Pipekit: a Deployment Tool with advanced scheduling and Inter-Service Communication for Multi-Tier Applications

Pablo Chico de Guzmán
IMDEA Software Institute, Spain
pablo.chico@imdea.org

Felipe Gorostiaga
IMDEA Software Institute,Spain
felipe.gorostiaga@imdea.org

César Sánchez
IMDEA Software Institute, Spain
cesar.sanchez@imdea.org

***Abstract*—Modern cloud applications are based on microservice architectures. The deployment of these microservice based applications often requires that every constituent service starts after all its dependencies are configured and running properly. It is also common that these dependencies generate dynamic data that needs to be supplied to other services too at starting time. More complex scenarios require additionally interchanging data in other phases of the microservices lifecycle.**

One alternative to solve these dependencies is to describe the deployment of microservice applications manually—using scripts—which allows IT operators to precisely define when a service is ready to start serving other components. However, synchronization by scripting is tedious, error prone and hard to maintain. Other solutions offer specific languages to describe service dependencies, along with tool support that interpret scripts in these languages to take care of starting services in the proper order. These tools are either very rich but complex to use, or fail in providing sophisticated ways to describe what it means for a service to be ready.

Moreover, the communication layer between services, if supplied, is based on intermediate entities and non-trivial network protocols.

This paper proposes *pipekit* as a solution, by offering a container orchestration language which focuses on simplicity (*pipekit* is similar to Docker Compose) and is equipped with directives to define when a service is ready. The *pipekit* tool provides a communication layer for moving data between services, implemented using shared storage. This shared storage provides a very simple interface to move artifacts between services, and greatly simplifies the synchronization logic of *pipekit* by using semaphores at the file system level.

***Keywords*-orchestration; microservices; deployment; synchronization;**

I. Introduction

Microservice architectures are becoming increasingly popular, so large monolithic implementations are being replaced by smaller independent and reusable services that collaborate together [6], [12]. Even though (micro)services are designed as standalone components, more often than not these services need to interact with each other to work as intended for the general combined application. This combination entails a set of dependencies between services, which has to be taken into account when the application is deployed. One important case is starting a service only when those services upon which it depends have started and are *ready*. The set of coordination activities necessary to deploy a microservice application is usually known as cloud service orchestration[1] [11].

One possibility is to describe the activities involved in starting the application via a launching script crafted by hand. This script must take care of organizing the booting times, postponing the execution of a service until all the services it dependes on are up and ready to provide the required functionality. This approach offers complete fine grain control over the deployment steps. However, synchronization via scripting tends to be tedious, error prone and very difficult to maintain and debug, specially when many services are involved and the application is susceptible to suffer changes.

Many tools have been proposed to deal with this problem in the context of containers [9], [13], such as Docker Compose [1] or Helm [3]. These solutions typically offer a declarative language that allows describing the important information to boot and connect services as well as their dependencies. These solutions also offer a tool—which interprets deployment descriptions written in the corresponding declarative languages—and launches the microservices in the order described. However, even if a container implementing a certain service has been launched this does not necessarily mean that the service is ready in terms of the application logic. For starters, there is typically a non-negligible initialization phase after the server boots. During this phase, the service is not ready to attend requests, potentially leading to a race condition if any of the dependent

[1]Even though the term orchestration can refer to a more general notion of describing the whole control flow of service oriented applications.

services tries to communicate too soon. Even worse, the service might fail to initialize successfully, a situation that could go unnoticed by its client services cascading the error and producing a failing trace that is very difficult to diagnose and manage. The diagnosis is difficult to perform because the symptoms may be detected in different components than the root cause (the service that failed to initialize properly).

One solution, suggested for example by Docker Compose, is to install additional code inside the dependent service (alongside the application code) to check whether a service is ready, which pollutes the service codebase (the service true functionality) with orchestration details and infrastructure management logic. Moreover, this approach introduces a new source of potential bugs, forces developers to handle non-business issues, and spreads significant part of the orchestration logic across all components.

