Uni Web Topics Presentation

Felix Pojtinger

October 21, 2021

Contents

1 Introduction

1.1 Contributing

These study materials are heavily based on professor Heuzeroth's "Spezielle Themen für Web-Anwendungen" lecture at HdM Stuttgart.

Found an error or have a suggestion? Please open an issue on GitHub (github.com/pojntfx/uni-webtopics-notes):

Figure 1: QR code to source repository

If you like the study materials, a GitHub star is always appreciated :)

1.2 License

Figure 2: AGPL-3.0 license badge

2 Overview

- What is DevOps?
- Which parts of the software lifecycle does it cover?
  - Development
  - Distribution (I will focus on this today)
  - Operation
- What is "cloud native"?
- Why are "traditional" distribution methods still relevant?

3 Development

- DevOps: Also includes development!
- Modern development should not be bound to any client attributes
- It should not matter if the client is a RISC-V Linux machine, a locked-down Windows workstation or an Android phone
- Development should be possible from any platform, for any platform
- The only truly cross-platform application framework is the web
- PWAs make it possible for web apps to have all the features native apps have
- PWAs work offline by default
- Why not make our development environments PWAs?
- Virtual machines and user-friendly hypervisors and containers make it possible to run the editor's backend locally too
- Source code can for example never leave the company's system
- Development environments can be quickly updated and tightened to prevent supply chain attacks and increase reproducibility
- Imagine: You find a Free Software project, and all you have to do in order to contribute is press ".”!
- Onboarding new developers becomes much easier
- Independence of client choice enables the use of much cheaper or constrained client devices
- Open standards and web technologies enable the adoption of new client and server hardware (i.e. RISC-V chips) easier and enables the easy use of and testing on multiple architectures
- Autoscaling, ballooning etc. can be used server-side: There is no need to provision lots of development servers if no one is using them, and if there is a need for a lot of resources (for example if someone is compiling say a

C++ project) the provisioner (i.e. Kubernetes) can dynamically decide to scale up the container or VM

- There is no need to trust a project's build system, everything can be sandboxed!
- There are already multiple "cloud IDEs"
- Most are based on VSCode (or, to be more precise, VSCode's API specification)
- VSCode (or its libre forks, like VSCodium) is already based on web technologies (Electron), so adapting it to run in the browser is possible
- Theia is an example of an alternative implementation of VSCode's API, which serves as a vendor-neutral implementation of VSCode
- Cloud-Native IDEs can either be self-hosted or public SaaS, so lets take a look at some of them!
- GitPod: Live demo
- Codespaces: Live demo
- pojde: Live demo
- But what if we want to develop things that one can't normally develop remotely?
- Apps which require Android devices as a target, require a programmer, USB or Bluetooth and are not using Web Bluetooth/Web Serial (i.e. Android apps, smart home projects, IoT devices, Arduinos)
  - Forward USB over IP
  - Formward DBus over IP for BlueZ
  - Use SSH tunnels
- Apps which require a Wayland compositor/a screen (i.e. desktop Linux apps, GTK/QT apps)
  - Waypipe
  - Use SSH tunnels
- Apps which require public ports
  - Reverse HTTPS/TLS/UDP/TCP proxies to the public web
  - Use SSH tunnels

4 Distribution

4.1 Basic Distribution Principles

- Binaries

  - Compiled forms of software
  - On Linux: ELF binaries, PE binaries on Windows and MACH-O binaries on macOS
  - Binaries can be statically or dynamically linked
    * Statically linked: Since the Linux ABIs are stable, one can depend on them not changing - this allows not linking against any specific C library and makes the resulting binary portable across distributions. It also allows including all external dependencies into the binary, effectively making it a "single-file" distribution method
    * Dynamically linked: Thanks to `dlopen` and package management, dynamic linking can also be used. Most of the time (especially on non-Linux OSes), at least the C library and external dependencies (i.e. `SQLite`) thus need to be available in `LD_LIBRARY_PATH` at runtime; if they are not, the application can't continue. This makes the binaries non-portable across distributions; for example, if a binary is built on a Debian 11 host, it most probably won't run on a Debian 10 host due to the different versions of the GNU C library used. This does however also have a few big advantages, which apply especially to Linux distributions.
    * Demo: Creating a statically-linked (`CGO_ENABLE=1`) Go binary, running `ldd` on it and running it in two containers (Debian and Alpine Linux), then retrying it with a statically-linked (`CGO_ENABLE=0`) binary

- GPG signing
  - GPG: GNU privacy guard; a Free Software implementing GPG (RFC 4880)
  - Signatures allow the user to verify the author of a piece of software
  - To increase security, only signed software should ever be installed - as we'll see later, this is already the case on Linux distributions and their repositories
  - For example: If author Alice publishes an app (lets call it "scihab") and user Bob wishes to be able to verify that the binary has actually been produced by Alice, he can verify that the binary has actually been produced by Alice and hasn't for example been infected with malware by a malicious actor, in which the case the signature (usually a `.asc` file) no longer matches.
  - Demo: Creating a signed binary, verifying it (hydrapp), tampering with it (adding bytes to end), and re-verifying it
  - Show gridge (https://pojntfx.github.io/gridge/)

