Deployment model for distributed applications

Todor Ivanov[1], Nikola Valchanov [2]

[1] *Plovdiv University, Faculty of mathematics & Informatics, 236, Bulgaria Blvd, Plovdiv, Bulgaria*
[2] *Plovdiv University, Faculty of mathematics & Informatics, 236, Bulgaria Blvd, Plovdiv, Bulgaria*

Abstract
This article reviews pros and cons of monolith and distributed architectures and points out potential problems when it comes to deployment of applications. It proposes a deployment model for distributed applications that will solve some of the most common problems and will provide an easy way to manage dependencies, configurations and authentication.

Keywords
Deployment, distributed applications, service discovery, cloud

1. Introduction

Nowadays, one of the most important steps of the development of an application is the deployment. Thanks to this process, the application becomes useful and available for the clients. There are many different types of deployment processes which can be used in order to build and provision our application in the cloud, but people nowadays tend to use distributed deployment strategies because they are managed easier.

When it comes to non-centralized deployments, we tend to use different tools for the specific services (e.g TeamCity for .NET, Heroku for JS and etc.), and that makes the whole process more complex. Compared to that, in the centralized deployments we have a single source of tasks that manage all of the services. We can use them both for monolith and distributed applications.

We have a lot of pros when comparing distributed and monolith architectures. The main ones are noticeable when it comes to resources, scaling, partial updates and error handling. In the monolith architecture, when trying to execute some of the operations above, it's quite possible to have some system downtime. This is because the application components are tightly coupled and cannot easily be used independently. This, however, is not the case with distributed applications [2]. In them, each part is a service that can be managed separately. Usually, we have more than one instance of the application running at any moment and all updates, scalings and resource management can be applied for a specific instance and part of the application components [5].

In order to configure a distributed architecture deployment, we need to solve a few problems [3]. These days every service is being deployed to the cloud [4] and, happily, this cloud is protected. For us, to manage these services, it's a bit hard because we need to authenticate ourselves or our deployment pipelines to the cloud. This happens with authentication keys or secrets. Unfortunately, these credentials cannot be added as a part of the project itself because this is not secure. The practice here is to encrypt the credentials and include them in the CI pipelines additionally where they will be decrypted and used. Moreover, if we have multiple environments, we need to duplicate those encrypted files for each environment.

In order to understand the problem easier, we can examine the following frequently met use case - we have an existing architecture presented in Figure 1 of a distributed application that is deployed somewhere in the cloud. By reviewing the architecture, we can see that the application consists of a web client, API Gateway, EmailService and a DB. In order to deploy that application to the cloud, we need a pipeline created by any CI tool. There are a few mandatory steps and configurations that need to be done [1]. These steps may include configuring building scripts, setting up deployment procedures and consuming a vault for secrets and configurations.

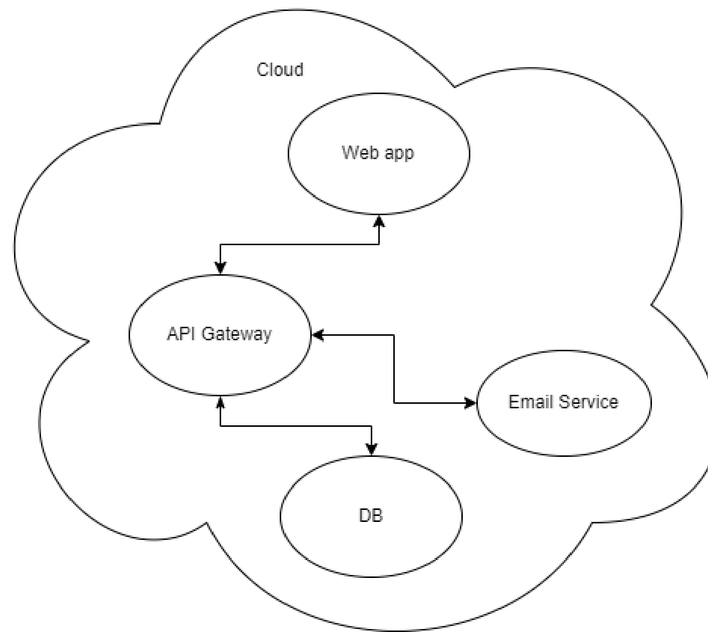

Figure 1: Distributed application architecture

The end result of each pipeline is an artifact that should be deployed to the remote server. There are a few already existing solutions that handle the process of deployment and configuration of the services. Sometimes pipelines are configured in a way that administrators have to include additional files with secrets that are used for access control, configuration variables and etc. which is an expensive procedure in terms of time because first of all the secrets are encrypted manually and second of all, the pipeline should include additional logic to read those files and add them to the artifacts.

