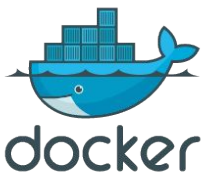


Docker für Angular-Testing und Deployment

Thomas Kruse
(trion development GmbH)



Thomas Kruse

- Entwickler, Trainer, Berater
 - www.trion.de
 - @everflux
- Java User Group Münster
- Frontend Freunde Münster



Agenda

- Ganz kurz: Docker
- Angular CLI ohne Docker
- Angular CLI mit Docker Build-Container
- Tests mit Docker
- Deployment mit Docker

Docker

- Prozess Isolation
- Idee: Build-Ship-Run
 - Container abstrahieren von Inhalt
 - Handhabbare Einheit zur Verteilung von Software
 - Isolation zur Laufzeit

Docker Begriffe

- Image
 - Vorlage für einen Container (Filesystem)
- Container
 - Instanz eines Image, lauffähig als Prozess
- Dockerfile
 - Beschreibt Build/Aufbau eines Image

```
EXPOSE 4242
ADD .
RUN apt-get update
MAINTAINER "John Doe"
FROM ubuntu:latest
```

Benefits Container

- Reproduzierbare Ergebnisse durch Images
 - Wichtig für Build-Container: Bereitstellung von Umgebung/Werkzeugen
- Gleichartige Behandlung von Anwendungen mit unterschiedlichen Technologien
 - PHP, Java, node.js, ruby: Alles nach außen ein Container
- Gute Ressourcennutzung physischer Maschinen
- Grundlage für Cloud-Umgebungen

Angular-CLI

Angular CLI



- Opinionated Build
 - Abstraktion von konkreter Implementierung
 - Optimierungen einfach nutzbar
- Deckt Lebenszyklus eines Projektes vollständig ab
 - Projekt erstellen
 - Testen
 - Production Build

Lebenszyklus eines Projektes

App Erzeugen

ng new DemoApp



```
ng new DemoApp
File Edit View Search Terminal Help
→ projects ng new DemoApp
Installing ng
create .editorconfig
create README.md
create src/app/app.component.css
create src/app/app.component.html
create src/app/app.component.spec.ts
create src/app/app.component.ts
create src/app/app.module.ts
create src/assets/.gitkeep
create src/environments/environment.prod.ts
create src/environments/environment.ts
create src/favicon.ico
create src/index.html
create src/main.ts
create src/polyfills.ts
create src/styles.css
create src/test.ts
create src/tsconfig.app.json
create src/tsconfig.spec.json
create src/typings.d.ts
create .angular-cli.json
create e2e/app.e2e-spec.ts
create e2e/app.po.ts
create e2e/tsconfig.e2e.json
create .gitignore
create karma.conf.js
create package.json
create protractor.conf.js
create tsconfig.json
create tslint.json
You can 'ng set --global packageManager=yarn'.
Installing packages for tooling via npm.
Installed packages for tooling via npm.
Successfully initialized git.
Project 'DemoApp' successfully created.
```

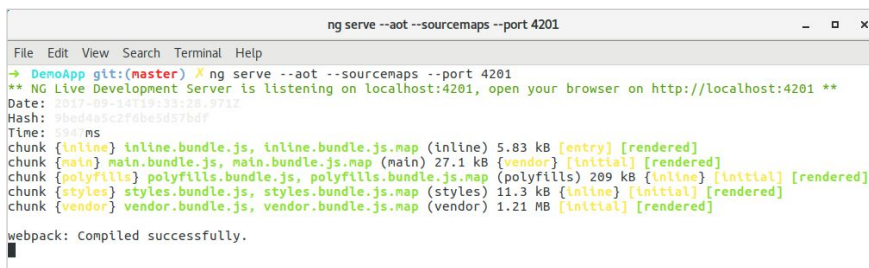
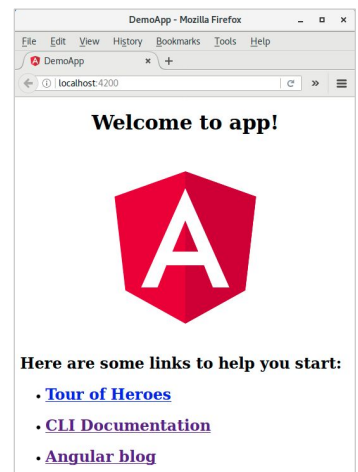
Struktur aufbauen/generieren

```
ng generate component greeting  
  
ng g c greeting  
  
ng g service user  
  
ng g s user  
  
ng g module MyFeature  
  
ng g m MyFeature
```

Komponenten
Direktiven
Services
Pipes
Guards
Module
Klassen
Interfaces
Enums
...

