# bpm: BLZ Package Manager

A Senior Project Report presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

Of the Requirements for the Degree

Bachelor of Science in Computer Engineering

by

Kenneth Huang

June 2019

# Table of Contents

# Introduction

## Project Overview

bpm (BLZ Package Manager) is a package manager for the open-source programming language BLZ, built in Java. It allows users of the BLZ programming language to create and upload their own packages, as well as downloading necessary dependency packages for their packages. To do this, the program communicates with the "cardiovascular", a web server designed for users to upload and download BLZ packages.

The program has three primary functions. The first one, "init", initializes a package directory for use with the package manager. Part of this initialization is creating a "heartbeat" meta file, which holds information about the package's name, version, and its dependencies.

The second function, "beat", compares the dependencies in the "heartbeat" meta file against packages currently installed in the directory, and downloads any missing dependencies.

The third function, "pump", uploads the current package to a web server that hosts packages.

## Client and Background

My client is the creator and maintainer of the BLZ programming language, Alex "blazingkin" Gravenor, a Cal Poly computer science major. BLZ is an open-source programming language built in Java, similar to how Python is built in C, created with an emphasis on readable

and easy to understand code. He started working on BLZ in 2015, and has been continually releasing updates ever since.

Package managers are an important part of project management for many programming languages: Java has Maven, Python has pip, JavaScript has npm, Ruby has RubyGems, and so on. With that in mind, Alex wanted to have a package manager for his programming language, bpm. (Its name being the same as the acronym for "beats per minute" has led to the naming of its components to be heart-related.) Because of the open source nature of BLZ, there were several programmers who previously tried to create package managers for BLZ, written in C, Haskell, and TypeScript, but were unable to finish them. Although no version of the package manager was previously completed, one of these developers was able to produce a completed version of a web application, the "cardiovascular", and is the web app that this project will communicate with to send and receive packages.

## Project Goals and Objectives

The goal of this project is to create a complete, functioning version of the BLZ package manager (bpm).

Objectives of this project include understanding how to create symbolic links, how to read, parse and write "heartbeat" meta files, and re-learning HTTP requests. These objectives allow for the uploading of packages, and the downloading of required ones, working toward the goal of creating a functional bpm.

# Product Definition

Customer Requirements

My client Alex provided me with the following requirements for bpm:

- bpm will be a command-line program.
- bpm will use the “cardiovascular” web server cvds.blazingk.in to store packages.
- bpm must support the following arguments:
    - **init:** creates a meta “heartbeat” file
        - This “heartbeat” file will contain information about the package and dependencies (which dependencies are needed, and what version(s) are required?)
            - Requires notation to denote version requirements (i.e. only one version, a range of versions, latest)
    - **beat:** compares installed dependencies against the “heartbeat” meta file; installs packages from the “cardiovascular” if packages listed in the “heartbeat” are not installed or are the wrong version
        - The “download” part of the package manager
    - **pump:** Uploads current package to “cardiovascular”
        - The “upload” part of the package manager

# Design

## Design Decisions

There were a few design decisions that I made for this project. Though I was not restricted on what programming language I could use for bpm, this project has been written in Java. As BLZ itself is also written in Java, I chose to write bpm in the same language to reduce the number of dependencies needed to create BLZ projects. Additionally, I decided to use YAML ("YAML Ain't Markup Language") as the file format for the meta "heartbeat" files for its simplicity and readability, which line up with BLZ's focus of being readable and easy to use.

## Final Design

bpm can be split up into three primary functions: `init`, `beat`, and `pump`. Each of these functions can be invoked by running the program with one of these three functions as the first command line argument (after the program name).

- `init`

Command line arguments: `bpm init [directory]`

The init function creates the folders and files necessary for a BLZ package. The overall program flow for this command can be seen below in Figure 1.