Another solution would be to store these configurations in environment variables and to implement storage for the artifacts which can later be accessed by another task or pipeline in order to fully deploy the service. In terms of implementation, this is easier but it requires human intervention or additional scripts that should run the secondary deployment tasks.

2. Architecture of an application that manages distributed systems

In order to solve these problems and present a unified way of storing configurations, access tokens and keys we can take advantage of the architecture presented in Figure 2.

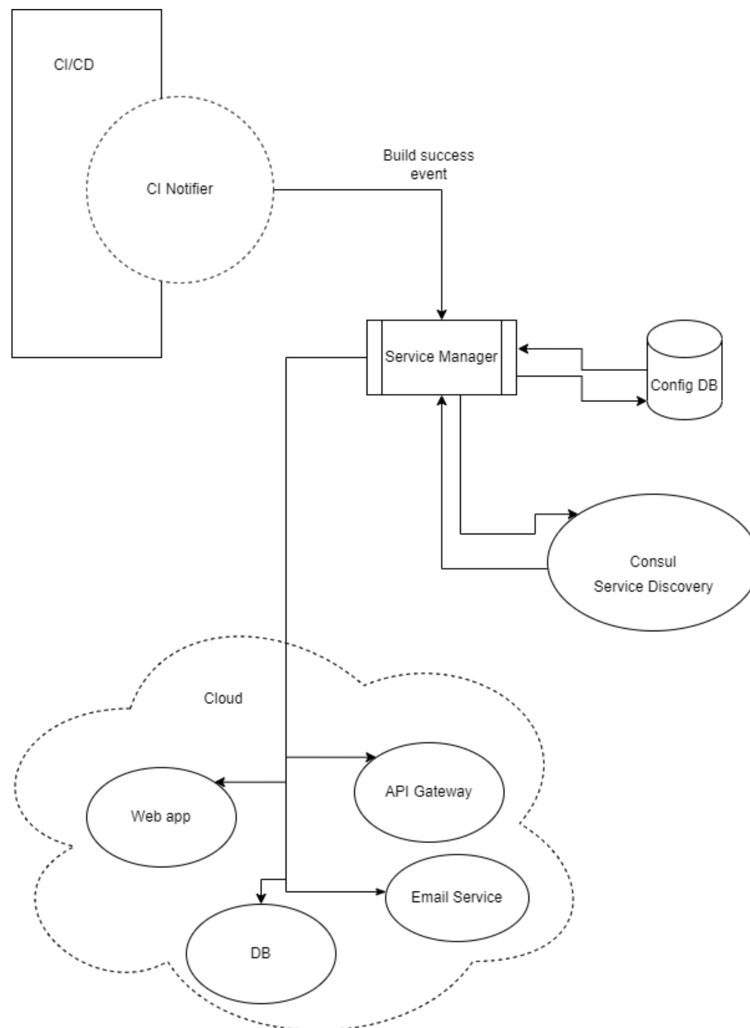

Figure 2: Architecture of application for managing distributed systems

The proposed architecture extends the default CI pipeline behavior by adding a CI Notifier [1]. The Notifier allows us to attach different kinds of middlewares on every step of the pipeline. In this particular case we are attaching a middleware that will be interested in the build success events from the CI Notifier. The end result of these events will be the already built codebase that is ready for deployment.

The other parts of the architecture are as follows:

- A Consul server [6] [7] handles service addresses. This is done by taking advantage of the Service Discovery feature of Consul where clients of Consul can register a service, such as API Gateway or DB, and other clients can use Consul to discover providers of a given service. Using either DNS or HTTP, applications can easily find the services they depend upon.
- A Database - will be responsible for the storage of configurations, access tokens and keys
- A Service Manager - it is responsible for obtaining the configuration for the specific service and its proper deployment.

In order to obtain a better understanding of the proposed architecture and its workflow, we can look at the sequence diagram in Figure 3.

