DEV-PRO

Software Delivery Process 101

What it takes to deliver the code change

Sergii Sinienok
Cloud Architect @ Dev-Pro.net

Agenda

We're going to wear developer's hat. Take a look and touch the source code, deliver the real app.

- Software Delivery and Development - Theory
- Delivery Workflow for various types of apps
- Live development and delivery demos
- Lessons Learned and Best Practices

Software Delivery and Development - Definitions

Development - programming the chain of actions and conditions in order to achieve the expected system reactions upon the defined inputs.

Delivery - programming the chain of actions and conditions in order to cost and time effectively satisfy the end customer needs, so to generate maximum revenue

Software Delivery and Development - Side by side

Development

- Strives to tech perfection
- Is a good place for mental fights
- The result fulfills only your and your mama demands
- Expected to work on your local environment

Delivery

- Strives to "Value on-time"
- Is a good place for collaboration
- The result fulfills a demand of thousands people
- Needs an **infrastructure** in order to work

Delivery Workflow for various app types

- Desktop App
- Mobile App
- Website
- Public facing API
- The service with UI Single Page Application (SPA), API, and other features.

Software Delivery - Build step

- Construction of something that has an observable and tangible result.
- Process of converting source code files into standalone software artifact(s) that can be run on a computer. Produce artifacts.

Functions of the Build step:

- Version control/Tagging.
- Code qualit check.
- Compilation/Translation of the source code into executable instructions.

Delivery Workflow - Build Artifacts per Application type

- **Desktop App** - EXE or MSI files.
- **Mobile App** - .app (iOS) or .apk (Android) bundles.
- **Website** - a folder with files or archive.
- **Public facing API** - a folder with files or archive.
- **The service with UI SPA, API, and other features** - a folder with files or archive.

"Build Artifacts per Application type"

Software Delivery - Hosting Infrastructure

- Facility(ies) that houses IT equipment, such as computing servers and networking kits. Provides basic utilities for it to work (electricity, cooling, etc.).
- Software applications, such as web servers, DB servers, etc., installed on computing servers.
- Maintenance service for all the above.
- Physical and/or remote access to the IT equipment to configure and operate.
- All the otther perks that make one infrastructure better from another.

Hosting Infrastructure for Desktop App

- Communicates with an "outside world" via Internet.
- All configuration is packed along with the main app file.
- Distribution server - main source for end users to download the app from.
- Runs on End User's local environment (Computer).
- Extremely versatile envs, no control for the app.

"Hosting Infrastructure for Desktop Application"

Hosting Infrastructure for Mobile App

- Communicates with an "outside world" via Internet.
- All configuration is packed along with the main app file.
- App Store - main source for end users to download the app from.
- Runs on End User's local environment (Phone).
- Computation Power and Network bandwith are strictly limited.

"Hosting Infrastructure for Mobile Application"

Hosting Infrastructure for Website

- Configuration becames an Issue because of Secrets.
- Static VS dynamic websites.
- HTTP protocol is the main communication channel.
- HTTP vs HTTPS.
- Domain names.

"Hosting Infrastructure for Website"

Hosting Infrastructure for SPA with API and Services

- SPA vs Website.
- At least two domain names instead of one.
- Website demo with dev tools (`static-website-example`).
- SPA demo with dev tools.

["Hosting Infrastructure for SPA with API and services"]()

Software Delivery - Deploy Step

All of the activities that make a software system available for use. Activities examples: Release, Installation, Uninstallation, Update, State Tracking, etc.

We take our **Build Artifacts** and make it **work as expected** on existing **infrastructure**.

!!! Deploy Step MUST be automated as much as possible !!!

Demo #1: Manual app deployment from scratch

Demo Application - Requirements

There should be the service to manage information about Marvel Heroes. This service should do the following:

- Has the UI as SPA that will talk to API to manage data.
- List all the Heroes available.
- Create new Hero entries.
- View details and Edit existing Hero BIOs.

What we'll be doing:

1. Infrastructure:
    1. Server for UI app.
    2. Server for API.
    3. DB Server - cloud-hosted.
    4. Access to manage infrastructure
2. Environment configuration:
    1. Web server, corresponding packages and config for UI app.
    2. API server, environment and app-level configs for API.
    3. DB instance set up and config in the cloud.
3. Network configuration and Integration testing.

Note: We'll be using GCP for our infrastructure for this particular example.

UI Server - Infrastructure

1. Spin up `n1-standard-1` Virtual Machine in GCP (Name: code-delivery-ui; Region: europe-west1; Boot Disk - Ubuntu 19.04; Firewall - Allow HTTP traffic)

2. Configure SSH access there to be able to manage the server. Generate SSH key, then add to Compute Engine > Metadata > SSH Keys, test connection:

```
ssh-keygen -t rsa -b 4096 -C "ansible"
cat ~/.ssh/ansible.pub | pbcopy

ssh -i ~/.ssh/ansible ansible@{public-ip}
```

Note: Usually it takes months to get the server up and running with Bare Metal.

