined in `pytest.ini` and are described more in section 3.4 [95].

Finally, a custom CLI command `deploy_docker_bases` is defined by the configuration option `cmdclass` [96]. The command builds and pushes Docker images to the registry, as is described in section 3.3.14.

3.6.2 Uploading to PyPI

The final step is uploading the package to PyPI. An account is required to upload packages to PyPI, registration is available at its main page.

Then the distribution archives are required – they are generated by calling the commands `sdist` and `bdist_wheel` (as described above). To upload these archives to PyPI, the package `twine` [97] is required. Its usage [97] is shown in code example 3.8, username and password are automatically prompted when the command is called.

```
twine upload --repository-url https://upload.pypi.org/legacy/ dist/*
```

Code example 3.8: Uploading packages to PyPI

After the upload command is finished the package is available immediately on PyPI at `https://pypi.org/project/arca`.

Chapter 4

Integration into Naucse

This chapter contains the details of the integration of Arca into Naucse.

I submitted the changes in a series of pull requests to the GitHub repository of Naucse, all of them were approved and merged by the maintainers. There were three preparatory pull requests (#347, #348, #349[†]) with changes that could be submitted in advance to make the set of changes in the main pull request, #349, more manageable – they helped, but not massively. Then there were four follow-up pull requests (#385, #389, #393, #405) which fixed issues not discovered during testing or review and updated the used version of Arca. Finally, the metacourse with Czech instructions on how to add a run was submitted in pull request #394.

Ideally, there would have been more pull requests, the main one changed 3,500 lines, but the changes were all so interconnected that they could not have been reasonably separated into smaller chunks.

4.1 Changes of content rendering

The way some parts of the website were rendered had to be changed to make the features required by the assigned solution possible or easier. This section will describe them and the reasons why they were needed.

First, let us establish some terminology – it will differ slightly from the one used in the analysis to closer match the terms used in the rendering code. A *course* is still either a *canonical course* or a *run* – comes from the class used for both of them, `Course`. A course is comprised out of one or more *sessions*, groupings of several

[†]The pull requests can be accessed by replacing the number of the PR in the following link: `https://github.com/pyvec/naucse.python.cz/pull/347`.

lessons together. A session is defined by a name, an identifier and the list of lessons, for runs also the place, date and time are added. A lesson is a text about a certain topic, consisting of a main page and subpages. The text can be extended with static files, like images or prepared files. Lesson are organised in *collections*, groups of lesson related to each other.

4.1.1 Static files not shared between courses

Originally, the static files of a lesson were shared among all courses that used it, each file was available from one single URL inside the `Flask` application. Now they are available at a URL specific to the course that is using it, so courses in arbitrary branches can change the contents of these static files or define new ones.

This was done by extending the `lesson_static` route with an optional argument `course`, adding another decorator with a template for the URL and modifying the function that is used to get the URLs to the static files to include the course.

4.1.2 Separated rendering of content from the whole page

Previously, the `Flask` application rendered the whole page (headers, footers, menu etc.) together with the content of the specific URL requested. That was changed, the content is first rendered from a separate template for the specific URL, and the result is then used in rendering the whole page with the footers and headers. This was done to mimic how content from arbitrary branches is rendered, where just the content is rendered first in the branch and then it is put to the template for the whole page – more about that further.

There is a new folder in the `templates` folder, where all the templates for the `Flask` app are, `content`. This folder contains individual templates for the insides of pages – for the overview of a course, the front page of a session, the back page of a session and a course calendar. The content rendered from these templates, plus the content rendered from the sources of lessons, is placed into templates located directly in `templates`. Different pages also have different templates here, because the content is displayed a bit differently for each page. But all of those templates inherit from a new `_base_course.html` template, which sets up common elements of the page – breadcrumbs, a block for warning messages and the footer. This template still inherits from the old `_base.html` template.

4.1.3 Relative URLs inside lesson contents

Originally all URLs inside the contents of a lesson (texts converted from `Markdown` or `Jupyter Notebooks`) were absolute – they included the full path, including the course they were in. This had to be changed because the rendered contents then could not be shared between multiple courses.

The change is primarily implemented using the `naucse.routes.get_relative_url` function which accepts two arguments, the current URL and the target URL and generates a relative path to the target URL. This function is then used everywhere where URLs are generated for links inside the lesson content.

4.1.4 Dynamic link to edit page

Naucse includes a link at the bottom of every page that leads either to the GitHub repository or if the page that has a definitive source in the repository (like the definition file of a course or the source material for a rendered lesson page), a link to the file on the master branch. This URL was based on a hardcoded function which always returned a URL to the base repository, to the master branch. The URL was passed to each rendered template where it was put inside a link predefined text with a tiny icon.

This mechanism was updated to enable having links to other arbitrary branches. Instead of the URL, a dictionary is passed to the templates with an URL, a name of the service where the link is pointing to (GitHub by default) and the icon. This dictionary is generated by the `naucse.utils.routes.get_edit_info` function based on the relative path in the repository to the target file.

The URL is generated by the `naucse.utils.routes.edit_link` function. This function, instead of having hardcoded values, returns the URL based on the current git environment. A new `MetaInfo` class is used. It provides two dynamically generated attributes which:

- return the slug and branch of the current build if Naucse is running on Travis,
- return the slug and branch of the current environment using the `gitpython` and `giturlparse` [98] packages or
- return the default slug and branch, which are pointing to the base repository and branch.

4.1.5 Dropped Python 3.5 support

Arca was written in Python 3.6, using its backwards-incompatible features like f-strings, so the support for Python 3.5 had to be removed. This was done by removing a metaclass that was providing the compatibility with Python 3.5 and updating the README file to require 3.6.

4.2 Content from arbitrary branches

Rendering content from arbitrary branches has two major components. The first is the "client" – the Naucse instance that is rendering the whole page. The second is the "server" – the code from arbitrary branches that is launched using Arca.

Naucse was modified in such a way that courses (both canonical courses and runs) can be defined by a `link.yml` file as well by `info.yml`. When Naucse encounters this file, it loads it to a different *model* (a class for a specific kind of data) and uses that model differently. While courses rendered from the same repository are loaded to a `Course` model, courses rendered from arbitrary branches are loaded to a `CourseLink` model. This goes back to the directory structure 1.1 on page 25 – the `link.yml` is still either in `courses` or `runs` under a identifier.

This is done in `MultipleModelDirProperty`, a new subclass of `DirProperty`. While the original property could only return instances of one model class, the new subclass can return multiple models, based on the file that is located in the folders. The original was initialized with a model and a path where to search for instances, the new subclass is initialized with a list of models and the path. The list also determines preference – if there are multiple definition files in the same directory, the model that is first in the list is returned. The definition file is set by the class attribute `data_filename`.

The new `CourseLink` is mostly compatible with `Course` in the basic attributes, like the title or description. The difference is in the way these attributes are populated. `Course` uses `YamlProperty`, a lazy-evaluated way to load the attributes from the file `info.yml`. `CourseLink` uses `YamlProperty` too, but only to load the repository URL and the branch name of the target arbitrary branch. A new `ForkProperty` class is then used to load the attributes using Arca.

The `ForkProperty` creates a `Task` (the class from Arca) instance using the arguments it gets – the target callable and arguments. The task is then executed using Arca in the arbitrary branch. The callable used to get attributes about the course

is `course_info` from `naucse.utils.forks`. The function is present in the base repository as well, but quite confusingly, it is not executed elsewhere than in the arbitrary branches. The repository can change the function to do something else, but let us assume it will work – error handling will be explained later.

This was all on the "client" side, now about the "server" side. The target arbitrary branch has to contain a `info.yml` in the same exact folder where `link.yml` was created on the "client" side. The `course_info` function from `naucse.utils.forks` loads the `Course` instance and returns a dictionary of serialized attributes of the instance which are important for the type of course – more information is returned about runs then about canonical courses.

The `CourseLink`'s compatibility with `Course` is first utilized in lists of courses, both of canonical courses and of runs. The courses from arbitrary branches are displayed there next to courses rendered from the base branch.

4.2.1 Pages related to a specific course

Another perk of `CourseLink`'s compatibility is that the routes related to courses work even for courses from arbitrary branches.

Here comes in handy the change described in section 4.1.2. The routes were modified in such a way that if the course is rendered from an arbitrary branch, the inside of the page is rendered in the branch, and if not, it is rendered as previously, from a template inside `templates/content`. The content, whether it was rendered locally or in a branch, is then put inside a common template.

The "client" side of rendering content from a branch is done using `CourseLink`'s methods, there is one for each type of page that can be served. These methods create a `Task` instance to render that specific page requested and execute it using Arca. The result is then returned to the route and the route fills the whole page including headers and footers with it.

The "server" side is managed by the `render` function, in the `naucse.utils.forks` module. This function returns the rendered contents of the pages and some extra meta-data, like titles, licenses or the dictionary about where the source code described in section 4.1.4.

4.2.2 Gathering URLs to freeze

`elsa`, the package that is handling the process of *freezing* the dynamic page into a static one, only extends another package, `Frozen-Flask`[99]. This package works

by hooking into the `Flask` mechanism for generating URLs to gather what pages should be rendered and frozen. The freezing does not work when some of the links are not generated using the standard `Flask` `url_for` function that returns an URL to a certain route with some arguments – like when those links are generated in a different process completely or when the content containing the links is retrieved from the cache.

This was solved in two parts. The function on the "server" side uses the same mechanisms as `Frozen-Flask` does, and collects all the URLs that were generated in it. These URLs are returned with the content of the page and metadata. The "client" side then extends the mechanism for generating the URLs to freeze by also returning URLs returned from arbitrary branches.

