ANKARA UNIVERSITY

ENGINEERING FACULTY

DEPARTMENT OF COMPUTER ENGINEERING

INTERNSHIP REPORT

CI/CD

Ömer Faruk UYSAL

19290337

18.07.2022 – 12.08.2022

ABSTRACT

In modern application of product development, the goal is to have multiple team members working simultaneously on different parts of the same product. However, this collaboration brings other problems. At the end, when separately carried jobs need to be combined, the resulting work can be tedious, manual, and time intensive. This is where CI/CD (Continuous Integration/ Continuous Delivery) method comes into our lifes. This method helps us by introducing automation into the stages of product development.

During the internship, I was given multiple tasks which are all in the context of CI/CD. The main purpose of these tasks, is to automate works which require manual handling. After the automation, development process speeds up and the development team get rid of time-consuming operations.

The first task I was given, is in the deployment stage of CI/CD workflow. During the deployment stage of the product, each product has a specific and characteristic deployment folder. This situation results with the problem that in each deployment the folder hierarchy needs to be constructed again. So, I designed an XML format describes the hierarchy of the deployment folder and then developed I generic parsing mechanism with Python programming language to automate this process.

The second task I was given, is about the branching mechanism of Git version control system. I was expected to automate creating and then approving pull-requests. So, I wrote a script that makes requests to version control system server.

Finally, I wrote a dead branch detection code that checks the branchs whether if they are still being in use or not. If not these branchs are automatically deleted.

I present the internship report to the internship commission, in which I explain my observations during the internship and the work carried out within the scope of the given tasks.

INSTITUTION INFORMATION

Institutions;

Name	: Milsoft Software Technologies
Deparment (if it can be stated)	: DLP
Address	: MilSOFT Yazilim Teknolojileri A.S. Ihsan Dogramaci bul. Universiteler Mah. No:25 Teknokent – ODTU 06800 Cankaya-Ankara / TURKIYE
Telephone	: +90 (312) 292 30 00
E-mail	: milsoft@milsoft.com.tr
Web Page (if exists)	: www.milsoft.com.tr

MILSOFT Softfare Technologies is a %100 Turkish national firm founded in 1998. The firm operates in the fields of system integration and software development. Also, Milsoft continues to work in defense industry, with having the CMMI-5 level.

TABLE OF CONTENTS

1. INTRODUCTION

In modern application of product development, the goal is to have multiple team members working simultaneously on different parts of the same product. This means multiple team members could have multiple tasks. In order to ensure efficiency and continuity this whole process could be thought as a development life cycle. The Development Life Cycle simply outlines each task required to put together a software product. This helps to reduce waste and increase the efficiency of the development process. Monitoring also ensures the project stays on track and continues to be a feasible investment for the company. The steps of development cycle can be subdivided into smaller units but generally whole process looks like in the figure 1.

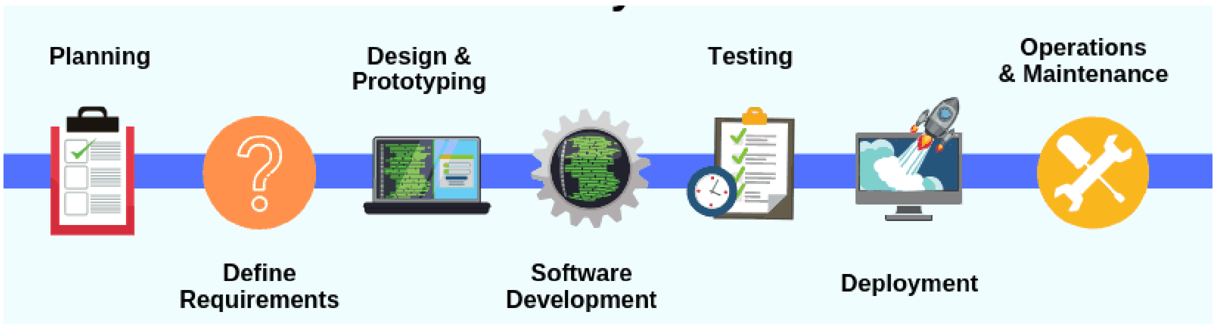


Figure 1. Stages of Development Cycle

Fine tunning the development cycle is another concept, and this is the place where CI/CD is needed. So, the general concept of my internship was making improvements in the development cycle by automating process. During the internship, the first task I performed is in the deployment stage of development cycle, second and third ones are in the integration stage of development cycle. For the first task I designed an XML format specifies the deployment folder hierarchy and then I genericly parsed this XML file. For the second task, I coded a script that automize the branching mechanism in the git flow. And lastly, I coded a script that detects and deletes the dead branchs in the given Git repository. Before getting into more details about the tasks, the general concepts and basics of the CI/CD will be covered in the next chapter.

2. ABOUT THE CI/CD

2.1. What is CI/CD?

CI/CD is a method to frequently deliver products to customers by introducing automation into the stages of development cycle. The main concepts attributed to CI/CD are continuous integration, continuous delivery, and continuous deployment. CI/CD is a solution to the problems integrating new code can cause for development and operations teams.

Specifically, CI/CD introduces ongoing automation and continuous monitoring throughout the development cycle, from integration and testing phases to delivery and deployment. Taken together, these connected practices are often referred to as a "CI/CD pipeline" and are supported by development and operations teams working together in an agile way.

2.2. What's the Difference Between CI and CD?

The acronym CI/CD has a few different meanings. The "CI" in CI/CD always refers to continuous integration, which is an automation process for developers. Successful CI means new code changes to an app are regularly built, tested, and merged to a shared repository. It's a solution to the problem of having too many branches of an app in development at once that might conflict with each other.

The "CD" in CI/CD refers to continuous delivery and/or continuous deployment, which are related concepts that sometimes get used interchangeably. Both are about automating further stages of the pipeline, but they're sometimes used separately to illustrate just how much automation is happening.

Continuous delivery usually means a developer's changes to an application are automatically bug tested and uploaded to a repository (like GitHub or a container registry), where they can then be deployed to a live production environment by the operations team. It's an answer to the problem of poor visibility and communication between dev and business teams. To that end, the purpose of continuous delivery is to ensure that it takes minimal effort to deploy new code.

Continuous deployment (the other possible "CD") can refer to automatically releasing a developer's changes from the repository to production, where it is usable by customers. It addresses the problem of overloading operations teams with manual processes that slow down app delivery. It builds on the benefits of continuous delivery by automating the next stage in the pipeline.

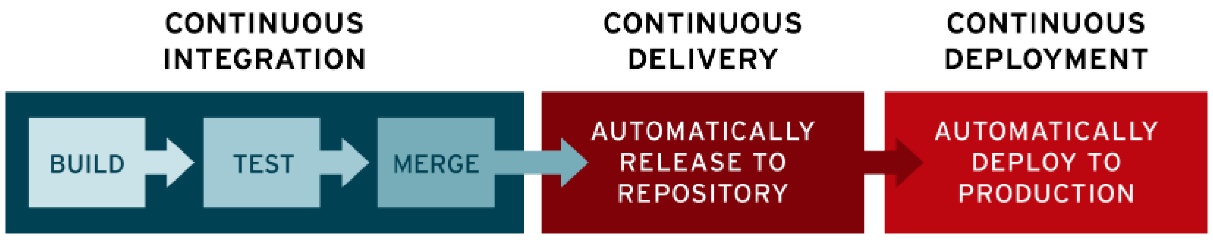


Figure 2. Continuous Integration/Delivery/Deployment

It's possible for CI/CD to specify just the connected practices of continuous integration and continuous delivery, or it can also mean all 3 connected practices of continuous integration, continuous delivery, and continuous deployment. To make it more complicated, sometimes "continuous delivery" is used in a way that encompasses the processes of continuous deployment as well.

2.3. <u>Continuous Integration</u>

In modern application development, the goal is to have multiple developers working simultaneously on different features of the same app. However, if an organization is set up to merge all branching source code together on one day (known as "merge day"), the resulting work can be tedious, manual, and time intensive. That's because when a developer working in isolation makes a change to an application, there's a chance it will conflict with different changes being simultaneously made by other developers. This problem can be further compounded if each developer has customized their own local integrated development environment (IDE), rather than the team agreeing on one cloud-based IDE.

