DATASTAX®
DataStax Monday Learning

DataStax Monday Learning

Upgrade yourself, unlock new skills

- Every Week
- Best Instructors
- Most Important Topics
- From Engineers to Engineers
- Absolutely Free

DATASTAX®
DataStax Monday Learning

Docker Containers

From Basics to Best Practices

5-weeks Learning Path: 28.09-23.10.2020

Speakers:

- Aleks Volochnev
- Developer Advocates of DataStax

Schedule:

- **Week I 28.09.2020** Docker Fundamentals I
- **Week II 05.10.2020** Docker Fundamentals II
- **Week III 12.10.2020** Application Development with Docker
- **Week IV 19.10.2020** Best Practices + Final Assignment
- **Week V 26.10.2020 Introduction to Kubernetes**

DATASTAX®
DataStax Monday Learning

Docker Containers

From Basics to Best Practices

- 1.250 registrations
- Over 5K views on Youtube
- Almost 2,000 HOURS overall watch time

Thank you!

DATASTAX®
DataStax Monday Learning

Docker Containers
From Basics to Best Practices

Week III

Application Development with Docker

3 Questions to know you better

Simple Deployment

It should be a Question of Minutes to run your App on a new Machine

Simple, reproducible environment is the key to successful development. There should be two commands to run your App on a new machine:

- git clone
- docker-compose up -d

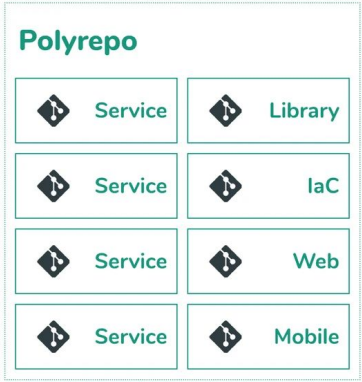
It also should be easy to switch it to development mode, like uncomment a bind mount.

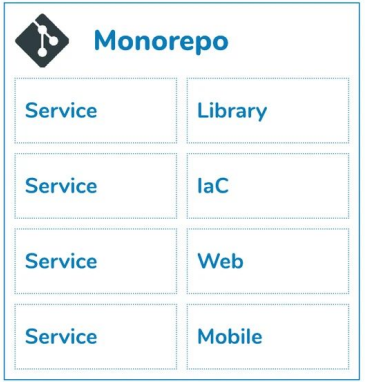
Monorepo vs Polyrepo

- **Polyrepository** - when you or your team works with multiple git repositories within THE SAME project.
- **Monorepository** - when all the code is in the same one-and-only repository

Given Monorepo, it's easy. Given Polyrepo, it may require an additional repository to define the system with *docker-compose.yaml*.

Traditional names are "*all-in-one*", "*main*" etc.



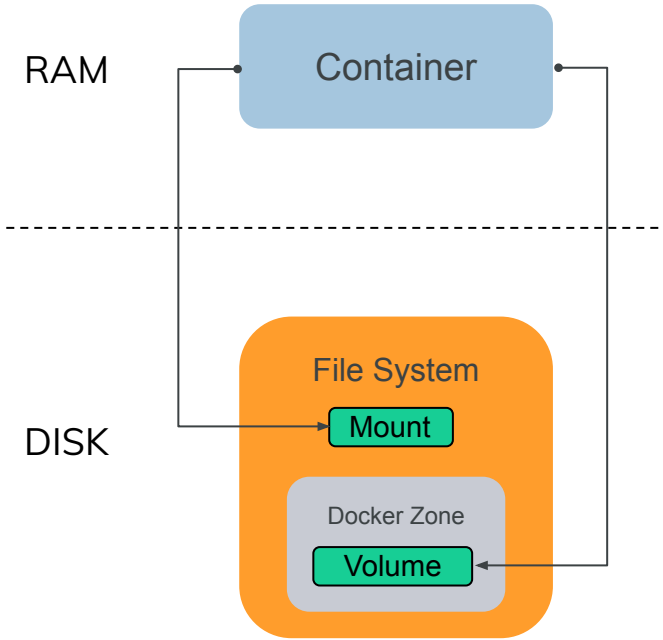


Source Code

Bind Mounts

Last week slide, remember?

The simple and “old-school” way to mount a local folder or a file into the container file system. Have limited functionality but usually enough for most of the use-cases. Allows direct access to the files from both host and container, very often used for development purposes. Default access mode is RW (read-write) but can be configured to be RO (read-only). **Managed by user,** so you can’t use Docker CLI commands to directly manage bind mounts. **Host mount precedes container mount.** Prefer for the cases when data comes from host.