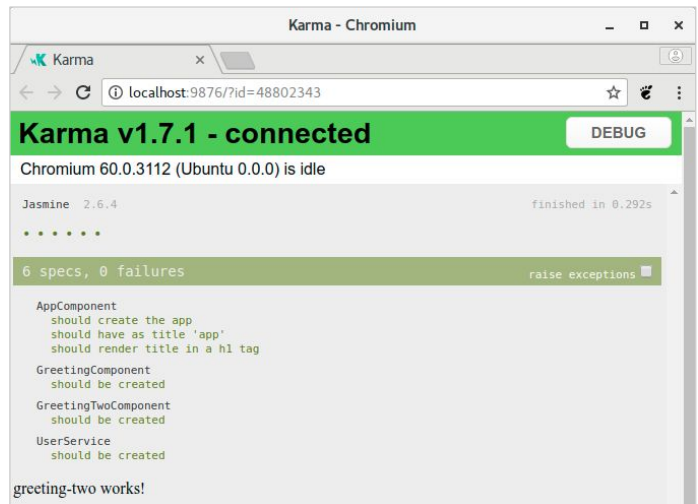
Entwicklung

```
ng serve (--aot, --port 4201, --env=prod)  
ng serve (--sourcemaps, --preserve-symlink)
```



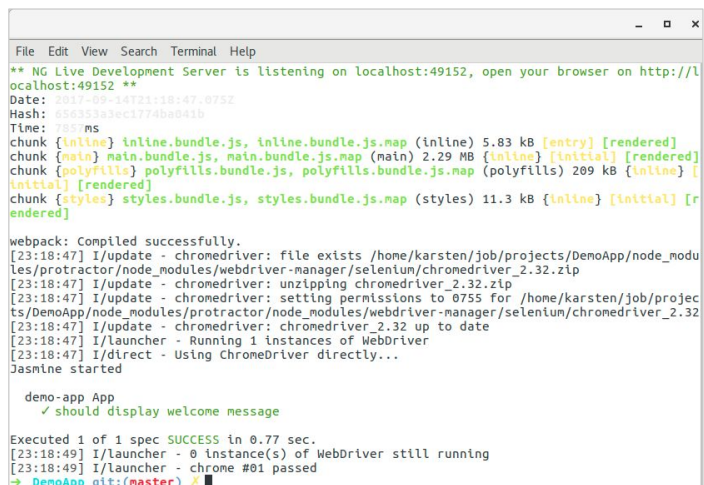
Unit Tests

```
ng test --watch false
```



