docker

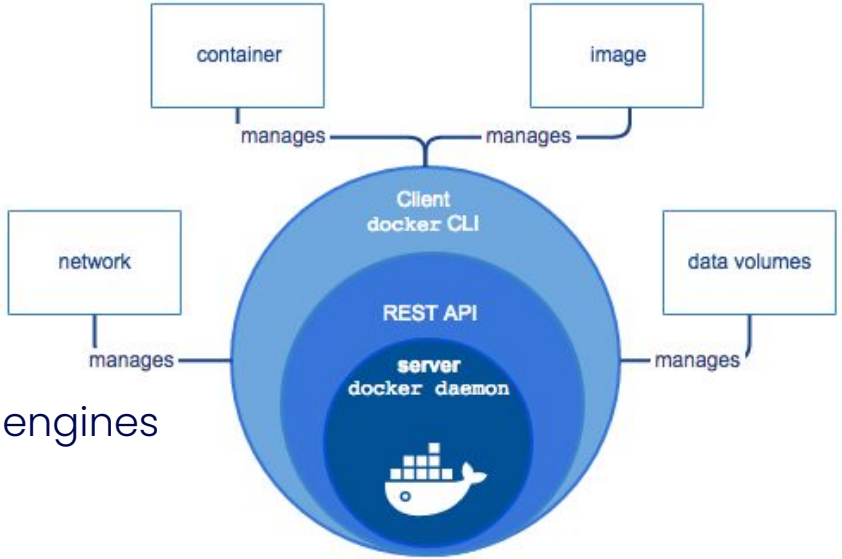
Module Seven

Compose and Testcontainers

Docker Engine API

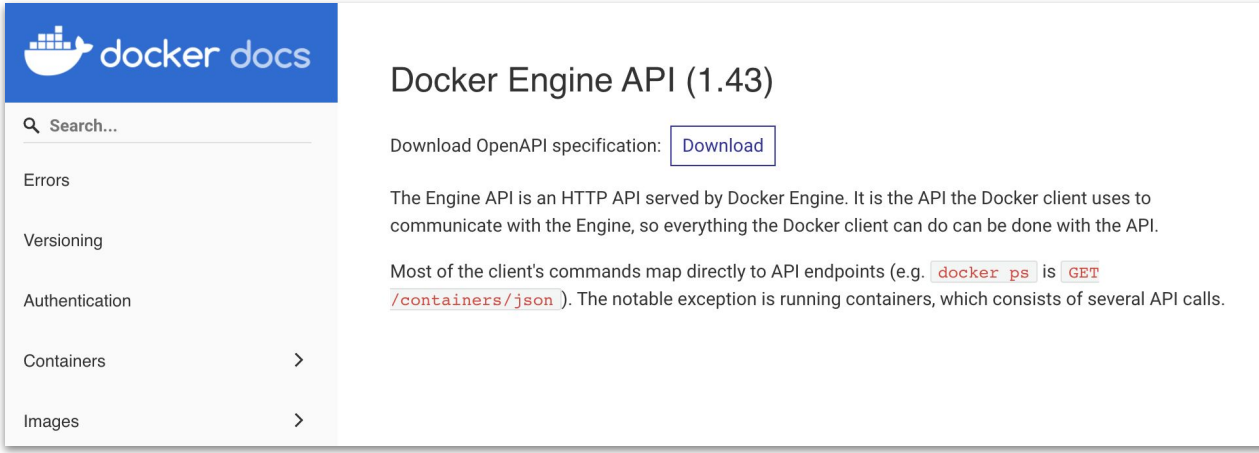
The Docker Engine

- The Docker Daemon/Engine
  - Manages everything
  - Exposes a REST API
- The Docker CLI
  - Interacts with the API
  - Provides a CLI-based user interface
  - Can be configured to point to remote engines
- The Docker GUI/Dashboard
  - Interacts with the API
  - Provides a GUI-based user interface