Change from Image to Bind Mount

Main docker-compose file should be prepared and support simple switch from image to build mode.

```yaml
services:
  # The KillrVideo Sample Data Generator
  generator:
    #image: killrvideo/killrvideo-generator      You, 2 years ago • Reworked
    build: ./generator # Uncomment this line to change to local build
    volumes: # Uncomment volumes to change to local build
      - "$(pwd)/generator":/opt/killrvideo-generator
    ports:
      - "5858:5858"
```

Environment-Specific Data

Environment Variables

Some values may and should change from environment to environment. For example, you may want to have different password on your production database and on a developer's laptop development database.

DO NOT KEEP SENSITIVE DATA IN SOURCE CODE!!!

All passwords, API tokens etc. belong to environment, not to source code.

```
version: '3'
services:
  # The KillrVideo Sample Data Generator
  generator:
    build: .
    #image: killrvideo/killrvideo-generator
    volumes:
      - .:/opt/killrvideo-generator
    ports:
      - "5858:5858"
    depends_on:
      - dse
      - backend
    environment:
      KILLRVIDEO_YOUTUBE_API_KEY: SET_YOUR_YOUTUBE_KEY_HERE
      KILLRVIDEO_LOGGING_LEVEL: debug
```

Environment Variables

Some values may and should change from environment to environment. For example, you may want to have different password on your production database and on a developer's laptop development database.

DO NOT KEEP SENSITIVE DATA IN SOURCE CODE!!!

All passwords, API tokens etc. belong to environment, not to source code.

```yaml
services:
  # The KillrVideo Sample Data Generator
  generator:
    build: .
    #image: killrvideo/killrvideo-generator
    volumes:
      - .:/opt/killrvideo-generator
    ports:
      - "5858:5858"
    depends_on:
      - dse
      - backend
    env_file:
      - ./dev.env
```

dev.env

```
KILLRVIDEO_YOUTUBE_API_KEY=MYSUPERPROTECTEDKEY
KILLRVIDEO_LOGGING_LEVEL=debug
```

.gitignore!!!

Default Env File

You may use default file `.env` to be loaded automatically. It doesn't need to be attached via env_file directive.

"Convention over Configuration"

```yaml
services:
  # The KillrVideo Sample Data Generator
  generator:
    build: .
    #image: killrvideo/killrvideo-generator
    volumes:
      - .:/opt/killrvideo-generator
    ports:
      - "5858:5858"
    depends_on:
      - dse
      - backend
```

.env

```
KILLRVIDEO_YOUTUBE_API_KEY=MYSUPERPROTECTEDKEY
KILLRVIDEO_LOGGING_LEVEL=debug
```

.gitignore!!!

Use Env Vars in your App

Usage of Environment Variables is vary from language to language, but all modern languages offer enough tools to work with them.

Don't forget to set "default" values if needed.

```python
def serve():

    dse_username = os.getenv('KILLRVIDEO_DSE_USERNAME')
    dse_password = os.getenv('KILLRVIDEO_DSE_PASSWORD')
    dse_contact_points = os.getenv('KILLRVIDEO_DSE_CONTACT_POINTS', 'dse').split(',')
    service_port = os.getenv('KILLRVIDEO_SERVICE_PORT', '50101')
```

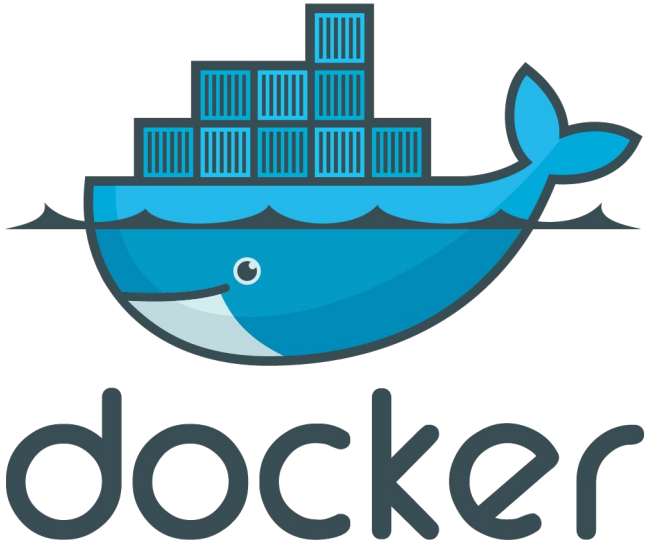
```java
package com.datastax.app;

import java.util.Map;

public class ReadEnvironmentVariable {
    public static void main(String[] args) {

        System.out.println("Read Specific Enviornment Variable");
        System.out.println("JAVA_HOME Value:- " + System.getenv("JAVA_HOME"));

        Map <String, String> map = System.getenv();
        for (Map.Entry <String, String> entry: map.entrySet()) {
            System.out.println("Variable Name:- " + entry.getKey() + " Value:- " + entry.getValue());
        }
    }
}
```