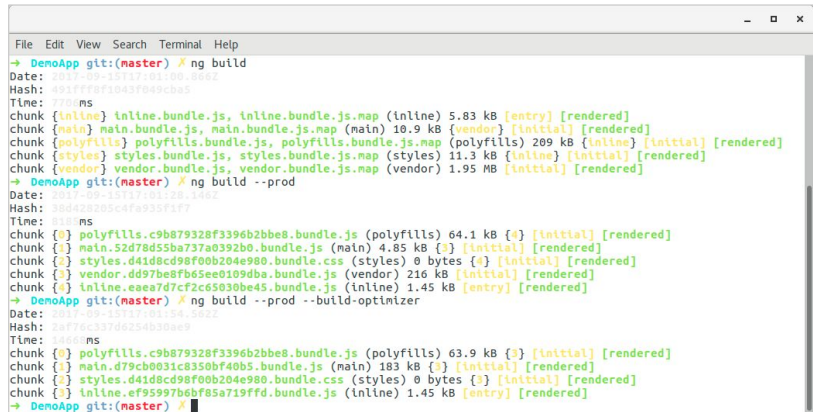
Ende-zu-Ende Tests

```
ng e2e
```



Deployment

`ng build (--prod)`

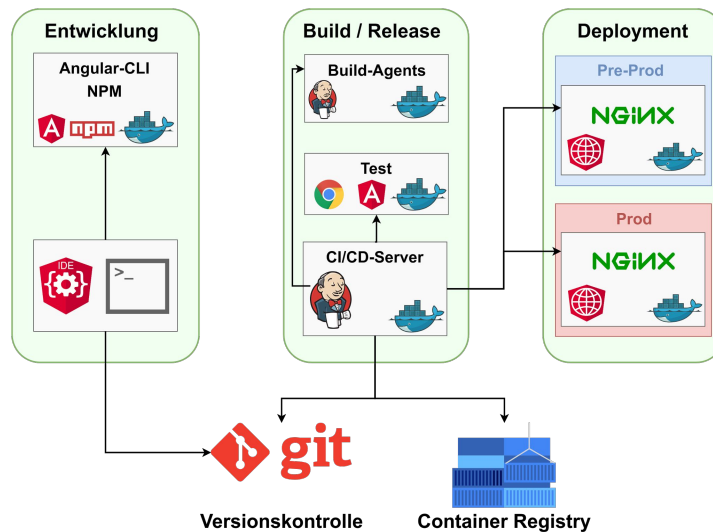


```
File Edit View Search Terminal Help
→ DenoApp git:(master) # ng build
Date: 2017-09-15T17:01:00.000Z
Hash: 491f9f9f1043f049cbe5
Time: 778ms
chunk {inline} inline.bundle.js, inline.bundle.js.map (inline) 5.83 kB [entry] [rendered]
chunk {main} main.bundle.js, main.bundle.js.map (main) 10.9 kB [vendor] [initial] [rendered]
chunk {polyfills} polyfills.bundle.js, polyfills.bundle.js.map (polyfills) 209 kB [initial] [rendered]
chunk {styles} styles.bundle.js, styles.bundle.js.map (styles) 11.3 kB [initial] [rendered]
chunk {vendor} vendor.bundle.js, vendor.bundle.js.map (vendor) 1.95 MB [initial] [rendered]
→ DenoApp git:(master) # ng build --prod
Date: 2017-09-15T17:01:28.146Z
Hash: 384428205c4fa935f1f7
Time: 1ms
chunk {0} polyfills.c9b879328f3396b2bbe8.bundle.js (polyfills) 64.1 kB {0} [initial] [rendered]
chunk {1} main.52d78d55ba737a0392b0.bundle.js (main) 4.85 kB {1} [initial] [rendered]
chunk {2} styles.d41d8cd98f00b204e980.bundle.css (styles) 0 bytes {2} [initial] [rendered]
chunk {3} vendor.d997be8f65ee0109dba.bundle.js (vendor) 216 kB [initial] [rendered]
chunk {4} inline.e5ea7d7cf2c65030be45.bundle.js (inline) 1.45 kB [entry] [rendered]
→ DenoApp git:(master) # ng build --prod --build-optimizer
Date: 2017-09-15T17:04:04.000Z
Hash: 2af76c337d6254b39ae9
Time: 1466ms
chunk {0} polyfills.c9b879328f3396b2bbe8.bundle.js (polyfills) 63.9 kB {0} [initial] [rendered]
chunk {1} main.d79cb0031c8350bf40b5.bundle.js (main) 183 kB {1} [initial] [rendered]
chunk {2} styles.d41d8cd98f00b204e980.bundle.css (styles) 0 bytes {2} [initial] [rendered]
chunk {3} inline.ef95997beb65a719ffd.bundle.js (inline) 1.45 kB [entry] [rendered]
→ DenoApp git:(master) #
```

Deployment Herausforderungen

- Pragmatisch: Build Ergebnis ("**dist**") auf Produktionssystem ausrollen
 - rsync, ftp, ...
- Potentiell **inkonsistent** während Deployment läuft
 - Nicht atomar - kann ergänzend implementiert werden
- Was geschieht mit **alten** Dateien? Löschen?
 - Nutzer mit Lazy-Loading erhalten potentiell Fehler
- Was ist bei Fehlern - **Rollback** möglich?
- **Schrittweiser** Rollout möglich?
- Passt **Umgebung** zur Anwendung? (Libs, Konfiguration, ...)

