Packagebird Final Report

Cpt_S 423 – Dr. Bolong Zeng

Mentor Brandon Busby

Team Apollo

Elisha Aguilera, Setenay Guner, Denish Oleke

Table of Contents **Page Number**

Executive Summary

In this document we are introducing Packagebird based on the original requirements given at the beginning of the project versus the end of the year requirements. Throughout the development of the project many aspects and requirements have changed significantly.

After the necessary context and the required details are provided, we will detail the final prototype, test and validations applied against the prototype, and what insights were gleaned from the battery of tests.

The contributions made to the project by the team members are listed in Project Management, differences between the initial and final design of the project are discussed in Results. We are also looking at unit tests and validation results to analyze in the same section.

As far as Packagebird is concern, its broader Impacts focuses on the future potential of the Package and its variety of use. The future of the project currently seen by the team is discussed in Limitations and Recommendations for Future Work. The changes and issues we have not been able to solve are also mentioned in this section.

Introduction and Background

Packagebird is a dependency management system, principally by relaying sources to and from a registry server, providing a similar yet more constrained service in comparison to existing package managers, which are often centralized language specific repositories. If further details are needed, Spinellis (84) provides a thorough cross section of their utilities. While the field is now solidified with projects like NPM and NuGet there are still attempts to improve the concept. Original requirements for this project were:

- Design a build system for an organization which produces a large software product
- Develop a dependency management system so that a build of package always knows what version of the package's dependencies to use.
- Look up files, add, remove, and edit packages and projects.
- Design a dependency graph

The problem Packagebird is aiming to resolve is the management of dependencies for projects, as teams rely on code assembled by the team or an individual to implement applications. On occasion, there are issues where certain developer environments may have missing dependencies or dependency version conflicts. Though many dependency package managers exist, they are usually bound to a particular programming language or framework and are too complex and intricate to replicate and deploy internally.

Throughout the development of the project, it changed significantly. The dependency graph was not fully implemented and one of the major changes was incorporating a graphical front-end interface to the project. While exploring solutions we went through an iteration of a Go client implementation using Cobra-cli. We've made drastic changes to the database in the middle of the project. In the final requirements:

- server has successfully integrated into MongoDB, as the document-based schema which also effectively stores package metadata.
- Implemented build, test, help commands to the CLI
- GUI implemented using C#

Description of Final Project Design

The project consists of three parts: the graphical client, the command line client, and the server. The client is written in Go language using the Cobra-cli framework for generating the command line application interface. Network exchanges and the server use gRPC stubs generated by the Protoc compiler, with interfaces defined in Protocol Buffers version 3.