Build && Run

Build && Run

Important to understand that build and run are two different steps. For example, java applications build requires a lot of build-time dependencies whose aren't meant to be in the production container.

In this case, we may need one image to build an application and another - much smaller - to run it.



Debug

Debug

Debug containerised application is different, but not hard at all (well, depends on your app 😅)

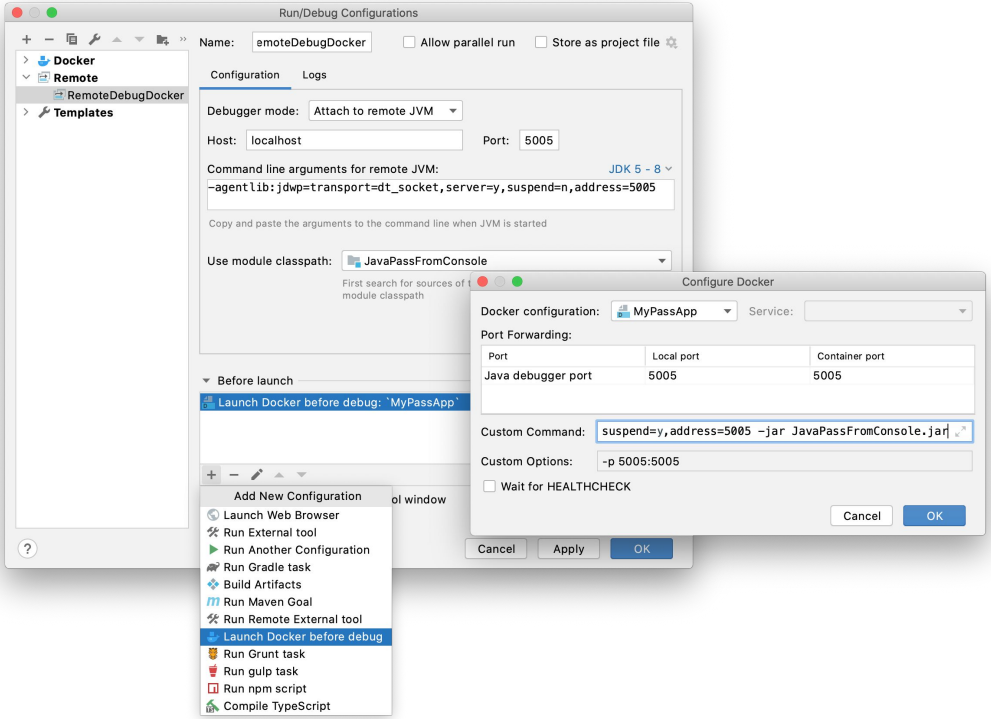
1. We will need a port to connect
2. We need a service to run with debug enabled
3. We need a remote debugger session

DATASTAX

Debug Java

Java Example:

1. Build the Service in the Debug Mode
2. Expose debug port (f.e. 8000)
3. Start the service in debug mode
4. Run remote debugging session



CI / CD

Autobuild

Local build is a great way to fail - local environment affects your build so it's only acceptable for tiny personal projects.

We really recommend to stay lazy and use automated builds. There are plenty of services to build it for you:

- Jenkins
- Travis CI
- Github Actions
- And many others!