Einsatz von Docker



Docker Build-Container

- Build-Container: Kurzlebiger Container für Build-Tasks
- Stellt Abhängigkeiten zur Verfügung, die (nur) im Build benötigt werden
 - Keine (globale) Installation nötig
 - Versionswechsel leicht möglich
- Isolation
 - (Parallele) Ausführung ohne Seiteneffekte

Docker Angular-CLI Images

- trion/ng-cli
 - Angular-CLI, node, npm, yarn
 - <https://hub.docker.com/r/trion/ng-cli/>
- trion/ng-cli-karma
 - Chrome Browser, xvfb
 - <https://hub.docker.com/r/trion/ng-cli-karma/>
- trion/ng-cli-e2e
 - Java, webdriver
 - <https://hub.docker.com/r/trion/ng-cli-e2e/>
- node.js, derzeit Version 6*
- npm und yarn Package Manager
- Getestet mit Linux, macOS, Windows
- Version folgt Angular-CLI Version
 - z.B. trion/ng-cli:1.4.2

Feste Versionen

```
docker run --rm trion/ng-cli:1.0.0 ng -v
```

```
→ ~ docker run --rm trion/ng-cli:1.0.0 ng -v

Angular CLI
@angular/cli: 1.0.0
node: 6.10.1
os: linux x64
→ ~
```

```
docker run --rm trion/ng-cli:1.4.1 ng -v
```

```
→ ~ docker run --rm trion/ng-cli:1.4.1 ng -v

Angular CLI
@angular/cli: 1.4.1
node: 6.11.3
os: linux x64
→ ~
```

Verwendung trion/ng-cli

```
docker run \
  -u $(id -u) \
  --rm \
  -v "$PWD":/app \
  trion/ng-cli \
  ng new MyDemo
```

run: Erstelle aus dem Image **trion/ng-cli** einen Container und starte ihn

-u: Verwende **\$(id -u)** als User

--rm: Lösche Container und Volumes nach Ausführung

-v: Monte das Verzeichnis **"\$PWD"** vom Host nach **/app** im Container (Volume mount)

ng new MyDemo: Argumente für Container

App erzeugen

```
docker run -u $(id -u) --rm \
  -v "$PWD":/app trion/ng-cli:1.4.2 ng new DemoApp
```

- Gedacht für Entwicklerrechner, Volume-Mount mit lokalem Daemon
- Syntax: Linux / macOS
- Unter Windows: -u \$(id -u) kann entfallen

Bestehende App bereitstellen

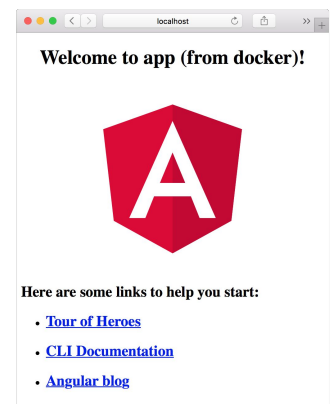
```
docker run -u $(id -u) --rm \
-v "$PWD":/app trion/ng-cli:1.4.2 npm install
```