UI Server - Environment configuration. NGINX installation

- Greating NGINX user, allowing SSH for it:

```
sudo adduser nginx

sudo usermod -aG sudo nginx


sudo rsync --archive --chown=nginx:nginx ~/.ssh /home/nginx/
```

- SSH as nginx user:

```
ssh -i ~/.ssh/ansible nginx@{public-ip}
```

UI Server - Environment configuration. NGINX installation

- Install NGINX:

```
sudo apt update

sudo apt install nginx


sudo ufw allow 'Nginx HTTP' // Adding firewall rule.
```

- Test NGINX installation by opening http://{public-ip}/ in the browser.

UI Server - Environment configuration. App deployment

- Build the app:

```
npm install && npm run build:prod:en
```

- Upload the build to the server:

```
sudo chown -R $USER:$USER /var/www/html // While SSHd to the server,
                                        // Change the web folder owner
rsync -avz -e 'ssh -i ~/.ssh/ansible' dist/browser/ nginx@35.240.97.91:/var/www/html
```

- Test the deployment by opening http://{public-ip}/ in the browser.

API Server - Infrastructure

1. Spin up `n1-standard-1` Virtual Machine in GCP (Name: code-delivery-api; Region: europe-west1; Boot Disk - Ubuntu 19.04; Firewall - Allow HTTP traffic)

2. Configure SSH access there to be able to manage the server. Generate SSH key, then add to Compute Engine > Metadata > SSH Keys, test connection:

```
ssh-keygen -t rsa -b 4096 -C "ansible"
cat ~/.ssh/ansible.pub | pbcopy

ssh -i ~/.ssh/ansible ansible@{public-ip}
```

Note: Usually it takes months to get the server up and running with Bare Metal.

API Server - Environment configuration. NGINX installation

- Greating NGINX user, allowing SSH for it:

```
sudo adduser nginx
sudo usermod -aG sudo nginx

sudo rsync --archive --chown=nginx:nginx ~/.ssh /home/nginx/
```

- SSH as nginx user:

```
ssh -i ~/.ssh/ansible nginx@{public-ip}
```

API Server - Environment configuration. NGINX installation

- Install NGINX:

```
sudo apt update

sudo apt install nginx

sudo ufw allow 'Nginx HTTP' // Adding firewall rule.
```

- Test NGINX installation by opening http://{public-ip}/ in the browser.

API Server - Environment configuration. NodeJS installation.

NodeJS is an engine to run our API. Will not work without it:

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.3.8/install.sh | bash //NVM
source ~/.profile
nvm ls-remote | grep -i "latest lts"

nvm install v10.16.3
nvm install v4.9.1 // For env inconsistency demo
```

API Server - Environment configuration. App deployment.

- Make a Build

```
npm install && npm run build
```

- Upload the build to the server:

```
rsync -avz -e 'ssh -i ~/.ssh/ansible' package.json nginx@{public-ip}:/home/nginx/code-deli
rsync -avz -e 'ssh -i ~/.ssh/ansible' dist nginx@{public-ip}:/home/nginx/code-delivery-api
```

API Server - Environment configuration. App deployment.

- Run the application (Uses the DB from the cloud)

```
export DATABASE_URL={url}

npm i && npm run start
```

- Test locally - Works!

```
curl http://localhost:3000/
```

API Server - Public access.

In order to access our API via Internet, we need to set up a Reverse Proxy to route our HTTP calls from outside to localhost. Let's use NGINX for this purpose.

```
sudo nano /etc/nginx/sites-available/default
```

API Server - Public access.

```
server {
    ...
        location / {
            proxy_pass http://localhost:3000;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection 'upgrade';
            proxy_set_header Host $host;
            proxy_cache_bypass $http_upgrade;
        }
    ...
    }
```

API Server - Public access.