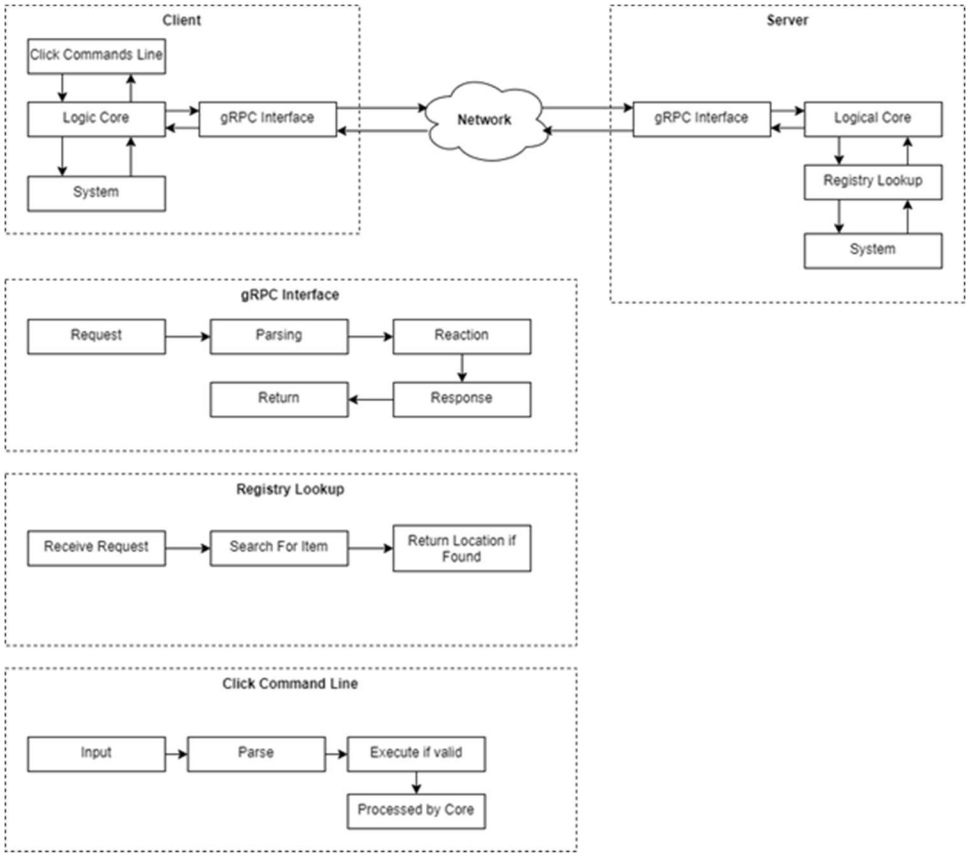

Figure 2.1 – Process Diagram, Components and Subcomponents exchange as illustrated.

The server acts as a central node and has multiple clients connecting to access and manipulate data stored in a MongoDB database. It is language agnostic and follows a client-response model to allow bidirectional communication which was a main factor of why we chose Go. Server can easily communicate with multiple clients using Protobuf call methods. Client and the server streams are separate, and they both can stream files and messages in any order. Files are split into smaller chunks and buffered to prevent network conditions to prevent blocking the client or server process due to network interruptions, along with avoiding over-consuming main memory.

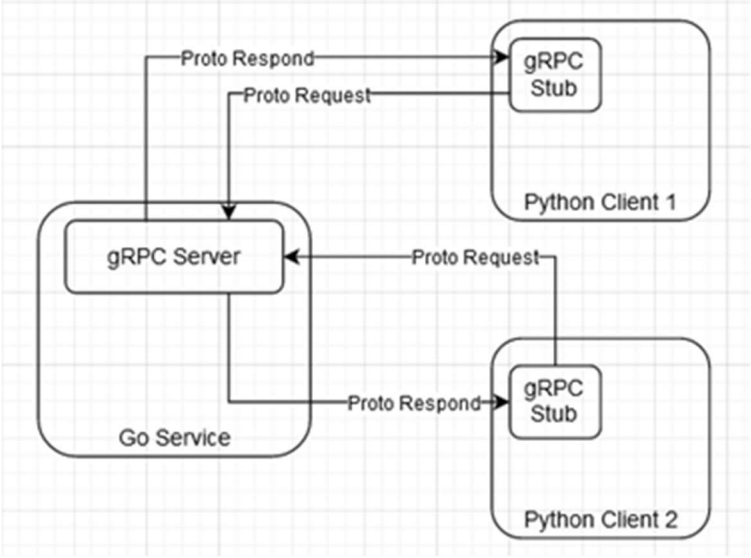

Figure 2.2 Go service communicating with Python clients.

This project will deliver a simple trimmed down alternative with a single server executable and client executable for desktop and server environments that is language agnostic, along with easily understood documentation for deployment and use. Below figure shows how straightforward it is to setup a project in Packagebird.

