**DEPENDENCY MANAGEMENT IN A NUTSHELL**

|  |  |
| --- | --- |
| Package Bird  Dependency Management | Abstract   * The most well-built software that has been on service for decades now ‘Package Bird’ Package Bird gives you the best experience with dependency Management both on client and server perspective. Best since the age of time.   Set, Denish, Eli  CPTS\_421 |

Table of Contents

**Project Overview 1**

Project Acknowledgement 2

Preliminary Description 3

Package implementation 4

**Project Design 5**

Process Breakdown 5-6

Implementation Framework 6

Project Integration 6

**Intentionally left blank**

**Project Acknowledgement.**

As the name says, package bird is a dependency manager that models the Client’s behavior by interacting with the server, which we will see in the next few pages. Just to remind ourselves, we will once more define to you what a server and client means to us pertaining to this project.

We are looking at the server as the central node, that multiple clients connect to access and manipulate same data. In other words, the server – will be our back-end interface-handling request from various clients, performing evaluation and lookup functionality, and directing files to and from the system over the network to the remote clients. Whereas the Client – will be our front-end executable that serves as the interface between a user, the user’s machine, and the remote registry.

The other thing we will be interacting with is the Registry – which is a database, which will act as a metadata address book for our existing packages, contained in the system. Not forgetting the project, itself which is – a continuous ledger of bundled packages consisting of dependencies for the system. And finally, the Packages – these are containerized formats for storing and accessing source files, often code language agnostic libraries.

With all that put aside or in mind, we have also visualized that data may split into atoms if we may say and considering that fact, it is one reason we have decided to call this project Package-Bird so that we can collect those atoms as a package. Without assumption, we know that whatever is on server has priority (to avoid getting user into conflict solving). On one hand, we may prefer partial synchronization due to potentially substantial amounts of data.

**Preliminary Processes.**

The main primary processes which are envisioned for Package-Bird are *the configuration of a developer’s environment*, and *the “creation | registration” of packages*. These processes are envisioned to execute in linear fashion, as there is no foreseeable circumstance where a user would configure a development environment while packaging it in the same step.

On the other hands, Secondary processes will include *browsing listed projects* and *packages registered* in the system for research purposes like other existing package managers. To relate with, Examples of package managers like Package-Bird can be found in the Package Management Systems, and while the field on Package-Bird is now solidified, there are still attempts to improve the concept such as the OPIUM: Optimal Package Install/Uninstall Manager project. However, there are still many challenges involving package managers, including the handling of dependencies. Not to just mention it but for Package-Bird we worked around possibilities of overcoming some of those challenges which is why Team Apollo decided on deploying gRPC. For this project, the focus is on simplicity and ease of use, from setup and configuration to ongoing use factoring in the scalability and maintainability of the project in general. Deploying the server should be straightforward and simple, running a configuration executable to point at the appropriate file directory, and initiating. Client’s setup on remote machines can quickly dial-in, configure and begin pulling packages from projects defined on the server.

**Package-Bird Functionality**

Before we dive any further, let us say the application involves many clients interacting with a centralized server, and itself interacting with a database along with the file system referred to as the registry. If that is the case and we are running on a local machine, then at this stage in development, the file system will be at a specified location on disk, and the server will be a process on the same machine unless otherwise, the process is vice versa. On multiple occasions or scenarios, for this set up of the project, the database will act as an address book for the packages, storing metadata including the version number, package name, authoring team, unique identifier string, additional miscellaneous information, and most importantly other dependencies. The server will be MongoDB, as the document-based schema effectively translates the JSON based package metadata.

Clients will formulate requests by inspecting a manifest file stored at the root level of the development directory if and only if the operation is bound to a local machine otherwise vice versa. Meanwhile, exchange of information between the client and the server will utilize remote-procedure calls over hypertext-transfer-protocol, received by the server in Go/gRPC and processing the information therein. The server will also interact with the file system using the standard library which will allow it to interact with the database using language bindings for MongoDB. When a request for packages is received from a client, the server will automatically attempt to find the described packages within the registry. If they are located within and reachable, the server will recursively build a dependency graph for each of the packages. Once completed, the source files will be bundled and compressed, then transmitted back to the client over the SSH protocol.

**Design Choice**

Criteria on package-bird design was chosen based on the customer needs we identified after discussion with the stakeholders and the team and the needs of keeping the minimal risk of liability. Our highest priority is to transfer packages between the server and the client in a short amount of time in the safest way possible and we solidify our decision to FTP for easy file transfer.

This concept screening shows the different options we considered while building the server. HTTP represents a primary functioning HTTP server. Others use HTTP API calls to communicate with the client while utilizing other aspects that we later found crucial to our system. REST API uses resources from the server to provide clients. It utilizes many HTTP methods to perform CRUD operations. A simple HTTP server, which we use as a reference concept, could use all the methods it offers, but it lacks certain elements. RPC is a distributed system; when the client calls a stub with the same signature as the function in the remote server, the message is encoded and passed to pass back a response within the server as quickly as possible. Transferring large files over HTTP can be an issue. While all the concepts we defined above could make use of different file transfer protocols when transferring large files, such as dependency packages, REST falls behind. While REST API supports 2GB chunks of files, the browser memory cannot and the connection between the client and server is not as efficient as gRPC.

A picture containing shoji, kitchen appliance

Description automatically generated

**Process Breakdown.**

Out of the many design choices listed above and like mentioned earlier, the gRPC gives us the flexibility of implementing the server in many ways while still using HTTP transport protocol. However, we do not have to specify the details for the remote interaction. However, we still write the same code as if the subroutine is local. While we are receiving data on the server, it needs to have the name of the function and the list of parameters being passed to that function from the client.

Since we are building a Command Line Interface, we decided to implement the client using Python since scripting, creating executables are easier, and the team has previous experience with it. However, implementing gRPC within the Python client was challenging and after discussions with the team, gRPC seemed like the better option. It is language agnostic and offers more along with the base procedure calls. It follows a client-response model of communication, which allows bidirectional communication. It separates the client and the server streams apart and they both can transmit messages in any order. This means multiple clients can request multiple calls from the server. Since we are building an app for developer projects this was a decision factor. gRPC uses Protocol Buffer to serialize the payload data instead of relying on JSON or XML files.

On further analysis, it appears and seems like the coordination of file transformation is complicated; however, Protobuf’s auto code generation is a helpful tool for solving such complications. In addition, Boilerplate code creates objects that represent the gRPC API defined in the .proto file. These objects contain serialization and deserialization logic to decode data on client’s machine from a Proto Binary. Overall, it gives us a fast and reliable file transferring between server and multiple clients.

Graphical user interface

Description automatically generated

**Package-Bird Implementation Framework**

As far as Package-Bird implementation is concerned, the client will be implemented using Python 3 along with the request and click frameworks for server interaction and command-line interface, respectively. The server will be written in Go language and interact with the previously stated MongoDB database. The data components will revolve primarily around packages; however, there are also authenticated clients with certain privileges.

Unlike the client implementation, the design choice and implementation will evolve around the gRPC framework, which we hope its library will offer us an easier and highly scalable high performance server client system like mentioned earlier in the Project Acknowledgement.

**Project Integration**

Much as we have not yet laid out the gadget/device specifications, we do believe that once the system is fully running it should be able to run on most of the machines that may support the legacy operating system. However, as of now, our client – server system has no restrictions since it is only functional on the local machines but once we scale it to run remotely, we will decide which operating system and versions the project will require at most or bare minimum.