4.3 Fragment caching

The requirements for the integration say that individual content fragments should be shared across arbitrary branches, the assignment also says that arbitrary branches that share the identical code can also share the cache of fragments.

The fragment caching was implemented to cache the contents of lessons after being turned into HTML from the source materials. It does not make sense to cache other pages like the overview of the course, the session cover pages or the calendar, because they always depend on the specific course. These other pages are cached only for arbitrary branches, by the Arca caching mechanisms, which caches whole results of calls to the `render` function.

The Arca's caching region is reused here for caching of the fragments – it would not make sense to have two different caches inside one system.

4.3.1 The key

The key to the cache is generated by the `page_content_cache_key` function from the `naucse.utils.routes` module, which generates a key based on the following (described in detail further):

- the Git tree hash of the rendering code,
- the specific page,
- the course variables,
- the Git tree hash of the lesson.

The Git tree hash is a hash of a directory inside a repository, including all its contents. Meaning if two directories have the exact same contents in two repositories, even if they came to it in different ways, their tree hash is the same. These tree hashes are retrieved by two functions in `naucse.utils.routes`, `get_naucse_tree_hash` and `get_lesson_tree_hash`, which both utilize the Arca's `get_hash_for_file` util.

A page that should be rendered is defined by the identifier of the lesson and its specific subpage.

The course variables are mechanisms inside of Naucse which can modify the content slightly. These variables are defined by a course and apply to all its lessons, so only courses that share variables can share content fragments.

4.3.2 Value

The values in this fragment cache are dictionaries that contain two values. The rendered HTML fragment and a list of relative URLs used in the fragment. The relative URLs are included, so when the fragment is used from the cache, the absolute URLs can be generated and frozen. The links still work when Naucse is launched in the "serve" mode (section 1.1.2 on page 26), but when the "freeze" mode is used the URLs are not frozen because they were not generated using `url_for`.

The relative URLs are retrieved again by the mechanism used by `Frozen-Flask`. A temporary hook is inserted into the `Flask` app that records the requested URLs, which are then turned into relative ones by the `get_relative_url` function.

4.3.3 Usage

The cache is used in two locations because there are two ways how a lesson can be rendered.

The first location is the `naucse.routes.page_content` function which handles rendering lesson content from the current branch. The function only uses the cache if the **without_cache** is not set (more further) and the current repository is not in a dirty state. A dirty state of a repository is such a state where files were modified or added but not committed – when somebody is modifying a lesson, the lessons should be rendered directly, not from cache. If a repository is dirty is determined by the Arca's `is_dirty` util.

The second location is where lesson content is requested from arbitrary branches, in the `naucse.routes.course_page` function. For lesson contents from arbitrary branches caching works in the following way:

Client estimates the cache key and retrieves the value if it exists.

Client adds the key, if the value was present in the cache, to the list of arguments sent to the `render` function in the arbitrary branch, indicating an *offer* of previously rendered content.

Server can reject the offer and render the content again or returns `None` instead of the content – metadata is returned anyway not depending on the content. **Server** always renders content when no offer is made.

Client uses either the content returned by the server or the value from the cache if `None` was returned.

Client updates the value in the cache with the content used in the previous step.

The `render` function internally calls `naucse.routes.page_content` as well, with the **without_cache** argument, which disables cache in that function. Even if the arbitrary branch changed this so cache would be used, the cache would be non-permanent and would not alter the "client" cache – the callables are called in a separate environment.

4.3.4 The nothing frozen error

There was a slight issue after this fragment caching was implemented. When the content of lessons was retrieved entirely from cache, `Frozen-Flask` started raising an error "*flask_frozen.MissingURLGeneratorWarning: Nothing frozen for endpoints lesson_static. Did you forget a URL generator?*", even though the URLs to the route for static files were correctly generated and frozen.

This is caused by the fact that when URLs are passed to the freezing function of `Frozen-Flask` as absolute URLs instead of the standard way the package registers them, they do not mark the endpoint as used and this error is raised.

This problem was solved by creating a generator, `lesson_static_generator` from `naucse.routes`, which generates the URLs in such a way `Frozen-Flask` marks the endpoint as used.

4.4 Error handling

By default, all errors in arbitrary branches are silenced. If errors are silenced or raised is determined by the `naucse.utils.routes.raise_errors_from_forks` function. The function returns `True` indicating exceptions from arbitrary branches should be re-raised if the environment variable `RAISE_FORK_ERRORS` is set to `true`.

When errors are silenced, they are logged into the `naucse` Python logger, which saves the logs to `.arca/naucse.log`, alongside `.arca/arca.log` where logs from Arca are stored.

4.4.1 List of courses

The first location where an error can occur is in a list of canonical courses or runs. Before the courses from arbitrary branches are rendered in the template they are first checked by the `naucse.utils.routes.does_course_return_info` function, which checks that the branch can be pulled and that the `course_info` function on the "server" side returns required data – the title and description and the start and end dates for runs. If the `course_info` function does not return these attributes, the course is excluded from the list. While in the "serve" mode of running Naucse the course can then be accessed directly, in the "freeze" mode this means the course is ignored completely and is not included in the finished rendered static website.

4.4.2 Lessons

When an error occurs during the rendering of a lesson page inside the arbitrary branch, there are two ways of handling the error.

If the lesson exists in the base branch, the version from the base branch is rendered instead and a warning is added to the top of the page. Naucse then also tries to handle the footer, the links to the previous lesson, next lesson and the current session. The "server" side can return these links when the `naucse.utils.forks.get_footer_links` function is called – when this function throws an error, the footer is ignored as it is not that critical.

If the lesson does not exist in the base branch, an error page is rendered with the information about the arbitrary branch and the lesson which was attempted to be rendered. The template for this page is in `templates/error_in_fork.html`.

When a static file cannot be retrieved from an arbitrary branch, the error is handled similarly. If the lesson and the file exist in the base branch, it is returned, otherwise, a 404 status is returned and the file is ignored by `elsa`.

4.4.3 Other pages

Other pages, like the overview of a course, session cover pages and calendars are only handled by rendering the `templates/error_in_fork.html` template with the correct information about what page was supposed to be rendered.

4.5 Launching Naucse locally

When running Naucse locally, by default the `CurrentEnvironmentBackend` is configured for Arca and courses from arbitrary branches are not shown at all. This is implemented in accordance with the requirement NN6.

Whether courses from forks are allowed is defined by the `forks_enabled` function from `naucse.utils.routes`, which returns `False` by default and `True` when the environment variable `FORKS_ENABLED` is set to `true`.

This function is used in the list of canonical courses or runs to exclude all courses from arbitrary branches. In other places the `forks_raise_if_disabled` function from `naucse.utils.routes` is used, which raises an exception if courses from arbitrary branches are not allowed. This is necessary because while there are no links anywhere to the courses, a user could still request the URL for a page from the course directly.

4.6 Testing

The integration of rendering courses from arbitrary branches is tested with an extension of tests that were already used in Naucse. The previous tests were written using the `pytest` package, so the new tests are also written using this package to match previous tests.

New tests were added in two files in the `test_naucse` folder, `test_route_utils.py` and `test_forks.py`.

The tests in `test_route_utils.py` test the `AllowedElementsParser` class, which will be mentioned later.

The tests in `test_forks.py` are integration tests of courses from arbitrary branches. The courses are tested by creating a local copy of the repository with all the current changes, adding new courses (both canonical courses and runs), committing these changes to the copy of the repository and then referencing these courses in the tests. This way, even the uncommitted changes in the repository are tested.

Parts of the integration that are tested:

- Attributes of `CourseLink` – the integration with `ForkProperty`.
- The render methods of `CourseLink`, that they return values when they should and do not when they should not – like the calendar for a canonical course, which should not be returned.
- Offering of cached content.
- The pages with overviews of courses, that they display courses when arbitrary branches are allowed and alternatively that courses are hidden when arbitrary branches are disabled.
- The ignoring of errors, both in lists of courses and on individual pages.
- The `get_footer_links` function.

The local fork and its availability are managed in several `pytest` fixtures. The first fixture creates the local fork, it is called `fork` and works as follows:

1. The entire contents of the repository is copied, including local changes.
2. Two working courses are added, one canonical course and one run. These courses are generated using the `generate_course` and `generate_run` as a dictionary and then serialized to YAML and saved to `info.yml`.
3. The local changes and the new courses are committed in the `test_branch` branch.
4. Two more courses are added to a new branch, `test_broken_branch`.
5. The rendering is broken by removing contents of the `naucse.utils.render` file.
6. The broken courses are committed.
7. The local repository is removed once the fixture is not used.

The second fixture, `model`, creates an altered instance of a `Root` model, with only the courses from the local fork present. This fixture is then used in the third fixture, `client`, which creates a `Flask` testing client for making requests to specific routes.

4.7 Changes of deployment on Travis CI

A couple of changes had to be made in the configuration of *Travis CI*.

4.7.1 Docker

The most prominent were changes to enable Docker. First, a transition was made from container-based environment to a sudo-enabled one. While the former was launched inside a Docker container, the latter is launched in a full VM, meaning Docker can be used [100].

This was done precisely because of Docker because the next change was to configure `DockerBackend` as the backend that should be used by Arca, by setting the environment variable `ARCA_BACKEND` to `arca.DockerBackend`. Courses from arbitrary branches were enabled by configuring the `FORKS_ENABLED` environment variable.

