Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Faculty 4
**School of Computing,
Communication and Business**

Project

Robert Dolibog
Author

DevOps and Site Reliability Engineering
Course

Automated build and Testing Pipeline for a 3D Unity Project
Title

Abstract:

This project aims to develop an automated build and testing pipeline for a large 3D Unity project, made for the IMI showtime this semester. The Unity Project requires multiple gb of storage, due to a new 3D scanning method being used, Thus Git LFS enhancement is necessary. The central part of the project will be the Continuous Integration (CI) on a server, such as Jenkins, which will automate the Unity build process using scripts that interact with Unity's CLI. After the CI pipeline has executed the process will go into automated tests to ensure the health of the build. This process will streamline the development and maintain high quality builds. To further the streamlined approach a branching workflow inspired by git flow will be put into place. In the documentation I will go into detail about the challenges faced working the pipeline out, and deploying a branching workflow in a team of 6.

Introduction:

Disclaimer:
I sadly couldn't make the whole repository accessible as the repository size still exceeded the limit set by GitHub. To compensate, I provided examples and directions to the files when needed. Further I uploaded only the Ci necessary files to the project repository to give at least access to them. The directions are made in the following format: (project/screenshots/GitLabStorageUsage.png)

This project was about simplifying the development process for a large 3D Unity game that my IMI Showtime group and I were creating. Although the Unity project was a team effort, I took on the responsibility of setting up the Continuous Integration (CI) pipeline by myself. Initially, we had ambitious plans to incorporate a new 3D scanning technology, but due to difficulties in coordinating with the service provider, we had to adjust our approach. We used a different realistic avatar as our replacement, and used motion capture to animate the avatar. Thus we still generated a lot of high storage files (.fbx)

Setting up the CI pipeline was my first deep dive into automating the build process for a Unity project. The challenge was not just about learning the technical aspects of GitLabs Runner/ Pipeline integration, Debugging and scripting for automation but also about managing and optimizing the large amounts of data our project was producing.

One of the key aspects of this project for me was learning to work within a coding team's structure. Although the Unity project was collaborative, the CI pipeline was a solo venture. This dual role allowed me to see both the collaborative and individual contributions to software development. I guided my teammates through using Git for version control, facilitating a workflow that allowed us to work on different parts of the project simultaneously and efficiently.

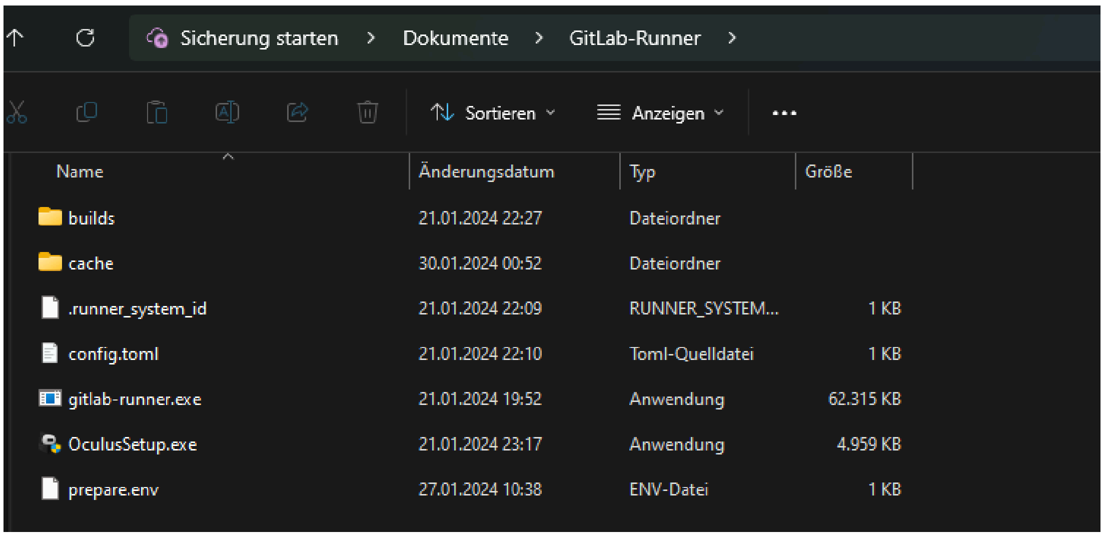
This document will cover the journey from the initial challenge of adjusting our project plans to successfully implementing an automated CI pipeline. It will detail the technical hurdles, and the solutions I found working through the project.