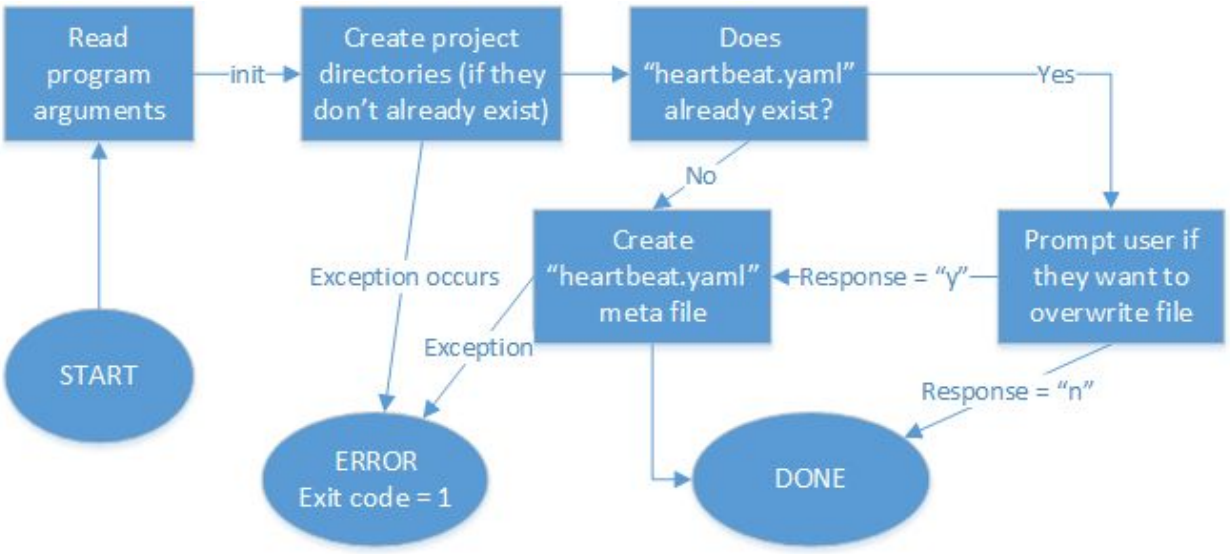


Figure 1. Program flow for `init`.

First, init creates the directories "Packages" and "Source", directories that house dependency packages and project source code, respectively. init takes an optional directory argument; if this argument is not set, these directories will be made in the current working directory; otherwise, they will be created in the specified path, creating directories as necessary.

init will not overwrite directories if they already exist, but if the directories cannot be made as a result of a non-directory file sharing a name with a directory to be created, or as a result of permission problems, the program will print out an appropriate message and exit.

Next, init checks if a "heartbeat.yaml" meta file already exists. If said file exists, it will prompt the user if they would like to overwrite the existing file. If the user replies "n", the program will finish. If the user replies "y" or there was no heartbeat file to begin with, then a new heartbeat file will be put into the directory specified in the arguments, or the present working directory if there was no directory specified.

- Heartbeat YAML format

The heartbeat.yaml meta file is organized into several categories: the name of the package, the version of the package, and the dependencies of the package. The dependencies are divided further into different environments: development (“dev”), testing (“test”), and production (“prod”), and each of these environments contains a list of dependency packages, with their name, minimum required version, and maximum required version. An empty string can be used to signify the lack of a minimum or maximum version; “latest” signifies to use the latest version.

An example file can be seen below in Figure 2, and the heartbeat created with `init` can be seen in Figure 3 below. A heartbeat file created with `init` will have its name field match the name of the directory of the initialized package.

```
name: TestProject
version: 0.1.0

deps:
  dev:
    - {name: json, min: 2.3, max: latest}
    - {name: readline, min: 0.4.2, max: 0.4.2}
    - {name: jquery, min: 12.2.0, max: 13.0}
    - {name: rubycopy, min: "", max: 2.2.1}
    - {name: noconstraint, min: "", max: latest}
  test:
    - {name: unittest, min: 2.0, max: 2.0}
  prod:
    - {name: pq, min: "", max: latest}
```

Figure 2. Example heartbeat YAML file.

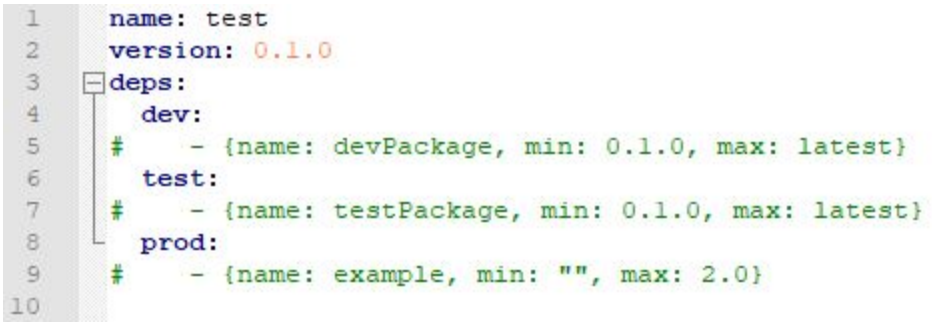


Figure 3. The heartbeat YAML file generated by `init`.