Another thing that was configured for Docker was login, so images can be pushed to a registry using the `DockerBackend`'s **use_registry_name** configuration option. An account was created for the project on *Docker Hub* [101], Docker's free and public registry. The credentials for were added to the configuration file for *Travis CI*, `.travis.yml`, by defining two environment variables, `DOCKER_HUB_USERNAME` and `DOCKER_HUB_PASSWORD`. The former is added in a plain text format, with the name of the account, `naucse`. The latter is added as a secret variable using Travis' CLI tool [102].

The two variables are then used in the `before_script` part of the configuration to login to Docker, so the images can be pushed. The secret variables are not available in builds of pull requests [70], so the login is conditional – it is only executed when the password variable exists, otherwise the `ARCA_BACKEND_REGISTRY_PULL_ONLY` variable is set so `DockerBackend` only attempts pulling from the registry and does not push.

4.7.2 List of courses

One issue with the usage of Docker is time. Pulling or building one image can take up to several minutes, and several images might have to be required for one build if the forks start to have diverging requirements. The combination of pulling or building multiple images can take a long time, enough to timeout the entire build.

For projects using the free version of *Travis CI* the limit is 10 minutes of nothing being printed [103] and the process of freezing Naucse does not print anything until everything is done. That means if the images were pulled or built during the freeze, the build could timeout.

Because of this, a new command was added to the Naucse's CLI that lists all courses – the command works even locally and can be launched using the code example 4.1. The command lists the identifier of each course, its title and if the course is a run, also the dates. If the course is from an arbitrary branch, the repository URL and the branch name is added, but more importantly, since the title is printed, a container has to be launched to run the task that returns the title and therefore the image is not acquired during freezing.

```
python -m naucse list_courses
```

Code example 4.1: Listing all courses in Naucse

The command goes one-by-one and prints the information, meaning that even if there are multiple Docker images that need to be pulled or built, as long as some build does not take over 10 minutes, the whole Travis build does not timeout.

The command is implemented by extending the `elsa`'s CLI in the `naucse/cli.py` file.

4.7.3 Cache

Travis has a persistent cache between builds [104]. Previously only installed Python packages were stored in this cache. Arca in Naucse is configured to use the filesystem backend for cache, in the `.arca/cache` folder, which was added to the persistent cache on *Travis CI*. The cache is only available between builds in the same branch [104], but anything to speed things up is good.

4.7.4 Logs

After the website is frozen, the Naucse log is printed with the information if and how the rendering of some contents from arbitrary branches failed.

4.8 Vectors of attack and their prevention

The assignment states that the rendering of courses from arbitrary branches has to be done securely since the code in forks is unreviewed and should not be

trusted. This section describes individual vectors of attack and how those threats are prevented or reduced.

That said, some of the vectors are not completely mitigated because there is still some level of trust in the people submitting the courses from arbitrary branches – their pull requests with the link to the branches still need to be reviewed and, at least for now, the maintainers will usually know the people submitting courses. The solutions described in this section are implemented more to prevent accidental errors opposed to malicious attacks, with an inherent bonus of preparing work for the future, when actual malicious attacks will have to be prevented. This section analyses even the unsolved vulnerabilities, so when a time comes when anybody will be able to submit courses, they can be solved completely.

4.8.1 Cross-site scripting

Cross-site scripting (XSS) is a vulnerability in web applications which enables injecting JavaScript scripts to the websites [105]. These scripts can then disrupt the functionality of the website. Since the whole point of this thesis is to include HTML fragments from untrusted sources, this is one of the obvious attacks.

This vector is prevented by only allowing certain HTML elements and attributes. Each piece of HTML from the arbitrary branches is parsed and checked.

The parsing and whitelist are managed by the `AllowedElementsParser` class, which implements `HTMLParser` from the standard Python library. The class is used to parse HTML and raise errors if there are disallowed elements or attributes, or if the HTML cannot be parsed.

The whitelists (one for elements and one for attributes) is defined in this class and it was generated from the existing content of Naucse. Main objects that are missing from the whitelists, on purpose, are the `script` tag and the event attributes like `onhover` etc., simply the elements and attributes from HTML that can launch JavaScript.

To make sure no new elements are being introduced without being added to the whitelist, the same pieces of content that are checked from arbitrary branches are also checked when rendered from the base branch. This way when a new element is added, the programmer will have to add the element to the whitelist. This is important because if new elements were introduced without adding them to the whitelist, any content made from a new arbitrary branch would not pass the check.

If a disallowed object is used in content from arbitrary branches, Naucse treats the result as if the render failed and the error message is displayed instead. If a disallowed object is used in content from the base branch, the exception is raised so the integration test of freezing the content fails.

4.8.2 Disrupting the appearance of the website

It is important that the general appearance stays the same everywhere on the website. It would not make sense for the header and footer and other common control elements to be different for different courses, the layout and style have to be consistent so users can navigate the page easily. The original Naucse has a way for lessons to add custom Cascading Style Sheets (CSS) for that specific lesson. Further, the conversion from `Jupyter Notebook` to HTML adds `style` elements and attributes to output to modify the appearance of individual elements.

The custom CSS was only used in a couple of lessons, but the usage is very effective to make things clearer in the lesson, however, the styles are really only specific for those lessons and it would not make sense to move the styles to generic styles. Further, it would not make sense to disable the feature for courses from forks, because the lessons do rely for the clarity on the styles and it would very counter-intuitive. One would expect that the lessons will look completely the same when rendered from an arbitrary branch when nothing changes in the branch.

I chose to reduce the possibility of the disruption of appearance by limiting the scope of the custom styles to only the content of the lesson, leaving the overall appearance of the website the same. Both of the checks are also performed on content from the base branch, same as the checks for elements and attributes above.

Lesson CSS

The part of the website that contains lesson was placed in a `div` element with the class `lesson-content`. This element was styled to be positioned relatively so if any elements of the lesson are positioned absolutely, they are only absolute inside that element.

The CSS defined for the lesson is parsed and validated. This is done using the `cssutils` package [106]. All the individual selectors are then prefixed with '`.lesson-content `' (without apostrophes, space intentional) which makes the selectors only apply to elements inside the element. The snippet doing the prefixing is showing in code example 4.2.

```python
@staticmethod
def limit_css_to_lesson_content(css):
    parser = cssutils.CSSParser(raiseExceptions=True)
    parsed = parser.parseString(css)

    for rule in parsed.cssRules:
        for selector in rule.selectorList:
            # the space is important - there's a difference between for example
            # ``.lesson-content:hover`` and ``.lesson-content :hover``
            selector.selectorText = ".lesson-content " + selector.selectorText

    return parsed.cssText.decode("utf-8")
```

Code example 4.2: Prefixing custom lesson CSS

The space is very important because it ensures that the styles are only applied to elements inside the `lesson-content` element. Two operators that would be problematic are + and ~. They select either the next or previous element, but for them to be used they would have to be at the start of selectors defined by the lesson (like in code example 4.3), but that would not pass the validation by `cssutils`.

```css
+ div {
    color: red;
}
```

Code example 4.3: Invalid CSS starting with +

Style elements and attributes

The whitelist of allowed elements used in `AllowedElementsParser` includes `style`, however, a further check is made for those elements. The element is only allowed if all the individual selectors start with '`.dataframe` ' – also by `cssutils`. The dataframe element is generated by `Jupyter Notebook` for the outputs of individual cells. By checking that there is a space after the class of the element, it is ensured that only elements inside the dataframe or at most, the previous and next elements, are modified.

Styling applied by the `style` attribute of elements only applies to the element itself, meaning the attribute is allowed in the whitelist of attributes managed by `AllowedElementsParser`.

Problems with the solution

This is the first on the unresolved vulnerabilities – it is not sufficiently secure. I was originally operating under the assumption that CSS is "only styling", but that is not

the case. As was shown by Max Chehab, malicious CSS can be used for example for creating a keylogger [107].

So, even though I tried to limit the scope of the CSS provided by arbitrary branches, this solution cannot be considered safe. But after a consultation with the maintainers of Naucse, I did not extend the solution for managing CSS any further – one possible extension would be whitelisting specific rules to provide a safe subset. The solution is not safe against malicious and untrusted input, but it does provide at least some basic protection.

4.8.3 Code injection

Code injection is a vulnerability that enables attackers to execute any arbitrary code in the application [108]. Since the point of this thesis is to allow exactly that, the execution of that code has to be very careful.

Naucse relies on Arca to handle this vulnerability. Docker is used on *Travis CI*, separating the environment and putting time limits on the execution. Locally, courses from arbitrary branches are disabled by default and the process of enabling them is not documented in the public-facing meta-course, meaning only someone who knows what they are doing can enable them and it is assumed they will know what they are doing.

If the environment was not properly isolated the following could happen:

- The secret variables defined for Travis could be revealed. Currently, there are two of them: `GITHUB_TOKEN` which serves for deploying content to *GitHub Pages* and `DOCKER_HUB_PASSWORD` which serves for pushing images to a Docker registry.
- The whole build on *Travis CI* could be disrupted, stopping build and causing a Denial of Service (DoS).
- The content deployed to *GitHub Pages* could be replaced with something malicious.

4.8.4 Cache poisoning

Cache poisoning is a vulnerability resulting in an attacker being able to modify content not belonging to them by attacking the cache [109].

There are two levels of caching in this thesis that could be affected by this vulnerability.

Arca cache

Arca can cache entire results of tasks (described in section 3.3.5) and the Naucse integration does use this feature. If the attacker could make Arca return a value for a different repository using cache, that would be cache poisoning.

However, that should be impossible because of the way Arca implements this cache. The cache is entirely in control of Arca, which retrieves and saves the values.