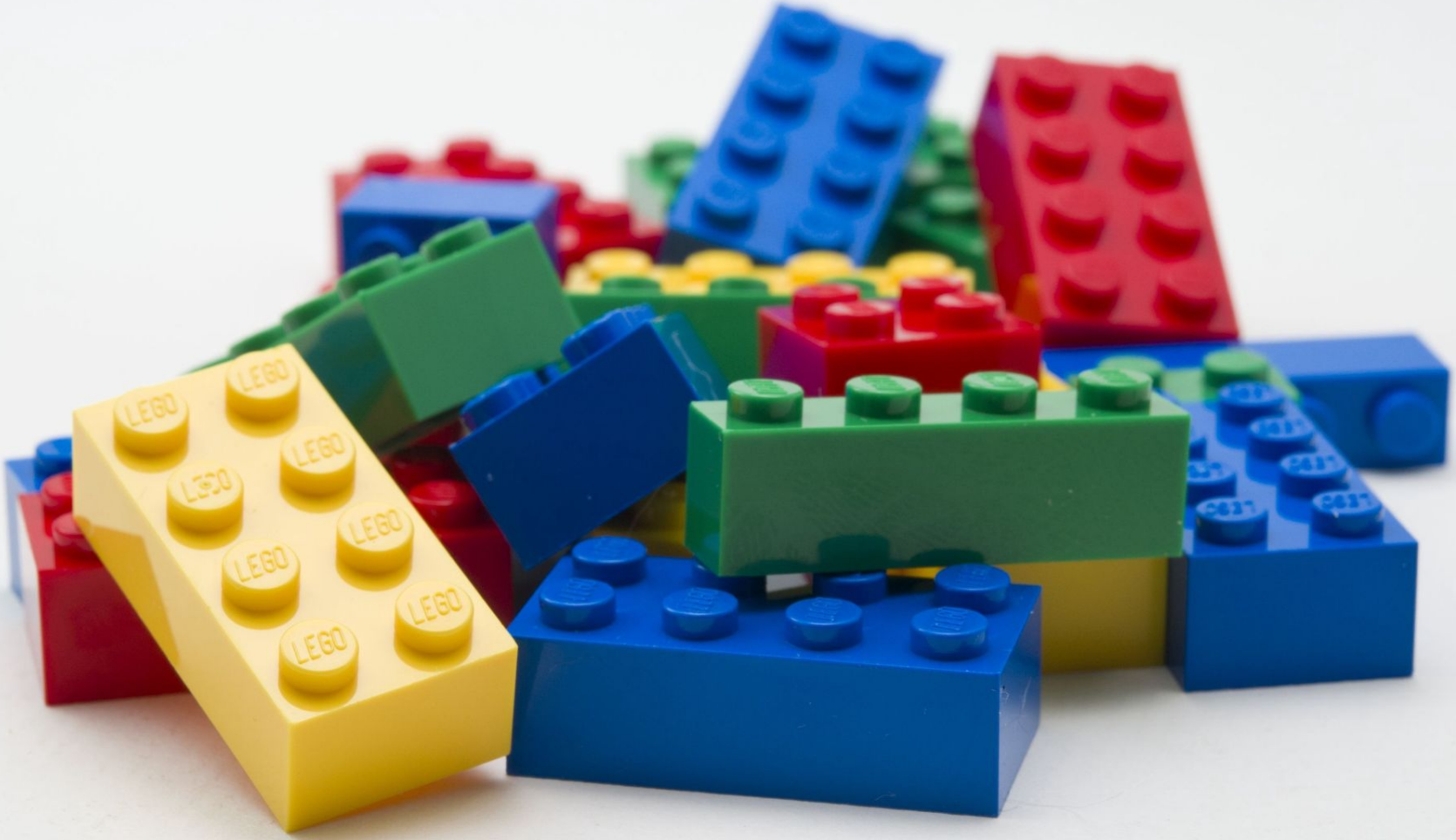


The Engine API

- The API is documented online
- The API provides the ability to do anything on Docker daemon
  - Provides ability to build other tooling/automations on top of the engine



Compose Basics


LEGO
LEGO
LEGO
LEGO
LEGO
LEGO
LEGO

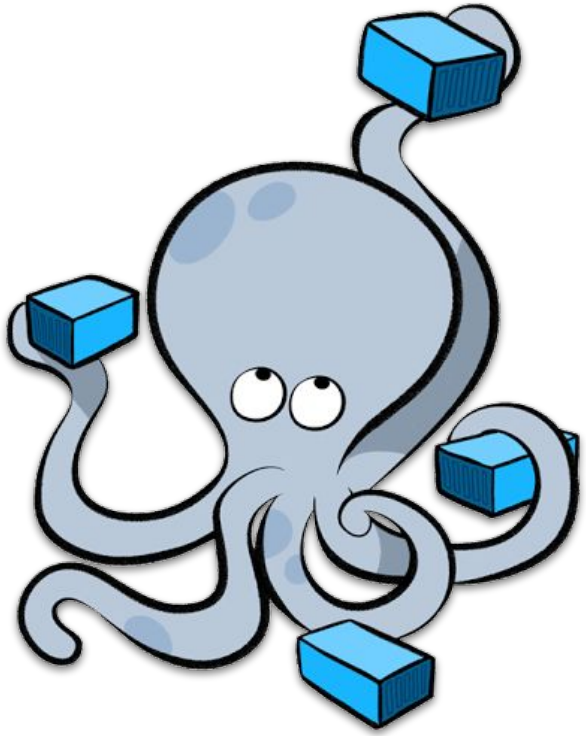
1. git clone

2. docker compose up

3. Do cool stuff!

Docker Compose 101

- TL;DR - I simply declare what I want and Compose figures out how to make it happen
- Basic commands include…
  - `docker compose up` – “make it so!”
  - `docker compose down` – “tear it down!”
  - `docker compose logs` – show me logs



A simple command

```
docker run -d -p 3306:3306 \
    -e MYSQL_ROOT_PASSWORD=superSecret \
    -e MYSQL_DATABASE=memes \
    -v mysql-data:/var/lib/mysql \
    mysql:8.2.0
```

Converting the command

```
docker run -d -p 3306:3306 \
    -e MYSQL_ROOT_PASSWORD=superSecret \
    -e MYSQL_DATABASE=memes \
    -v mysql-data:/var/lib/mysql \
    mysql:8.2.0
```

```
services:
  mysql:
    image: mysql:8.2.0
```

The services represent each of the building blocks of my application.