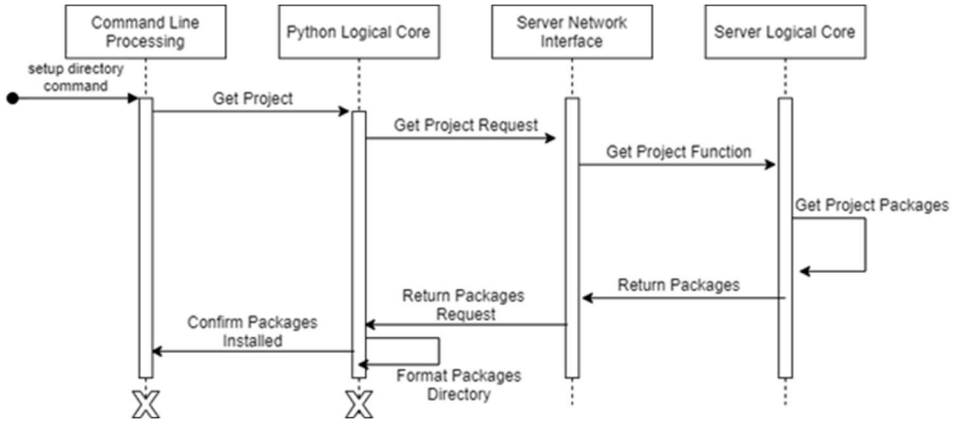

Figure 2.3: Sequence diagram of a project setup process.

The most significant changes to the overall system architecture were a revamp and extension of the data persistence accessors; iterations towards the begging of the second semester failed to properly leverage MongoDB's strengths as a NoSQL database, instead being constrained with relational thinking. Near the latter half of the semester, changes were made including more detailed accessors, combinations into common queries defined as methods, and generic methods for inserting, modifying, and extracting structures from the database, detailed in the following section. Relations are enforced by object identifier reference fields in select documents, with some documents like Authentications having references to Projects and Users. A graphical illustration of these relations is shown below in Figure 2.3.

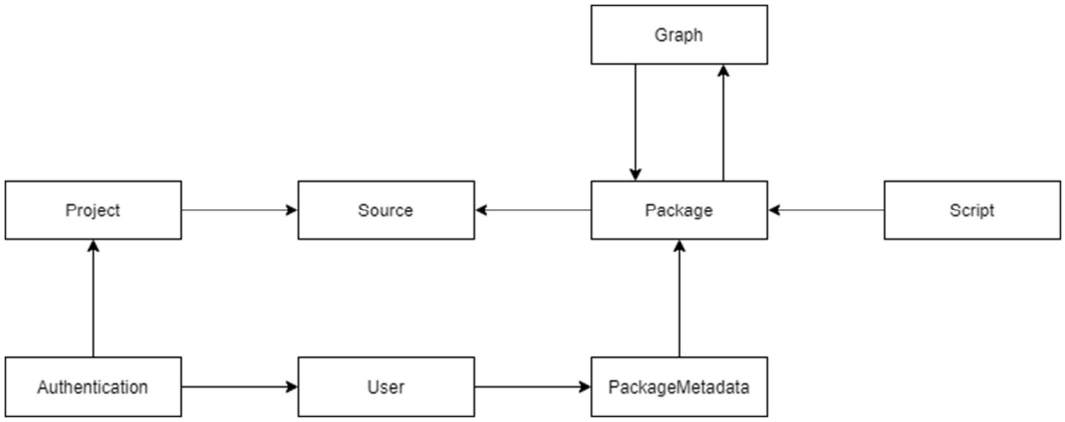

Figure 2.3: Relationship schema between document collections in MongoDB.