The key for the cache is strictly namespaced for specific repository (by using a SHA256 hash of the repository URL). Then the branch name and the commit hash of the last state of a repository is included in the cache. Finally, another SHA256 hash is used to specify for what specific task the result is stored under the key. The method creating the key can be seen in code example 3.4 on page 52.

The security of this key relies on the SHA265 hashing algorithm and will be broken only if the attacker is able to find a conflict using this algorithm applicable to this key.

Naucse cache

The second cache that could be affected is the cache of individual lesson fragments. By poisoning cache, an attacker could possibly deliver malicious content to pages outside their courses.

This problem is solved by namespacing the cache key, so the fragments only apply to their courses or, as was permitted in the #175 issue [13], to courses that share the exact same rendering code. This is done by including the tree hash of the folder containing the rendering code. The key is described in more detail in section 4.3.

4.8.5 Disrupting the build

Even with isolation being handled there are ways how the arbitrary branches could disrupt the build on *Travis CI*.

Invalid returned values

The arbitrary branches can be modified in any way, so the functions that are called by Naucse do not have to return what is expected of them. If the returned values were not validated, the `Flask` app could fail to render some of the pages, like if

keys were missing from dictionaries or `None` was in a variable that is not handled to enable `None`.

So before the values returned by tasks from arbitrary branches are passed along to rendering in the full page, they are first checked to have a specific structure so the rendering does not fail. Most of the checks are done using functions from the `naucse.utils.links` module, but occasionally some checks are made at specific locations where they are relevant. If some of these checks fail, the entire result is treated as if the function failed and the error page is displayed otherwise.

Timeouting the build

Travis CI has time limits for builds. A build is killed after 10 minutes of nothing being printed and the total limit is 120 minutes [103].

Naucse uses Arca's timeouts to reduce the possibility of the build being killed by Travis, but the solution is not complete. Even though installation of requirements is limited to 5 minutes and the execution of task limited to 5 seconds (the defaults in Arca), the attacker could trigger the task being killed by ensuring a lot of tasks would timeout. The process of freezing does not print anything until everything is frozen, meaning only 120 tasks would reach the limit resulting in the build being killed. Even if the process printed the progress, the problem would not go away. The amount of lesson is dictated by the arbitrary branch itself, so an increase of required tasks to kill the build to 1440 would do nothing.

This is the second unresolved vulnerability. It could be solved by creating a pool of time available for a certain branch, but after a consultation with the maintainers, I did not implement this.

4.9 Meta course

Finally, a metacourse was added in pull request #394. This course, although defined as a canonical course, is not displayed in the list of canonical courses, but instead, there is a link to it on the main page. It is available at `https://naucse.python.cz/course/meta/`.

This course describes the complete process of adding a run to Naucse using the new system of adding courses. The contents of the course:

- installing Naucse for local usage,
- creating a run locally, including custom lessons,

- forking Naucse on GitHub,
- pushing changes to the fork,
- submitting a `link.yml` file to the base branch in a pull request,
- installing a webhook and
- making further changes to the run.

Chapter 5

Webhooks for Naucse

Naucse is being deployed as a static app on *GitHub Pages*. The deployment is done through *Travis CI*, on each update of the main `master` branch of the base repository, but that means that changes in the forks would not propagated to the website until an update in the `master` branch of the base repository.

This was be solved by utilizing *GitHub Webhooks* and the *Travis CI* API. Webhooks are a service by GitHub that calls a specific URL every time a certain event occurs, one of these events being an update of the repository. A webhook can be installed for the arbitrary branches with a certain URL. Then an application can listen on the URL and do anthing with the information.

Since Naucse is deployed as a static app, a different application had to be used to react to these webhook calls. This application also handles the automatic installation of the webhook to the forks. This chapter contains the description of the functionality and implementation of the two features in the new application – generating build based on updates in forks and installing the webhooks to GitHub. At the end of the chapter, the deployment of the application is also described.

The application uses `Flask`. This framework is ideal for small web apps in Python and is also used in Naucse. The code is in a separate repository available at `https://github.com/mikicz/naucse-hooks/` and is licensed under the terms of the MIT license [110].

5.1 The app

The whole app is contained in a file in the root of the repository, `naucse_hooks.py`. It is initialized by creating an instance of the `Flask` class, the instance is then used to defined routes and to initialize external packages.

Defining routes in `Flask` is done by using the decorator `route` of the `Flask` instance. The decorator defines on what URL the route is accessible at, the name of the function defines how the route is named. When the URL is requested, `Flask` calls the function matching the route. The request, sessions, cookies and etc. are accessible as global variables in the `flask` module.

The app can be launched locally by running code example 5.1, once requirements from the file `requirements.txt` are installed.

```
FLASK_DEBUG=1 FLASK_APP=naucse_hooks.py python -m flask run
```

Code example 5.1: Running the Naucse Hooks app locally

5.2 Triggering a new build

The hook is served by the route `push_hook` which listens to the path `/hooks/push`. This route reacts to POST requests, first verifying the incoming payload and if everything is valid, triggering a new build of Naucse on *Travis CI*.

5.2.1 Veryfing the payload

The first round of verification is that the request matches what GitHub should send, at least the information required for identifying an update in a repository used in Naucse. This includes checking that the `X-GitHub-Event` HTTP header is set. The header can be set to many different values, but this hook is only interested in `push` and `ping` events. `ping` events are sent to all new webhooks installed to GitHub to verify the hook exists – the route can return a `OK` (status code 200) response right away and stop processing the request. If the header is not set or its value is not `push` or `ping`, the route returns a `Bad Request` (status code 400) response with the description of the error.

Once the header is checked, the body of the request is converted from JSON. If the body is not valid JSON, `Bad Request` is returned again – it can happen if the webhook is not properly configured in GitHub or that somebody else sent the

request. Then the repository and the branch is retrieved from the body, again returning a `Bad Request` if the information is missing.

The second round of verification is that the arbitrary branch from the body is actually used in Naucse. The hook app utilizes `Arca` for cloning or pulling the latest version of the production branch of the Naucse repository, and then checks all the `link.yml` files. If the repository and branch is not used, a `Bad Request` is returned, otherwise, the final check comes.

The last check is that the commit last pushed to the branch was not used previously to trigger a build through this app. This is done by checking GitHub API for the last commit (since the body of the request cannot be trusted) and checking against local records. More about this in the section 5.2.3.

If all the checks and verifications pass a new build is triggered.

5.2.2 Triggering a new build

New builds are triggered on *Travis CI* using their API. There is a Python package for interacting with this API, `TravisPy` [111], but unfortunately, it does not support triggering builds. Instead, the `requests` [112] package is used to make the POST request. The request can be seen in code example 5.2. The required functionality from the trigger is so trivial that the example almost exactly matches the provided example using `curl` from the Travis API documentation [113]. The `app` variable is the global `Flask` application – the individual keys from the configuration are described more in section 5.4. The `repo` and `branch` are parameters of the function which contains the code and are included in the message so it's clear what triggered the build.

```
requests.post(
    "https://api.travis-ci.org/repo/{}/requests".format(
        urllib.parse.quote_plus(app.config["TRAVIS_REPO_SLUG"])
    ),
    json={
        "request": {
            "branch": app.config["NAUCSE_BRANCH"],
            "message": f"Triggered by {repo}/{branch}"
        }
    },
    headers={
        "Authorization": f"token {app.config['TRAVIS_TOKEN']}",
        "Travis-API-Version": "3"
    }
)
```

Code example 5.2: Triggering a build on Travis CI

When builds are triggered automatically by Travis based on pushes to Git, it first checks if there are builds already running for the specific branch and stops them – only the last commit matters. This is not the case for builds triggered using the API. Travis can be configured to limit concurrent builds, however, the new builds are placed in a queue instead. But from the point of Naucse, concurrent builds on the same branch are undesired. Concurrent builds could even result in faulty behaviour – the build time is not constant on Travis, in theory, an outdated build could finish after the latest build, overwriting the version deployed. Because of this, the hook stops all running builds for the production branch.

The `TravisPy` package can be used here – it implements listing builds and stopping them. How the package is used can be seen in code example 5.3. The condition makes sure only builds that would deploy the frozen static pages are stopped – finished builds, builds of pull requests or different branches are excluded.

```
t = TravisPy(app.config['TRAVIS_TOKEN'])
for build in t.builds(slug=app.config["TRAVIS_REPO_SLUG"]):
    if (build.pending and
            not build.pull_request and
            build.commit.branch == app.config["NAUCSE_BRANCH"]):
        build.cancel()
```

Code example 5.3: Stopping pending builds on Travis CI

5.2.3 Security

Since the URL for the webhook is public (it is described in the metacourse in Naucse and in the README file of the repository) the hook has to be secured against attacks. The primary attack the hook is vulnerable to is DoS because of the functionality that stops currently pending builds. If an attacker was able to constantly trigger new builds and cancel pending builds, they could effectively stop deploying altogether.

GitHub provides a way of securing hooks – if a secret key is provided with the webhook URL, it signs the payloads using Hash-based Message Authentication Code (HMAC). The route could then check the signature and refuse any requests that are not signed correctly. The problem is that the webhooks have to be installed automatically for the method to work – if the secret key was public so users could install the hook manually, it is no longer secret. Additionally, the hooks would still have to include checks if the repository is used in Naucse. In fact, this solution does not solve the problem completely, people with push permissions to one of the

repositories used in Naucse can run a DoS attack just by pushing new commits repeatedly.

