Uni Web Topics Presentation

Presentation on Cloud Native Development

Felix Pojtinger

2021-11-19

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 let's 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
  - Forward 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: Create 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 using keygaen
- 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/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
  - Builds should be reproducible
  - Source packages contain all information necessary to build the application (including source code, build-time dependencies, patches, metadata for package creation)
  - Tarballs: `tar`-files, tape archives: A linear storage format used internally for physical tapes but also for files. Contains all source code; often zipped (`.tar.gz`)
  - Depending on the package format the source package can contain the source code (RedHat) or use a separate tarball (Debian)
  - There can be additional tarballs, i.e. one with the original source and one with the distro’s patches (i.e. `.debian.tar.gz`)
  - Many systems allow installing from the binary package (see following) and by downloading & rebuilding the source package

- Binary packages
  - Contains the compiled program, data files and metadata (i.e. dependencies on other binary packages)
  - Is usually what is being used to install the software

- Documentation packages
  - Are often separate packages as documentation may not be required for running the software and documentation can be large
  - Can have different license from rest of software, i.e. GFDL
  - Often ends in docs; i.e. in Alpine Linux, for the binary package `mariadb` the documentation package is called `mariadb-docs`

- Dependencies
  - There are two basic types of dependencies: Build-time and runtime
  - Build-time dependencies are required to compile the software; i.e. compilers or dependencies vendored by the distro (i.e. LaTeX packages, headers)
  - Runtime dependencies are required to run the software; i.e. dynamically-linked libraries (i.e. OpenSSL) or other programs that can be launched

  - Dynamic linking is useful in distros as it allows the distro to update all important dependencies (i.e. the SSL library) at once as long as the ABI didn't change
  - Versions and alternatives can be specified; i.e. Go and `gccgo` as one of the Go compilers to choose from or OpenSSL and LibreSSL as one of the SSL libraries to choose from

- Metadata
  - Packages often have metadata: ID, Name, description, author, license, version, URL, changelog etc.
  - AppStream metadata (and to a certain degree, `.desktop` files) is a standard for this data
  - `.desktop` files provide shortcuts, application categories and window menu options (mostly of use in a desktop context, but can also be used as an alternative to launching binaries directly)

- systemd and systemd Units
  - Are the basic buildings blocks of a Linux system
  - Describe a service/daemon
  - Can be either system services (managed by root or an authorized user) or user services (managed by a user)
  - Can describe dependencies on other services; i.e. dependency of backend on database
  - Can launch services in parallel if dependencies allow for it
  - Supports socket activation: Services will only be started if a user makes a request and is stopped afterwards ("scale to zero")
  - Can be enabled, started, stopped etc. based on user input or targets (i.e. after system is turned on, gets an internet connection, has started the shutdown process)

- Demo: Downloading, updating, extracting a package

4.3 Distribution to RedHat Enterprise Linux

- RHEL is a very popular distribution and serves as the upstream of many other distros (CentOS, Rocky Linux etc.)
- Fedora Linux is its upstream
- Is based on the RPM package format and the DNF package manager
- Commercial
- Very long support cycles (at least ten years per major release)
- RPM package format: Demo
- DNF package manager and repositories: Demo

4.4 Distribution to Debian GNU/Linux

- Debian is another very popular distribution that also serves as the upstream of many other distros (Ubuntu, Linux Mint, Pop!_OS etc.)
- Is based on the DEB package format and the APT package manager
- Community-Driven, completely Free Software
- 5 years support per major release
- DEB package format: Demo
- APT package manager and repositories: Demo

4.5 Distribution to Linux (universal)

- Flatpak is a universal package format for Linux (“apps for Linux”)
- Sandboxed
- Standalone/runtime-based (does not depend on OS-provided libraries)
- Runs on all Linux distributions, even older major releases: Allows the user to install modern software on stable systems like Debian
- Permission system: Camera, File Access, Terminals
- Portals: Allow interaction with host system (i.e. screen sharing, file access etc.)
- Includes a universal repository system
- Demo: Setting up Flatpak and installing “Video Downloader”
- No concept of source packages; a single YAML/JSON manifest (reverse FQDN) is the source package and serves as the build system too
- Tight integration with GNOME Builder
- Demo: Creating and building a Flatpak with GNOME
- Why is Flatpak relevant in a web context?: Similar to Docker, which I will show later