Project Management

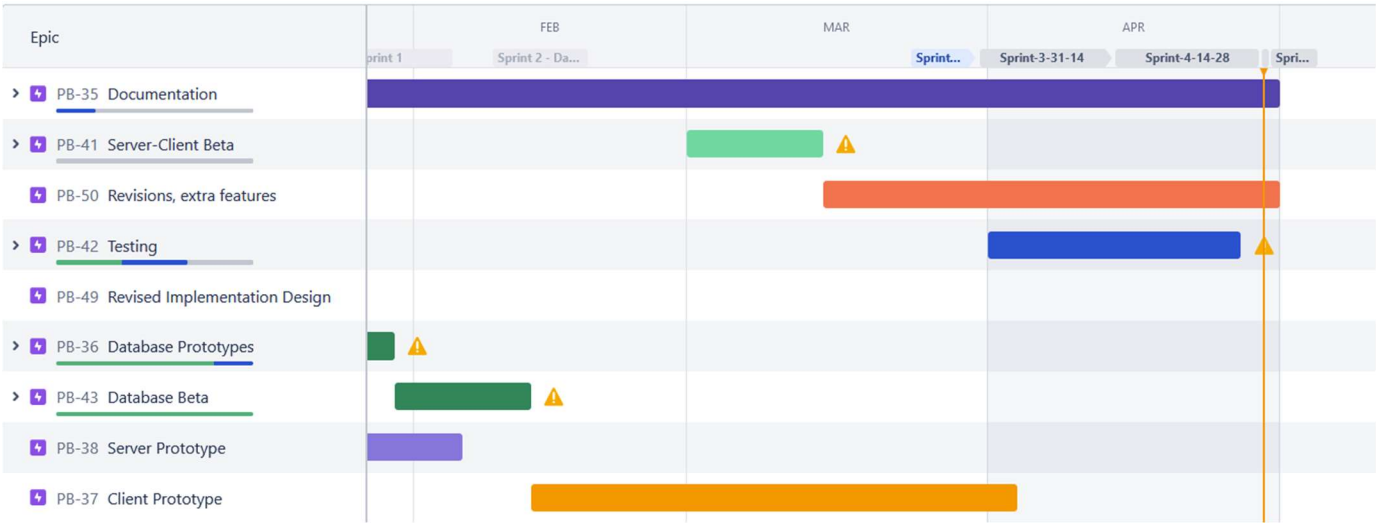

Figure 3.3: Jira Gantt Chart of planned and conducted work during the Spring semester.

Description

The initial quarter of the semester was incorporating lessons learned from the alpha prototype, accounting for resource and skill limitations gleaned from the previous experience and laying out a plan to complete the project. We attempted to utilize Story Points to help gauge development velocity and difficulty, but the chaotic and varied schedules of team members, combined with the reshuffling of items resulted in story points being dropped as an additional metric. We instead opted for direct estimation of difficulty and very lax time budgets for items.

Iterations were originally composed into a series of Epics, to be listed. As the semester progressed, some tasks had to be reshuffled, shrunk or cut. A dramatic change was the inclusion of a graphical user interface client, which forced a redesign of the client and some changes to the server. We originally underestimated the difficulty of implementing a graphical user interface, this led to a period of increased development urgency colloquially known as ‘crunch’. Communication was a weakness of our team, with members having conflicting obligations and demands, further shrinking the time budget to implement features. Were the project to be redone, given available resources and experience, we might have opted for a daily SCRUM model to keep a faster and less disjointed pace of work.

Member Contributions

Elisha Aguilera primarily contributed to end implementation of the code, extending and refactoring the server, and rewriting the client in Go language, in addition to implementing the graphical user interface wrapper application. Precise technical additions include implementing generic interface creation and retrieval from Go language to MongoDB, defining the application programmer interface in Protocol Buffers using gRPC, and leveraging Cobra-Cli to generate the client-side command line application in Go language.

Outside of technical work, his contributions included serving as team liaison between the mentor, the instructor, and team members as needed. Technical work was the objective, but on occasion he assisted with documentation and other non-implementation deliverables, such as the regular Beta Prototype update documents, the monthly status reports, and the Final Report.

Contributions made by Setenay Guner mostly stayed in the backend implementation of the code as well. Utilizing unit tests along with fuzz testing as well as debugging database interactions accordingly with the results. Her contributions also included technical writing for the deliverables and research for the database as well as the final poster demo.

Contributions made by Denish Oleke were more on the project documentation which included Mongo Database integration descriptions, Client user instructions and the Server user instruction. In reference to implementations, He contributed to the package handler implementation and as we prepare to pass this project to the next team, he will be documenting the source code.

Results

Final Prototype Description

In terms of technical implementation, compared to the original alpha prototype, the final prototype uses the document-based NoSQL database solution MongoDB, compared to the original SQLite relational database. Inserting and extracting data leveraged the recently released features of Go version 1.18, generic types, and advanced type reflection, to support inserting and decoding structures without boilerplate code and additional method calls. Interaction is conducted through MongoDB's official Go language drivers.