- `beat`

Command line arguments: `bpm beat env`

The `beat` function is arguably the most complex: it creates a symbolic link (symlink) to BLZ's core packages in the project directory, reads and parses a heartbeat.yaml meta file, and compares it to a heartbeat.lock file, if one exists. If it does, it gets the difference between the two for the corresponding environment `env`; if not, it gets all packages under the specified `env`. It then recursively downloads the packages needed by the project, as well as dependencies of those packages, from the cardiovascular (cdvs) server, and unpacks them for use.

An overview of the program flow can be found in Figure 4 below.

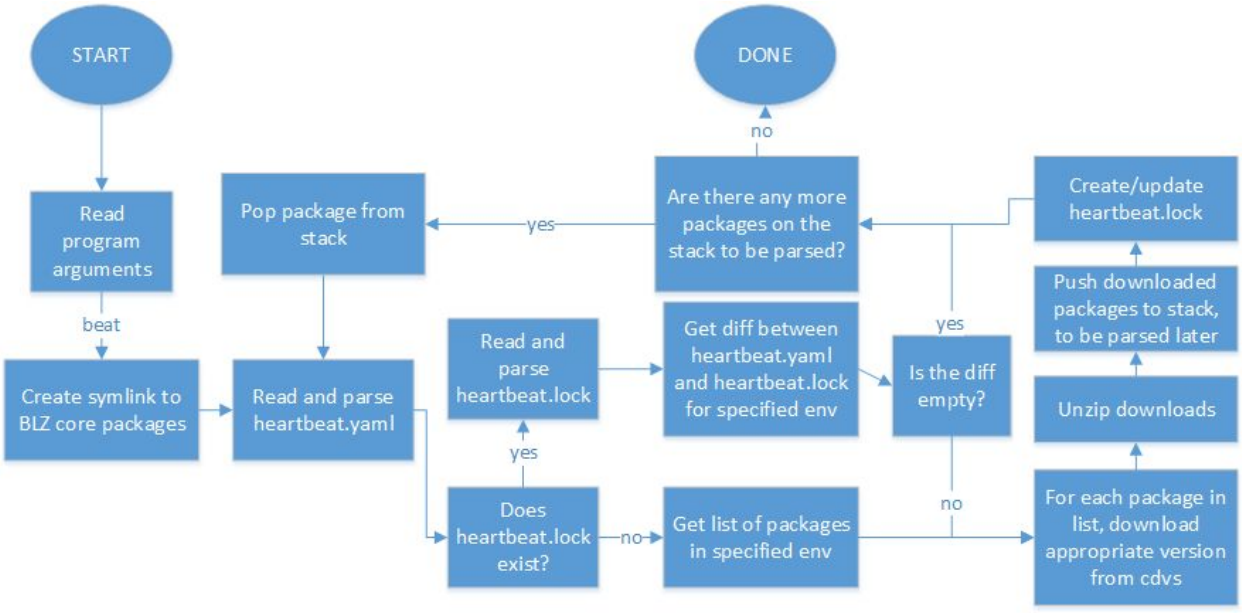


Figure 4. Program flow for `beat`.

To start off, the environment variable BLZPACKAGES, an environment variable set when installing BLZ and containing the path to BLZ's core packages, is read. A symbolic link is to the core packages directory is then created.

Next, the heartbeat.yaml file is read and parsed into a Heartbeat object with the help of the SnakeYAML library, which allows the parsing of a YAML file into a custom defined class.

A heartbeat.lock file is generated when `beat` downloads dependencies. It has a very similar format to the heartbeat.yaml file; it represents the currently installed dependencies for the package. For an environment specified in the arguments (the defaults being "dev", "test", and "prod"), the heartbeat.yaml file is compared against the heartbeat.lock file, if it exists, and generates a list of packages that need to be downloaded. This is done by checking the version range for each dependency in heartbeat.yaml against the versions for matching installed packages in the heartbeat.lock file. If the dependency is missing or is not the correct version, the dependency, along with its version range, is added to the aforementioned list.

Each dependency on the “to be downloaded” list is then looked up on the “cardiovascular” web server (cdvs); using an HTTP GET request, the program receives a JSON response which contains a list of versions of the requested package, as well as download links for each version. If there is an appropriate version (within the version range) inside this JSON response, `beat` will download the package. After the dependencies for the package are downloaded, the heartbeat.lock file is updated with the newly downloaded packages.

In order to ensure that all necessary dependencies are downloaded, the parsing of heartbeats and downloading of dependencies is recursive. When downloading dependencies, the name of each downloaded dependency is pushed onto a stack. After the necessary dependencies for one package are downloaded, if the stack is not empty, a package is popped off the stack, and its necessary dependencies are downloaded and pushed to the stack. This continues until there is nothing left on the stack.

- `pump`

`pump` allows the user to upload their package to the cardiovascular (cdvs). Because cdvs requires a login before being able to upload, the user will be prompted for the cdvs username and password before they can upload their package.