I define a service named **`mysql`** and specify the container image it will use.

Specifying ports

```
docker run -d -p 3306:3306 \
    -e MYSQL_ROOT_PASSWORD=superSecret \
    -e MYSQL_DATABASE=memes \
    -v mysql-data:/var/lib/mysql \
    mysql:8.2.0
```

```
services:
  mysql:
    image: mysql:8.2.0
    ports:
      - 3306:3306
```

OR

```
services:
  mysql:
    image: mysql:8.2.0
    ports:
      - target: 3306
        published: 3306
```

Next, we bring over the port mapping.

The Compose specification has both a short-form syntax and a long-form syntax that's more verbose.

Moving env vars over

```
docker run -d -p 3306:3306 \
    -e
MYSQL_ROOT_PASSWORD=superSecret \
    -e MYSQL_DATABASE=memes \
    -v mysql-data:/var/lib/mysql \
    mysql:8.2.0
```

Next, we bring over the environment variables.

The Compose specification allows you to define them as either a key/value mapping or as an array of **KEY=VALUE** strings.

```
services:
  mysql:
    image: mysql:8.2.0
    ports:
      - 3306:3306
    environment:
      MYSQL_ROOT_PASSWORD: superSecret
      MYSQL_DATABASE: memes
```

OR

```
services:
  mysql:
    image: mysql:8.2.0
    ports:
      - 3306:3306
    environment:
      - MYSQL_ROOT_PASSWORD=superSecret
      - MYSQL_DATABASE=memes
```

Defining volumes

```
docker run -d -p 3306:3306 \
    -e MYSQL_ROOT_PASSWORD=superSecret \
    -e MYSQL_DATABASE=memes \
    -v mysql-data:/var/lib/mysql \
    mysql:8.2.0
```

```
services:
  mysql:
    image: mysql:8.2.0
    ports:
      - 3306:3306
    volumes:
      - mysql-data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: superSecret
      MYSQL_DATABASE: memes
volumes:
  mysql-data:
```

Next, we define the volume, ensuring our data is persisted across environment restarts

Adding another service!

- With Compose, it's easy to add other services to help with debugging and troubleshooting
- This new service is adds a database visualizer

```
services:
  mysql:
    ...
  phpmyadmin:
    image: phpmyadmin:5.2
    ports:
      - 8080:80
...
```

Managing cross-service dependencies

- Using **depends_on** ensures the service doesn't start until the database service is running
- Additional options will allow you to indicate a dependent service needs to be healthy or even run to completion first

```
services:
  mysql:
    ...
  phpmyadmin:
    ...
    depends_on:
      - mysql
...
```

Advanced depends_on

- Other conditions can be specified
  - **service_healthy**: the dependency needs to be passing its healthchecks
  - **service_completed_successfully**: the dependency needs to successfully exit before starting this service

```
services:
  mysql:
  config-fetcher:
  phpmyadmin:
    ...
    depends_on:
      - mysql:
        condition: service_healthy
      - config-fetcher:
        condition: service_completed_successfully
```

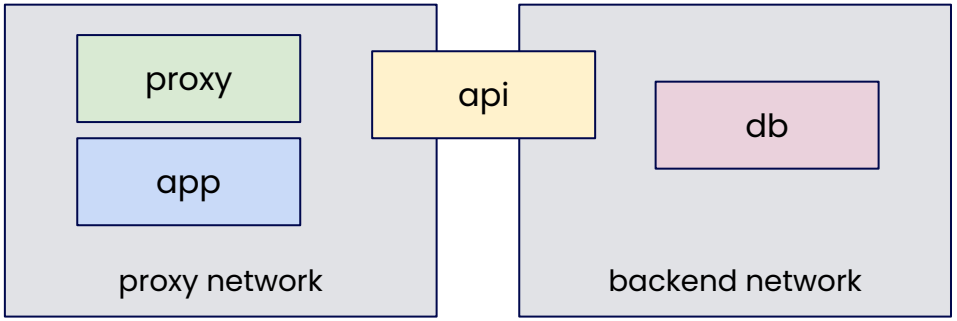
Networking

- Each Compose stack gets its own network
- Each service has a DNS entry added that matches its name
- This allows phpmyadmin to connect to the database simply using the hostname `mysql`

```
services:
  mysql:
    image: mysql:8.0
    ...
  phpmyadmin:
    image: phpmyadmin:5.2
    ports:
      - 8080:80
    depends_on:
      - mysql
    environment:
      PMA_HOST: mysql
      PMA_USER: root
      PMA_PASSWORD: superSecret
...
```

Advanced networking

- You can build more sophisticated networks to control what containers can talk to each other
- Aliases provide the ability to customize the DNS names




```
services:
  proxy:
    ...
    networks:
      - proxy
  app:
    ...
    networks:
      - proxy
  api:
    ...
    networks:
      proxy:
      backend:
  db:
    ...
    networks:
      backend:
        aliases:
          - database
networks:
  proxy:
  backend:
```

Adding a dev service

- Build a custom image and use it for the service using the `build` field
  - If you're using multi-stage builds, you can target a specific stage with `build.target`
- Mount in source code using a bind mount
  - As you make changes to the files on your host, they'll be updated in your container

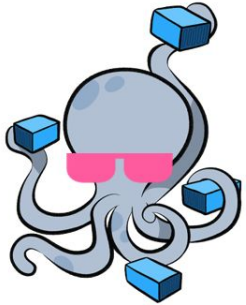
```
services:
  python:
    build:
      context: ./
      target: dev
    ports:
      - 8000:5000
    volumes:
      - ./app:/usr/local/app
  mysql:
    image: mysql:8.2.0
    ...
  phpmyadmin:
    image: phpmyadmin:5.2
    ...
```

Free Samples!

github.com/docker/awesome-compose

README | CC0-1.0 license

Awesome Compose awesome



> A curated list of Docker Compose samples.