```
sudo nginx -t

sudo systemctl restart nginx

npm i && npm run start // Restart the application
```

- Test our API:

```
curl http://{public-ip}/ //Should return Hello World!
```

Manual app deployment - Pros and Cons

- Pros:
  - No Pros! If you think there are, you wrong!!!
  - There's no excuse to deploy that way in a modern world!
- Cons:
  - Error-prone because of a ton of manual steps.
  - Takes time. A lot!
  - People owns expertize, not machines.
  - Hard to document: UI and OS envs are subject to shange;
  - Edge-cases.

Application Versioning?

How to version the app?

- Need a label in VCS (git tag): `build-1.0.1`
- Name the build artifact appropriately: `myapp_build-1.0.1`
- Make the app to show its version: UI label, API endpoint, etc.

Delivery Workflow - High-level cycle

Source Code Under VCS => Build Artifact => Versioning => Infrastructure => Deploy => Success!

Delivery Workflow - Common Best Practices

- All code shoul be under Version Control System (VCS). No exeptions.
- Every deploy should be versioned.
- At least two environments: Staging and Production.
- Document everything that requires documentation.
- Everything as a Code.
- Automation Everywhere.

Demo #1 - Lessons Learned

- Need to version my app appropriately.
- Want to easily find the version number.
- Want to configure my app somehow.
- Automate Infrastructure routines.
- Simplify the Deploy step.

Demo #2: Handling The Lessons Learned

Handling The Lessons Learned - Level Set

- Introduce the simplest Configuration Management possible - env vars.
- Code Package, Environment Isolation. Docker.
- App Versioning - the Right Way.
- Infrastructure Automation, Infrastructure as Code.
- Build and Deploy separation: `Build Once` best practice.
- Deploy: Easy-Pizy.
- Logging: Types and Troubleshooting.

Demo #2 - What we're going to do?

- Deploy our UI and API apps to the cloud
- Take a look at AWS Cloud
- Cover our "lessons learned" with corresponding theory and practical examples

Config Management - WTF?

Configuration management (CM) is a systems engineering process for establishing and maintaining consistency of a product's attributes throughout its life.

WAAAAAAAT?

To keep it simple, the majority of apps require some initial information in order to launch and execute.

Config Management - Examples

Our UI app configs. We need to know where the API lives (at least):

```
API_URI=http://localhost:3000/

API_JWT=...
```

Config Management - Examples

Our API app configs. We need to know where the DB is and how to verify tokens (private key):

```
DATABASE_URL=...

JWT_SECRET=your-secret-whatever
```

Config Management - Ways to store configs

- Hardcode/embed into source code
- Store in config file
- Store in DB
- Environment Variables
- Config Management systems (etcd, Hashi Corp Vault, etc.)

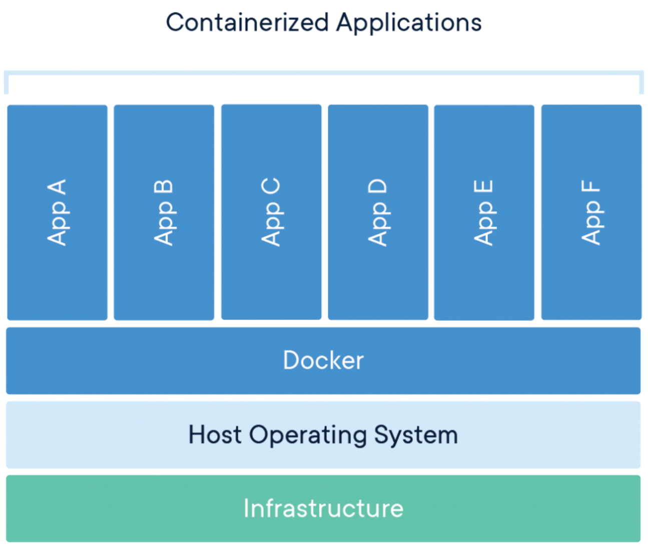
Code Package and Isolation - WHY?

Code Package - Standardize the delivery artifacts to simplify operation.
Isolation - Making our delivery artifct self-contained.

(Went through Deckerfiles for UI and API)

Package and Isolation - HOW? Docker

- **Standard:** Docker created the industry standard for containers, portable anywhere
- **Lightweight:** do not require an OS per application, driving higher server efficiencies and reducing server and licensing costs
- **Secure:** Docker provides the strongest default isolation capabilities in the industry



App Versioning - the Right Way

- Versioning approach - "SemVer" as industry standard
- Use Tags in Source Control (git, svn, tfs, etc.)
- Make the app version easily accessible for users
- Collect changelogs, make it widely accessible