I chose not to use the HMAC signing. Instead, I am using the checks described in section 5.2.1 to make sure the request is valid. The condition that the repository is used in Naucse is not sufficient (but necessary, even if HMAC was used), but the condition that a new commit was pushed before the hook was called limits the attacks in the same way HMAC would – to people with push permissions to a repository used in Naucse. However, this method allows for manual hook installation.

The current solution is still venerable to attacks. GitHub API, which is used to checks if there was indeed a new commit pushed to a branch, implements rate limiting – only a certain number of requests is allowed per some period of time [114]. An attacker could invoke the hook URL enough times to use up the limit and GitHub API would temporarily cut off the hook app. This time, deployment would not be stopped completely, updates pushed to the main branch of the repository would still trigger new builds, but any updates in arbitrary branches would be not.

After a consultation with the maintainers, I did not implement any precautions to prevent this, but there are several methods that could mitigate these attacks. The first would be caching the results and using conditional requests (requests with a header that says to only return the values if they changed), which do not use up the limit [114]. The caching would be only effective while the number of different arbitrary branches remains low because this API request is only made when the branch is used in Naucse. Once the number of arbitrary branches increases, the solution would be to rate limit incoming requests.

5.3 Automatic installation

The functional requirements for the Naucse integration say that the webhook should be installed in an automated fashion to the forks – this section will describe how the automation was implemented in the webhook `Flask` app.

5.3.1 Logging in using GitHub

I used two different packages to facilitate login using GitHub. The first is `Flask-Session` [115] which extends `Flask` with sessions, a key-value storage only valid for the current user. This package is used to store authentication token

from the second package, `GitHub-Flask` [116]. `GitHub-Flask` handles the OAuth2 authentication and authorization for GitHub.

The package `Flask-Session` adds server-side storage of session data. The data can be stored using different interfaces, but since the hook application only needs to store the authentication token, the filesystem is sufficient. Once the package is initialized, the global `Flask` variable `sessions` works like a regular dictionary, but storing the data persistently on the disk and making the data available to future requests with the same session cookie.

The `GitHub-Flask` package makes logins using GitHub very easy. The usage is shown in code example 5.4. Once the `GitHub` class is initialized, the programmer only has to call its `authorize` method with the required scope and the user is redirected to GitHub to approve the application. Then the callback URL (marked by decorator `github.authorized_handler`) is called with the access token – the programmer can do with the token as they wish (here it's saved to a session). Then the programmer has to tell the class how to retrieve the token from the selected storage, using decorator `github.access_token_getter`. This is required later for using the `GitHub` instance to interact with the GitHub API.

```
from flask_github import GitHub
from flask_session import Session
from flask import Flask, session, redirect, url_for, flash

app = Flask(__name__)
github = GitHub(app)
Session(app)

@app.route('/login')
def login():
    return github.authorize("public_repo,write:repo_hook")

@app.route('/github-callback')
@github.authorized_handler
def authorized(oauth_token):
    if oauth_token is None:
        flash("Login failed!", "error")
        return redirect(url_for('index'))

    session["github_access_token"] = oauth_token

    return redirect(url_for('index'))

@github.access_token_getter
def token_getter():
    return session.get("github_access_token")
```

Code example 5.4: Login using GitHub with `GitHub-Flask`

5.3.2 Listing repositories

Once the user is logged in, the `index` route lists all the public `naucse.python.cz` repositories they have access to (one user can be in multiple GitHub organizations which can have the repository forked as well). The appearance of the page is shown in figure 5.1.

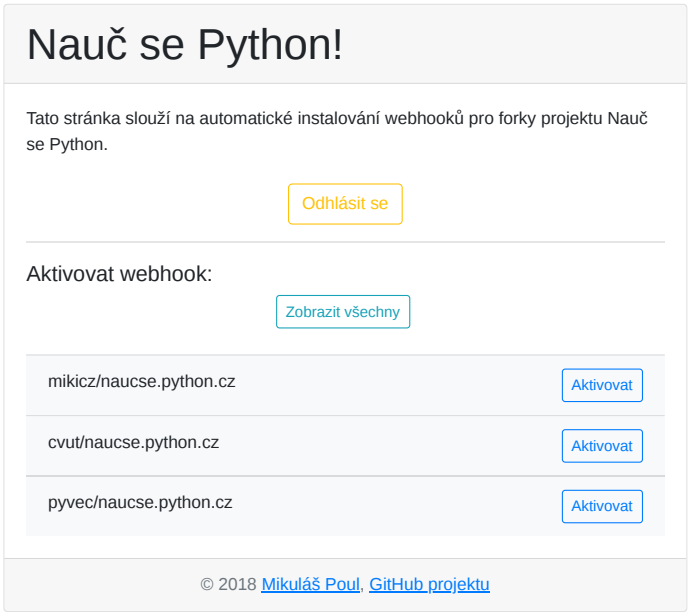

Figure 5.1: The page with automatic webhook installation

The repositories are listed using the `GitHub` instance, by calling the code shown in example 5.5. This snippet returns all the public repositories the current user has access to (is the owner, admin of the group or a collaborator). For this the scope `public_repo` (as can be seen in code example 5.4) is required. This scope, unfortunately, provides quite a lot of permissions (for example write permissions to code), but there is not another scope which would provide read-only permissions of all the user's repositories [117].

```
github.get("user/repos", all_pages=True, data={"visibility": "public"})
```

Code example 5.5: Listing public repositories accessible by the user

Forks usually retain the name of the original repositories, so primarily, only repositories called `naucse.python.cz` are listed, but the user can list all the other repositories if they choose so. The `naucse.python.cz` repositories are always listed first, ordered by the name of the owner, with the current user being shown first.

Then the other repositories are listed – first the repositories of the user and then the rest sorted alphabetically.

Ideally, repositories with already activated hooks would be marked as such, but a separate request would be required for each repository shown. That would slow down the loading of the page considerably since some users have a large number of repositories or have access to multiple organizations with a lot of repositories.

5.3.3 Activating webhooks

The last step is to actually install and activate the webhook. The `write:repo_hook` scope is required for adding new webhooks (the scope also allows to read existing webhooks). Once the user clicks on the button to activate the webhooks the route `activate` is invoked with the owner of the repository and the name of the repository. Before creating the webhook in the repository, a couple of checks are launched first:

- that the repository exists and the user can read it,
- that the existing webhooks can be read and
- that the webhook does not already exist in the repository.

If any of the checks fail then the user is redirected back to the page with listed repositories with a message explaining what went wrong. Othewise the webhook is installed using the `GitHub` instance – shown in code example 5.6. `login` and `name` are the variables containing the owner and the name of the repository, `app` is the global `Flask` application. When the call succeeds, the user is redirected back to the main page with a success message. If the request fails (a `GitHubError` exception is raised), it means that the user most likely does not have write access to the repository and an error message is shown instead.

```
github.post(f"repos/{login}/{name}/hooks", {
                "name": "web",
                "config": {
                    "url": app.config["PUSH_HOOK"],
                    "content_type": "json"
                }
            })
```

Code example 5.6: Adding a new webhook to repository

5.4 Configuration

Since the repository containing the code is public, it cannot contain the all the various secret tokens and keys that are required for running the page. Apart from that, other configuration is also required for the app to run. To separate the configuration from the code of the app and to enable overriding the configuration locally the following approach is used in the app.

There is a file with the default configuration in the root of the repository, called `settings.cfg`. Optionally, a file called `local_settings.cfg` can be created with overridden settings – this file is ignored and not added to Git. The two files are then loaded to the `Flask` app using the code in code example 5.7. The `silent` argument in the second `from_pyfile` call makes the file optional, but since it is second it overrides the default configuration.

```
from flask import Flask

app = Flask(__name__)
app.config.from_pyfile("settings.cfg")
app.config.from_pyfile("local_settings.cfg", silent=True)
```

Code example 5.7: Loading configuration to `Flask`

The files use Python syntax, any variables declared in them are loaded to the `app.config` attribute and can be then accessed from the app. The hook app utilizes this functionality for the secret keys and tokens (e.g. code examples 5.2 on page 97 or 5.3).

The following configuration has to be filled out for the `Flask` app to work properly (some values are pre-configured for the production Naucse instance):

NAUCSE_GIT_URL The URL of the repository Naucse is rendered from, used to check if the pushed repository is used in it.

NAUCSE_BRANCH The branch Naucse is rendered from in the **NAUCSE_GIT_URL** repository. Used to check if the pushed repository is used and to stop and trigger Travis builds.

TRAVIS_REPO_SLUG The slug used on *Travis CI* (owner/name).

TRAVIS_TOKEN The authentication token for interacting with Travis API.

SESSION_COOKIE_DOMAIN The domain the app is deployed on or `None`.

SENTRY_DSN The Data Source Name (DSN) for Sentry, an error handling service (`None` otherwise). See section 5.6 for more.

SECRET_KEY The key for singing session cookies.

GITHUB_CLIENT_ID The ID of the GitHub OAuth2 app for the login functionality.
GITHUB_CLIENT_SECRET The secret key for the GitHub OAuth2 app.
PUSH_HOOK The full URL for the push hook (including protocol and domain).

5.5 Testing

The critical part of the app – the hook route – is tested using the `pytest` package.

The tests are located in the `test_naucse_hooks.py` file. There are three fixtures located in the file, one that returns the `Flask` app configured for testing, one that returns a client for the app and one that returns a location to a folder, a "fake repository", which has the same structure as Naucse, but the contents can be relied on to stay the same.

These fixtures are then used in the tests functions, either testing individual functions defined in `naucse_hooks.py` or the entire route by calling it using the test client.