Even though I have no plans to pursue a career specifically in game development, this experience was valuable. It provided me with a practical understanding of CI pipelines and the dynamics of teamwork in software development. The project wasn't just about building a game; it was about discovering how to make the process smoother, learning the value of automation in development, and enhancing my skills in problem-solving and project management.

Methodology and Implementation:

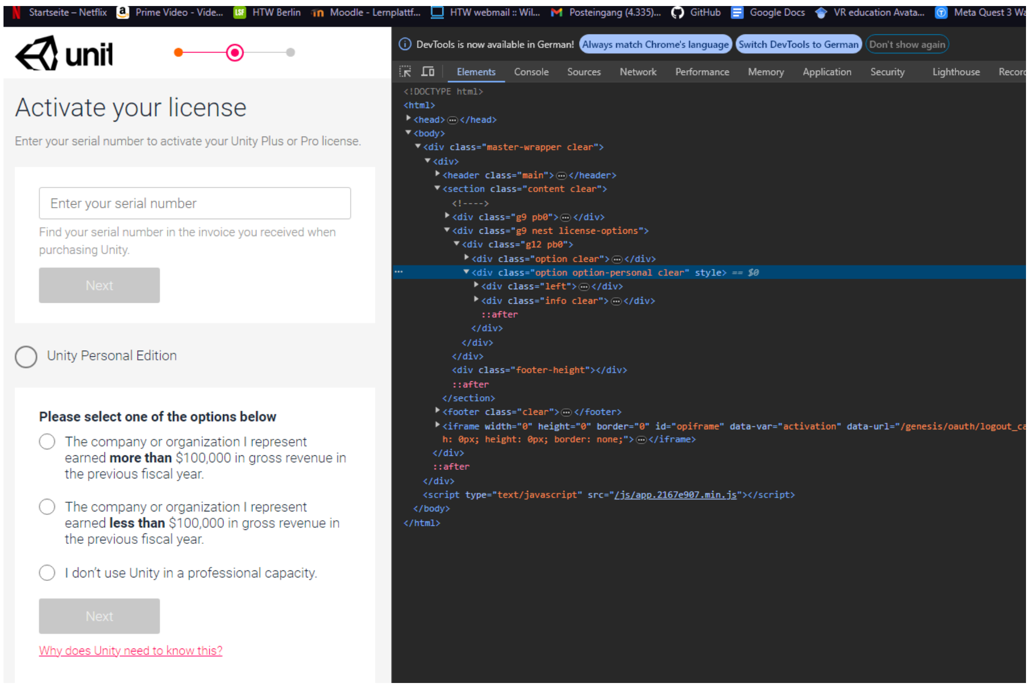
Now come to the methods used and implementation of those. The CI pipeline for our Unity project involved a large learning curve, where the documentation by Game CI served as guidance. (Game CI, n.d.) They provided me with a great starting point and gave great insight and tools to work with, like their docker unity editor image, as well as their code examples. But before I could go into the coding I first had to acquaint myself with GitLab's CI/CD documentation, which laid the groundwork for understanding the complexities and potential of the CI/CD process.(Game CI, n.d.)

To put this to practice, I began with the implementation of a GitLab runner. This is a program that is installed either locally or server side and allows code from a gitlab repository to be built, this is then orchestrated by the git-lab.yml file.(project/git-lab.yml) My first attempt was to use a Docker runner, which seemed ideal in theory. However, it quickly became clear that the Docker runner was not ideal for the demands of our large Unity project. After trying to implement it this way I ran into severe performance bottlenecks. This led me to a shift to a Windows runner. This helped the performance issues, but introduced a new challenge: the need to translate all my existing Linux-based scripts to their Windows PowerShell counterparts. This translation was not straightforward and consumed a considerable amount of time and effort, as well as a lot of debugging on the server side.
(project/scripts)



Name	Änderungsdatum	Typ	Größe
builds	21.01.2024 22:27	Dateiordner	
cache	30.01.2024 00:52	Dateiordner	
.runner_system_id	21.01.2024 22:09	RUNNER_SYSTEM...	1 KB
config.toml	21.01.2024 22:10	Toml-Quelldatei	1 KB
gitlab-runner.exe	21.01.2024 19:52	Anwendung	62.315 KB
OculusSetup.exe	21.01.2024 23:17	Anwendung	4.959 KB
prepare.env	27.01.2024 10:38	ENV-Datei	1 KB

An important task was integrating GitLab secrets into our pipeline to manage the Unity licensing. Unity's system tried to actively prevent the download of the personal license file, a .ulf, which was necessary for authentication in my building process. To still access the file, I looked into the HTML code of Unity's licensing page, disabling the hidden elements to access the download of the license file manually.