4.6 In Comparison: Comparison to Distribution with Android, Windows and macOS

- As we are in a web context I will only take a short look at proprietary platform’s distribution mechanism
- The intention here is to show why these platforms are not viable for secure usage
- Android
  - APKs can be published to a “app store” which is comparable to a repository, i.e. F-Droid, Huawei AppGallery, Samsung Galaxy, Amazon App Store or Google Play.
  - APKs are ZIP files with a Manifest, Dalvik Bytecode, Resources and native libraries
  - Updates are handled by the app store

  - Permissions are handled by the operating system

- Windows
  - MSI packages or self-extracting installers are used
  - No concept of source or binary packages, no reproducibility - most software is fully proprietary and violates the user's rights
  - Files are tracked through the registry, but arbitrary code can be run at install time
  - Has an app store without apps
  - Almost all apps implement their own update systems which replaces the app itself with the update

- macOS
  - DMG images or `.pkg` installers are used
  - No concept of source or binary packages, no reproducibility - most software is fully proprietary and violates the user's rights
  - Files are not tracked, but in the case or a DMG image are extracted
  - Sandboxing is possible, but not widely used
  - Has a heavily restricted app store which costs $100 per year and a $2000 computer to send apps to
  - Most apps implement their own update systems which replaces the app itself with the update

4.7 Distribution to Kubernetes/the Cloud

- Native
  - `scp` to server
  - Configure systemd Unit
  - Get status and logs using `systemctl` and `journalctl`

- Docker
  - Like VMs, but the host kernel is shared (isolation taking place by using CGroups)
  - "Apps for servers"
  - Workflow
    * Define Dockerfile
    * Create OCI image by using `docker build`
    * Push using `docker tag` and `docker push`
    * Pull using `docker pull`

        * Start using `docker run`
        * Get logs using `docker logs`

- Kubernetes
    - A declarative orchestration system for containerized applications
    - Objects
        * Pods: One or multiple containers running an app
        * Deployments: Configuration of how pods will be created
        * Service: Makes an app (i.e. a gRPC API server) reachable from other pods within the cluster
        * Ingress: Makes a service reachable from the outside
        * Autoscaler: Allows vertical or horizontally scaling a deployment
    - Internal Components
        * CRI: Container runtime interface, manages containers (historically Docker)
        * CNI: Container network interface, manages networking between containers (i.e. Flannel)
        * CSI: Container storage interface, manages volumes and attaches them to containers (i.e. OpenEBS)
    - Scheduler: Efficiently scales the pods across the cluster
    - Workflow
        * Declarative configuration using JSON or YAML
        * Define YAML
        * Setup Kubeconfig
        * Deploy objects to the cluster using `kubectl apply`
        * Get objects using `kubectl get`
        * Delete objects using `kubectl delete`
        * TUIs and GUIs like `k9s` and Lens can make management easier

- Helm
    - “APT/DNF for Kubernetes”
    - Workflow
        * Define k8s YAML
        * Add Go template syntax
        * Define stack (YAML + Go templates)
        * Define values (template values)
        * Define chart metadata (version etc.)
        * `helm package`

        * Upload `.tar.gz` to repo
        * `helm repo add`
        * `helm install repo/chart`
        * Take inventory using `helm list`
        * Delete using `helm delete`
        * TUIs and GUIs like `k9s` (which also supports Helm) and Kubeapps can make management easier

- Skaffold
    - "Skaffold handles the workflow for building, pushing and deploying your application, allowing you to focus on what matters most: writing code."
    - Workflow
        * Invoke `skaffold` (dev or production)
        * Builds image using Docker
        * Build Helm chart
        * Deploys Helm chart to Kubernetes
        * Repeats when source files change (in development)
    - Features
        * No need to configure development clients
        * Can use ressources of the cluster
        * Offline development is possible by using a local cluster
        * Debugging is universal across languages
    - Configuration
        * Define Skaffold YAML
        * Develop with `skaffold dev`
        * Deploy using `skaffold run`
        * Delete using `skaffold delete`