Continuous integration (CI) helps developers merge their code changes back to a shared branch, or "trunk," more frequently—sometimes even daily. Once a developer's changes to an application are merged, those changes are validated by automatically building the application and running different levels of automated testing, typically unit and integration tests, to ensure the changes haven't broken the app. This means testing everything from classes and function to the different modules that comprise the entire app. If automated testing discovers a conflict between new and existing code, CI makes it easier to fix those bugs quickly and often.

2.4. Continuous Delivery

Following the automation of builds and unit and integration testing in CI, continuous delivery automates the release of that validated code to a repository. So, in order to have an effective continuous delivery process, it's important that CI is already built into your development pipeline. The goal of continuous delivery is to have a codebase that is always ready for deployment to a production environment.

In continuous delivery, every stage—from the merger of code changes to the delivery of production-ready builds—involves test automation and code release automation. At the end of that process, the operations team is able to deploy an app to production quickly and easily.

2.5. Continuous Deployment

The final stage of a mature CI/CD pipeline is continuous deployment. As an extension of continuous delivery, which automates the release of a production-ready build to a code repository, continuous deployment automates releasing an app to production. Because there is no manual gate at the stage of the pipeline before production, continuous deployment relies heavily on well-designed test automation.

In practice, continuous deployment means that a developer's change to a cloud application could go live within minutes of writing it (assuming it passes automated testing). This makes it much easier to continuously receive and incorporate user feedback. Taken together, all of these connected CI/CD practices make deployment of an application less risky, whereby it's easier to release changes to apps in small pieces, rather than all at once. There's also a lot of upfront investment, though, since automated tests will need to be written to accommodate a variety of testing and release stages in the CI/CD pipeline.

2.6. Common CI/CD Tools

CI/CD tools can help a team automate their development, deployment, and testing. Some tools specifically handle the integration (CI) side, some manage development and deployment (CD), while others specialize in continuous testing or related functions.

One of the best-known open-source tools for CI/CD is the automation server Jenkins. Jenkins is designed to handle anything from a simple CI server to a complete CD hub.

3. WORK CARRIED OUT

3.1. Automating the Packaging Script

The main motivation for this task is to automize the packaging process for the deployment of the project. The folder hierarchy for the deployment is changeable. So, during the packaging process, manual changes may be required in the packaging code. To overcome this issue, I was expected to design an XML file format to desciribe folder hierarchy. With this XML format, even though the folder hierarchy changes, the code to parse it and package, does not change.

So, to be able to achive this task, first I needed to understand what XML is and how it works. The abbrevation XML stands for Extensible Markup Language. Briefly it is a markup language and file format for storing, transmitting and reconstructing arbitrary data. It defines a set of rules for encoding documents in a format that is both human and machine readable. With the help of the XML, hierarchical structures can be constructed. Since the folder structure can be expressed in a hierarchical way, XML is suitable for this task.

So, I declared a set of rules for XML format. XML uses features called tags and attributes to decleare rules. Tags are the features generates the structure and attributes are the features that specialize the tags. So, I defined 2 tags and 3 attributes initially.

The first tag “ofu” indicates the directories. For example, consider the folder hierarchy given in the figure 3.

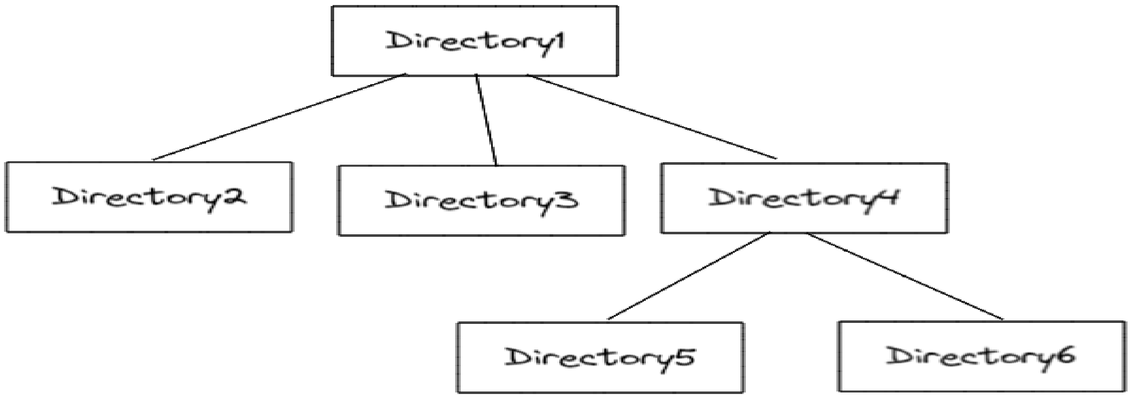
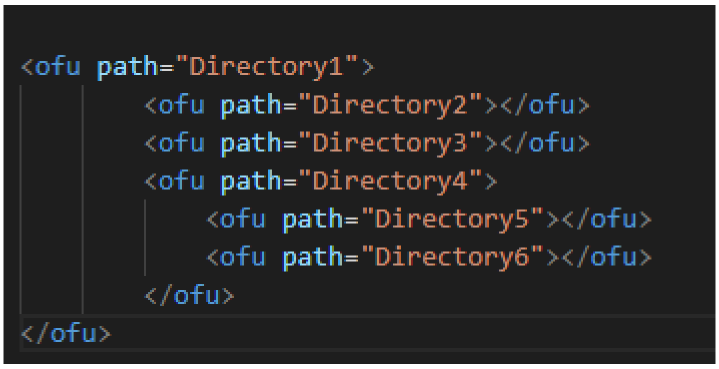


Figure 3

This hierarchy can be represented with the structure created by using “ofu” tags as given in the following figure 4.



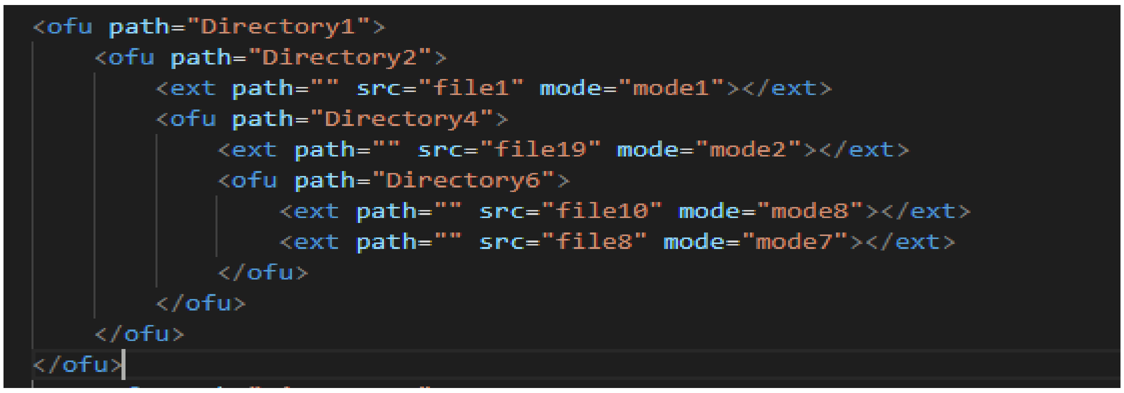
```
<ofu path="Directory1">
        <ofu path="Directory2"></ofu>
        <ofu path="Directory3"></ofu>
        <ofu path="Directory4">
            <ofu path="Directory5"></ofu>
            <ofu path="Directory6"></ofu>
        </ofu>
</ofu>
```

Figure 4

The second tag is “ext” tag and it identifies the places that requires extract operation. Extract operation is a function used later in the deployment process to move files to their appropriate positions in the hierarchy.

In the tags, I used 3 attributes: the “path”, “src” and “mode”. The first attribute path is simply the name of directories. While parsing the XML file, these attributes are added each other appropriately and absolute path is constructed. Absolute path referred here as the path from most outer folder to the desired folder. The “src” attribute indicates the places that the extraction operation moves files from. And lastly the “mode” attribute identifies the mode while file extraction happening. The attributes “src” and “mode” are only used in the “ext” tag since there is no need them in the “ofu” tag.