Activate your license

Enter your serial number to activate your Unity Plus or Pro license.

Enter your serial number

Find your serial number in the invoice you received when purchasing Unity.

Next

Unity Personal Edition

Please select one of the options below

The company or organization I represent earned **more than** $100,000 in gross revenue in the previous fiscal year.

The company or organization I represent earned **less than** $100,000 in gross revenue in the previous fiscal year.

I don't use Unity in a professional capacity.

Next

Why does Unity need to know this?

```
<!DOCTYPE html>
<html>
 ▶<head> ⋯ </head>
 ▼<body>
   ▼<div class="master-wrapper clear">
     ▼<div>
       ▶<header class="main"> ⋯ </header>
       ▼<section class="content clear">
          <!---->
         ▶<div class="g9 pb0"> ⋯ </div>
         ▼<div class="g9 nest license-options">
           ▼<div class="g12 pb0">
             ▶<div class="option clear"> ⋯ </div>
             ▼<div class="option option-personal clear" style> == $0
               ▶<div class="left"> ⋯ </div>
               ▶<div class="info clear"> ⋯ </div>
                ::after
              </div>
            </div>
          </div>
          <div class="footer-height"></div>
          ::after
        </section>
       ▶<footer class="clear"> ⋯ </footer>
       ▶<iframe width="0" height="0" border="0" id="opiframe" data-var="activation" data-url="/genesis/oauth/logout_ca
        h: 0px; height: 0px; border: none;"> ⋯ </iframe>
      </div>
      ::after
    </div>
    <script type="text/javascript" src="/js/app.2167e907.min.js"></script>
  </body>
</html>
```

With the CI pipeline's structure in place, testing revealed various logic errors across multiple files. An example was the absence of an exit condition for the Unity game builds, which I resolved by implementing a check within the UnityBuild.log(project/UnityBuild.log)
file for a specific success message. This step was vital to ensuring that the pipeline would only return passed when the build was actually finished

Code-excerpt success handling build.ps1:

```
do {
   Start-Sleep -Seconds 30 # retry every 30 seconds
   $buildSuccess = Get-Content $unityLogPath | Select-String -Pattern
$buildSuccessMessage -Quiet

   if ($buildSuccess) {
      Write-Host "Build completed successfully."
```

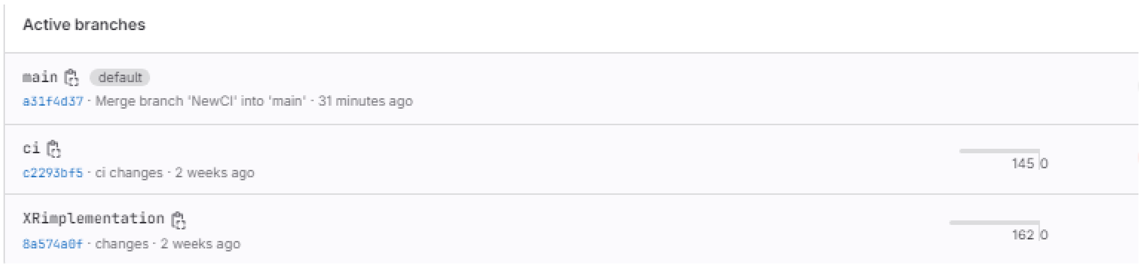
```
        break
    }

    $currentTime = Get-Date
    $elapsed = ($currentTime - $startTime).TotalSeconds
    if ($elapsed -gt $timeout) {
        throw "Build process timed out."
    }
} while ($true)

# Check build result
switch ($UNITY_EXIT_CODE) {
    0 { Write-Host "Run succeeded, no failures occurred" }
    2 { Write-Host "Run succeeded, some tests failed" }
    3 { Write-Host "Run failure (other failure)" }
    default { Write-Host "Unexpected exit code $UNITY_EXIT_CODE" }
}
```