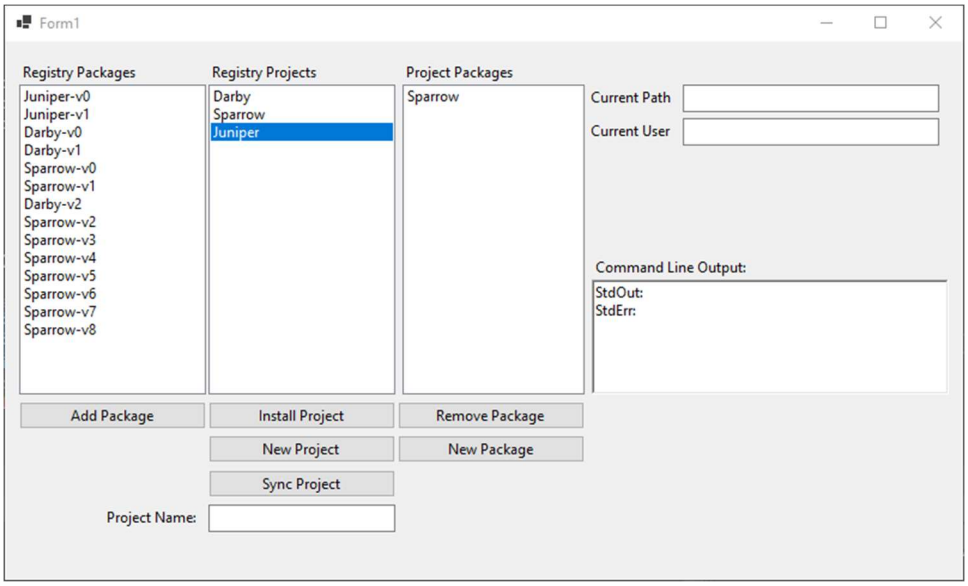

Figure 4.1: Graphical user interface wrapper atop command line client.

The client was rewritten in Go version 1.18 to support a graphical user interface wrapper written in Visual C# using WinForms. To replace the Python-only Click framework, the reimplementation instead uses Cobra-Cli, and the standard library to interact with the filesystem. The graphical interface was a late addition to improve user-friendliness with less technically inclined users. It uses the standard library to interact with the filesystem and initiate the command line client for sending and accessing data from the server. The output is displayed in lists and text boxes, with input gathered from text boxes and buttons.

The client and server communicate over a gRPC application programming interface defined in protocol buffers version 3, with few modifications to the original alpha prototype version. Additional functions of note include listing values stored on the server, such as Projects and Packages.

Final Prototype Test Results

	Line Coverage	Branch Coverage
>Database Interface	76%	
>PackageDBInterface	77%	90%
>ProjectDBInterface	74%	82%

Figure 5.1: Line and branch coverage of the unit tests.

Unit and integration tests were primarily conducted by Setenay Guner with assistance from Elisha Aguilera due to varying levels of familiarity with components of the application. Due to some components containing auto-generated code, namely the gRPC server stubs, not every line was covered in a conventional sense. In addition, some of the components rely on side-effects involving the filesystem and MongoDB database documents, and as such those tests may not be considered “pure” in a traditional sense.

Figure 5.2: Fuzz testing results.

After receiving feedback from our mentor, we also implemented fuzz testing into our program. It is a way of automating the process of sending random and invalid inputs to the functions we are manipulating. It can be used to find vulnerabilities and edge cases. It is not very UI friendly, the screenshot above shows corpus, contains input fuzzer thinks useful for the target function, it also shows the coverage fuzzer got to explore and you can see this number increase as you’re running the tests. Fuzz testing can also be used on network protocols, however, since this is a new feature in Go language, we decided to go with a concept we’re more familiar with.

Network operations testing was primarily conducted by Elisha Aguilera. As the application consists of two networked components it is prudent to test the responsiveness of a battery of operations in various conditions. As stated in the *Beta Prototype Update* document, to precisely control network conditions, we utilize *Clumsy version 0.2* to simulate increased latency and jitter in increments. The final case is testing operations against a server hosted on a Virtual Private Server located in Los Angeles, approximately 884 miles away from the client’s location in Sultan, Washington with an average round-trip ping of 34 milliseconds as measured from PowerShell.