These samples provide a starting point for how to integrate different services using a Compose file and to manage their deployment with Docker Compose.

> **Note** The following samples are intended for use in local development environments such as project setups, tinkering with software stacks, etc. These samples must not be deployed in production environments.

Contents

- Samples of Docker Compose applications with multiple integrated services.
- Single service samples.
- Basic setups for different platforms (not production ready - useful for personal use).

Samples of Docker Compose applications with multiple integrated services

docker

Compose Details

Running integration tests

- Define a service that runs the test cases
- Use Docker Compose to spin up all of the services
- Use `docker compose up --exit-code-from <service>`
  - When service exits, everything else is shut down and the exit code is relayed

```
services:
  tests:
    image: test-runner
    depends_on:
      - api
      - client
      - selenium
    ...
  selenium:
    image: selenium/standalone-chrome
    ...
  api:
    image: app-api
    ...
  client:
    image: app-client
    ...
  db:
    image: database
    ...

> docker compose up --exit-code-from tests
```

Adding Environment Variables

- Hard-coding parameters is a bad idea
- Let's replace them with environment variables
  - We can bring them in with a .env (or different) file
  - We can pass them on the command line
  - We can get them from the shell
- Use `docker compose config` to validate

```
.env file
DBNAME=memes
```

OR

```
docker compose run -e DBNAME=memes
```

OR

```
export DBNAME="memes"
```

```
services:
  mysql:
    environment:
      MYSQL_DATABASE: ${DBNAME}
```

Adding Secrets

- Using Environment Variables for secrets can also be a bad idea
  - Available to all processes
  - Can show up in logs
- Let's use secrets files instead
- Please see the secrets section of the compose docs for more information

```
services:
  mysql:
    image: mysql:8.2.0
    environment:
      MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_root_password
      MYSQL_PASSWORD_FILE: /run/secrets/db_password
      MYSQL_DATABASE: ${DBNAME}
    secrets:
      - db_root_password
      - db_password
secrets:
  db_password:
    file: db_password.txt
  db_root_password:
    file: db_root_password.txt
```

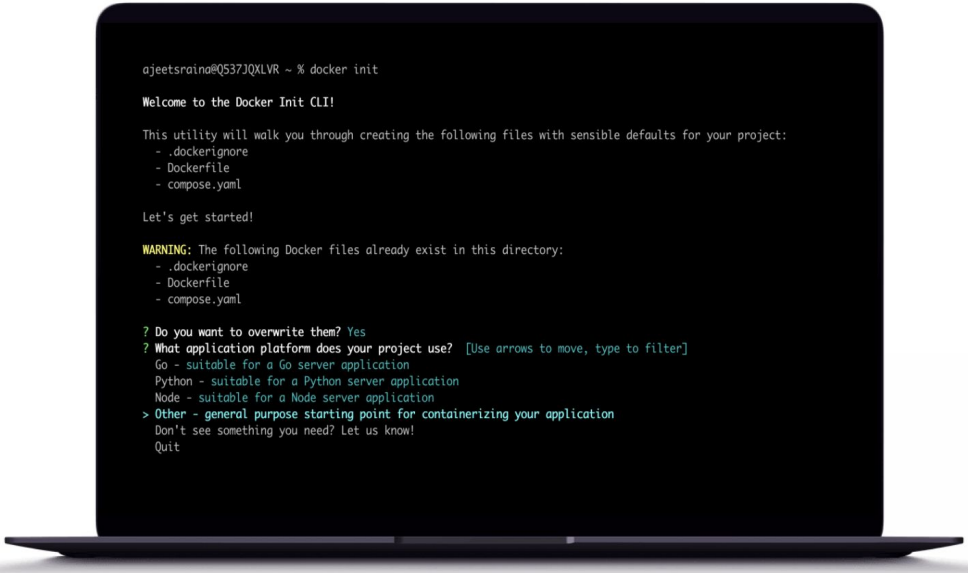
Compose - What’s New?