This was the final version of the code and worked quite well, but to come to this point I also had to do a lot of debugging. A good example is an error that came up when building the project. I used the GitLAb Jobs feature to get real time updates on the state of the building process. In this documentation I got the error that the .log file was not found. This message then led me to understand that the scripts were executing faster than the .log file was taking to be created. This led me to implement the sleep for 30 seconds step. This also helped the performance as the script didn't have to check on every tick, as the file took 7-10 minutes to be completed. (project/screenshots/unsuccessfulPipeline.log)

When it came to dealing with the CLI of Unity I used the example file given by Game.ci. The BuildCommand.cs file contains all the necessary logic for dealing with the CLI and gives an Interface for building Unity projects. I used this file in conjunction with my build.ps1 file to build the game with specified Build options defined by variables. (Game CI, n.d.) (project/CLI_script)

Now coming back to the actual build pipeline. This process took initially more than 23 minutes to complete a singular build. This was way too long of a duration, to be practically applicable. To address this, I installed GitLab's caching mechanism, which promised to reduce build times significantly. After looking into it myself through GitLabs caching documentation, I successfully integrated it into the pipeline, as well as storing it locally to be used on every new build. This cut the build time down to only 7 minutes. (project/screenshots/successfulPipeline_optimized.png)

Now come to another challenging part of this project. Bringing a git workflow structure into our Showtime team. The team I worked with had strongly varying levels of experience when it came to working from and with git. This was a problem as before I could start implementing a branching workflow I had to tutor some members on the in and outs of Git. After the basics were understood, it came down to actually working with the remote repository. Despite the initial coaching, the team still faced challenges with Git, leading to frequent merge errors. Recognizing that a full Git-flow structure was too complex in this fast-paced environment, I adapted our approach. I implemented a mix of central workflow, where most changes were directly pushed to the main branch, and combined it with trunk-based development. This approach was really useful for large features like the VR/XR controller logic and scene management, as well as bigger CI changes. It worked well for the team and gave even the newbies a structured development process in Git.



```
Active branches

main  default
a31f4d37 · Merge branch 'NewCI' into 'main' · 31 minutes ago

ci
c2293bf5 · ci changes · 2 weeks ago                          145 | 0

XRimplementation
8a574a0f · changes · 2 weeks ago                             162 | 0
```

<u>Conclusion</u>

In summary, it is clear that I had to make some cut backs on the original scope. Manual user testing for example, turned out to be the most effective method for our project, it allowed immediate feedback and direct issue resolution. This was an essential feature given the rapid evolution of our application. I sadly didn't have the time to implement Automated tests, while they were still valuable, our project did not require them, as the speed of development made such tests less relevant. To elaborate on this, most of the coding was made in day long sprints, where the time between pushes was pretty short. Each new version brought different changes, rendering extensive automated testing for earlier versions useless.

This project was challenging, as it pushed me to apply the knowledge I had gained from the course, such as YAML syntax and the use of Git / secrets. But, it also required me to delve into topics that were new to me, such as writing and implementing powershell scripts as well as the in and outs of the GitLab CI process and a considerable amount of debugging. This process was not just a learning experience but also gave me a clearer view into the practical aspects of CI pipelines and the depth of collaborative development.

In conclusion, I learned the importance of flexibility and the value of adjusting project goals when faced with real-world scenarios. It has been a great look into the iterative aspect of software development, where the goal is often changing and the way to achieve this is filled with learning. To bring this project to an end, I want to say that I enjoyed working through both projects. The CI project and the Showtime project. At first the group collaboration was pretty lacking, but as we familiarized ourselves we became a better team, and I could implement some of the scope I had previously defined for this project.

Sources:

-"Documentation." Game CI, n.d., https://game.ci.

-"GitLab CI/CD." GitLab, n.d., https://docs.gitlab.com/ee/ci/.

-"Caching." GitLab, n.d., https://docs.gitlab.com/ee/ci/caching/.

Code:
-https://code.fki.htw-berlin.de/cm/studierendenprojekte/ws23_24-b6/b6_npc