This paper proposes *pipekit*, an alternative for defining service bootstrap orchestration. One of the key design principles of *pipekit* is to minimize the learning curve to aid its adoption. In particular, *pipekit* extends Docker Compose by adding a more precise way to specify what it means for a service *to be ready* using a novel "depends_on" clause. By doing so, a larger part of the booting logic remains in the hands of the orchestration tool, and the description of the orchestration and the service functionality remains properly separated. The *pipekit* orchestrator starts the services in the order described, delaying their execution until all the respective dependency conditions have been met. If any of the conditions is taking too long, *pipekit* can be instructed to timeout and abort the deployment reporting the issue. Then, the IT operator can inspect the *pipekit* logs (which do not contain application logic but only orchestration messages) to diagnose the error and quickly identify the offending component as the root cause of the failure.

Additionally, the deployment of microservice archirectures often requires the exchange of data between different services for initial configuration. This problem can be managed by configuration tools such as Puppet [8] or Chef [10], but these tools involve the management of a centralized entity to communicate this data using non-trivial network protocols. This approach is unsuitable in many of cases, such as moving a dynamic secret. Instead, the *pipekit* orchestrator provides a very simple communication layer between services by mounting a shared storage layer in every service. This communication layer can be also used for developing complex continuous integration [7] tasks that require moving artifacts between different execution steps, where each execution runs on a different machine, possibly concurrently. This communication layer based on shared storage greatly simplifies the implementation of *pipekit* by allowing the usage of synchronization mechanisms implemented at the file system level, which is more secure and easily traceable.

Section II introduces *pipekit* by an example. Section III describes the *pipekit* communication layer. Section IV shows the value of *pipekit* by solving a complex continuous integration task.

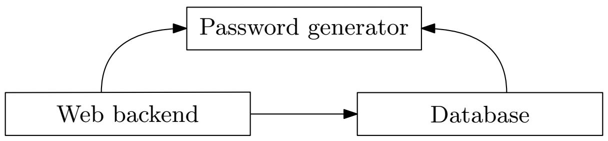

Figure 1. Example of Multi-Tier Application.

II. Pipekit by an Example

In order to improve the learning curve of *pipekit*, the syntax of its configuration file has been chosen to be similar to that of Docker Compose. A *pipekit* configuration file is a yaml file where the services that are composed for a given application are listed and named. For each of them, the file specifies the docker image that the service container must run, a set of signals that expose the milestones through which the service goes, and a set of conditions that have to be witnessed before the service is started. Other fields from the Docker Compose format such as ports, environment variables or labels are also supported by *pipekit*, but we will focus here on the `signals` and `depends_on` clauses, since they are the novelty of *pipekit*.

A signal is defined by a name, that must be unique in the scope of the service defining it, and a signal constructor from the set of constructors that are available in the *pipekit* tool. The supported constructors at the time of writing are:

- `Grep(fd, string)`: scans the file descriptor passed as the first argument and looks for a match of the string supplied as the second argument against the input from the descriptor.
- `Exists(path)`: becomes *true* when the path supplied as the argument corresponds to an existing file. The path is relative to the shared storage layer mount point.

For example, to create a signal that fires when a service generates the log *"Database is ready"* to the standard output, we can create the signal `Grep(stdout, "Database is ready")`. This kind of interaction is very common. For example, when a database like mysql finishes initialization and is ready to listen to requests on a port, it sends a message to the log which can be scanned. In order to wait for a configuration file to be generated, we could issue the signal `Exists("path_to_file")`.

Optionally, every signal can receive an additional argument specifying a maximum timeout. For example, if we want to fail the orchestration if the *"Database is ready"* log is not generated after 30 seconds, we could write the signal `Grep(stdout, "Database is ready"):30`.

Consider an application consisting of three services: *web*, *database* and *password-generator*, with the dependencies

```
version:"2"
services:
  password-generator:
    image:password-generator
  database:
    image:mysql
    signals:
      ready:Grep(stdout,"Database␣is␣ready"):30
    depends_on:
      - password-generator.ExitCode(0)
  web:
    image:web-backend
    depends_on:
      - password-generator.ExitCode(0)
      - db.ready
```