- Anwendungsfall: Frischer SCM checkout
- Per npm Abhängigkeiten installieren

Entwicklung

```
docker run -u $(id -u) --rm -p 4200:4200 \
-v "$PWD":/app trion/ng-cli \
ng serve --host 0.0.0.0
```

```
1. docker run -u $(id -u) --rm -p 4200:4200 -v "$PWD":/app trion/ng-cli:1.2.7 ng (docker)
--host 0.0.0.0
** NG Live Development Server is listening on 0.0.0.0:4200, open your browser on http://localhost:4200 **
Hash: ac0b1c872357f9102494
Time: 9046ms
chunk {0} polyfills.bundle.js, polyfills.bundle.js.map (polyfills) 178 kB {4} [initial] [rendered]
chunk {1} main.bundle.js, main.bundle.js.map (main) 5.28 kB {3} [initial] [rendered]
chunk {2} styles.bundle.js, styles.bundle.js.map (styles) 10.5 kB {4} [initial] [rendered]
chunk {3} vendor.bundle.js, vendor.bundle.js.map (vendor) 2.19 MB [initial] [rendered]
chunk {4} inline.bundle.js, inline.bundle.js.map (inline) 0 bytes [entry] [rendered]
webpack: Compiled successfully.
webpack: Compiling...
Hash: 696508ef25e70f2d3ecb
Time: 167ms
chunk {0} polyfills.bundle.js, polyfills.bundle.js.map (polyfills) 178 kB {4} [initial]
chunk {1} main.bundle.js, main.bundle.js.map (main) 5.27 kB {3} [initial] [rendered]
chunk {2} styles.bundle.js, styles.bundle.js.map (styles) 10.5 kB {4} [initial]
chunk {3} vendor.bundle.js, vendor.bundle.js.map (vendor) 2.19 MB [initial]
chunk {4} inline.bundle.js, inline.bundle.js.map (inline) 0 bytes [entry]
webpack: Compiled successfully.
```



Test

Unit Test

```
docker run -u $(id -u) --rm -v "$PWD":/app \
trion/ng-cli-karma ng test --watch false
```

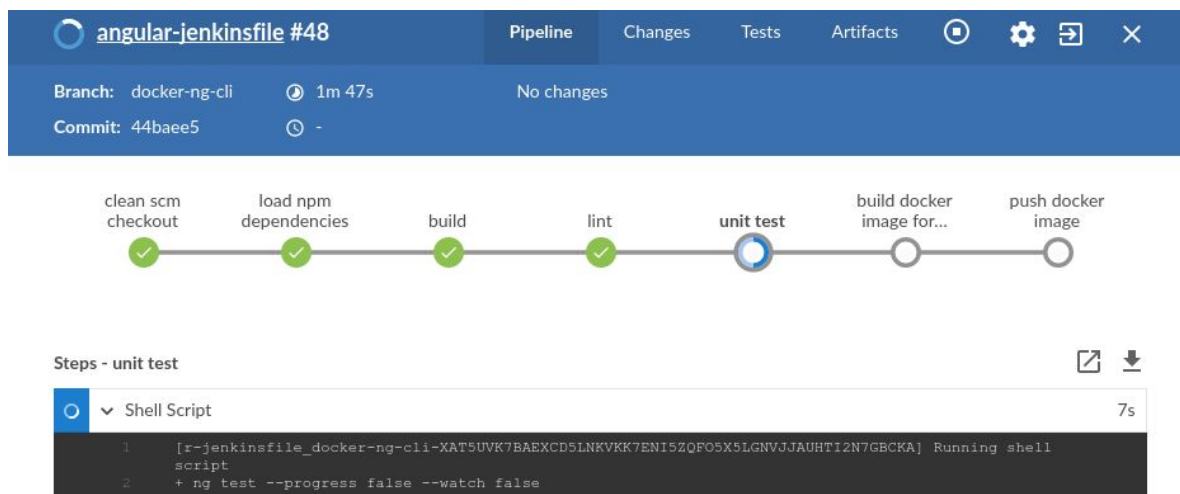
- Auf Entwicklerrechner: Volume Mount + watch möglich
 - Windows: Polling notwendig :(
- Performance wie lokale Ausführung

Ende-zu-Ende Tests

```
docker run -u $(id -u) --rm -v "$PWD":/app \
trion/ng-cli-e2e ng e2e
```

- e2e Tests benötigen in jedem Fall Browser
- Oder sogar mehrere Browser...
- ... oder gar Plattformen
- Für Smoke-Tests in jedem Fall geeignet, Rest über Matrix-Tests

CI Pipeline



Anforderungen CI Server

- Gemischter Einsatz unterschiedlicher Versionen
 - Umgebung für alle Jobs einheitlich
 - Umgebung soll nicht speziell für Angular vorbereitet werden müssen
- Headless
- Parallele Ausführung
- Reproduzierbare Ergebnisse