Similarly, as in Arca and Naucse, *Travis CI* is used to run the tests automatically once updates are pushed and for pull requests, `flake8` is used to check code's compliance with PEP8.

5.6 Logging

To catch the errors within the `Flask` app, `raven` [118] is used to catch errors and send the logs to *Sentry* [119]. *Sentry* is an error tracking software which has integrations with many different langauges and frameworks, `raven` is its error catching package. The integration of `raven` into the `Flask` app can be seen in the code example 5.8. If the configuration **SENTRY_DSN** is set with the Data Source Name of the `Sentry` project, all exceptions are caught and sent to `Sentry` with debug information.

```
from flask import Flask
from raven.contrib.flask import Sentry

app = Flask(__name__)
...
sentry = Sentry(app, dsn=app.config["SENTRY_DSN"])
```

Code example 5.8: Integration of the `raven` package to `Flask`

`Flask` logging (which uses Python logging from the standard library) is used to log incoming hook events and if they were rejected and why. The logs are saved to the file `naucse_hooks.log`, using a `RotatingFileHandler` handler. The handler is registered to the `Flask` app using code from example 5.9.

```
import logging.handlers
from flask import Flask

app = Flask(__name__)
...
handler = logging.handlers.RotatingFileHandler("naucse_hooks.log")
formatter = logging.Formatter("[\%(asctime)s] {\%(pathname)s:\%(lineno)d}"
                              " \%(levelname)s - \%(message)s")

handler.setLevel(logging.INFO)
handler.setFormatter(formatter)

app.logger.addHandler(handler)
```

Code example 5.9: Registering a custom handler to `Flask`

5.7 Deployment

The hook app can be deployed anywhere using Web Server Gateway Interface (WSGI) – a Python convention for web servers to call Python apps. I deployed the app on my own Virtual Private Server (VPS) for the purposes of Naucse, using the domain `hooks.nauc.se` provided by one of the maintainers. My server runs on Debian 8 and uses Apache 2 as its web server. I used *Let's Encrypt* [120] to generate a SSL certificate to serve the app over HTTP Secure (HTTPS).

Conclusion

The goal of this thesis was to solve several issues with the unsatisfactory situation of rendering content for the *Nauč se Python!* (*Learn Python!*) project . The solution was to render parts of the website, materials for some specific courses, from other sources – forks of the base Git repository. I accomplished this and the changes I implemented are already used to offset the issues. But that is getting ahead of myself.

First, I analysed the situation, the issues and why alternative solutions were unsuitable for this project. I defined a set of requirements based on the assignment of this thesis and analysis of what was needed.

I tried to find tools for rendering content from Git repositories in an isolated environment, but did not find any that were suitable. Most of the existing tools used very old Python or placed too many restrictions on the code. An isolated environment was required because the code in forks cannot be trusted – it cannot be reviewed by the maintainers of the project.

Instead, I developed a tool called *Arca* that matches everything in the requirements and actually does more than was required. I made Arca very configurable and extendible, which will hopefully help it spread into other projects.

Arca efficiently clones Git repositories, sets up isolation and can execute anything from the repository that can be called by Python in the isolated environment. Arca can cache the results, so execution does not have to be repeated. Arca is also configurable in the level of isolation. The callables can be executed without isolation, in Docker containers or in a full virtual machine.

Arca is thoroughly tested with a combination of unit and integration tests, which are launched automatically using *Travis CI*. It has extensive documentation deployed to *Read The Docs*, and can be installed from the Python Packaging Index.

Then I integrated Arca into the project, enabling rendering of individual courses from forks. In the base branch, the courses can now be also defined by a single file containing a link to the target repository and the branch name.

Individual courses can share content fragments, if they also share the rendering code. This saves computing time but also protects courses from cache poisoning. The content from forks is separated from the rest of the website to not disrupt the overall appearance of the website. If an error occurs during rendering from a fork, an error message is displayed instead of build failure.

I also added a detailed metacourse to the project, which describes in Czech how to use this new functionality. This course explains to organisers how to add their course to the website.

To propagate changes from forks to the website I developed a new, dynamic application that reacts to changes in the forks. The app triggers a new build of the production website when an update is made in a fork used in the project. This app also activates GitHub Webhooks, the service that notifies the application of changes.

The maintainers approved and merged my changes and the solution had already been used. Three past courses were frozen and archived and one new course was already taught using new materials from a fork.

The assignment of the thesis was fulfilled completely. The tool, thanks to its configurability, can do even more than was requested. The integration implements even features marked as optional in the assignment.

I must say I spent a lot of time working on this thesis but I am very happy with the result and I think it will really help the community. I believe in the community's mission of teaching Python. I have taken part in some courses myself and anything that can help the community to make these courses better is worth the time.

That is not to say all work is done, there are definitely things to improve, both on the tool and on the integration.

A flaw I see with Arca is that it requires a completely separate configuration for every structure of the target repository. Another is the inefficiency of the isolation

in a virtual machine, which could definitely be sped up, but there just was not enough time to make this happen.

The integration, as was described in more detail in section 4.8, still needs some work on security. While the new situation is fine as long as the maintainers know who submitted the courses from the forks, if at any point completely unknown people were allowed to submit courses, suddenly there are possible holes.

Bibliography

1. VIKTORIN, Petr; HRONČOK, Miroslav, et al. *Nauč se Python: MI-PYT (Pokročilý Python)* [online] [visited on 2018-04-04]. Available from: `http://naucse.python.cz/2017/mipyt-zima/`.
2. VIKTORIN, Petr; HRONČOK, Miroslav, et al. *Nauč se Python: Asteroidy na InstallFestu* [online] [visited on 2018-04-04]. Available from: `http://naucse.python.cz/2018/installfest/`.
3. VIKTORIN, Petr et al. *GitHub: naucse.python.cz* [online] [visited on 2018-04-04]. Available from: `https://github.com/pyvec/naucse.python.cz/`.
4. RONACHER, Armin. *Flask: A Python Microframework* [online] [visited on 2018-04-04]. Available from: `http://flask.pocoo.org/`.
5. HRONČOK, Miroslav et al. *elsa* [online] [visited on 2018-04-04]. Available from: `https://github.com/pyvec/elsa`.
6. EVANS, Clark C. *YAML: YAML Ain't Markup Language* [online] [visited on 2018-04-04]. Available from: `http://yaml.org/`.
7. GRUBER, John. *Markdown* [online] [visited on 2018-04-04]. Available from: `https://daringfireball.net/projects/markdown/`.
8. PROJECT JUPYTER. *Project Jupyter* [online] [visited on 2018-04-04]. Available from: `http://jupyter.org/`.
9. TRAVIS CI, GmbH. *Travis CI: Test and Deploy with Confidence* [online] [visited on 2018-04-04]. Available from: `https://travis-ci.com/`.
10. GITHUB, Inc. *GitHub Pages: Websites for you and your projects.* [online] [visited on 2018-04-04]. Available from: `https://pages.github.com/`.

11. MESSNER, Petr. *Materials of archived courses should be freezed/should not change: Issue #214* [online] [visited on 2018-04-26]. Available from: `https://github.com/pyvec/naucse.python.cz/issues/214`.

12. DEVIQ. *DevIQ: Don't Repeat Yourself* [online] [visited on 2018-05-09]. Available from: `http://deviq.com/don-t-repeat-yourself/`.

13. VIKTORIN, Petr. *Render courses from forked repositories: Issue #175* [online] [visited on 2018-04-26]. Available from: `https://github.com/pyvec/naucse.python.cz/issues/175`.

14. THE PIP DEVELOPERS. *pip 1.1 documentation: Requirements files* [online] [visited on 2018-05-03]. Available from: `https://pip.readthedocs.io/en/1.1/requirements.html`.

15. FLINT, Alex. *Process Isolation in Python: Elegant process isolation in pure python* [online] [visited on 2018-04-12]. Available from: `https://alexflint.github.io/process-isolation/`.

16. BRESSERS, Joshua. *Red Hat Security Blog: Is chroot a security feature?* [online]. 2013 [visited on 2018-04-12]. Available from: `https://access.redhat.com/blogs/766093/posts/1975883`.

17. ZOPE FOUNDATION. *RestrictedPython* [online] [visited on 2018-04-12]. Available from: `https://github.com/zopefoundation/RestrictedPython`.

18. ZOPE FOUNDATION. *Zope Foundation* [online] [visited on 2018-04-12]. Available from: `http://www.zope.org/en/latest/foundation.html`.

19. PLONE FOUNDATION. *Plone* [online] [visited on 2018-04-12]. Available from: `https://plone.org/`.

20. ZOPE FOUNDATION. *Zope* [online] [visited on 2018-04-12]. Available from: `https://github.com/zopefoundation/Zope`.

21. PLONE FOUNDATION. *Plone Documentation v5.1: Sandboxing and RestrictedPython* [online] [visited on 2018-04-12]. Available from: `https://docs.plone.org/develop/plone/security/sandboxing.html`.

22. STINNER, Victor. *pysandbox* [online] [visited on 2018-04-14]. Available from: `https://github.com/vstinner/pysandbox`.

23. STINNER, Victor. *[Python-Dev]: The pysandbox project is broken* [online] [visited on 2018-04-14]. Available from: `https://mail.python.org/pipermail/python-dev/2013-November/130132.html`.

24. THE PYPY PROJECT et al. *PyPy* [online] [visited on 2018-04-14]. Available from: `https://pypy.org/`.

25. THE PYPY PROJECT et al. *PyPy documentation: PyPy's sandboxing features* [online] [visited on 2018-04-14]. Available from: `http://pypy.readthedocs.io/en/latest/sandbox.html`.