Figure 2. *pipekit* yaml example.

```
FUNCTION pipekit(path_to_yaml_file):
   fired:=[]
   waiting:=GetAllServices(path_to_yaml_file)
   ready,waiting:=GetReadyServices(waiting,fired)
   Deploy(ready);
   computing:=GetSignals(ready)
   WHILE(computing||waiting):
      signal:=select(computing);
      computing:=computing-signal;
      IF timeout(signal): CONTINUE;
      fired:=fired+signal;
      ready,waiting:=GetReadyServices(waiting,fired);
      IF ready!=[]:
        Deploy(ready);
        computing:=GetSignals(ready)
   RETURN waiting!=[]
```

Figure 3. *pipekit* pseudo-code.

depicted in Fig 1. A correct orchestration has to make sure that *password-generator*—which is responsible of generating a secret password—boots first. Afterwards, the secret is consumed by *database* for initialization of access credentials, and by *web* for configuring the connection to *database*. The *database* initialization might take some time after the container is spawned, when the initial phases of the database take place. During this time, *web* should not try to connect to *database*. Finally, the orchestration finishes by starting *web*.

The orchestration directives to deploy such application can be written in *pipekit* via a yaml description file shown in Fig. 2. Each service defines the `signals` that other services can consume for synchronization. For example, the *database* service defines a *ready* signal indicating that it can start accepting connections. The signal *ready* is implemented by using the directive `Grep`. In our example, the *ready* signal will be canceled if it takes longer than 30 seconds to fire. The *ready* signal is consumed by *web* using a `depends_on` clause. This way, *web* will not be started until the signal *ready* is fired, ensuring that *database* is ready to receive connections before *web* is started.

By default, *pipekit* implicitly defines an *ExitCode* signal for every service to easy the definition of sequential dependencies. The parameter of the *ExitCode* signal is the *exit code* of the service. *ExitCode* is used by *database* and *web* to force that these services are only started after *password-generator* has successfully finished its execution. Note that the secret is directly accessible by *database* and *web* because it is saved in the shared storage layer that *pipekit* configures between every service.

The *pipekit* algorithm will start *password-generator* first because it has no dependencies. If *password-generator* finishes successfully, *pipekit* will start *database* since it satisfies all its conditions. After a few seconds, if *database* is initialized properly the code injected in *database* for the `Grep` constructor will activate the *ready* signal. At this time, the orchestration can finish by starting *web*. The pseudo-code for the *pipekit* algorithm is shown in Figure 3.

III. Pipekit Communication Layer

The *pipekit* tool provides an *out of the box* inter service communication layer, implemented by mounting a shared file system in every service. A file system offers a very simple interface to read and write dynamic data generated at deployment time by different services.

We consider two different scenarios for running *pipekit*: (a) in a single-host environment; (b) a multi-host environment. When *pipekit* runs in a single-host environment, for example, in a developer machine, a shared file system is provided by a Docker Volume, which is simply a shared mount point in the machine file system. The case when *pipekit* runs in a multi-host environment is more complicated. Some Container Cluster Management tools, such as Kubernetes [4] or Docker Swarm [2] offer Docker Volume plugins to support distributed storage layers, but this technology is not mature and reliable yet. Instead, the current *pipekit* implementation integrates with *i2kit* [5], a deployment tool that transforms every service into its own set of virtual machines, in order to simplify the integration of containers with cloud vendor technologies such as virtual networking or load balances. The current *i2kit* implementation supports AWS deployments. AWS offers EFS, an elastic file system implementation that can be mounted by thousands of virtual machines, which *pipekit* uses for providing a shared file system for every service.

The *pipekit* tool organizes the shared storage layer in different folders, using the name of each service as the folder's name, to simplify the access to the data generated by each service. Also, any service can access a `.env` file in the folder of any other service, populating the variables included in the file as environment variables. Following the *password-generator* example, the *password-generator* service could write the line `PASSWORD=secret-value`

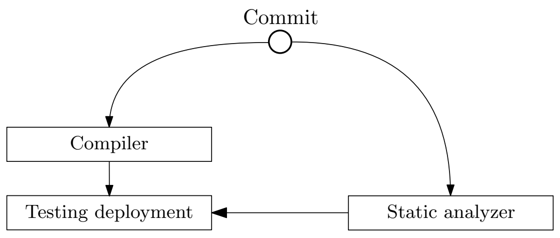

Figure 4. Continuous Integration Pipeline.

into *database/.env* and *web/.env*, and the environment variable `PASSWORD=secret-value` would be populated to both services, *database* and *web*. This mechanism avoids polluting the container logic with code to read data from the communication layer. Moreover, most of the containers available nowadays receive their input data via environment variables.