Little Changes, Big Results

- Docker Init
  - Accelerates the code -> container process
  - Lowers friction for new developers
- Dockerfile HEREDOC
  - Streamlines Dockerfiles, removes confusion, improves readability
- Docker Compose Include
  - Improves separation of concerns…
  - …while allowing easy integration
- Docker Compose Watch
  - Accelerates the code-sync-build-test cycle
- Docker Debug
  - Easy debugging of running containers and images
  - Build your own “debugging toolkit”

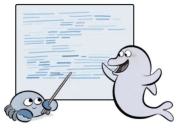
$ docker init

- Generates Docker assets for projects
- Allows you to choose application platform
- Makes it easier to create Docker images and containers

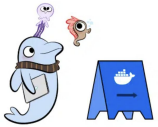


$ docker init

Simplified Docker Assets Creation



Saves Time and Effort



Better Project Organization

Enhanced Portability

Compose Include

- Include Compose files from other locations for composable apps
  - Reference local files
  - Pull from remote git repos (as of v2.21.0)
- Each Compose file can reference their own relative paths

```
include:
  - another-compose.yaml
  - git@github.com:namespace/repo.git
services:
  ...
volumes:
  ...
```

Compose Watch

- Automatically updates your compose service containers while you work
- Blazing-fast file synchronization supporting live update

How it works?

- Automatically builds a new image with BuildKit and replaces the running service container
- Add a `develop` section to your services in the `compose.yaml` file
- Configure it with a list of paths to watch and actions to take
- Watch rules allow ignoring specific files or entire directories within the watched tree.

```yaml
services:
  web:
    build: .
    command: npm start
    x-develop:
      watch:
        - action: sync
          path: ./web
          target: /src/web
          ignore:
            - node_modules/
        - action: rebuild
          path: package.json
```

Compose Watch Actions

Sync

Specifies a path to watch for changes in the host file system, and a corresponding target path inside the container to synchronize changes to.

Rebuild

The "rebuild" action specifies a path to watch for changes in the host file system, and triggers a rebuild of the container when changes are detected.

sync+restart

The "sync+restart" action specifies a path to watch for changes in the host file system, and a corresponding target path inside a container to first synchronize changes to and then restart the container

Compose Bridge

- Enables the use of compose.yaml files to create resource definitions for other platforms, primarily Kubernetes.

- Converts compose.yaml into Kubernetes manifests and Kustomize overlays.
  - compose-bridge -f compose.yaml convert.

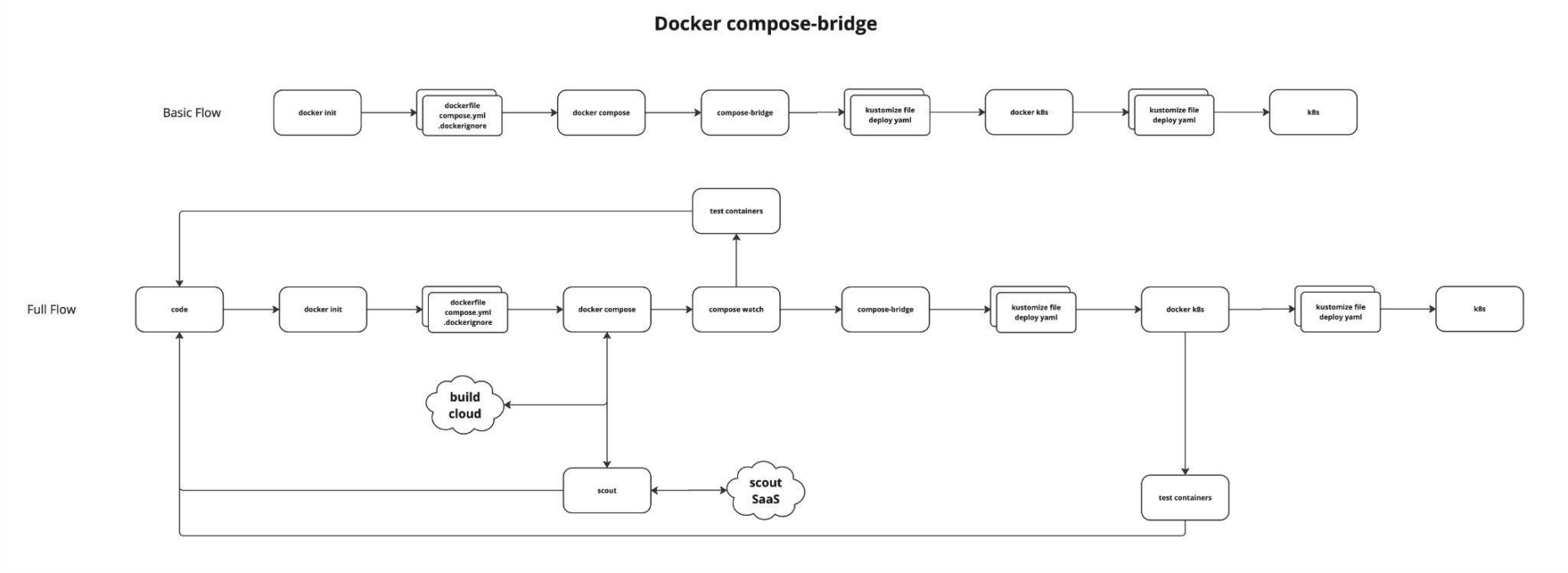
- Modify default templates, add new templates, or build custom transformations to match infrastructure needs.

- Extract and customize transformation templates using compose-bridge transformations create my-template.