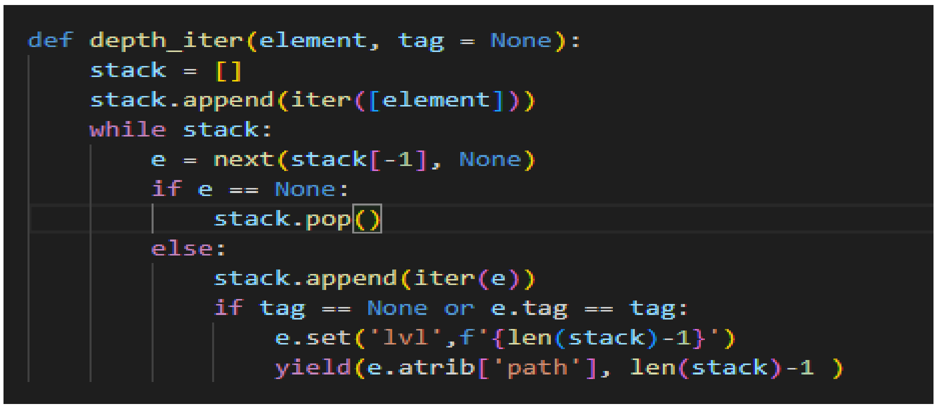
With all definitions given above, the sample XML file used for the description of folder hierarchy can be in the figure 5.



```
<ofu path="Directory1">
    <ofu path="Directory2">
        <ext path="" src="file1" mode="mode1"></ext>
        <ofu path="Directory4">
            <ext path="" src="file19" mode="mode2"></ext>
            <ofu path="Directory6">
                <ext path="" src="file10" mode="mode8"></ext>
                <ext path="" src="file8" mode="mode7"></ext>
            </ofu>
        </ofu>
    </ofu>
</ofu>
```

Figure 5

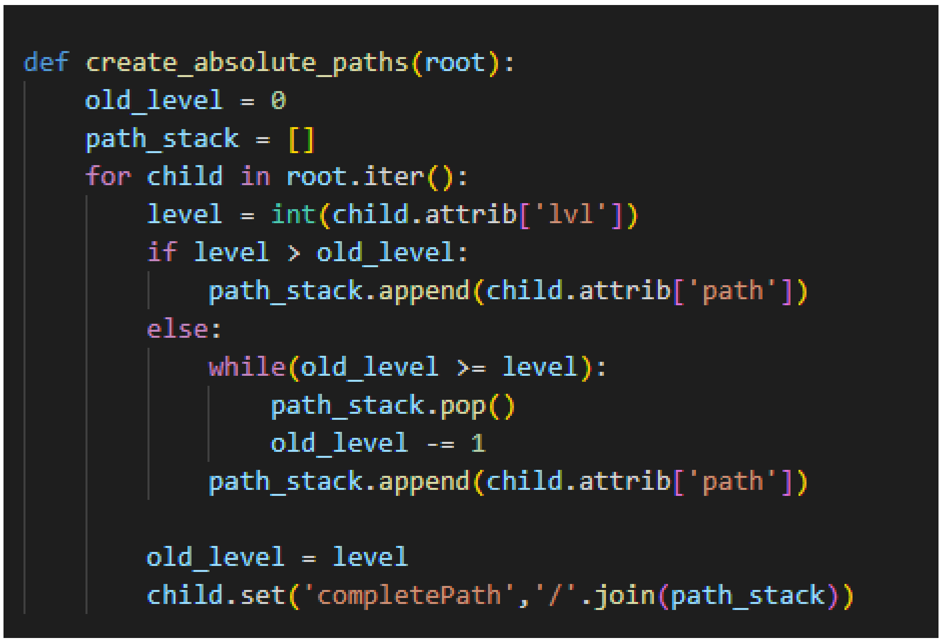
After constructing the XML format, the parsing mechanism is needed. To parse an XML file, I used Python programming language and its xml.etree library. With the help of the library, an XML file can be easily parsed into a tree data structure. Then, to create absolute paths for the “ext” tags, I wrote a traversal function. Since the tree structure returned from xml.etree library, has no connections between parent and child nodes, I walk through the tree by depth-first traversal and then create a new attribute for each node in the tree. This attribute is named “lvl” and stores the depth of the corresponding node in the tree. The function is given in the figure 6.



```python
def depth_iter(element, tag = None):
    stack = []
    stack.append(iter([element]))
    while stack:
        e = next(stack[-1], None)
        if e == None:
            stack.pop()
        else:
            stack.append(iter(e))
            if tag == None or e.tag == tag:
                e.set('lvl',f'{len(stack)-1}')
                yield(e.atrib['path'], len(stack)-1 )
```

Figure 6.

After labelling the depth of all the nodes, I created a function that finds the complete path for all the nodes. The function takes the root of the three, then starts to iterate over all the child nodes of the root. For each node and the previous node during the iteration, depth information compared. With the help of this comparison the complete paths are created for each node in the tree and set as attribute. The function definition is in the figure 7.



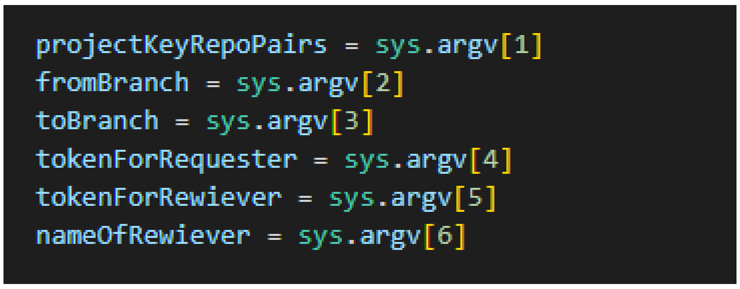
```python
def create_absolute_paths(root):
    old_level = 0
    path_stack = []
    for child in root.iter():
        level = int(child.attrib['lvl'])
        if level > old_level:
            path_stack.append(child.attrib['path'])
        else:
            while(old_level >= level):
                path_stack.pop()
                old_level -= 1
            path_stack.append(child.attrib['path'])

        old_level = level
        child.set('completePath','/'.join(path_stack))
```

Figure 7

With the last step, all the preminal preparations have been completed. We have parsed the XML file into a tree structure, we have walked through the tree and found the depths of each node, and finally we have looped over all the nodes and labelled the complete path for each node. So, the tree structure is ready for the file extraction operation and task is completed. The packaging script for deployment process, has became independent from folder hierarchy. That means, eventough the folder hierarchy changes the packaging code does not need to change. Just another XML file definition according to the given hierarchy is enough for deployment.

3.2. Auto Pull Request Creating and Approving

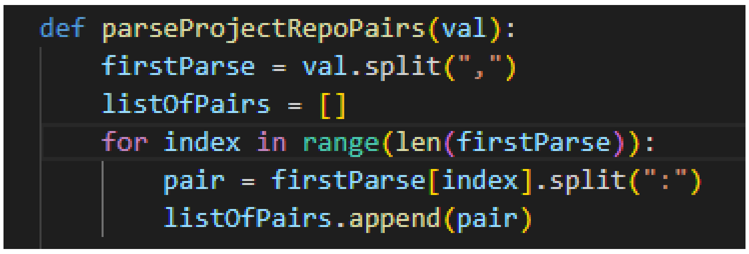
For the second task I was expected to automize pull-request creation and approval process in the git flow. To achive this task, I coded a Python script. Since the script will be entagreted to Jenkins and run on the command line, firstly, I take command line arguments which give the related information about the requests. I used the built-in “sys” Python module to take command line arguments. The code snippet in the figure 8 describes this action.



```python
projectKeyRepoPairs = sys.argv[1]
fromBranch = sys.argv[2]
toBranch = sys.argv[3]
tokenForRequester = sys.argv[4]
tokenForRewiever = sys.argv[5]
nameOfRewiever = sys.argv[6]
```

Figure 8

Project and repository pairs taken as one string, so this string needs to be parsed. To parse project and repository pairs, I defined the following function which is given in the figure 9.