An overview of the function logic can be seen below in Figure 5.

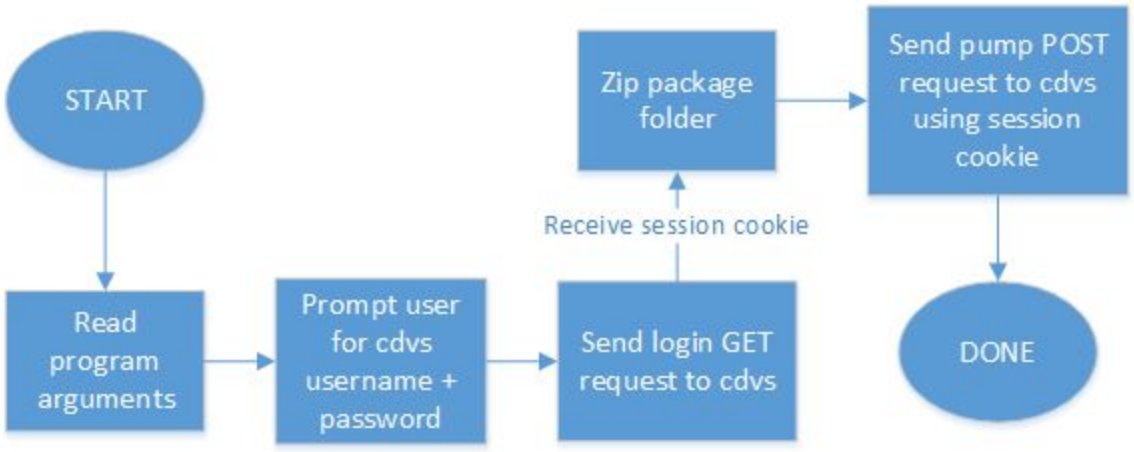


Figure 5. Program flow for `pump`.

After the user inputs their cdvs username and password (if they don't have one, they can register on the website cdvs.blazingk.in), these fields are used to send a login GET request to cdvs, which responds with a session cookie used for authenticated HTTP requests. This cookie is stored in program memory for "pumping" the package into the cardiovascular. After successfully logging in, the package's source code directory and heartbeat.yaml file are zipped into a tarball, then uploaded to cdvs via a HTTP POST request.

# Testing and Issues

While testing the `init` function, one of the issues I ran into was the package name on the heartbeat.yaml file being incorrect, with a single period instead of the actual package name. This is where I remembered about file systems: I had been using "." as my default directory, and while the "." directory refers to the current directory, it still adds to the absolute path. In order to handle cases where the "." directory is used, I added code to use the parent directory of "." instead (which provides the actual directory name).

Although I ran into a number of issues while developing and testing the `beat` function, the most notable has to do with differences in operating systems. The development for this project has been done on a Windows machine; when attempting to create a symlink to the core packages folder, I would get exceptions that access was denied. After doing some research into the issue, I discovered that the creation of symbolic links requires administrator permissions on Windows. (For comparison, Linux and Mac OS don't require permissions to make symlinks). To work around this, I changed the function's behavior so that if it is run on a Windows machine, `beat` will instead copy the core packages directory rather than create a symbolic link to it.

# Future Work

If this project were to be extended, a feature that I would like to implement would be the ability to handle different versions of the same package. However, this is currently very difficult to do due to BLZ's current package loading logic: it looks for packages within the Packages directory via searching by name, without considering the version of these packages, meaning that multiple versions of the same package cannot be placed in the same Packages directory, even if they were differentiated by measures such as appending their version number to their respective package directories. Thus, the possibility of adding this feature within the time limit of the senior project is unlikely. It would take significant additions or reworks to BLZ in order to implement this feature in a cleaner manner.

Another feature I would like to implement would be more detailed documentation. Additions such as more detailed help messages, and a more detailed README on the GitHub repository would assist users in using the application.

# Conclusion

bpm (BLZ Package Manager) is a package manager I have created for the BLZ open-source programming language. Running on Java, it consists of three primary functions: `init`, to set up a new package directory, `beat`, to get a package's required dependencies, and `beat`, to upload a package to the cardiovascular. This program uses YAML files as metadata that facilitate these functions.

Throughout the development, testing, and debugging of this project, I have been able to reinforce the knowledge that I have learned at Cal Poly; this project has deepened my understanding of file systems, file manipulation, and networking via web requests. This project has also allowed me to obtain new knowledge and understanding: of package managers in general, of YAML (and of reading and parsing it), of symbolic links, and of the extra work it takes to write a program that support multiple operating systems.