=> Anforderungen lassen sich gut mit Docker-Container erfüllen

CI Server - Beispiel Jenkins

- Scripted pipeline
- Mit Jenkins Docker Integration
- Nutzung via Shell ebenfalls möglich

```
node {
    stage('Clean checkout') {
        deleteDir()
        checkout scm
    }
    docker.image('trion/ng-cli-karma:1.4.2').inside {
        stage('NPM Install') {
            withEnv(["NPM_CONFIG_LOGLEVEL=warn"]) {
                sh 'npm install'
            }
        }
        stage('Test') {
            sh 'ng test --progress=false --watch false'
            junit '**/test-results.xml'
        }
    }
}
```

CI Server - Beispiel bitbucket ci

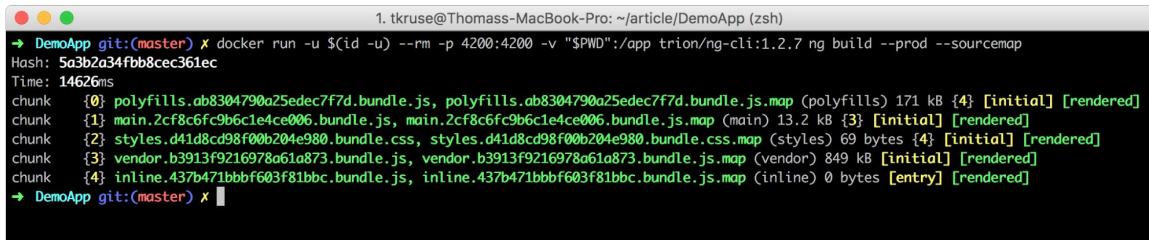
- Analog für andere Build-Server mit yml Konfiguration
 - gitlab ci
 - travis ci
 - ...

```
image: trion/ng-cli-e2e:1.4.2
pipelines:
  default:
    - step:
        script:
          - npm install
          - ng build
          - ng lint
          - ng test
          - ng e2e
```

Deployment

Build

```
docker run -u $(id -u) --rm -v "$PWD":/app \
trion/ng-cli ng build
```

A terminal window titled '1. tkruse@Thomass-MacBook-Pro: ~/article/DemoApp (zsh)' shows the execution of the command 'docker run -u \$(id -u) --rm -p 4200:4200 -v "\$PWD":/app trion/ng-cli:1.2.7 ng build --prod --sourcemap'. The output displays the build hash '5a3b2a34fbb8cec361ec' and a time of '1462ms'. It then lists four chunks with their respective file names, sizes, and status: chunk {0} (polyfills) 171 kB [initial] [rendered], chunk {1} (main) 13.2 kB [initial] [rendered], chunk {2} (styles) 69 bytes [initial] [rendered], and chunk {3} (vendor) 849 kB [initial] [rendered]. A fourth chunk {4} (inline) is listed as 0 bytes [entry] [rendered]. The terminal ends with the prompt 'DemoApp git:(master) x'.

Angular Deployment mit Docker Image

- Image ist leicht handhabbare Einheit
- Atomares Deployment
- Versionswechsel einfach möglich
- Reproduzierbare Ergebnisse
 - Konfiguration, Anwendung identisch in jeder Umgebung/Stage
- Schrittweiser Rollout (bspw. mit Kubernetes)

Angular Anwendung als Docker Image

- Webserver zur Auslieferung
nginx
- Für HTTP Anfragen:
Port **8080**
- Build-Ergebnis der Angular-App befindet sich im
dist Ordner

Dockerfile

```
FROM nginx:alpine

EXPOSE 8080
COPY nginx/default.conf /etc/nginx/conf.d/default.conf
RUN chown -R nginx /etc/nginx

COPY dist /usr/share/nginx/html/
```

nginx config

- Konfiguration für Port 8080 - wie in dockerfile
- Document Root, Index File
- Regel für Angular-Router mit HTML5 History-API (reload-safe)

```
server {
    listen 8080;

    location / {
        root /usr/share/nginx/html;
        index index.html index.htm;
        try_files $uri$args $uri$args/ $uri $uri/ /index.html =404;
    }
}
```