- kubectl Plugin is available

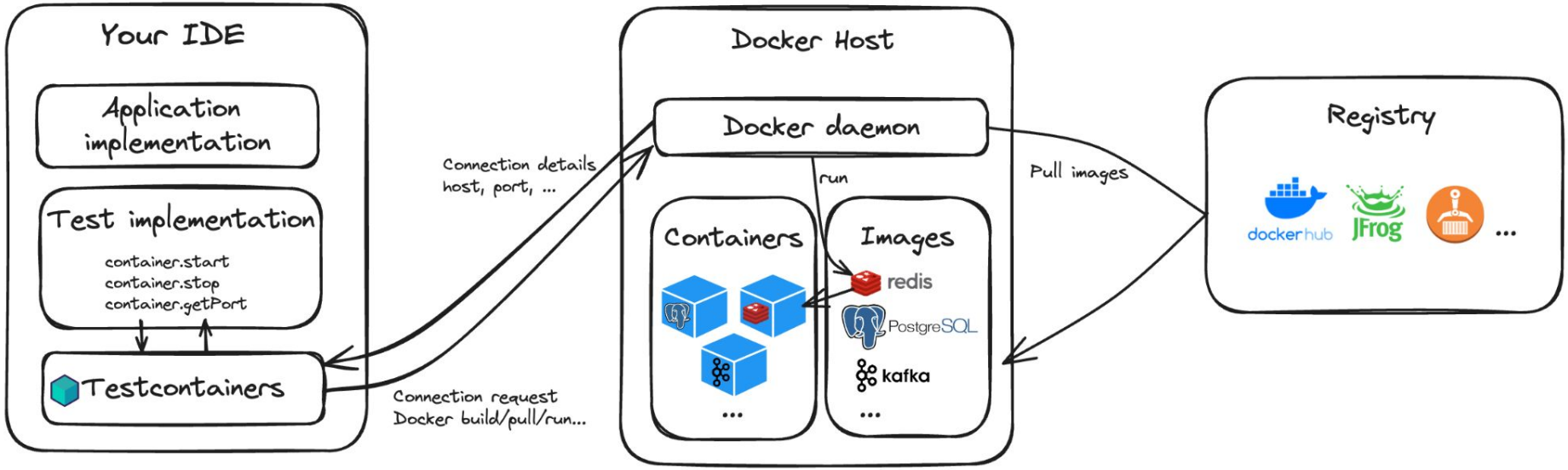
```yaml
services:
  web:
    build: .
    command: npm start
    x-develop:
      watch:
        - action: sync
          path: ./web
          target: /src/web
          ignore:
            - node_modules/
        - action: rebuild
          path: package.json
```

docker init - docker compose - compose bridge



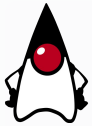
Test Containers

How it works?


Your IDE
Application implementation
Test implementation
container.start
container.stop
container.getPort
Testcontainers
Connection details
host, port, ...
Connection request
Docker build/pull/run...
Docker Host
Docker daemon
run
Containers
...
Images
redis
PostgreSQL
kafka
...
Pull images
Registry
dockerhub
JFrog
...

For any language

Testcontainers libraries exist for all popular languages including Java, Go, .NET, NodeJS, and more.


Java


Go

.NET


Node.js


Elixir


Python

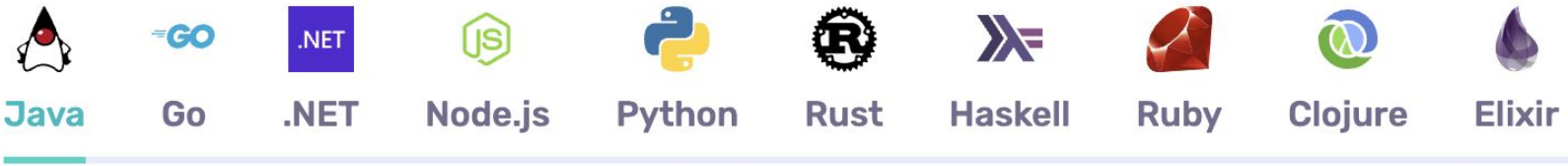

Rust


Haskell


Ruby

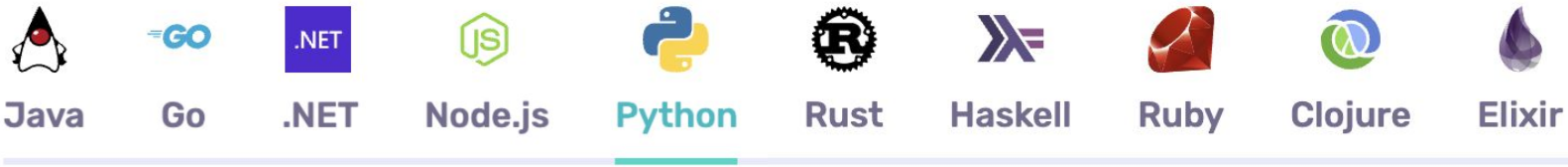

Clojure

Testcontainers for 10+ languages



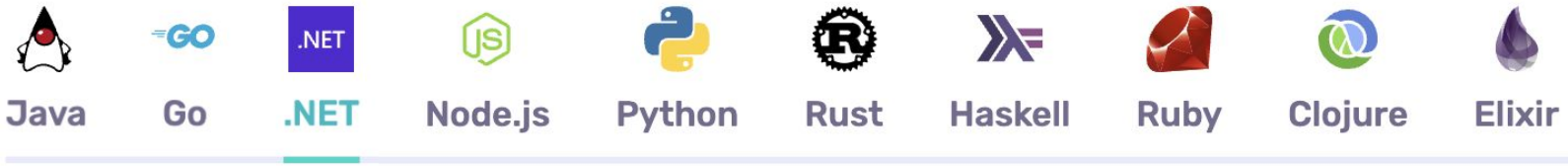

```
GenericContainer redis = new GenericContainer("redis:5.0.3-alpine")
    .withExposedPorts(6379);
```