Final Prototype Validation Results

The prototype with the graphical user interface was validated with a series of operations we deemed relevant to the commonly expected use of the prototype. From a clean database state, a new project is created, with a corresponding directory created in the working directory of the executable. This is reflected in the database, where corresponding documents for the project, and its source file metadata document.

To validate is the creation of a packages, first a clean state project and database are generated, and the project is populated with a few files containing some basic functions that return string values. The create package process is initiated and the project source is copied into a package, assigned a version number, and the graphical list is updated to get the updated list of packages. We then performed a tar extraction on the source file to confirm the source files were created and the original project directory structure was preserved.

Validating the addition of a package is performed by repeating the previous process, and then performing the add package process on a new project. The package source is downloaded and extracted to the ‘packages’ subdirectory within the project directory. The extracted package was then inspected to ensure that the extracted and installed package matches the package that was created and stored in the server’s file system.

Project installation is validated by creating a project and populating it with simple single function JavaScript files that return strings. These files are then synced with the project in the server, and in a new directory the project is installed. The resulting project directory was inspected and confirmed to match the original project created in a different directory.

Broader Impacts

We envision this tool will be used alongside traditional version control solutions such as Git in team-oriented environments to deliver software for clients. Traditional software package management is very granular and centralized in external registries and is difficult to leverage for smaller firms. Packagebird allows system administrators to quickly create a package management service and provide team members to deliver modular units of software that combines into a partial or complete deliverable. While this tool was created and tested for use in software creation, it can be used anywhere that software is written and distributed.

While this project is not a complete solution, it has made attempts to curb the effect known colloquially as dependency hell, wherein dependencies begin to cause naming conflicts and errors due to inter-package compatibility issues, often addressed using graph operations as described by Fan and others. By permitting a high level of control and inspection of what packages are involved in the project, developers can carefully monitor and trace what offending packages cause issues with their project, and rollback additions as needed.

This project doesn't directly address any contemporary issues from the time of writing outside of software development and distribution, but the additional productivity it imparts on teams that successfully integrate and leverage its abilities will decrease non-productive development time. This saves their employer or client expensive engineering hours. In turn, the added value can be passed on to the end consumer by reducing the prices of their end-products, or by delivering innovative features at a faster rate.

Limitations and Recommendations for Future Work

To remind you once again, Packagebird works as a package management system that allows users to manage their software packages in a more efficient and organized manner. While the project is still in its initial stages, we have encountered a few limitations and would like to recommend some future work for the project.

As of now, one of the main limitations of Packagebird is that it does not currently support offline package management. In other words, users cannot add or remove packages when they are not connected to the internet. This limitation can be addressed by adding support for offline package management in future versions of this project.

On the other hand, the second limitation is that Packagebird does not currently support binary packages. This means that it cannot be used to install software that is not available for user end consumptions.

Leverage Redis caching to speed up queries

The current implementation makes direct calls to a MongoDB server, and as a result regenerates each and every query regardless of the state of the data being accessed. At small scales this is reasonable, but as load increases likewise bandwidth increases in scarcity. This can be addressed by caching common results that don't frequently change and delivering them to clients should their request be a valid fit.

A solution is to use existing server technologies, the most popular of which is current Redis. It can serve as a database in its own right, but it is usually relegated to caching due to its speed and lack of expressiveness when storing and retrieving data. Another solution is to incorporate more of MongoDB's advanced features or redesigning certain schema to take advantage of common items, such as embedding documents if the use case permits.

Implement a production viable security model

The current user authentication and security model is purely for demonstration and proof-of-concept, exemplified by the use of plain-text for password matching. Production grade authentication will use password hashes with added salt values, and increasingly use second channels like email and mobile phone numbers to verify abnormal connection attempts and for password recovery.

The present schema could incorporate these changes by adding additional collections with documents related to users by object identifier strings, or by altering the existing user document schema. If these values cannot be stored in the database, they can be stored in files on the filesystem and accessed by the server executable.

Implement a generic database structure accessor

Go language is a very capable language, however some developers may feel it lacks expressiveness. To compensate for this quality, we have implemented a series of methods that can access generic interfaces and structures from the database using reflection and generic parameters. However, it is not production grade, and there is still some brittle code required to access items from the database. This can be addressed by further research and experimentation into advanced Go language features or taking advantage of the 'Mgo' MongoDB drivers as opposed to the current official MongoDB drivers.