4.8 Pipelines

- bagop & bagccgop
    - Tools to build for a lot of platforms
    - Examples in Go, but these kind of tools exist for almost all programming languages
    - Cloud environments and Kubernetes clusters are heterogeneous: They can have many different CPU architectures (x86_64, arm64, riscv64, ppc64 etc.)
    - Portability is the key factor; your apps should not assume their target platforms and use standard build systems, which makes this much easier

  - bagop and bagccgop try to build for all possible targets (i.e. Linux on riscv64, FreeBSD on x86_64 etc.) by default and then allows you to disable platforms which one can't compile for
  - bagop works for pure Go, is fast and builds for ~40 targets by default
  - Demo: Build a Hello World Go app with bagop
  - C libraries further complicate the situation: These require dynamically linked libraries and headers at compile/runtime, which means that they also require related dependencies at compile time for the specific target architecture and operating system, even if static linking is used
  - The solution is to either use tweaked `chroot`s and a cross-compiler (fast but tricky to set up) or `binfmt` and `qemu-user-static` (slow but easy to set up)
  - bagccgop is an automated way to use the first solution
  - Can build for all Debian-supported platforms (including "esoteric" ones like 32-bit PowerPC)
  - Can build static binaries with CGo thanks to GCCGo
  - Allows the use of pretty much any C library with "cross-compilation by default" (OpenSSL, SDL2, Vulkan etc.)
  - Demo: Build a Hello World CGo app (calls `printf`) with bagccgop and run it on the PowerBook
  - Using a tool like this makes sure that your app is actually portable and notifies you if dependencies are introduced which break portability

- Hydrun

  - Tool to run a command in the current directory on another processor architecture or operating system
  - Is very useful for both building software and testing software (the "runtime equivalent" of tools like bagop and bagccgop)
  - Enables reproducibility of the build system and decreases reliance on available (public) build environments like the VMs available on GitHub actions
  - Can be used to run your build locally and prevent issues which would otherwise only be "testable" by pushing to i.e. the Git repository connected to the CI system
  - Can be used to test built binaries
  - Demo: Build a static binary on Debian GNU/Linux through hydrun, bagop and bagccgop and then test if the binary is actually static by running it in Alpine Linux using it

- GitHub Actions

  - Allows you to run commands on remote, ad-hoc machines in response to triggers (i.e. a commit has been pushed to a Git repository or a schedule)

  - Can be used to build, test and publish software projects
  - Is fairly generic and can be extended to use custom actions written in i.e. JavaScript
  - Is configured using YAML
  - Can use GitHub's hosted machines or self-hosted nodes
  - **on**: Describes the triggers for the build system
  - **jobs**: Describes the commands to run and actions to execute for a build configuration
  - **strategy**: Describes a matrix of build configurations that can be executed in parallel (i.e. different binaries for the project, different build commands)
  - Pre-written actions can be very useful as a low-maintenance way to add complex functionality:
    * `actions/checkout`: `git clone`s the source code of the branch on which the event has been triggered
    * `docker/setup-qemu-action`: Installs QEMU, which allows the pipeline to run binaries for different target architectures
    * `docker/setup-buildx-action`: Installs `buildx`, the next-generation build command for Docker with better support for multiple architectures
    * `actions/upload-artifact` and `actions/download-artifact`: Upload/download an artifact to the current run's cache (i.e. to exchange it between jobs)
    * `marvinpinto/action-automatic-releases`: Create as a GitHub releases and uploads assets; see the next section

- Semantic Versioning and Semantic Release
  - Defines a formalized versioning scheme
  - Used almost universally
  - Three parts of a version: MAJOR.MINOR.PATCH
    * MAJOR: Increment if incompatible API changes were made
    * MINOR: Increment if backwards-compatible features were added
    * PATCH: Increment if backwards-compatible bug fixes were added
  - Is not only useful to those writing & releasing software, but also those consuming it (i.e. distributions like Debian or projects which depend on external libraries)
  - Semantic Release is a tool to make using it easier
    * 1: `git tag` (i.e. `git tag v0.1.0`)
    * 2: Push
    * Semantic Release will create a GitHub release, corresponding changelog and upload-/publish release assets (i.e. source code or binaries)
    * Demo: Release example software using GitHub Actions it