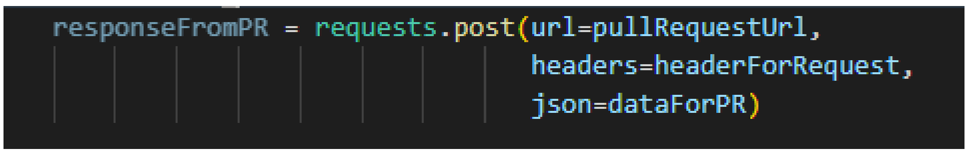


```python
def parseProjectRepoPairs(val):
    firstParse = val.split(",")
    listOfPairs = []
    for index in range(len(firstParse)):
        pair = firstParse[index].split(":")
        listOfPairs.append(pair)
```

Figure 9

To complete this parsing action, project and repository pairs should be given in specific format. Each project-repository pair should be separated with colon and if more than one pairs exist, they should be separated with comma.

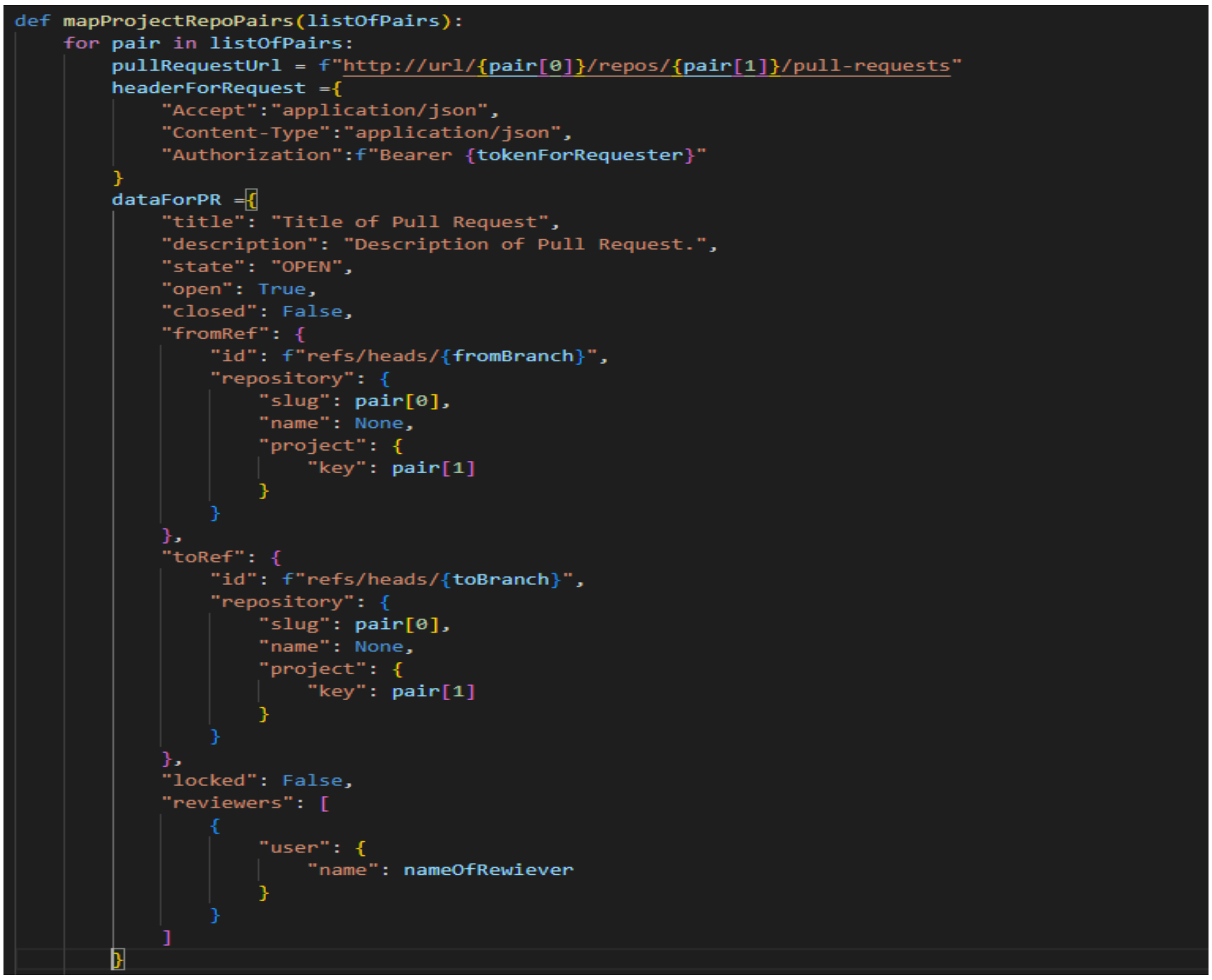
After parsing pairs, I mapped all the pairs to make requests. In order to make requests to related git hosting service’s api, I used Python’s “requests” module and its post method.



```
responseFromPR = requests.post(url=pullRequestUrl,
                               headers=headerForRequest,
                               json=dataForPR)
```

Figure 10

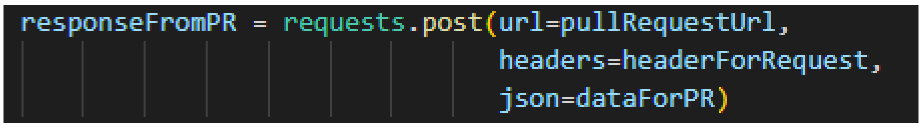
Post method takes 3 essential parameters: url, headers, and json data. Url just indicates the end point of the request. Headers defines the properties during the post action such as accepted file format or authorization method. Json data stands for the data send to the git hosting service. All of them defined with the lines in the figure 11.



```
def mapProjectRepoPairs(listOfPairs):
    for pair in listOfPairs:
        pullRequestUrl = f"http://url/{pair[0]}/repos/{pair[1]}/pull-requests"
        headerForRequest ={
            "Accept":"application/json",
            "Content-Type":"application/json",
            "Authorization":f"Bearer {tokenForRequester}"
        }
        dataForPR ={
            "title": "Title of Pull Request",
            "description": "Description of Pull Request.",
            "state": "OPEN",
            "open": True,
            "closed": False,
            "fromRef": {
                "id": f"refs/heads/{fromBranch}",
                "repository": {
                    "slug": pair[0],
                    "name": None,
                    "project": {
                        "key": pair[1]
                    }
                }
            },
            "toRef": {
                "id": f"refs/heads/{toBranch}",
                "repository": {
                    "slug": pair[0],
                    "name": None,
                    "project": {
                        "key": pair[1]
                    }
                }
            },
            "locked": False,
            "reviewers": [
                {
                    "user": {
                        "name": nameOfRewiever
                    }
                }
            ]
        }
```

Figure 11

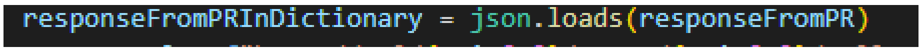
After these declerations, all parameters are satisfied to make post request. Post request sended to the git hosting service's api with the code in the figure 12.



```
responseFromPR = requests.post(url=pullRequestUrl,
                               headers=headerForRequest,
                               json=dataForPR)
```

Figure 12

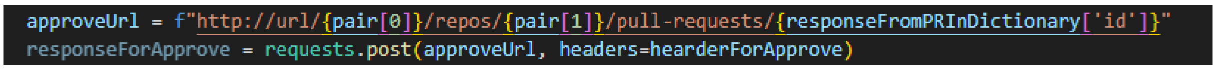
The response from the api is stored in a variable but response is in the JSON format. So, to manipulate response, it is first converted in the Python dictionary format with the help of Python's built-in "json" module.



```
responseFromPRInDictionary = json.loads(responseFromPR)
```

Figure 13

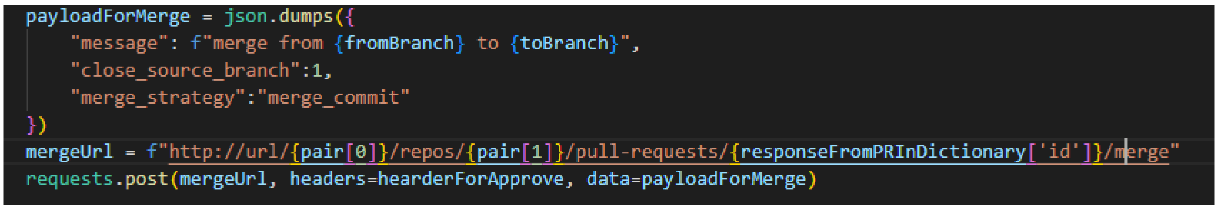
With this last step, pull-request is created successfully. Next step is sending the post request that approves the just created pull-request. To approve the pull-request, the id of the pull-request is required. So, I extracted the id of the created pull-request from the previous post request's response. Then, another post request is sent with the help of "requests" module.



```
approveUrl = f"http://url/{pair[0]}/repos/{pair[1]}/pull-requests/{responseFromPRInDictionary['id']}"
responseForApprove = requests.post(approveUrl, headers=hearderForApprove)
```