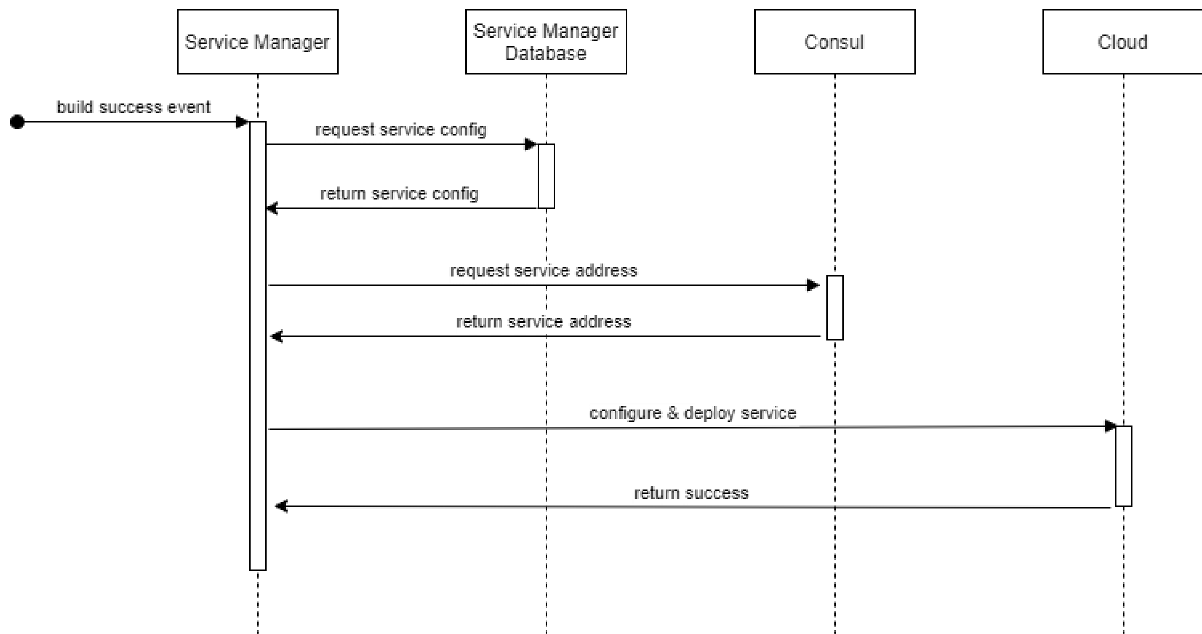

Figure 3: Sequence diagram of application architecture

With the presented diagram we can see that the flow will consist of several different steps. The CI Notifier emits an event at every step of the pipeline, but the start of the process is set when a success event has been dispatched. This event will have the artifact that contains the built codebase of a specific service as a payload. What is important here is that the service manager is attached as a middleware only to this success event and that allows us to filter and discard all of the unneeded events.

When the Service Manager receives the success event two main actions are triggered:

- The Service Manager searches for configurations, access tokens and keys in the config database and retrieves them. Each configuration may consist of compose files, helm charts, YAML files, etc. and the access tokens and keys are responsible for the proper authentication and authorization in front of the cloud and its infrastructure.
- Once the proper configurations and access tokens are acquired the Service Manager needs to know where the artifact for the specific service should be deployed. Using the access tokens, it can contact the Consul server which will return the proper address for deployment.

3. Conclusion

By reviewing the use case presented in this article we can come to the conclusion that there is still a problem when implementing deployment models for distributed applications and the problem is that credentials and configurations needed for each deployment are managed ineffectively. In order to solve that problem we can take advantage of the suggested architecture for an application that manages distributed systems. Using that architecture we are building an application called ServiceManager whose main responsibility is to manage service configurations and credentials. That way we will have a global and centralized way of credential management which will improve the currently existing processes.

4. References

[1] T. Ivanov, N. Valchanov, Extended Model of Code Orchestration and Deployment Platform, TEM Journal 11 (2022), 856-861, DOI: 10.18421/TEM112-45

[2] I. K. Aksakalli, T. Çelik, A. B. Can, B. Tekinerdoğan, Deployment and communication patterns in microservice architectures: A systematic literature review, Journal of Systems and Software 180 (2021), 111014. doi: 10.1016/j.jss.2021.111014

[3] L. Chen, Continuous Delivery: Overcoming adoption challenges, Journal of Systems and Software 128 (2017), 72-86. doi: 10.1016/j.jss.2017.02.013

[4] T. A. Lascu, J. Mauro, G. Zavattaro, Automatic deployment of component-based applications, Science of Computer Programming 113 (2015), 261-284. doi: 10.1016/j.scico.2015.07.006

[5] A. Avritzer, V. Ferme, A. Janes, B. Russo, A. Hoorn, H. Schulz, D. Menasché, V. Rufino, Scalability Assessment of Microservice Architecture Deployment Configurations: A Domain-based Approach Leveraging Operational Profiles and Load Tests, Journal of Systems and Software 165 (2020), 110564. doi: 10.1016/j.jss.2020.110564

[6] O. Al-Debagy, P. Martinek, A comparative review of microservices and monolithic architectures, in: 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), IEEE, 2018, pp. 000149-000154

[7] M. Schumacher, H. Helin, H. Schuldt, Service Discovery, in: CASCOM: intelligent service coordination in the semantic web. Springer Science & Business Media, 2008