IV. Pipekit Continuous Integration Example

Many times, the definition of a continuous integration pipeline requires artifacts to be passed among components. For example, the pipeline shown in Fig. 4 gets executed when code is committed to a branch of a repository. This pipeline starts two services, *compiler* and *static-analyzer*. The service *compiler* is responsible of generating the application binaries, and *static-analyzer* performs a static analysis on the source code to detect, for example, type errors and other statically-checkable errors and warnings. Once the binaries are generated, the compiled code is passed to *testing-deployment*, a service responsible of deploying the application in a single machine. This machine can be used, for example, by product managers to validate that the functionality added by the code commit works as expected.

The *pipekit* tools make it very simple to synchronize the pipeline execution, by adding `compiler.ExitCode(0)` and `static-analyzer.ExitCode(0)` as dependencies of the *testing-deployment* service. Additionally, note that each service has a very different infrastructure requirements. The service *compiler* is network intensive, in order to download the code dependencies, and its execution might take several minutes. The service *static-analyzer* is highly CPU intensive, and its execution might last for hours. Finally, *testing-deployment* requires little CPU and network bandwidth, but its execution might last for several days.

Ideally, *compiler* and *static-analyzer* should run in large machines to reduce their execution time, but *testing-deployment* can run on a very small machine to reduce costs. In order to take advantage of running each service in different machines, the binaries and the static analysis reports need to be passed to the *testing-deployment* service. We argue that it is over killing to implement a complex communication layer to support a typical scenario like this. In contrast, the *pipekit* shared file system allows providing this functionality *out-of-the-box*.

V. Conclusions and Future Work

Docker has recently "democratized" the container technology. Containers are a great solution to execute processes in a portable manner, and distribute these processes along different machines. However, native Docker orchestration tools, such us Docker Compose, are not expressive enough to support complex deployment scenarios. There are other solutions, designed before the *container era*, that solve this problem, such us Puppet or Chef, but the learning curve of these tools is highly non-negligible.

In contrast, *pipekit* follows the Docker Principle of simplicity. The input to *pipekit* is designed as an extension of the Docker Compose format for easy adoption. The implementation of the communication layer as shared storage is simple, powerful and more importantly, it requires no pre-configuration steps.

Future work includes evolving the current *pipekit* implementation from a *proof of concept* to a tool *usable* in production. We are also working on adding more directives, such as checking if a port is open, and also providing the ability for extending with user custom directives. Other research line is to provide support for other parts of the application life cycle (beyond deployment). For example, error handlers for service failures, or the description of controlled shut down of the system would be valuable extensions. Finally, we plan to work on a modeling the semantics of the *pipekit* algorithm in order to apply formal techniques like model checking and runtime verification. These techniques would enable, for example, designing algorithms that generate diagnosis reports to provide concise explanations of run-time errors at the orchestration of the deployment.

References

[1] Docker compose. https://docs.docker.com/compose/overview.

[2] *Docker Swarm.* https://github.com/docker/swarm.

[3] Helm. https://helm.sh.

[4] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Commun. ACM*, 59(5):50–57, April 2016.

[5] Pablo Chico de Guzmán, Felipe Gorostiaga, and Cesar Sanchez. i2kit: A tool for immutable infrastructure deployments based on lightweight virtual machines specialized to run containers. *CoRR*, abs/1802.10375, 2018.

[6] Martin Fowler. Microservices a definition of this new architectural term. http://martinfowler.com/articles/microservices.html.

[7] Martin Fowler. Continuous integration, 2016. https://martinfowler.com/articles/continuousIntegration.html.

[8] Jes Fraser. Puppet. *Linux J.*, 2011(207), July 2011.

[9] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[10] Stephen Nelson-Smith. *Chef: The Definitive Guide*. O'Reilly Media, Inc., 2013.

[11] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, October 2003.

[12] Johannes Thönes. Microservices. *IEEE Software*, 32(1):113–116, 2015.

[13] Chenxi Wang. Lxc and docker explained. http://www.infoworld.com/article/3072929/linux/containers-101-linux-containers-and-docker-explained.html.