Figure 14

As the last step, after approving the pull-request, it needs to be merged. To merge pull-request into the desired branch, one last post request is sent to the git hosting service's api. In the figure 15, the declerations of data and url to send the post request are given. With this last step, automizing the creation of pull-request and approving process is completed. The script maps all the given project-repository pairs, creates a pull-request for the given branch in these repositories and after approving pull-request it merges the branch.



```python
payloadForMerge = json.dumps({
    "message": f"merge from {fromBranch} to {toBranch}",
    "close_source_branch":1,
    "merge_strategy":"merge_commit"
})
mergeUrl = f"http://url/{pair[0]}/repos/{pair[1]}/pull-requests/{responseFromPRInDictionary['id']}/merge"
requests.post(mergeUrl, headers=hearderForApprove, data=payloadForMerge)
```

Figure 15

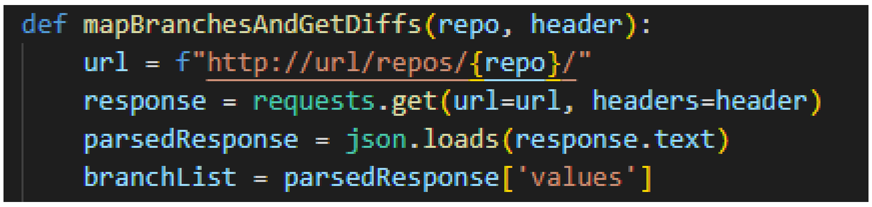
3.3. Dead Branch Detection

The last task I was given, is detecting, and deleting the branchs that are not used anymore. Eventough, deleting these branchs is not necessarily universal, it is good practice to keep git flow clean and understandable. Two main reasons for deleting these branches are as follows:

- They're unnecessary. In most cases, branches, especially branches that were related to a pull request that has since been accepted, serve no purpose.
- They're clutter. They don't add any significant technical overhead, but they make it more difficult for humans to work with lists of branches in the repository.

For this task, I coded a Python script that maps all the branches in the given repository and than compares their latest commit with the master branch's latest commit. According to the comparison, if the branch is detected as dead branch which means it is not used anymore, the branch is safe to delete.

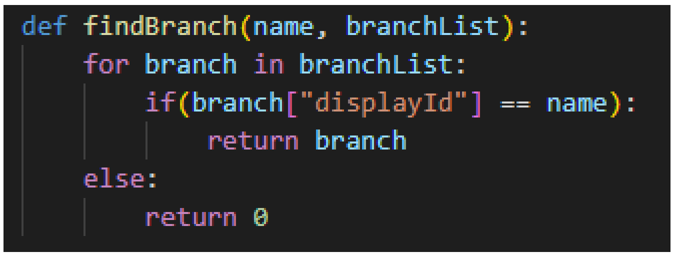
Firstly, a get request sent to the git hosting service's api, in order to get all the existing branchs in the given repository. To do that, again an url and header for the get request are needed. The lines given in the figure 16, send the get request, parse the response, and then extract the branchs into a list.



```python
def mapBranchesAndGetDiffs(repo, header):
    url = f"http://url/repos/{repo}/"
    response = requests.get(url=url, headers=header)
    parsedResponse = json.loads(response.text)
    branchList = parsedResponse['values']
```

Figure 16

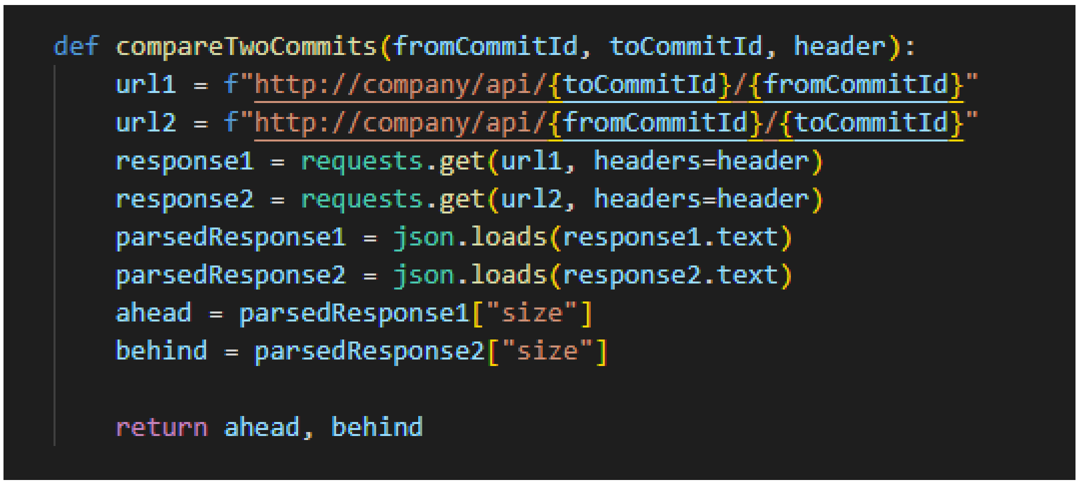
Next step is finding the master and development branchs. The code snippet given in the   figure 17, searchs the branch list to find desired branch.



```python
def findBranch(name, branchList):
    for branch in branchList:
        if(branch["displayId"] == name):
            return branch
    else:
        return 0
```

Figure 17

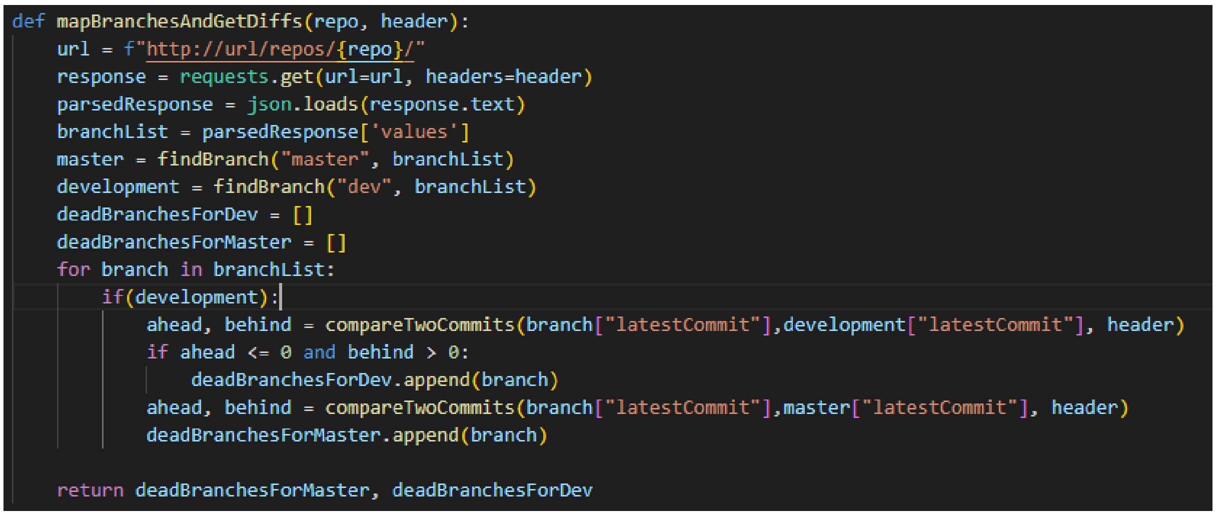
After finding the master and development branchs, each branch in the branch list is compared with them. The comparision is done by getting difference between last commits of the branches. To compare two commits, a get request send to the git hosting service's api. The function given in the figure 18, sends the requests and compares the two commits.



```python
def compareTwoCommits(fromCommitId, toCommitId, header):
    url1 = f"http://company/api/{toCommitId}/{fromCommitId}"
    url2 = f"http://company/api/{fromCommitId}/{toCommitId}"
    response1 = requests.get(url1, headers=header)
    response2 = requests.get(url2, headers=header)
    parsedResponse1 = json.loads(response1.text)
    parsedResponse2 = json.loads(response2.text)
    ahead = parsedResponse1["size"]
    behind = parsedResponse2["size"]

    return ahead, behind
```