Testcontainers for 10+ languages




```
redis = (
    DockerContainer("redis:5.0.3-alpine")
        .with_exposed_ports(6379)
)
redis.start()
wait_for_logs(redis, "Ready to accept connections")
```

Testcontainers for 10+ languages

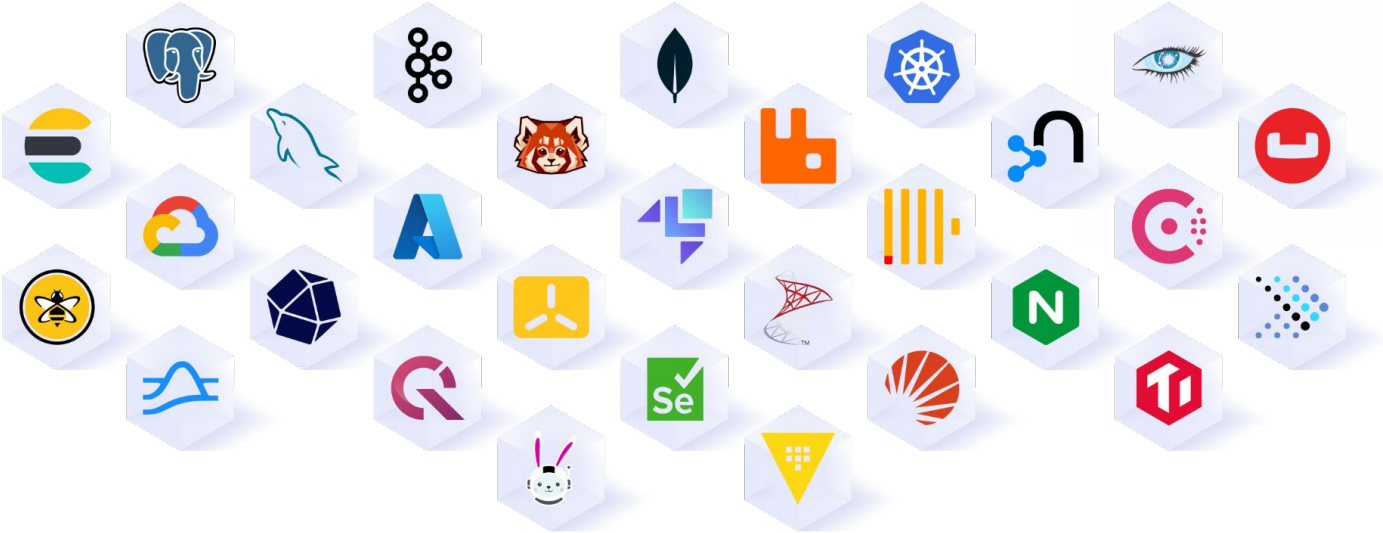



```
RedisContainer redisContainer = new RedisBuilder().Build();
await redisContainer.StartAsync();
```

For your entire stack

Test against any database, message broker, browser…

or just about anything that runs in a Docker container!



Official Modules

These companies maintain their Testcontainers Module implementation and recommends to use it for testing

Official Modules

Languages

- All
- .NET
- Go
- Java
- Node.js

Categories

- All
- Relational Database

CockroachDB
Relational Database
Java Go

Dapr
Cloud
Java

LocalStack
Cloud
Java Go .NET Node.js

Microcks
Cloud
Java Go Node.js

Neo4j
NoSQL Database, Vector Database
Java Go .NET Node.js

Pulsar
Message Broker
Java Go

Qdrant
Vector Database
Java Go Node.js

Redpanda
Message Broker
Java Go .NET Node.js

Synthesized
Relational Database, Other
Java

Weaviate
Vector Database
Java Go Node.js

WireMock
Cloud, Web
Java Node.js Go Python

YugabyteDB
Relational Database
Java

Testcontainers VS Docker Compose

Similar approach but…

Testcontainers	Docker Compose
Focus on **simplifying test environment** management	Focus on application **packaging, deployment, and delivery**
Programmatic compose: test dependencies as code without leaving your IDE	**YAML configuration file:** multi-container app management but you need to know how to 'cook' it
Self-contained: automated lifecycle of test dependencies	docker compose up -d ./run_tests docker compose down
Isolated testing environments: can run tests in parallel easily, every time new environment	**Caches the container configuration:** great for development, challenge for parallel testing
Pre-configured modules: Wide range of supported technologies	**Portability across environments:** customize composition for different environments

