Universal Package Manager

Removing the hard parts. Flexible software solutions.

Neal Viswanath

Less Code, More Config: A Platform for Unified Dependency Management

Container-based solutions to build and tooling systems using a universal package management engine

Market

- Developer tools are an under-utilized field. More tools and systems branch even more tools and systems, it is a self-feeding cycle.
- Branch off or update existing open source tools.
- Number of developers on GitHub is over 40M, the world probably has more than 60M developers.
- Tools can be used by hobbyists and professionals.
- Tools can be duplicated and edge out current solutions the more simple and modularizable they are. If you offer a plug-and-play tool there is no loyalty among the user base.
- Companies pay large amounts to cozy up devs.

Flow

1. Describe your environment with dotfiles and config files and commit them to VCS like GitHub.
2. Use the “unified installation engine” to parse and install these files.
3. Verify and develop.

The idea is to have a single dependency management tool that aggregates other tools (pip, yarn, brew etc.). We aren’t unifying every package manager but rather creating an opinionated solution that builds your environment. If a node package is specified the underlying manager may be yarn or npm, this is not visible or customizable for the dev. The developer does not specify the package manager, only their environment - we figure out a way to instate it.

.envconf

- Describe your environment as a . file
- JSON/Yaml formatted
  - operating system/base image
  - global dependencies (brew, apt)
  - global applications (chrome, postman, docker etc.)
  - languages (node, python3, ruby)
    - global language dependencies (npm -g)
  - toolkits
  - custom commands to run on initial build
  - command aliases
  - scripts
  - files
- Namespace and tag your different .envconf sections so you can modularize your installs. Add deps to namespace ‘python’ or tagged ‘py’ to install only a python specific environment etc.

.envconf Hub

The Hub, like Docker Hub or npm registry, can host .envconf files, dotfiles or environment terminal images tied to git repos for users to install from.

Unified Installation Engine

- Aggregation of dependency and package management systems
- Uses the .envconf to install your dependencies.
- OS (mac, linux) - brew, apt, apt-get
- Language specific (python, node) - pip, npm, yarn, gems
- Command runner - run your own sudo apt or curl commands to pull sh files
- Binary installers - .deb, .tar.gz

Unified Installation Engine tools

- ‘export’: Auto generate your .envconf through our engine toolset. It looks in your /bin/ and path and fetches your OS specs to pull out dependencies.
  - -g: export to git repo
  - --file-format: json/yaml
- ‘install’
  - -g: from a git repo
  - --file-format
- ‘remove’: removes the dependency you specify (language, dep, toolkit, application)
- ‘list’
- ‘add’: adds the specified dependency/command to your .envconf locally/on GitHub
  - -d: dependency
  - -l: language

Developer “Swiss Army Knife”

- CLI tools, images, extensions, applications
- toolkits
- backpacks
- container management system
- .envconf
- VCS config management
- atomized terminal environments
- remote ssh
- gesture/voice-based developer taskrunner, OS event publishing
- enterprise solutions
- auto sync, auto gen
- log ingestion, container ‘spies’
- market assessment

toolkits

- packages
- /bin/
- languages
- aliases
- commands
- files
- dependencies
- configurations

`toolkits` are the concept of a configuration that describes a development environment you can install on your computer. Developers version control their dotfiles - toolkits are an extension of this concept. Toolkits can be namespace or tagged. A .envconf file is a superset of toolkits. A toolkit is simply a .envconf that describes a single namespace like ‘python’. If you want to have a python toolkit you can describe your deps and even namespace or tag sections in the python toolkit to only install the ‘scientific tools’ or ‘API building’ dependencies, files, configs.

Ephemeral Installations

- Only install a toolkit or backpack for a specified time period before a clean-up agent runs through and removes them from your environment.
- Ex. install .pythontools.envconf -e --time-period=1h

Examples of toolkits

- Language Specific Toolkits: Python, Node
  - python
    - scientific-tools
    - graphical-tools
    - datascience-tools
    - api-tools
    - textparsing-tools
    - scraping-tools
- File-based toolkits: files that help your development
- Script-based toolkits: scripts aliased to commands
- Alias toolkits

Backpacks

- The idea is to have a ‘bag’ to carry your tools around with.
- Backpacks are containerized environments that are tied to 1 or more toolkits or .envconf files. A .[backpack].envconf file will describe a list of repos/file paths/public toolkits to stick into a containerized environment.
- Obviously they will only work as long as the OS or base image described in the .envconf files are similar enough (cannot ‘theoretically’ mix mac/linux dependencies, although you may be able to mix them practically by building and running containers inside containers or fuzzy matching dependencies).
- Installation engine runs the container and runs inside the container to pull in dependencies.
- Backpacks are images you run when you want for however long you want to keep your environments alive