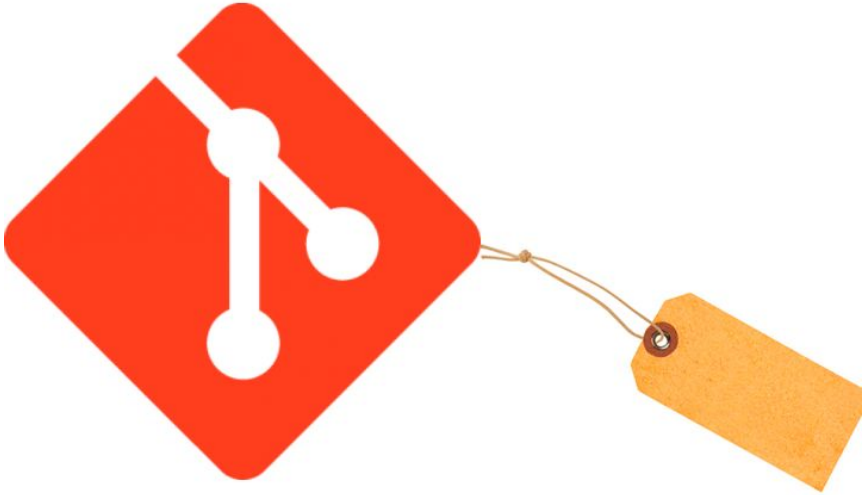
https://github.com/KillrVideo/killrvideo-java/blob/master/.travis.yml

```yaml
language: generic

# Sudo required for doing docker build
sudo: required
services:
- docker

# Build the app and a docker image
script:
- travis_fold start docker_build
- docker run -v ${PWD}:/opt/killrvideo-java -w /opt/killrvideo-java maven mvn install -DskipTests=true -
- docker build -t ${TRAVIS_COMMIT} -t killrvideo-java-local .
- travis_fold end docker_build
- travis_fold start docker_dependencies_up
- docker-compose -f docker-compose.ci.yml up -d dse dse-config
- sleep 180
- docker-compose -f docker-compose.ci.yml up -d backend
- sleep 180
- docker-compose -f docker-compose.ci.yml exec backend echo 'Still alive!' || { echo "Backend is down";
- travis_fold end docker_dependencies_up
- docker run --network killrvideo-java_default killrvideo/killrvideo-integration-tests

# If successful, see if we need to publish also
after_success:
- "[ \"$TRAVIS_EVENT_TYPE\" = \"cron\" ] && { echo \"Ignore nightly builds\"; travis_terminate 0; }"
- test -z $TRAVIS_TAG && { echo "Ignore non-tagged builds"; travis_terminate 0; }
- docker tag ${TRAVIS_COMMIT} killrvideo/killrvideo-java:${TRAVIS_TAG}
- echo "$DOCKER_PASS" | docker login -u "$DOCKER_USER" --password-stdin
- docker push killrvideo/killrvideo-java:${TRAVIS_TAG}
- "[ \"$(git tag --sort=-v:refname | grep -P \"^\\d+.\\d+.\\d+$\" | head -n1)\" == \"$TRAVIS_TAG\" ] &&

after_failure:
- travis_fold start docker_logs
- docker-compose -f docker-compose.ci.yml logs dse-config backend
- travis_fold end docker_logs

env:
  global:
  # DOCKER_USER & PASS
  - secure: hL9GzKnAuHP130bLzB1nK9eF6bfPD4yFdQ1IdRRp7QDZ+AJ2MUw483w5Uacw6/VNk+KlItO9ySxkAI8gaBcxjxcp+TmF
  - secure: hxakhOfACkkXcGc5T9cT0+a+ICw6X2ybsOXzJsgdju4QbdP0nD2iBSZCypDzfdzlF/y83yJGcSznegMega37ABomPd8B

notifications:
  slack:
    rooms:
      secure: N6FrJ0l0gw2t1wnty/xPDKvLBZlzOAmkIbHjd4ZrtaC4vLU8iZP+YCRvQd7Vwzc97yDhHIZ41eGtjHjBIIeMX/3rfu
```

git tag === docker tag

Docker images have no direct relation to git version control. The only way to connect them is to use proper tags: usually we need to have the same docker tag as the git tag we have: **my-service** version **1.12.3** should be released as **my-service:1.12.3**. It's for you to control it!

Release Candidates is a convenient way to test how system works without releasing a final version. For example, you could first build **my-service:2.0-RC1** and only if it works well release it as **my-service:2.0.**



Smoke Testing

The most underestimated kind of testing: easiest to implement, still discovers A LOT of issues.