But what does it look like?

```
from testcontainers.postgres import PostgresContainer
import sqlalchemy
import time

def print_hi(name):
    print(f'Hi, {name}')

if __name__ == '__main__':
    print_hi('Testcontainers ')

    # Postgres container
    with PostgresContainer("postgres:16") as postgres:
        psql_url = postgres.get_connection_url()
        engine = sqlalchemy.create_engine(psql_url)
        with engine.begin() as connection:
            version, = connection.execute(sqlalchemy.text("SELECT version()")).fetchone()
        print_hi(version)
        time.sleep(120)
```

But what does it look like?

```
$ python main.py
  warnings.warn(
Hi, Testcontainers
Pulling image postgres:16
Container started: 3553cb6152a0
Waiting to be ready...
Waiting to be ready...
Waiting to be ready...
Waiting to be ready...
Hi, PostgreSQL 16.3 (Debian 16.3-1.pgdg120+1) on aarch64-unknown-linux-gnu, compiled by gcc (Debian 12.2.0-14)
12.2.0, 64-bit
```

Testcontainers Cloud

Test without limits. Ship with confidence.

Testcontainers Cloud

Developer-first Testing

Test everything on your laptop without worrying about resources;

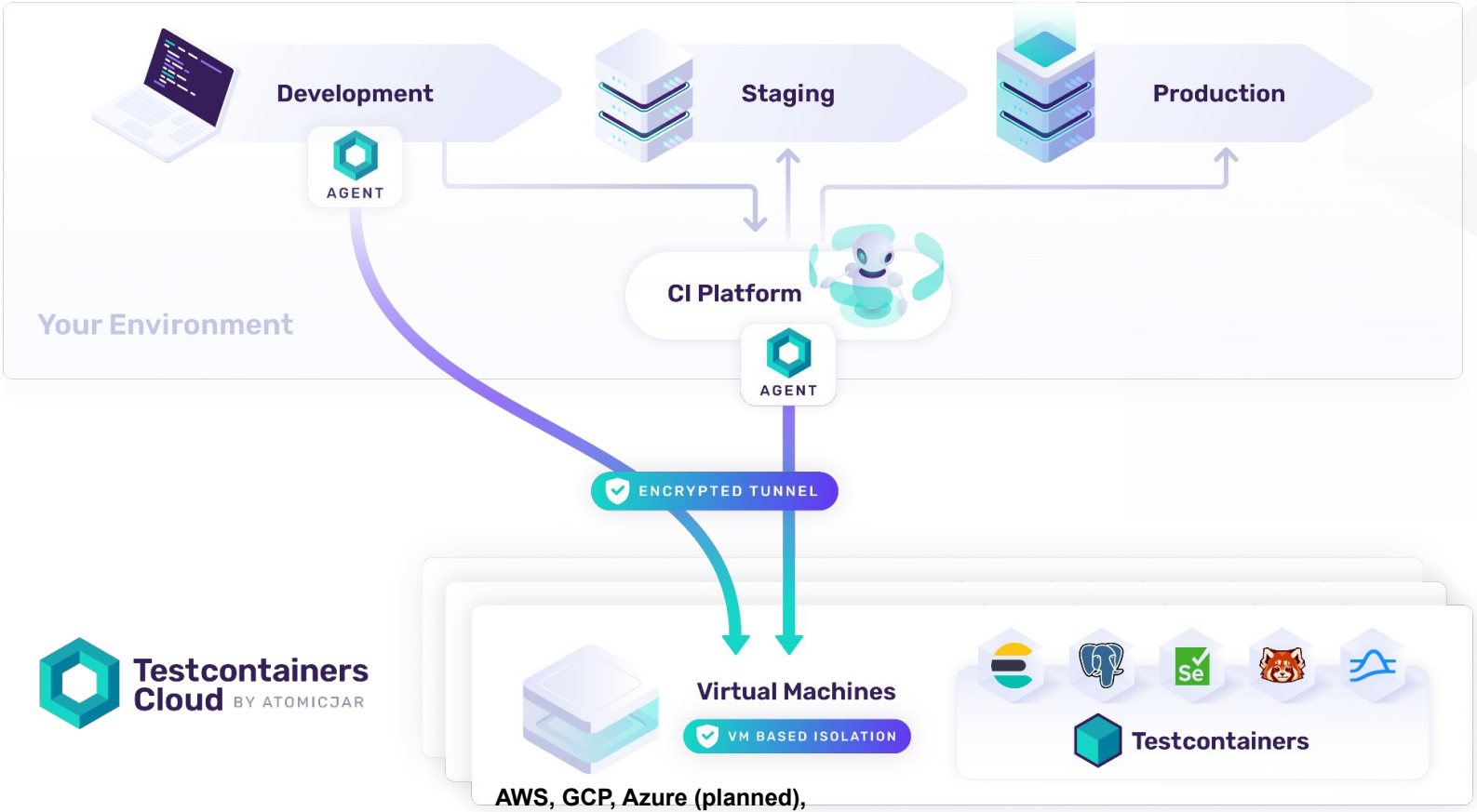
Effortlessly Fast CI

Run your ever-growing test suite without scaling your CI, and speed it up by running tests in parallel

Reliable Test Suites

Enhance team efficiency by getting rid of flaky tests and ensuring consistency from dev to CI

How it works?



AWS, GCP, Azure (planned),
private RedHat OpenShift cluster.

docker

Questions and Answers