26. THE PYPY PROJECT et al. *PyPy: What is PyPy?* [online] [visited on 2018-04-14]. Available from: `https://pypy.org/features.html`.

27. THE PYPY PROJECT et al. *PyPy: Python compatibility* [online] [visited on 2018-04-14]. Available from: `https://pypy.org/compat.html`.

28. BATCHELDER, Ned; DALY, Will, et al. *codejail* [online] [visited on 2018-04-14]. Available from: `https://github.com/edx/codejai`.

29. IMMUNIX et al. *AppArmor* [online] [visited on 2018-04-14]. Available from: `https://gitlab.com/apparmor/apparmor`.

30. PYTHON SOFTWARE FOUNDATION. *venv: Creation of virtual environments* [online] [visited on 2018-04-24]. Available from: `https://docs.python.org/3/library/venv.html`.

31. IMMUNIX et al. *AppArmor: Wiki* [online] [visited on 2018-04-14]. Available from: `https://gitlab.com/apparmor/apparmor/wikis/home/`.

32. TAFANI-DEREEPER, Christophe. *docker-python-sandbox* [online] [visited on 2018-04-12]. Available from: `https://github.com/christophetd/docker-python-sandbox`.

33. NODE.JS FOUNDATION. *Node.js* [online] [visited on 2018-04-12]. Available from: `https://nodejs.org/en/`.

34. DOCKER, Inc. *Docker: Build, Ship, and Run Any App, Anywhere* [online] [visited on 2018-04-12]. Available from: `https://www.docker.com/`.

35. LEWIS, Charlton T.; SHORT, Charles. *A Latin Dictionary*. Available also from: `http://www.perseus.tufts.edu/hopper/text?doc=Perseus:text:1999.04.0059:entry=arca`.

36. DEVIQ. *DevIQ: SOLID* [online] [visited on 2018-05-09]. Available from: `http://deviq.com/solid/`.

37. PETERS, Tim. *PEP 20: The Zen of Python* [online]. 2004 [visited on 2018-04-20]. Available from: `https://www.python.org/dev/peps/pep-0020/`.

38. SOURCEMAKING.COM. *Strategy Design Pattern* [online] [visited on 2018-05-09]. Available from: `https://sourcemaking.com/design_patterns/strategy`.

39. BAYER, Michael et al. *dogpile.cache* [online] [visited on 2018-04-25]. Available from: `https://bitbucket.org/zzzeek/dogpile.cache`.

40. BAYER, Michael et al. *dogpile.cache 0.6.6 documentation: API* [online] [visited on 2018-04-25]. Available from: `https://dogpilecache.readthedocs.io/en/latest/api.html#module-dogpile.cache.backends.memory`.

41. WALSH, Daniel J. *Opensource.com: Daniel J Walsh* [online] [visited on 2018-04-24]. Available from: `https://opensource.com/users/rhatdan`.

42. WALSH, Daniel J. Are Docker containers really secure? *Opensource.com* [online]. 2014 [visited on 2018-04-24]. Available from: `https://opensource.com/business/14/7/docker-security-selinux`.

43. WALSH, Daniel J. Bringing new security features to Docker. *Opensource.com* [online]. 2014 [visited on 2018-04-24]. Available from: `https://opensource.com/business/14/9/security-for-docker`.

44. DOCKER, Inc. *Docker documentation: Docker security* [online] [visited on 2018-04-12]. Available from: `https://docs.docker.com/engine/security/security`.

45. DOCKER, Inc. *Docker documentation: Install Docker* [online] [visited on 2018-04-12]. Available from: `https://docs.docker.com/install/`.

46. TRAVIS CI, GmbH. *Travis CI: Using Docker in Builds* [online] [visited on 2018-04-24]. Available from: `https://docs.travis-ci.com/user/docker/`.

47. HASHICORP, Inc. *Vagrant* [online] [visited on 2018-04-24]. Available from: `https://www.vagrantup.com/`.

48. HASHICORP, Inc. *Vagrant: Introduciton* [online] [visited on 2018-04-24]. Available from: `https://www.vagrantup.com/intro/index.html`.

49. HASHICORP, Inc. *Vagrant by HashiCorp: Download* [online] [visited on 2018-05-11]. Available from: `https://www.vagrantup.com/downloads.html`.

50. ORACLE CORPORATION. *Oracle VM VirtualBox* [online] [visited on 2018-04-24]. Available from: `https://www.virtualbox.org/`.

51. VMWARE, Inc. *VMware* [online] [visited on 2018-04-24]. Available from: `https://www.vmware.com/`.

52. RED HAT, Inc. *libvirt: The virtualization API* [online] [visited on 2018-05-02]. Available from: `https://libvirt.org/`.

53. MICROSOFT CORPORATION. *Microsoft Docs: Hyper-V on Windows 10* [online] [visited on 2018-04-24]. Available from: `https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/index`.

54. HASHIMOTO, Mitchell et al. *vagrant-aws* [online] [visited on 2018-04-24]. Available from: `https://github.com/mitchellh/vagrant-aws`.

55. HASHICORP, Inc. *Vagrant by HashiCorp: Docker - Provisioning* [online] [visited on 2018-05-02]. Available from: `https://www.vagrantup.com/docs/provisioning/docker.html`.

56. TRIER, Michael et al. *GitPython* [online] [visited on 2018-04-25]. Available from: `https://github.com/gitpython-developers/GitPython`.

57. KLUYVER, Thomas et al. *entrypoints* [online] [visited on 2018-04-06]. Available from: `https://github.com/takluyver/entrypoints`.

58. PYTHON PACKAGING AUTHORITY. *Python Packaging User Guide: Packaging and Distributing Projects* [online] [visited on 2018-04-06]. Available from: `https://packaging.python.org/tutorials/distributing-packages/#entry-points`.

59. KLUYVER, Thomas et al. *entrypoints: Documentation* [online] [visited on 2018-04-06]. Available from: `http://entrypoints.readthedocs.io/en/latest/`.

60. PYTHON SOFTWARE FOUNDATION. *Python 3.6.5 documentation: Glossary* [online] [visited on 2018-04-13]. Available from: `https://docs.python.org/3/glossary.html#term-iterable`.

61. GREENFELD, Daniel et al. *cached-property* [online] [visited on 2018-04-24]. Available from: `https://github.com/pydanny/cached-property`.

62. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *Secure Hash Standard* [online]. 2015 [visited on 2018-05-09]. Available from DOI: `10.6028/NIST.FIPS.180-4`.

63. AUTOMATE DOCKER DEPLOYMENT, How to. *Caleb Sotelo* [online] [visited on 2018-05-04]. Available from: `http://paislee.io/how-to-automate-docker-deployments/`.

64. DOCKER, Inc. *docker-py* [online] [visited on 2018-05-04]. Available from: `https://github.com/docker/docker-py`.

65. FREE SOFTWARE FOUNDATION, Inc. *timeout(1): run command with time limit: Linux man page* [online] [visited on 2018-05-04]. Available from: `https://linux.die.net/man/1/timeout`.

66. SPI. *Debian – Debian "stretch" Release Information* [online] [visited on 2018-05-04]. Available from: `https://www.debian.org/releases/stretch/`.

67. ALPINE LINUX DEVELOPMENT TEAM. *Alpine Linux* [online] [visited on 2018-05-04]. Available from: `https://alpinelinux.org/`.

68. PRZERADOWSKI, Paweł Piotr. *Issue #37: [Question] Any plans for musl based distros?* [online] [visited on 2018-05-04]. Available from: https://github.com/pypa/manylinux/issues/37.

69. YAMASHITA, Yuu; STEPHENSON, Sam, et al. *pyenv* [online] [visited on 2018-05-04]. Available from: https://github.com/pyenv/pyenv.

70. TRAVIS CI, GmbH. *Travis CI: Environment Variables* [online] [visited on 2018-05-08]. Available from: https://docs.travis-ci.com/user/environment-variables/.

71. DELUCA, Todd et al. *python-vagrant* [online] [visited on 2018-05-04]. Available from: https://github.com/todddeluca/python-vagrant.

72. HASHICORP, Inc. *Vagrant by HashiCorp: Basic Usage - Synced Folders* [online] [visited on 2018-05-11]. Available from: https://www.vagrantup.com/docs/synced-folders/basic_usage.html.

73. FORCIER, Jeffrey E.; HANSEN, Christian Vest; ERTL, Mathias, et al. *fabric* [online] [visited on 2018-05-04]. Available from: https://github.com/mathiasertl/fabric/.

74. AILISPAW. *Vagrant Cloud: Vagrant box ailispaw/barge* [online] [visited on 2018-05-11]. Available from: https://app.vagrantup.com/ailispaw/boxes/barge.

75. PYTHON SOFTWARE FOUNDATION. *Python 3.6.5 documentation: Logging HOWTO* [online] [visited on 2018-05-03]. Available from: https://docs.python.org/3.6/howto/logging.html.

76. KREKEL, Holger et al. *pytest documentation: pytest: helps you write better programs* [online] [visited on 2018-05-03]. Available from: https://docs.pytest.org/en/latest/.

77. KREKEL, Holger et al. *pytest documentation: pytest fixtures: explicit, modular, scalable* [online] [visited on 2018-05-03]. Available from: https://docs.pytest.org/en/latest/fixture.html.

78. KREKEL, Holger et al. *pytest documentation: Parametrizing fixtures and test functions* [online] [visited on 2018-05-11]. Available from: https://docs.pytest.org/en/latest/parametrize.html.