Implement improved workflow for testing against database

At present, the best means of unit testing is either to completely isolate the server application from the database, mocking all expected results, or by carefully setting up and tearing down the database manually. We are certain there exists a better workflow that can automate the latter process with a special mock test database. The motivation for this unorthodox test is to exercise any faults that occur during the translation, either during insertion or retrieval.

Better utilize Go language's concurrency features

This original implementation makes minor use of Go's special multithreading and concurrency features outside of the utilities provided by gRPC, and the operations provided by the MongoDB drivers. The biggest hindrance to its full utilization is performing write operations to data, which may cause race conditions, especially since it involves secondary side-effects. However, we believe the existing design is accommodating to incorporating multithreading. Research from Dilley and Lange describes common patterns using Go's concurrency model and may be a good starting point for a rework of the design if needed.

Conclusions

Initially, the project requirements were defined to an abstract scope, with general objectives but not too detailed to be fully defined. As research, iteration and exploration were conducted alongside the feedback of the client, these objectives were realized into a more concrete series of goals. From that point, these goals were further deconstructed into series of operations which could be satisfied with implementations.

Like many projects, the ambitions were grander than our abilities; some features that didn't make final implementation in the Beta Prototype include building and assembling complex relationships of packages, dependency graph analysis, and complex package permission levels.

The Beta Prototype is capable of exchanging files, structuring file system environments, and managing the packages associated with the source files. However, our implementation of the application is inelegant compared to industry solutions, and we hope that future teams can utilize our contributions as a foundation for improvement rather than being haphazardly discarded or incorporated without consideration.

Reductively, we view this project's greatest asset as its simplicity and ease of deployment; compared to the complex and bulky alternatives, the centralized and opaque language package registries. For example, if we consider popular npm package manager. It uses a combination of npm modules, Gulp tasks, and webpack to manage dependencies. However, there are some drawbacks related to that such as being hard to understand and being slow. In relation to the above, Packagebird is just two executables and a MongoDB database. In future iterations, these attributes can be extended and leveraged further to a potentially production- viable application.

Lessons learned from the first semester, to be incorporated to those inheriting this project, is to consider and plan the work carefully before anything is implemented, tempting as it is to start producing deliverables. This project's deliverables can serve as a foundation for future extension, and even if portions are carved off and removed, do attempt to internalize and recognize the rationale behind some of the design decisions made and the technologies used.

Acknowledgements

Our thanks to Brandon Busby for the advice, not only for his guidance for this project, but also for the ample advice on development concepts, employment, and financial management, far beyond the expectations of a project mentor.

Also, thanks to our instructor, Dr. Bolong Zeng, for additional guidance on project management, interpersonal skills and team management, and tolerance for our naïve antics.

Finally, thanks to the various contributors for the open-source technologies we used for this project, from the Go language to MongoDB, and the various libraries and frameworks therein. An additional note of appreciation for the authors and contributors towards the many, many pieces of documentation and tutorials we used to complete the project.

References

Dilley, Nicolas, and Julien Lange. “An Empirical Study of Messaging Passing Concurrency in Go Projects.” 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019. Crossref, https://doi.org/10.1109/saner.2019.8668036.

Fan, Gang, et al. “Escaping Dependency Hell: Finding Build Dependency Errors with the Unified Dependency Graph.” Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020. Crossref, https://doi.org/10.1145/3395363.3397388.

Spinellis, Diomidis. “Package Management Systems.” IEEE Software, vol. 29, no. 2, 2012, pp. 84–86. Crossref, https://doi.org/10.1109/ms.2012.38.

Appendixes

CPTS 421 Iterative Design

The process of iterative design engaged by the team was first to formulate the requirements of the project, find examples of similar solutions, compare the examples to find a potential solution that is achievable with our resources and skills, settle on a design that's do-able within the project, and begin the iterative process of implementing features, incorporating feedback into each iteration and potentially modifying the design if an insurmountable roadblock is reached.