Figure 18

The returned response from the api, indicates the number of commits that second commit in the url ahead of the first commit in the url. If we swap the commits place in the url, then we get the number of commits that the compared branch is behind of the master or development branch. With the help of these two information, it is possible detect that if the compared branch is still used or not. For example, if the number of commits that compared branch is ahead of the master branch, is less than zero and the number of commits that master branch is ahead of the compared branch, is greater than zero that means the compared branch is dead branch. So, the branch is added to the dead branchs list. All the process, is applied with the code snippet given in the figure 19.



```python
def mapBranchesAndGetDiffs(repo, header):
    url = f"http://url/repos/{repo}/"
    response = requests.get(url=url, headers=header)
    parsedResponse = json.loads(response.text)
    branchList = parsedResponse['values']
    master = findBranch("master", branchList)
    development = findBranch("dev", branchList)
    deadBranchesForDev = []
    deadBranchesForMaster = []
    for branch in branchList:
        if(development):
            ahead, behind = compareTwoCommits(branch["latestCommit"],development["latestCommit"], header)
            if ahead <= 0 and behind > 0:
                deadBranchesForDev.append(branch)
            ahead, behind = compareTwoCommits(branch["latestCommit"],master["latestCommit"], header)
            deadBranchesForMaster.append(branch)

    return deadBranchesForMaster, deadBranchesForDev
```

Figure 19

With this Python script, dead branch detection over the all branchs in the given repository, can be done automatically without any manual operation. That means, CI/CD is improved in the manner of automation.

4. CONCLUSION

The main goal and the scope of the work carried out during my internship, was to contribute the CI/CD cycle. And, I can surely say that the tasks I completed, made a considerable contribution to the CI/CD cycle. Beside that, I learned the details of CI/CD cycle.

Also, this internship was the first steps into my working life. Milsoft Software Technologies was a great company to observe professional life in the defense industry. I had doubts about working in the defense industry before my internship, but Milsoft Software Technologies cleared all my doubts. There exists a very friendly and welcoming working environment in the Milsoft Software Technologies. My supervisor and all the other engineers supported me during the internship and shared their experiences with me. Especially my supervisor paid attention that I had a productive and fulfilled internship. So, I can clearly say that my internship was very productive, and I gained lots of technical and professional experience.

I present the internship report to the internship commission, in which I explain my observations during the internship and the work carried out within the scope of the given tasks.

BIBLIOGRAPHY

Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, Massimiliano Di Penta (2021). *CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study*

https://developer.atlassian.com/cloud/bitbucket/rest/intro/

Red Hat, 2022, What is CI/CD, https://www.redhat.com/en/topics/devops/what-is-ci-cd, 22.07.2022

https://bitbucket.org/product/features/pipelines

https://requests.readthedocs.io/en/latest/

https://docs.python.org/3/library/xml.etree.elementtree.html

https://docs.python.org/3/using/cmdline.html

https://www.milsoft.com.tr/index.php/about/

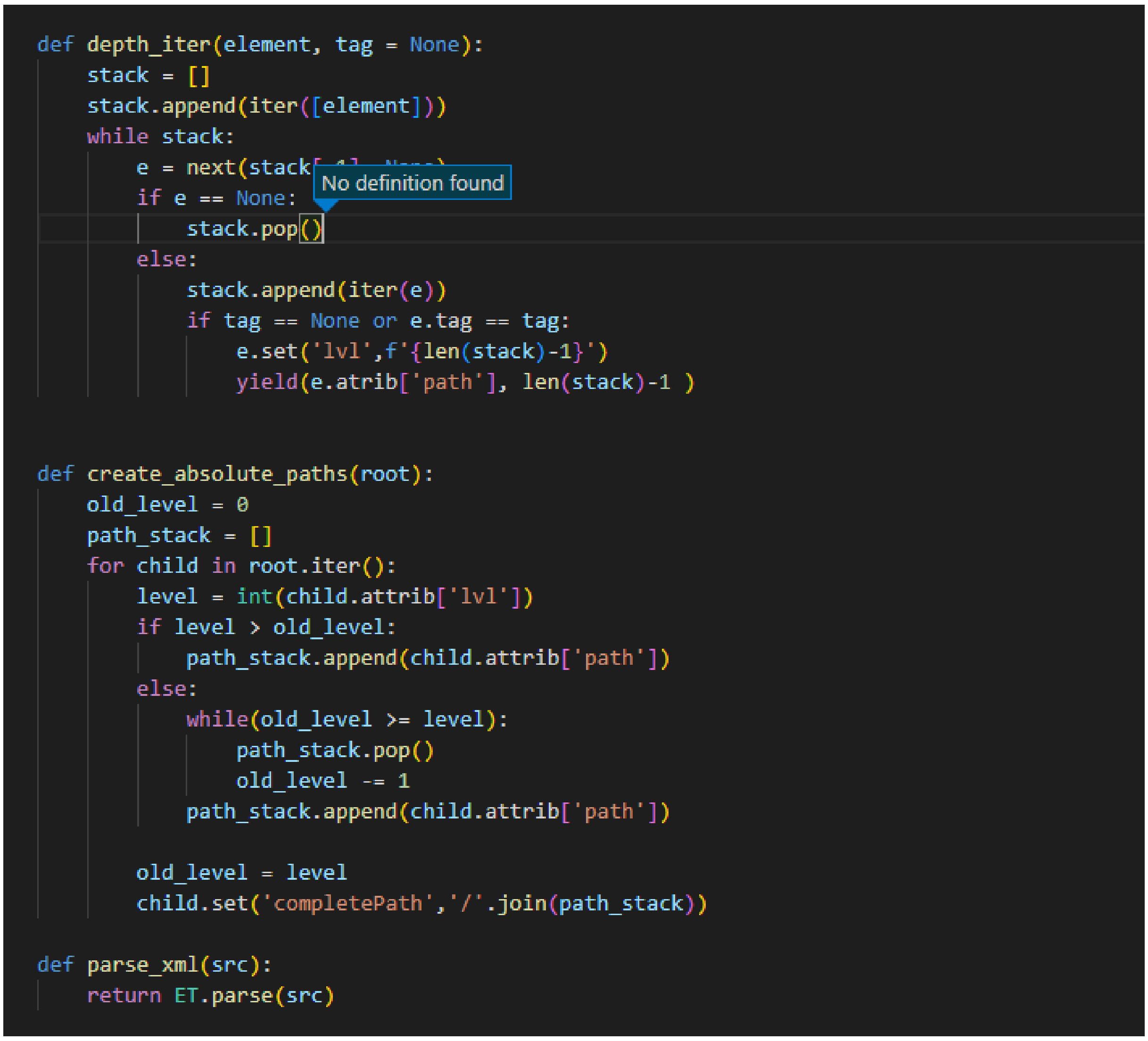
https://docs.python.org/3/library/json.html

AWS, 2021, What is DevOps, https://aws.amazon.com/devops/what-is-devops/, 21.07.2022