.dockerignore File

- Minimierung Datentransfer zu Docker-Daemon
- Vor allem in CI-Umgebungen ist Docker-Daemon oft nicht lokal
- Verringerte Build-Zeit

```
# .dockerignore

.git
.gitignore
.env*
node_modules
docker-compose*.yaml
```

Docker Image bauen

```
docker run -u $(id -u) --rm -v "$PWD":/app trion/ng-cli ng build
```

```
docker build -t demo/angular-app .
```

... ausprobieren ...

```
docker run --rm -it -p 8081:8080 demo/angular-app
```

... Image bereit für produktiven Einsatz

Image für Entwickler

- Bei heterogenen Teams kann Backend-Entwickler aktuelles Frontend mit Docker sehr einfach einsetzen
- Auf eigenen Arbeitsplatz mit beliebiger Backend-Version
- Exakt. Die. Version.
- Und umgekehrt: Frontend Entwickler kann Backend Image nutzen

Demo - <https://github.com/codecoster/ng-wegbl-demo>

- Verwendet WebGL
- Mit Docker problemlos in CI Umgebung testbar
 - Im Gegensatz zu PhantomJS



Demo - <https://github.com/codecoster/ng-wegbl-demo>

```
docker run -u $(id -u) --rm \
-v "$PWD":/app trion/ng-cli npm install
```

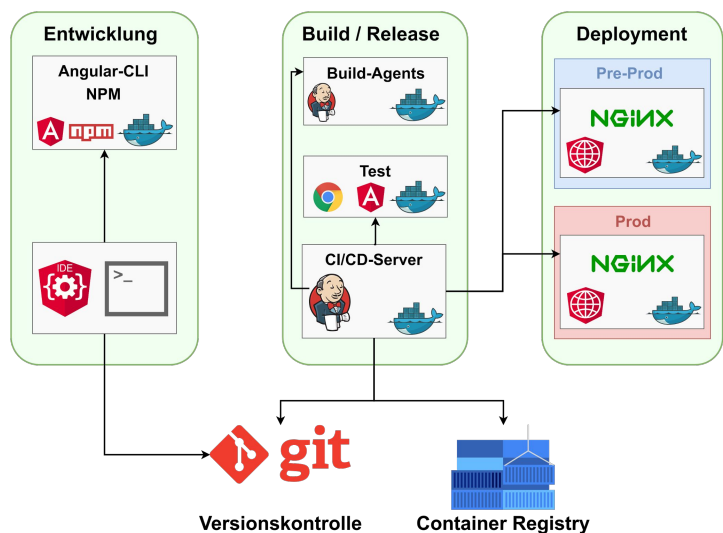
```
docker run -u $(id -u) --rm -v "$PWD":/app \
trion/ng-cli-karma ng test --watch false
```

```
docker run -u $(id -u) --rm -v "$PWD":/app \
trion/ng-cli-e2e ng e2e
```

```
docker run -u $(id -u) --rm -v "$PWD":/app \
trion/ng-cli ng build
```

Fazit

- trion/ng-cli
- trion/ng-cli-karma
- trion/ng-cli-e2e
- trion/nginx-angular



Danke.

Fragen?

tk@trion.de // @everflux

Backups

So much more

Multi-Stage Build

```
FROM trion/ng-cli:1.4.2 AS ngcli
WORKDIR /app
COPY . .
RUN npm install
RUN ng build --prod --aot --progress=false

FROM nginx:alpine AS app
COPY --from=ngcli dist /usr/share/nginx/html/
COPY nginx/default.conf /etc/nginx/conf.d/default.conf
RUN chown -R nginx /etc/nginx
```

Umgebungen

Heroku

Azure

GCE

Amazon EC2 Container Service

Kubernetes

Beispiel Heroku

- Heroku Deployment mittels eigener Registry
 - Image bauen
 - Image pushen

```
docker build -f heroku-docker/Dockerfile -t heroku-nginx .  
  
HEROKU_API_KEY="c0fefe" heroku-cli create  
  
docker login --username=_ --password=$HEROKU_API_KEY registry.heroku.com  
  
docker tag heroku-nginx registry.heroku.com/<app>/web  
docker push registry.heroku.com/<app>/web
```

docker-compose

- Laufzeitabhängigkeiten werden meist ebenfalls als Docker Image umgesetzt
- Handhabung wird dann aufwendig
- docker-compose erlaubt diese Abhängigkeiten zu konfigurieren...
- ... gemeinsam zu managen