- Portability
  - Applications should be portable
  - Portability can mean different things: Portability as in amount of

platforms it can be compiled for, platforms it can be compiled on, platforms it can run on in compiled form, constraints the compiled form needs
  - There are many reasons to make apps portable, both from a developer's and a user's point of view
  - Apps can be tuned for portability with a few simple steps (see in part https://drewdevault.com/2021/09/27/Let-distros-do-their-job.html)
    * Distribution as a simple tarball
    * Shipping static binaries
    * Use standard build systems and methodologies (Go, Cargo, Meson, Autotools, CMake etc.), *never* use custom bash scripts to build your software; this will ensure that packaging the software is much easier, as the tooling for the build system probably already exists. It also vastly increases the developer experience (DX).
    * Inclusion of good release notes makes it much easier from a distro's or developer's perspective to be aware of changes that might break the build system or new runtime or target platform requirements
    * Use dependencies carefully (i.e. use them to reduce maintenance overhead and security issues by having external tests on say usecases like IP or Email parsing); too many external dependencies and especially dependencies without a secure external supply chain lead to security issues in the app itself, which make it harder to build and decrease portability (i.e. cryptography libraries often require hardware-accelerated CPU support, which is unavailable in low-end CPUs)
  - Portability is however often overlooked; product owners mostly see no value in it, unless things break. It is up to the developer to take initiative
  - Demo: Compiling the Links browser from source with Autotools

- Reproducibility
  - Compiling the same source code should always reproduce the same binary, byte-for-byte
  - This allows the user and external developers to reproduce the binary
  - It ensures that the binary has actually been built using the source code in question
  - Without reproducibility, the only way to establish that the binary is "trusted" is trusting the developer who GPG signed the binary - they could have, for example, been paid to include telemetry or other malware, in which case the compiled binary would not match the output expected by the source code.
  - Reproducibility in combination with the points above also allow checking if changing the source code actually lead to different results
  - Demo: Compiling a Go binary multiple times leads to a binary with

the same SHA256 hash

- Why we need more than "just binaries"
  - Binaries themselves can be very portable, but are not the best solution
  - Binaries can't (without self-extraction) include assets other than the programs logic
    - Data files (i.e. databases)
    - Runtime-exchangable internationalization/translations
    - Config files
    - Media files
    - Metadata
    - Documentation
  - Binaries aren't self-describing
    - Runtime dependencies (libraries, binaries etc.)
    - Build-time dependencies (headers, compilers etc.)
    - Language, description etc. metadata
  - The solutions: Packages!

4.2 Packaging Overview

- What is a package?
  - Includes the binary, assets, metadata and signature
  - Is self-describing
  - Mostly some form of archive (i.e. RPM, `.tar.gz`) in combination with a metadata file and signature
- What is a package manager?
  - Can install, remove and update packages
  - Mostly two components: Low-level tool to install and remove package files (`dpkg` on Debian, `rpm` on Fedora) and a high-level tool to search, download, install and resolve dependencies (`apt` on Debian, `dnf` on Fedora)
  - Can resolve and install runtime and build-time dependencies (i.e. dependency on C library, SQLite, SDL2, headers for cURL etc.)
  - Can check GPG signatures of
- Repository
  - Can serve packages and their metadata (i.e. versions)
  - Large repository mostly provided by a distribution ("a distribution is the repositories"), with the ability to enable official community repos (i.e. Alpine Linux) and backports (Debian)
  - Custom repositories can also be installed and be included in individual packages, so that installing the package also installs the repository for further updates

- Source packages and tarballs
- Binary packages
- Documentation packages
- Dependencies (build-time, runtime, one-of-many i.e. multiple OpenSSL implementations)
- AppStream metadata and `.desktop` files
- systemd services
- Demo: Downloading, updating, extracting a package

4.3 Distribution to RedHat Linux

- RPM packages

4.4 Distribution to Debian GNU/Linux

- DEB package
- APT repository
- Yum repository

4.5 Distribution to Linux (universal)

- Flatpak
- Flatpak repository

4.6 Distribution to Android

- APK
- F-Droid repository

4.7 Distribution to Windows

- MSI package with auto-updates

4.8 Distribution to macOS

- DMG package with auto-updates

4.9 Distribution to Kubernetes/the Cloud

- Docker
- Kubernetes
- Helm
- Skaffold

4.10 Distribution to WebAssembly

- WASM-Binary
- WASI/wasm_exec equivalents

5 Operation

- Sentry
- OpenTelemetry
- Prometheus
- Grafana

5.1 Pipelines

- Bagop
- Hydrun
- GitHub Actions
- Semantic Release