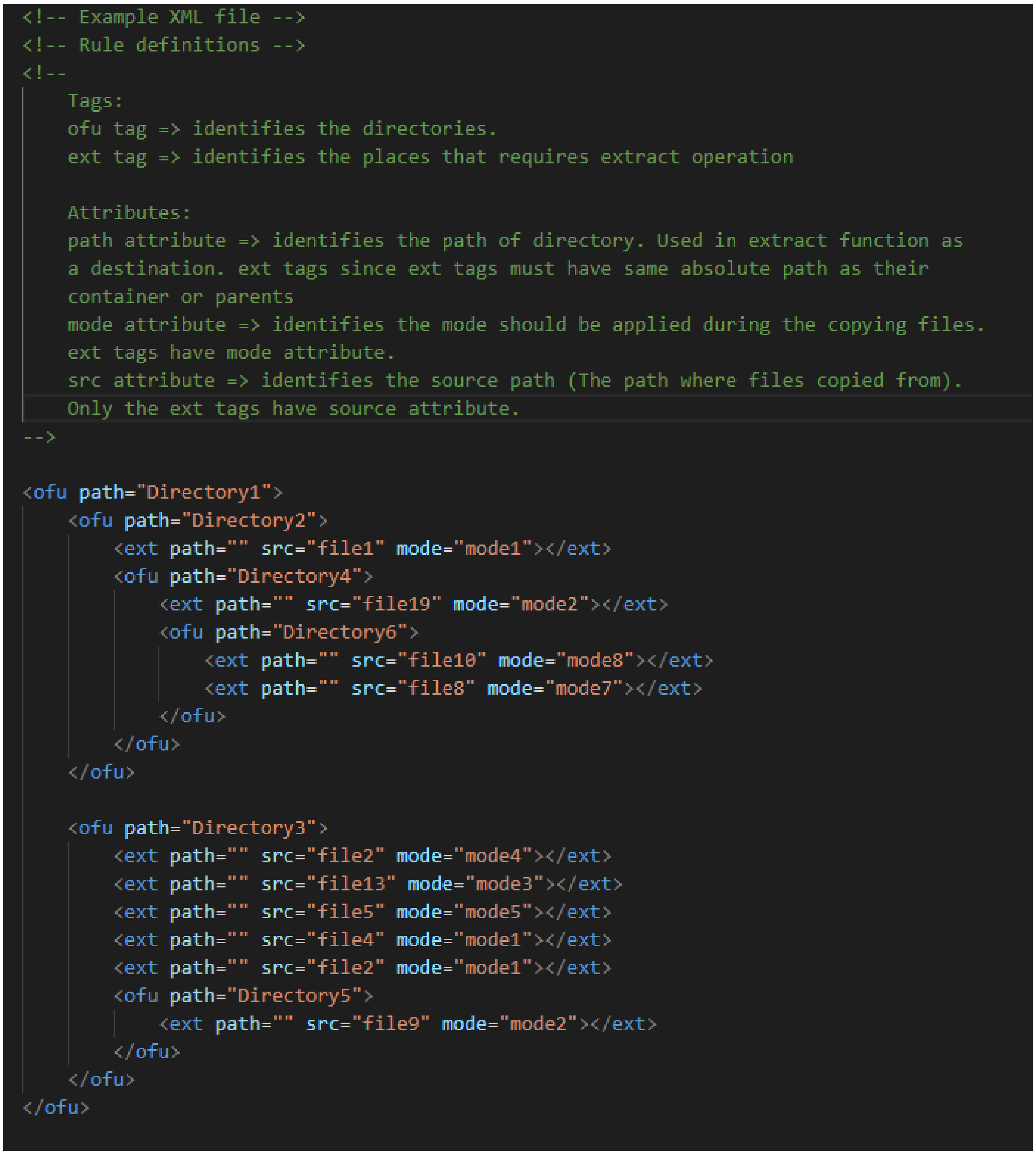
Thomas Hamilton, 2022, Jenkins Tutorial for Begginers, https://www.guru99.com/jenkins-tutorial.html, 08.08.2022 https://en.wikipedia.org/wiki/CI/CD

Mozilla, 2022, XML Introduction, https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction, 03.08.2022