CICD: Build/Test Inside a Container Pattern (1)

- Non-standardized and non-distributable environments are problems that developers face. Have a non-deterministic environment makes problems during engineering
- The build/test inside a container pattern spins up a container image with the environment you describe and runs your application build or tests inside of it, then takes the output and returns it back to you.
- Utilize a backpack with your application's dependencies to build/test it and and specify a buildCI or buildCD field in the .envconf file that returns the build files/binaries or test output logs or files back to you.
- Advantages of this (1) Self documenting environment configurations (2) Distributable and consistent building/testing environments (3) No build artifacts generated on your computer (i.e. .cache folders etc.)

CICD: Build/Test Inside a Container Pattern (2)

- Run multiple of the same container and namespace your tests to run specific subsets of them in different containers. Distribute your unit/integration testing workloads in the exact same duplicatable environments.
- Take for example a Node.js application. When building your application there is no need to reinstall ***all*** of your /node_modules/ dependencies during every build. Caching your dependencies between builds is efficient. Simply publish your branch and specify you want to build that branch to your running container. If the build container is always running with the most update-to-date modules inside, then all that is needed is to pull down the updated package.json/dep file and run npm install to manage your deps without reinstating the entire environment. Moreover, specify a cache and restart policy for our engine that restarts the container or re-runs all the initial dependency build commands in the .envconf on whatever timed-policy you want.

CICD: Build/Test Inside a Container Pattern (3)

- If you are creating a CLI or application that developers build and use locally, you can test your builds across 100s of images/OSes theoretically by swapping the base images of your environment builder files

“I brought my own beer” - Portable shells

- Completely export your entire shell (bash, sh, zsh w/ mods) with its configuration, commands and shell dependencies.
- Bring your entire shell as a dependency in a backpack or toolkit.
- Switch between shells in a single terminal.

Dedicated Environment Servers

- Remote SSH into Backpack container environments by running a server with a container management system. SSH directly into containers.
- Issue commands and builds to these containers.
- Allow multiple users into directly into container environments.
- If each developer has their own container environment for dev, they can debug each other’s code or errors without even having to pull down each other’s branches from git and run the code. You can simply ssh into their container and check the error messages and code and run it yourself (environment is already created)

Auto-Syncing

- If you install a dependency (brew install, sudo apt-get etc.) globally in your environment or make changes to any dotfiles (.bashrc, zsh, vim, git), the syncing agent will create a branch in your VCS repo or make a local diff with your PC's hostname and commit the changes to it. You can merge the changes at your own leisure.
- Auto-refresh: creates a hook into your environment's branch (master) on GitHub and installs the difference in dependencies from your last pull/change.
- Sync all your machine environments simultaneously.
- Moreover, run the agent inside a container and you can automatedly sync all your container environments as well

Container ‘Spies’

- Agents running in your containers
  - manage privileges of multi-user containers
  - auto-sync
  - log warnings or task running: if an error occurs what to do, sending emails etc.
  - log ingestion/export: transport your logs to a db/file etc.
  - caching and restart policies
  - REST commands: adds an HTTP server that executes commands inside the container over REST without SSHing or restarting

Security

- If you can describe your environment as a configuration, you can quantify and evaluate it
- Running an analysis on the tools/deps installed in your environment and comparing it to the .envconf that you've specified can shake out all the extraneous dependencies/files to create more consistent and secure environments
- Uncompromisingly automatically have your environment cleaned and keep it consistent across machines even if you temporarily install packages or need to get up and running on a new machine quickly

Add unpredictability to your environments

Like chaos-monkey, you can add chaos to your containers by specifying --add-chaos option. Replaces your base images with more chaotic ones to simulate unpredictable build environments or allows you to configure options that set the process utilization, random kill commands of existing processes, memory limits etc.

RPA to create containers

Sometimes your build commands of your application might be auto generated or have 1000s of steps. Start a command recording session in your terminal to capture all the commands executed until you stop and turn them into a script/container.

Taskrunner

Voice command/gesture based taskrunner: attach scripts to voice or gestures

- Alongside voice, use your OS as an event publisher to attach scripts to:
  - cron
  - log off
  - file update
  - application started

Market Assessment

- user voting board for integrations, languages and tools to be added to our build engine
- open source tools
- popular products and dev. tooling
- surveys
- data from GitHub/BitBucket