MapThat - Linux Kernel Best Practices

Shreya Someswar Karra
sskarra@ncsu.edu
North Carolina State University

Shubhangi Jain
sjain29@ncsu.edu
North Carolina State University

Anmolika Goyal
agoyal4@ncsu.edu
North Carolina State University

Anant Gadodia
agadodi@ncsu.edu
North Carolina State University

Srujana Marne Shiva Rao
smarnes@ncsu.edu
North Carolina State University

ABSTRACT

MapThat is a tool that integrates **Google Maps** with **Google Calendar**. The main objective of this project is to block the time taken by the users to travel from the default location to the required location. This is done so that no more appointments can be booked in that period of time. This is the **Phase I** of the project. The main aim of this phase is to make sure that other students understand the concept behind this project and want to develop/modify it further.

This document explains how the project aligns with the linux kernel practices.

CCS CONCEPTS

• **Software Engineering**; • **Linux Kernel Development-Best Practices**;

KEYWORDS

MapThat, Google Maps, Google Calendar, Linux kernel practices

1 INTRODUCTION

Writing a software project has made us realise that writing code is not an easy task. It has a lot of dependencies and takes a lot of time. Code changes, fixing errors, committing the changed code, delegating tasks, solving the open issues and documenting the same. The success of a project depends on how well it aligns with the maximum of the Linux Best Kernel practises. [1]. The following are the Linux kernel practises that need to be followed

- Short Release Cycles
- Zero Internal Boundaries
- The No-Regressions Rule
- Consensus Oriented Model
- Distributed Development

In this report we have tried to establish the connection between the requirements mentioned in the rubric and the Linux Kernel best practices. We have also tried to explain how our project is in sync with these Linux Kernel practices.

2 LINUX KERNEL BEST PRACTICES

2.1 Short Release Cycles

Short Release Cycle enables the developer to find code conflicts easily especially when they want to see the conflicts occurring due to integration from the latest stable build. Due to the short release cycle, the new commits which are released in the new cycle are not very different from previous commits and thus do not require major debugging efforts. Also, if release cycles are long, it creates a considerable amount of delay in getting the desired feature to the user. Our project had very short release cycles where we focused on pushing small yet functional changes to git so that the code is working on git for all the team members after every commit and every git push has a meaningful update behind it. It helped the teammates to add on to the project in an incremental manner. We focused on pushing the code after achieving every functionality milestone in our project with minimal disruptions.

2.2 Distributed Development Model

In the Distributed Development Model, an individual is responsible for different attributes of the development. However, it becomes very difficult for one developer to take care of hundreds of attributes related to the project. Hence in this model we assign different parts of the project to different developers who are responsible for code reviews, integration, maintenance etc. This ensures that each part does not lose any points when it comes to quality and management. Similarly, in our project the workload was distributed across the team. The project could be divided into several parts such as working with google maps API to retrieve current location, distance, time ; working with google calendar API to retrieve calendar entries and location mentioned in these calendar entries, integration of these APIs etc. The start of the project including brainstorming, division of parts was done together as a team. After which, the individual person was responsible for researching the execution and implementation. Since all the parts are connected, the code was constantly reviewed and integrated which helped all the team members be up to date with the project.

2.3 Consensus Oriented Model

Consensus Oriented Model says that **A proposed change will not be merged if a respected developer opposed it** [1]. Basically, if any developer is against the code changes, the alterations will be revisited and reviewed. Our project followed the same methodology. Our GitHub repository contains a markdown file called **CONTRIBUTING.md** which illustrates how to contribute to the software which adds on to the functionality of the project. Apart from this, the teammates have a WhatsApp and Discord chat channel and conduct frequent video calls to discuss and get feedback on any functionality change. Incase of any conflict, it was discussed on the spot and rectified accordingly.

2.4 Consensus-Oriented Model

Consensus Oriented Model says that " a proposed change will not be merged if a respected developer opposed it" [1]. Basically, if any developer is against the code changes, the alterations will be revisited and reviewed. Our project followed the same methodology. Our GitHub repository contains a markdown file called **CONTRIBUTING.md** which illustrates how to contribute to the software which adds on to the functionality of the project. Apart from this, the teammates have a WhatsApp chat channel and conduct constant video calls to discuss and get feedback on any functionality change. In case of any conflict, it was discussed on the spot and rectified accordingly.

2.5 The no regression rule

The no regression rule states that "If a given kernel works in a specific setting, all subsequent kernels must work there, too" [1] It means that as the functionality of the project grows, there is an assurance that the upgrades will not break the existing system. Similarly for our project, after every GitHub push, there was a check to ensure that the program was working properly in everyone's system and any intermediate commit did not break the system.

2.6 Zero Internal Boundary

When a project follows the concept of zero internal boundary, the developers have the flexibility to come in and make changes to other parts of the system as long as they are justifiable. This ensures that the changes made fixes the problems instead of compromising the kernel stability by looking for multiple workaround [1] Similarly for our project, we ensured that we practiced zero internal boundary where every member was aware of all the parts of the project and was welcome to contribute to any issues that arose. There was an understanding between the team members about all the updates done and thorough review of all team member actions were done via the chat channel and video call sessions.

3 CONCLUSION

Our project and project development follows Linux best practices as guided by the official documentation. There was zero internal boundary and all team members had full visibility of all the parts of the project. The project was developed using a distributed development model and released in short release cycles which ensured that we were pushing meaningful updates and the program was constantly evolving. The team members were constantly in touch via chat channels and video calls which ensured that there was proper communication regarding code changes, reviews, issues, feature ideas etc. Every git commit was done taking into account that the system should work in all settings.

REFERENCES

[1] Tim Menzies. 2021. *proj1rubric*. Retrieved September 29, 2021 from https://github.com/txt/se21/blob/master/docs/proj1rubric.md

[2] https://medium.com/@Packt_Pub/linux-kernel-development-best-practices-11c1474704d