```
git tag // Versions

git log // Changelog
```

Infrastructure Automation

- "Machines" MUST own infrastructure details, not people
- Document important infrastructure details along with its "topology"
- Automate the infrastructure routines as much as possible
- Make your Infrastructure Automation repeatable
- `Infrastructure as Code` best practice
- Infrastructure State versioning

(serverless.yml review + CloudFormation stack Designer)

Build and Deploy Steps Separation

- **Build** - producing and publishing application artifacts
- **Deploy** - delivering build artifacts to the environment, along with all the configuration required for it to launch
- `Build once - deploy everywhere` best practice

(AWS ECR review)

Deploy: Easy-Pizy

- Deploy is as much of "no-event" as possible
- `Anybody can deploy` as a best practice
- `Deployment guru` as dangerous antipattern

Logging - must-have for the app

- Log levels: DEBUG, INFO, WARN, ERROR, FATAL
- Structured Logging vs Plain Text logs
- Distributed Tracing
- Internal and Customer-facing logs
- Retention policy and costs

(AWS CloudWatch walkthrough)

How to deliver new feature? - Add Heroe's Favorite Color

- Create branch: `git checkout -b favorite-color`
- Modify UI - test with local API
- Modify API - test locally
- Add DB migration, run on local DB
- Create the Pull Request: description, approvals, merge
- Deliver UI, API and migrations to the cloud from `master`

Favorite Color Feature - new branch. Why?

- Isolates the scope of work
- Allows to save half-baked results remotely without affecting the other teams
- Locates the scope for dev and QA verification
- Sandboxed Point of collaboration

Favorite Color Feature - Modify UI

- Put the field on the form - `ui/src/app/modules/heroes/pages/hero-detail-page/hero-detail-page.component.html`(1)
- Add it to formBuilder - `ui/src/app/modules/heroes/pages/hero-detail-page/hero-detail-page.component.ts` (2) error
- Add field to data model - `ui/src/app/modules/heroes/shared/hero.model.ts` (3)

The field was sent to API, not persisted in DB yet. Move on.

Favorite Color Feature - Modify API

- Observe local DB, http://localhost:8080, `apidb.hero` table. No field
- Add field to the data model - `api/src/entity/hero.ts` (1), notice in JSON on UI. Valdation Error
- Add field processing to controller - `api/src/controller/hero.ts` (2). Not persisted in DB, hm?
- Observe local DB, http://localhost:8080, `apidb.hero` table. No field?
- Aha! Migration: `npm run typeorm migration:run`

Code Delivery vs DB changes delivery

- Create migration for new field: `npm run typeorm migration:generate -- -n favorite-color`
- Apply it to the local DB: `npm run typeorm migration:run`
- Observe local DB, http://localhost:8080, `apidb.hero` table. Here it is!
- Test UI + API locally. Bingo!

Feature delivery: AWS

- Deply API: `sh deploy-api.sh`
- Check with PostMan: fail, WTF?
- Migrate: `npm run typeorm migration:run`
- Check. Success!!!

Feature delivery: AWS

- Deply UI: `sh deploy-ui.sh`
- Check. Success!!!
- Done!

DEV-PRO

Feature delivery: Sum up

Good

- It is automagically versioned
- We can access version info easily
- High degree of automation
- Deployment is simple and straightforward for engineers

Opportunities

- Running on local machine
- Hard to use for non-engineers
- No Changelogs Automation and presentation
- Low visibility for business at scale

Delivery Pipeline - godmode

Does it make sense to invest at that degree?

- "Time to Market" matters .... more and more "Conway's law"
- DevOps Research and Assessment (DORA) State Of DevOps 2018, p.12
- "DORA State Of DevOps 2019: ELITE group growth from 7% to 20% (~285% growth)"

Delivery Pipeline - CI/CD. Where to start?

Pick a tool => Build on it => Done :)

How to select CI/CD tool:

- CI/CD Hosting Requirement - Do you want/can manage it or prefer it as a service?
- Budget for CI/CD - How often (once a months, week, day?) do you plan to produce builds/do deploys?
- Features - Everything as Code? Enterprise Auth? Cloud-native?

"Choosing a CI-CD Tool"

DEV-PRO

Top-5 CI/CD tools 2019

- Jenkins
- CircleCI
- TeamCity
- Bamboo
- GitLab

["Best 14 CI/CD Tools You Must Know | Updated for 2019"]

Delivery Pipeline - DevOps?

DevOps is a set of practices, not a role! "DevOps Manifesto".

Basic practices:

- CAMS Value (Culture, Automation, Measurement, Share Knowledge)
  "Culture eats strategy for breakfast"
- Continuous Delivery. Conveyor-like process.
- Everything as Code.
- Continuous Monitoring.

Demo #3 - Full Size ~~SUV~~ CI/CD pipeline

Git repository

- "Structure"
- Versioning: `git tag`

Release notes and knowledge sharing

- "Release Notes"
- "API Doc"
- "Project Knowledge Sharing"

CI/CD tool and process visibility

- "Build"
- "Deploy"
- "Test Reports"

Infrastructure - Integrated Platform

- Cloud-based hosting
- Logs
- Monitoring
- Integrations

Summary

- Code Delivery is not a Unicorn
- Project-based investments into Delivery Pipeline
- Automation is a key
- DevOps is a set of practices
- Use Cloud whenever possible

Q/A

DEV-PRO

Thank you!