Beispiel docker-compose.yml

```
# docker-compose.yml

version: "2"
services:
  ${serviceName}:
    build: .
    image: ${imageName}
    ports:
      - "${servicePort}:80"
```

Continuous Delivery

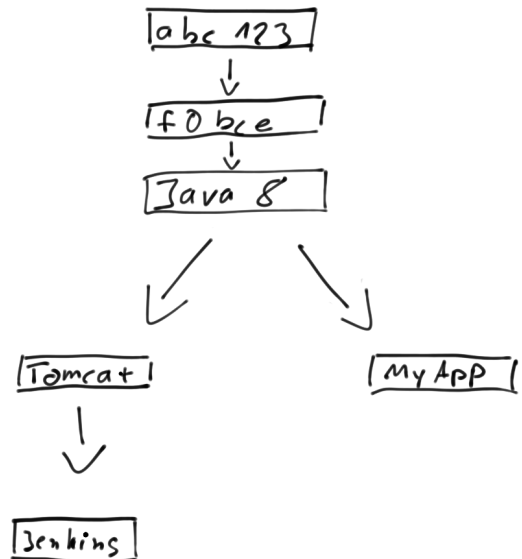
- Kontinuierliche Freigabe von produktionsreifem Stand
- Umsetzung: Bei jedem erfolgreichen Build wird Image in Registry gepushed
- Voraussetzung: Gute Testabdeckung
- Beispiel: Jenkins

```
stage ('Build Docker Image') {
    app = docker.build("demo/angular-app")
}
```

```
stage ('Push Docker Image') {
    docker.withRegistry('http://127.0.0.1:5000/') {
        app.push("latest")
        app.push("build-${env.BUILD_NUMBER}")
    }
}
```

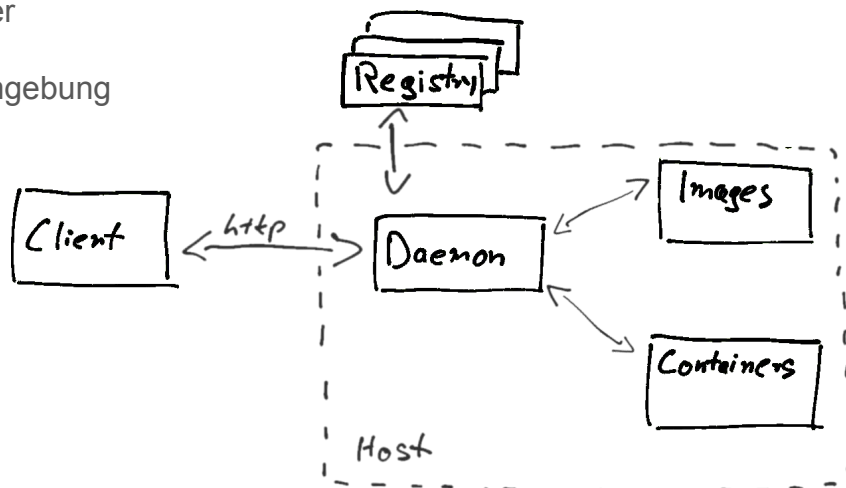
Docker Image Hierarchie

- Wie Layer in Images können auch Images geschichtet werden
- “Vererbung”
- Verwendung gemeinsamer Base-Images
- Beispielsweise für Angular
 - Custom base Image mit nginx Konfiguration
 - Anwendung kann Base Image nutzen und liefert lediglich “dist” Inhalt



Docker Architektur

- Client-Server
- Verteilte Umgebung



Angular Anwendung als Docker Image

- Base Image trion/nginx-angular
 - Webserver zur Auslieferung **nginx**
 - Für HTTP Anfragen: Port **8080**
- Build-Ergebnis der Angular-App befindet sich im **dist** Ordner
 - Ausgabe von Angular-CLI build

Dockerfile

```
FROM trion/nginx-angular  
  
COPY dist /usr/share/nginx/html/
```