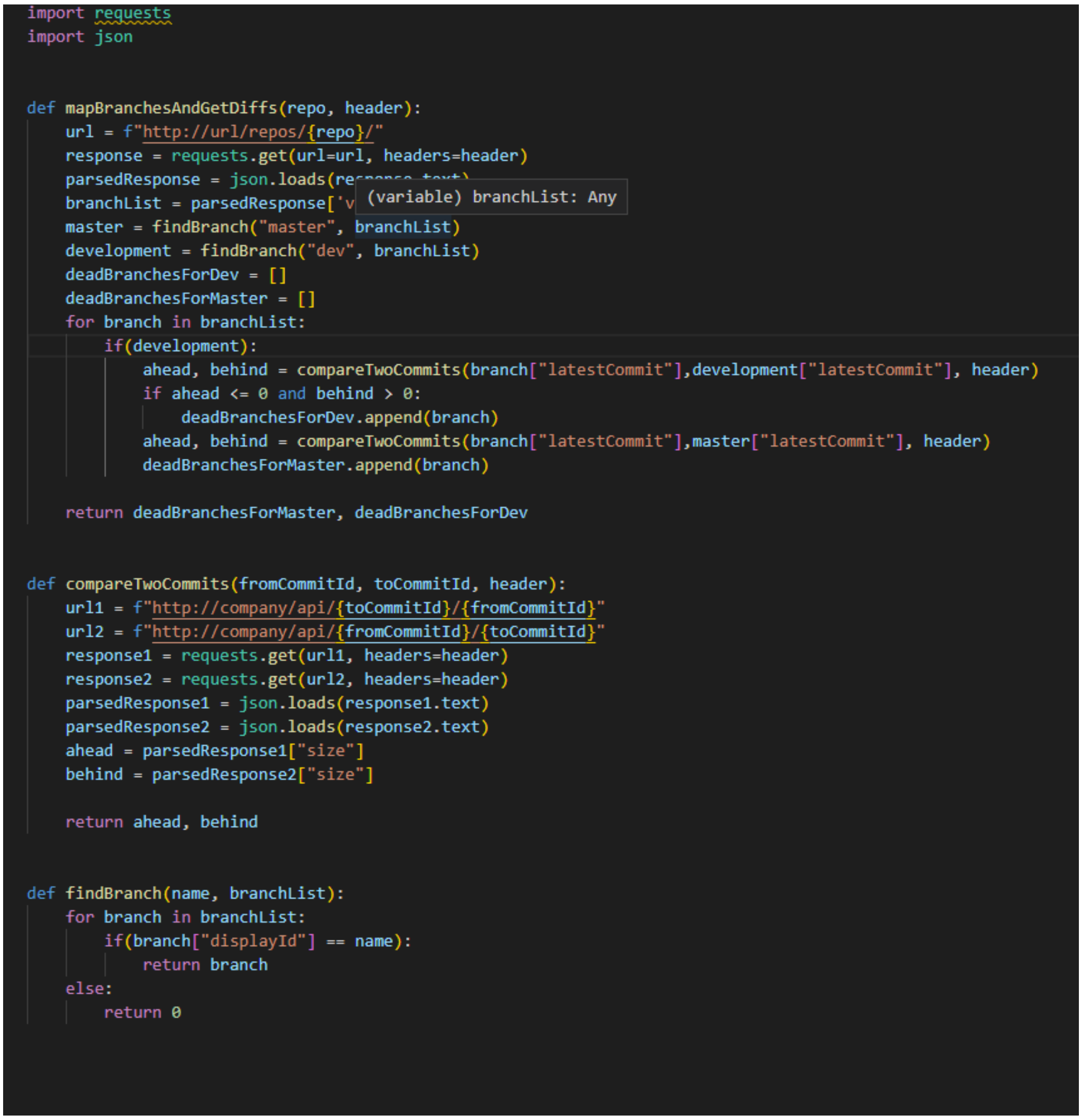
Appendix 1. The Codes Carried Out



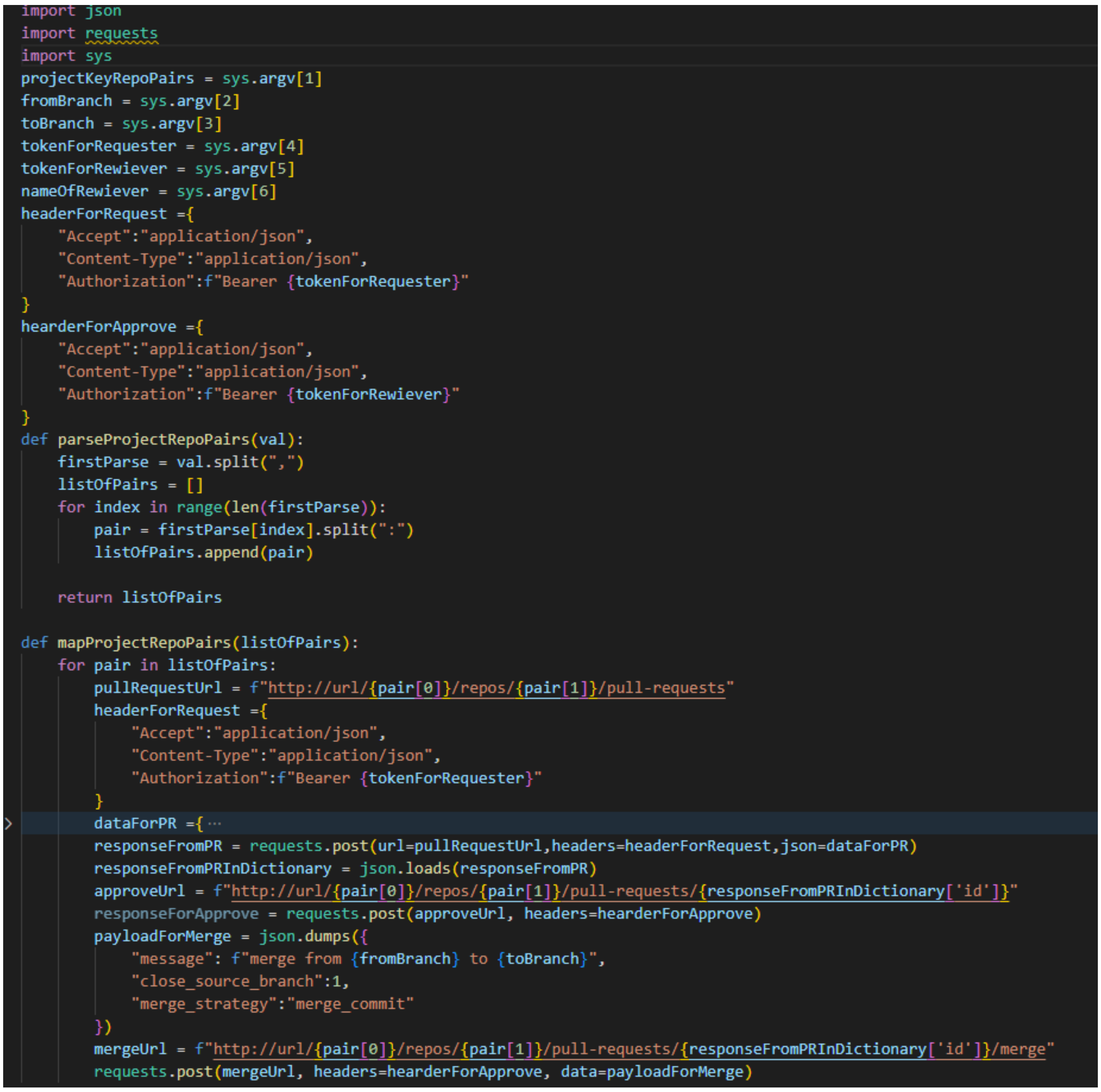
```python
def depth_iter(element, tag = None):
    stack = []
    stack.append(iter([element]))
    while stack:
        e = next(stack[-1], None)
        if e == None:
            stack.pop()
        else:
            stack.append(iter(e))
            if tag == None or e.tag == tag:
                e.set('lvl',f'{len(stack)-1}')
                yield(e.atrib['path'], len(stack)-1 )


def create_absolute_paths(root):
    old_level = 0
    path_stack = []
    for child in root.iter():
        level = int(child.attrib['lvl'])
        if level > old_level:
            path_stack.append(child.attrib['path'])
        else:
            while(old_level >= level):
                path_stack.pop()
                old_level -= 1
            path_stack.append(child.attrib['path'])

        old_level = level
        child.set('completePath','/'.join(path_stack))

def parse_xml(src):
    return ET.parse(src)
```



```xml
<!-- Example XML file -->
<!-- Rule definitions -->
<!--
    Tags:
    ofu tag => identifies the directories.
    ext tag => identifies the places that requires extract operation

    Attributes:
    path attribute => identifies the path of directory. Used in extract function as
    a destination. ext tags since ext tags must have same absolute path as their
    container or parents
    mode attribute => identifies the mode should be applied during the copying files.
    ext tags have mode attribute.
    src attribute => identifies the source path (The path where files copied from).
    Only the ext tags have source attribute.
-->

<ofu path="Directory1">
    <ofu path="Directory2">
        <ext path="" src="file1" mode="mode1"></ext>
        <ofu path="Directory4">
            <ext path="" src="file19" mode="mode2"></ext>
            <ofu path="Directory6">
                <ext path="" src="file10" mode="mode8"></ext>
                <ext path="" src="file8" mode="mode7"></ext>
            </ofu>
        </ofu>
    </ofu>

    <ofu path="Directory3">
        <ext path="" src="file2" mode="mode4"></ext>
        <ext path="" src="file13" mode="mode3"></ext>
        <ext path="" src="file5" mode="mode5"></ext>
        <ext path="" src="file4" mode="mode1"></ext>
        <ext path="" src="file2" mode="mode1"></ext>
        <ofu path="Directory5">
            <ext path="" src="file9" mode="mode2"></ext>
        </ofu>
    </ofu>
</ofu>
```



```python
import requests
import json


def mapBranchesAndGetDiffs(repo, header):
    url = f"http://url/repos/{repo}/"
    response = requests.get(url=url, headers=header)
    parsedResponse = json.loads(response.text)
    branchList = parsedResponse['v (variable) branchList: Any
    master = findBranch("master", branchList)
    development = findBranch("dev", branchList)
    deadBranchesForDev = []
    deadBranchesForMaster = []
    for branch in branchList:
        if(development):
            ahead, behind = compareTwoCommits(branch["latestCommit"],development["latestCommit"], header)
            if ahead <= 0 and behind > 0:
                deadBranchesForDev.append(branch)
            ahead, behind = compareTwoCommits(branch["latestCommit"],master["latestCommit"], header)
            deadBranchesForMaster.append(branch)

    return deadBranchesForMaster, deadBranchesForDev


def compareTwoCommits(fromCommitId, toCommitId, header):
    url1 = f"http://company/api/{toCommitId}/{fromCommitId}"
    url2 = f"http://company/api/{fromCommitId}/{toCommitId}"
    response1 = requests.get(url1, headers=header)
    response2 = requests.get(url2, headers=header)
    parsedResponse1 = json.loads(response1.text)
    parsedResponse2 = json.loads(response2.text)
    ahead = parsedResponse1["size"]
    behind = parsedResponse2["size"]

    return ahead, behind


def findBranch(name, branchList):
    for branch in branchList:
        if(branch["displayId"] == name):
            return branch
    else:
        return 0
```



```python
import json
import requests
import sys
projectKeyRepoPairs = sys.argv[1]
fromBranch = sys.argv[2]
toBranch = sys.argv[3]
tokenForRequester = sys.argv[4]
tokenForRewiever = sys.argv[5]
nameOfRewiever = sys.argv[6]
headerForRequest ={
    "Accept":"application/json",
    "Content-Type":"application/json",
    "Authorization":f"Bearer {tokenForRequester}"
}
hearderForApprove ={
    "Accept":"application/json",
    "Content-Type":"application/json",
    "Authorization":f"Bearer {tokenForRewiever}"
}
def parseProjectRepoPairs(val):
    firstParse = val.split(",")
    listOfPairs = []
    for index in range(len(firstParse)):
        pair = firstParse[index].split(":")
        listOfPairs.append(pair)

    return listOfPairs

def mapProjectRepoPairs(listOfPairs):
    for pair in listOfPairs:
        pullRequestUrl = f"http://url/{pair[0]}/repos/{pair[1]}/pull-requests"
        headerForRequest ={
            "Accept":"application/json",
            "Content-Type":"application/json",
            "Authorization":f"Bearer {tokenForRequester}"
        }
        dataForPR ={···
        responseFromPR = requests.post(url=pullRequestUrl,headers=headerForRequest,json=dataForPR)
        responseFromPRInDictionary = json.loads(responseFromPR)
        approveUrl = f"http://url/{pair[0]}/repos/{pair[1]}/pull-requests/{responseFromPRInDictionary['id']}"
        responseForApprove = requests.post(approveUrl, headers=hearderForApprove)
        payloadForMerge = json.dumps({
            "message": f"merge from {fromBranch} to {toBranch}",
            "close_source_branch":1,
            "merge_strategy":"merge_commit"
        })
        mergeUrl = f"http://url/{pair[0]}/repos/{pair[1]}/pull-requests/{responseFromPRInDictionary['id']}/merge"
        requests.post(mergeUrl, headers=hearderForApprove, data=payloadForMerge)
```