The server went through several iterations before we settled on gRPC. We first started out with a very simple HHTP server and further we explored using REST as an option. Even though REST and HTTP are used interchangeably, REST became too restrictive for our required functionality, and was abandoned after the first couple of iterations. After some further exploration, gRPC was selected for its compatibility with our client's recommended language, also solving the problem of controlled file transfers across a network. As we started implementing the server, SQLite was the initial choice for data persistence as it required almost zero configuration and is very lightweight. However, it came at the expense of rigid schema having to be encoded in the server.

One of the considerations that was brought to our attention by our stakeholder was relational versus document databases for this project. While Relational offers accuracy and consistency they work best with structured data. They also do not scale well. On the other hand, working with NoSQL gives us an advantage of storing large amounts of data with flexible structure. The two options are equally useful in their own way but for contrasting use-cases. On recommendation from our mentor and instructor after the Alpha prototype presentation, we opted to reimplement the data persistence using a NoSQL database.

The command line interface was implemented using Click and Python. The alpha build established a solid foundation for networking, file transferring and how we wanted to structure the server. A client-response model that allows bidirectional communication became our new default when restructuring the design in Beta.

For personnel and time management, we attempted to practice traditional a weekly Agile-based SCRUM system utilizing Atlassian Jira, decomposing major features into Epics, minor features into Tasks, and implementation steps into Subtasks. We applied Story Points to model the projected difficulty and effort expended for major items and grouped items into traditional two-week sprints with approximately eight story points per working team member. Our projected difficulties were usually far off from the actual work required for an Item, either too little, but more often we would underestimate the work required.

Given another opportunity to practice team management, we would likely not use the traditional Sprint model, given the varied schedules of the team members and non-standard working periods, instead using a stricter Kanban system with weekly deliverables reported on our mentor meetings. This was closer to the actual working pace towards the end of the first semester.

In reflection, our design approach would have certainly benefited from more communication and prototyping with abstractions over attempting to implement functionality as we were exploring potential technologies. Were we to have slightly more experience with modern networked applications, some of the initial implementation process would have been expediated significantly.

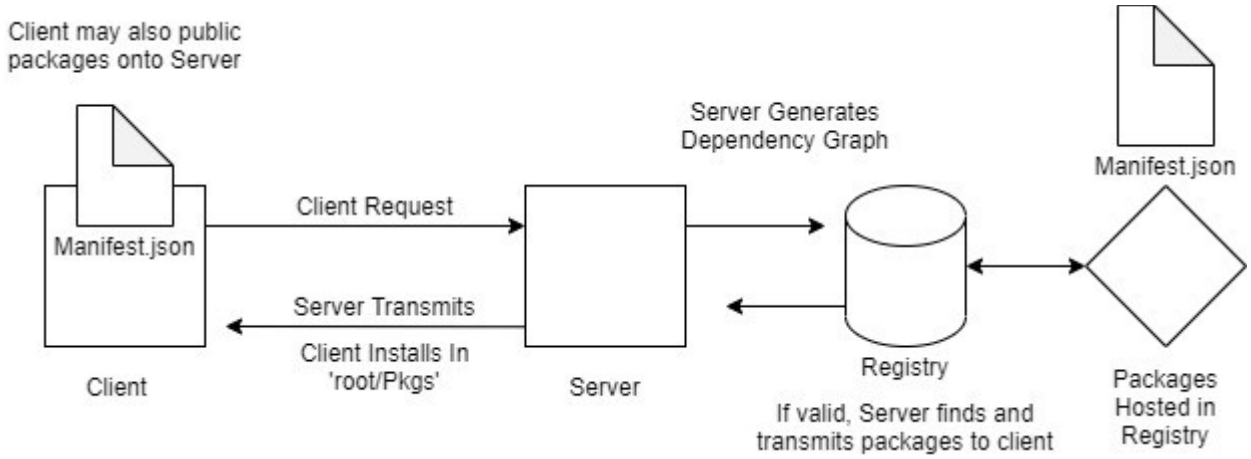

Figure Appendix-1.1: An early initial design of the Alpha version for Packagebird.

Team Photographs

Elisha Aguilera

Setenay Guner

Denish Oleke