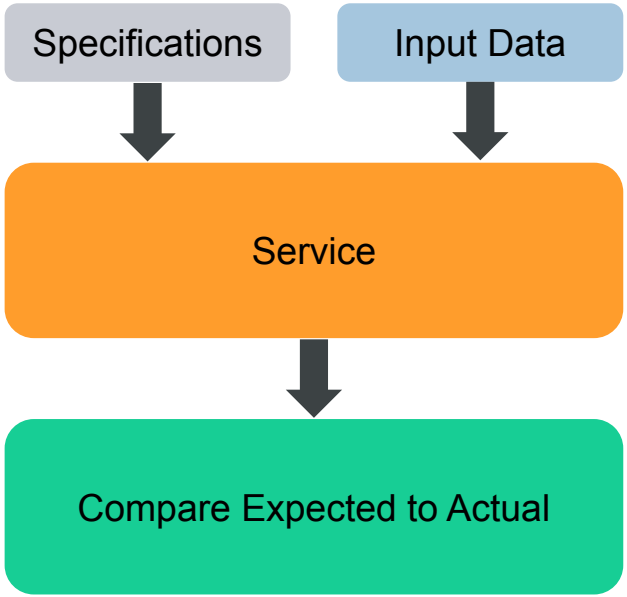
“Just plug it in a power socket. If it smokes, something is wrong”

Start the container, wait if it listens its port after N seconds. Fail build if it doesn’t.

Functional Testing

"Functional testing is a quality assurance process and a type of black-box testing that bases its test cases on the specifications of the software component under test. Functions are tested by feeding them input and examining the output, and internal program structure is rarely considered (unlike white-box testing).

Functional testing is conducted to evaluate the compliance of a system or component with specified functional requirements. Functional testing usually describes what the system does."

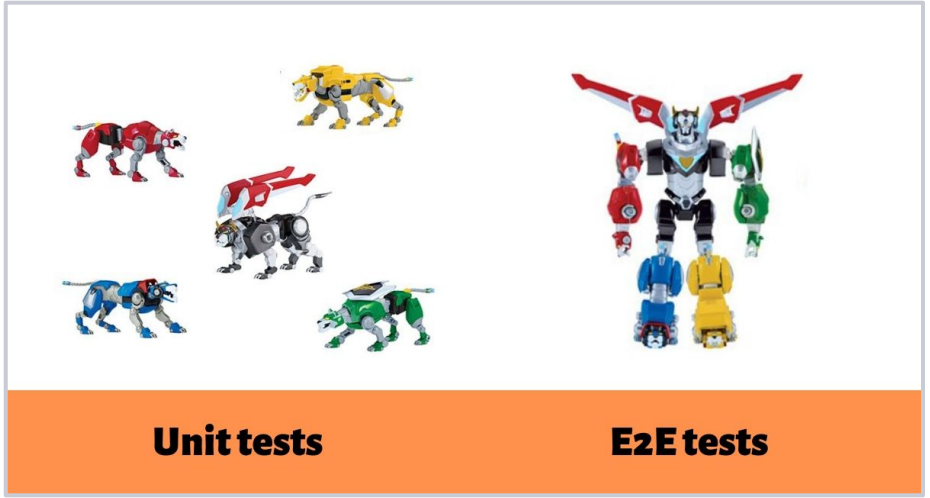


E2E Testing

"End-to-end testing is a methodology used to test whether the flow of an application is performing as designed from start to finish. The purpose of carrying out end-to-end tests is to identify system dependencies and to ensure that the right information is passed between various system components and systems."

Some frameworks to help:

- Protractor
- Cypress
- TestCafe
- And many others



LIVE QUIZ!

Week III Assignment

Week III Assignment

We proceed with the assignment of the Week II

1. Use the Week II Task I as a first step. Then improve it following this week steps: env variables, github actions or travis build, **tests**, etc. Please use our week II samples for the inspiration.

2. If possible, publish your code from p.1 on github in your repository. Create a new issue at github.com/datastaxdevs/docker-learning-path/issues. It may not be an option if you containerised a proprietary project, but please proceed to step III anyway.

3. Choose an issue from the list, write a comment that you have "taken" it. Review the project and think on how would you improve it. Write down your suggestions in the issue. Feel free to review multiple projects, also feel free to review a taken one! **Stay polite!**

4. If you want us to review your assignment publicly, send an issue link to me! We will pick some projects to discuss during week IV. We cover both mistakes and good decisions. :)

Resources:

- https://github.com/datastaxdevs/docker-learning-path
- https://discord.gg/va4vnsm

Thank You!

You are awesome!