79. ZIADE, Tarek; CORDASCO, Ian, et al. *flake8* [online] [visited on 2018-05-03]. Available from: https://gitlab.com/pycqa/flake8.

80. ROSSUM, Guido van; WARSAW, Barry; COGHLAN, Nick. *PEP 8: Style Guide for Python Code* [online]. 2001 [visited on 2018-05-03]. Available from: https://www.python.org/dev/peps/pep-0008/.

81. LEHTOSALO, Jukka et al. *mypy - Optional Static Typing for Python* [online] [visited on 2018-05-03]. Available from: http://www.mypy-lang.org/.

82. SPENCER, Chris. *Vagrant inaccessible: Issue #6060* [online] [visited on 2018-05-04]. Available from: https://github.com/travis-ci/travis-ci/issues/6060.

83. GOODGER, David. *reStructuredText* [online] [visited on 2018-04-29]. Available from: http://docutils.sourceforge.net/rst.html.

84. GOODGER, David. *PEP 287: reStructuredText Docstring Format* [online]. 2002 [visited on 2018-04-29]. Available from: https://www.python.org/dev/peps/pep-0287/.

85. BRANDL, Georg et al. *Sphinx documentation* [online] [visited on 2018-04-29]. Available from: http://www.sphinx-doc.org/en/master/.

86. BRANDL, Georg et al. *Sphinx documentation: Available builders* [online] [visited on 2018-05-11]. Available from: http://www.sphinx-doc.org/en/master/builders.html.

87. READ THE DOCS, Inc. *Read The Docs* [online] [visited on 2018-04-29]. Available from: https://readthedocs.org/.

88. READ THE DOCS, Inc. *Read the Docs 1.0 documentation: Read the Docs YAML Config* [online] [visited on 2018-04-29]. Available from: http://docs.readthedocs.io/en/latest/yaml-config.html#build-image.

89. PYTHON PACKAGING AUTHORITY. *Python Packaging User Guide: Tool recommendations* [online] [visited on 2018-04-28]. Available from: https://packaging.python.org/guides/tool-recommendations/.

90. PYTHON PACKAGING AUTHORITY. *setuptools* [online] [visited on 2018-04-28]. Available from: https://github.com/pypa/setuptools.

91. PYTHON PACKAGING AUTHORITY. *setuptools documentation: Developer's Guide for Setuptools* [online] [visited on 2018-05-03]. Available from: https://setuptools.readthedocs.io/en/latest/setuptools.html#install-run-easy-install-or-old-style-installation.

92. PYTHON PACKAGING AUTHORITY. *Python Packaging User Guide: Packaging and distributing projects* [online] [visited on 2018-05-03]. Available from: %7Bhttps://packaging.python.org/tutorials/distributing-packages/%7D.

93. PYTHON PACKAGING AUTHORITY. *setuptools documentation: Developer's Guide for Setuptools* [online] [visited on 2018-05-03]. Available from: https://setuptools.readthedocs.io/en/latest/setuptools.html#declaring-extras-optional-features-with-their-own-dependencies.

94. COOMBS, Jason R. et al. *pytest-runner* [online] [visited on 2018-05-03]. Available from: https://github.com/pytest-dev/pytest-runner.

95. KREKEL, Holger et al. *pytest documentation: Good Integration Practices* [online] [visited on 2018-05-03]. Available from: https://docs.pytest.org/en/latest/goodpractices.html#integrating-with-setuptools-python-setup-py-test-pytest-runner.

96. JI, Chuan. *How To Add Custom Build Steps and Commands To setup.py* [online] [visited on 2018-05-11]. Available from: https://seasonofcode.com/posts/how-to-add-custom-build-steps-and-commands-to-setuppy.html.

97. PYTHON PACKAGING AUTHORITY. *twine* [online] [visited on 2018-05-03]. Available from: https://github.com/pypa/twine.

98. NEPHILA et al. *giturlparse* [online] [visited on 2018-05-07]. Available from: https://github.com/nephila/giturlparse.

99. SAPIN, Simon et al. *Frozen-Flask* [online] [visited on 2018-05-07]. Available from: https://github.com/Frozen-Flask/Frozen-Flask.

100. TRAVIS CI, GmbH. *Travis CI: Using Docker in Builds* [online] [visited on 2018-05-07]. Available from: https://docs.travis-ci.com/user/docker/.

101. DOCKER, Inc. *Docker Hub* [online] [visited on 2018-05-07]. Available from: https://hub.docker.com/.

102. TRAVIS CI, GmbH. *travis.rb* [online] [visited on 2018-05-07]. Available from: https://github.com/travis-ci/travis.rb.

103. TRAVIS CI, GmbH. *Travis CI: Customizing the Build* [online] [visited on 2018-05-07]. Available from: https://docs.travis-ci.com/user/customizing-the-build#Build-Timeouts.

104. TRAVIS CI, GmbH. *Travis CI: Caching Dependencies and Directories* [online] [visited on 2018-05-07]. Available from: https://docs.travis-ci.com/user/caching/.

105. THE OPEN WEB APPLICATION SECURITY PROJECT et al. *Cross-site Scripting (XSS)* [online] [visited on 2018-05-09]. Available from: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

106. HÖKE, Christof et al. *cssutils* [online] [visited on 2018-05-08]. Available from: https://bitbucket.org/cthedot/cssutils.

107. CHEHAB, Max. *CSS-Keylogging* [online] [visited on 2018-05-08]. Available from: https://github.com/maxchehab/CSS-Keylogging.

108. THE OPEN WEB APPLICATION SECURITY PROJECT et al. *Code Injection* [online] [visited on 2018-05-11]. Available from: https://www.owasp.org/index.php/Code_Injection.

109. THE OPEN WEB APPLICATION SECURITY PROJECT et al. *Cache Poisoning* [online] [visited on 2018-05-09]. Available from: https://www.owasp.org/index.php/Cache_Poisoning.

110. OPEN SOURCE INITIATIVE. *The MIT License (MIT)* [online] [visited on 2018-04-24]. Available from: https://opensource.org/licenses/MIT.

111. MENEGAZZO, Fabio et al. *TravisPy* [online] [visited on 2018-04-24]. Available from: https://travispy.readthedocs.io/.

112. REITZ, Kenneth et al. *Requests: HTTP for Humans* [online] [visited on 2018-04-24]. Available from: http://docs.python-requests.org/.

113. TRAVIS CI, GmbH. *Travis CI: Triggering builds with API V3* [online] [visited on 2018-04-24]. Available from: https://docs.travis-ci.com/user/triggering-builds/.

114. GITHUB, Inc. *GitHub Developer Guide: GitHub API v3* [online] [visited on 2018-05-02]. Available from: https://developer.github.com/v3/.

115. FENG, Shipeng et al. *Flask-Session* [online] [visited on 2018-04-24]. Available from: https://pythonhosted.org/Flask-Session/.

116. ALTI, Cenk et al. *GitHub-Flask* [online] [visited on 2018-04-24]. Available from: https://github-flask.readthedocs.io/.

117. GITHUB, Inc. *GitHub Developer Guide: Scopes* [online] [visited on 2018-04-24]. Available from: https://developer.github.com/apps/building-oauth-apps/scopes-for-oauth-apps/.

118. FUNCTIONAL SOFTWARE, Inc. et al. *raven-python* [online] [visited on 2018-04-24]. Available from: https://github.com/getsentry/raven-python.

119. FUNCTIONAL SOFTWARE, Inc. *Sentry: Error Tracking Software* [online] [visited on 2018-04-24]. Available from: https://sentry.io/.

120. INTERNET SECURITY RESEARCH GROUP. *Let's Encrypt: Free SSL/TLS Certificates* [online] [visited on 2018-04-24]. Available from: `https://letsencrypt.org/`.

Appendix A

Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
AWS	Amazon Web Service
CD	Continuous Delivery
CI	Continuous Integration
CLI	Command Line Interface
CSS	Cascading Style Sheets
CTU	Czech Technical University in Prague
DoS	Denial of Service
DRY	Don't Repeat Yourself
DSN	Data Source Name
FIT	Faculty of Information Technology
HMAC	Hash-based Message Authentication Code
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
JSON	JavaScript Object Notation
MIT	Massachusetts Institute of Technology
PDF	Portable Document Format
PID	Process ID
PR	Pull Request
PyPI	Python Packaging Index
SHA	Secure Hash Algorithm
SSH	Secure Shell
SSL	Secure Sockets Layer

UML	Unified Modeling Language
URL	Uniform Resource Locator
VM	Virtual Machine
VPS	Virtual Private Server
WSGI	Web Server Gateway Interface
XSS	Cross-site scripting
YAML	YAML Ain't Markup Language

Appendix B

Contents of enclosed CD

```
README.txt ..... the file with CD contents description
arca ..... the repository with the sources of the Arca tool
    docs .....
        _build .....
            html ..... the rendered documentation of Arca in HTML format
naucse.python.cz ..... the repository with the sources of the Naucse project
naucse_hooks ..... the repository with the sources for the Naucse Hooks app
text_sources ..... the repository with the source of the PDF
BP_Mikulas_Poul_2018.pdf ..... the PDF of the thesis text
```

Directory structure B.1: Contents of enclosed CD

Contents also available from:

- `https://github.com/mikicz/arca`
- `https://arca.readthedocs.io/`
- `https://github.com/pyvec/naucse.python.cz`
- `https://github.com/mikicz/naucse-hooks`
- `https://gitlab.com/mikulaspoul/bachelor-thesis`
- `https://bachelor-thesis.mikulaspoul.cz/BP_Mikulas_Poul_2018.pdf`