Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

DevOps

Sean Skelton.
G00361169

Introduction

The project is a 3-tier web application consisting of a JSON authentication service and a chat service, broken down into login and register functionalities along with chat microservices. The frontend is a single-page application built on the Vue framework.

The entire application, including the database, is hosted on AWS Elastic Kubernetes Services (EKS).

This application is divided between two repositories, with both of them fully deployed via a continuous integration and continuous deployment (CI/CD) pipeline written in GitHub Actions. The frontend undergoes automated unit tests, while both the frontend and the services are built with Docker, uploaded to AWS, and automatically deployed through Terraform scripts. These Terraform scripts create the necessary resources on the cluster to make the frontend publicly accessible and enable internal communication with all the services.

Rough UML of deployed application.

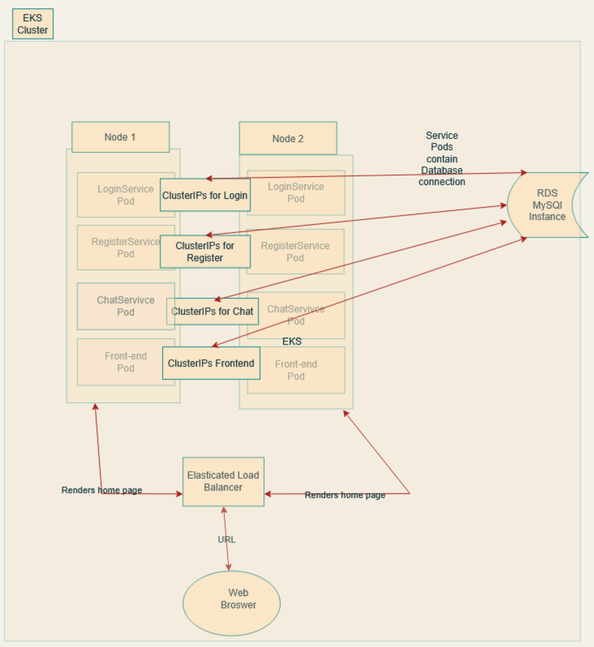

Figure 1

How this works.

The EKS cluster is the first component to be created, which orchestrates all the containerized applications. This is done through managed node groups, which are responsible for provisioning resources for the smallest unit Kubernetes can manage—the pods that run your containerized application. The replica factor for all my services was set to 2. By default, EKS tries to ensure that two pods from the same deployment do not end up in the same node group, providing a guaranteed level of fault tolerance in case one pod goes down. However, this behavior is not guaranteed unless explicitly configured, so when my services are deployed, recreated, or deleted—or even if the pods themselves go down—this internal structure is not guaranteed, but rather something that is aimed for.

A Cluster IP is configured and assigned to pods within the same deployment. This allows communication between the frontend, other pods, node groups, the RDS instance, etc. However, there are special rules for the ChatService. Since it requires a WebSocket connection to be maintained for a longer duration than an HTTP connection (which typically closes once the request is sent and the response is received), the chat component and the server must both stay actively listening for events. To avoid routing requests between the two instances of my chat service, I enabled "sticky sessions" on my frontend. This ensures that once a connection is established with one of the chat service pods, it continues to consistently communicate with that specific pod.

The frontend is exposed via a load balancer that has a public IP address for users to connect to. Requests are sent to the load balancer, which uses a combination of the Cluster IPs and defined relative routes to forward the requests to the correct service.

The DevOps Life Cycle.

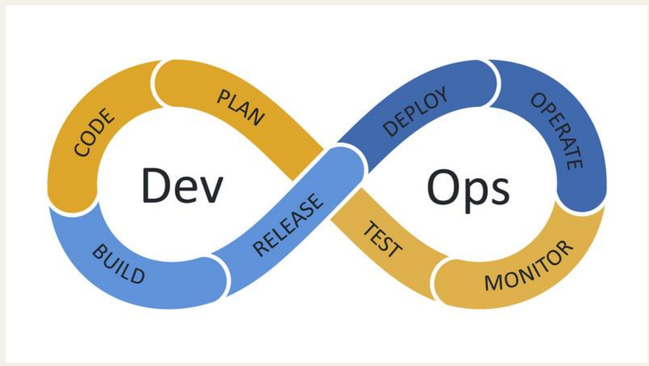

Figure 2

Defining Dev ops

his is a term that has been thrown around a lot in the software industry recently, and throughout the various stages of development for this application, I've read dev blogs, Medium articles, literature reviews, and case studies. It seems that it is not something that has been standardized with a set implementation. Different work cultures place varying emphasis on the practices associated with this methodology.

Some organizations focus more on the use of tools, such as pipelines and cloud deployments, done via automated scripts with Infrastructure as Code (IaC) tooling. Others view it as more of a cultural movement aimed at eliminating clashing and finger-pointing between teams, with tools serving as a way to break down historically siloed teams for testing, developing, and deploying. Still, others see it as a basket of IT practices to pick and choose from.

Most of the companies interviewed in the case studies that implemented their own flavors of DevOps seemed to increase overall productivity and streamline their development processes. While there were teething problems (I've created merge conflicts on Git before even writing a line of code), the key takeaway is that when you're making changes to an enterprise application with thousands of users, there will always be challenges—no matter what set of tools or practices you use.

What would I have done differently?

I think when I'm building applications, sometimes I can get hooked on trying to fix errors, logical inconsistencies, or getting a smaller part of the application working without stopping to evaluate how mission-critical it is. Is this the core functionality of the application? Is this the backbone of my communication? Or am I just trying to get a user interface to display a conditional message on the screen?

While untested and broken code should not be in production, identifying key functionality and prototyping it would have sped up my development process. This brings me to my next point.

I think creating rough architecture diagrams showing how the components of my application will interact both locally and in a development environment would have been extremely helpful. While I wouldn't worry about class definitions, functions, methods, etc., had I looked at how I architected my application locally versus how it might look when deployed, it would have highlighted problems much earlier. Local development often allows you to hardcode and bypass simpler configurations, which can create more work at a later stage in the development process when there is more pressure.

Lastly, working through the parts of your application in a way that makes sense is crucial. I started with my frontend and then tried to set up my AWS resources. While this approach did pay some dividends when it came to automating my AWS services, I had not developed enough functionality for this to be worthwhile at that stage. If I had started with a server and a simple database, I would have defined communication for my application, knowing what requests the server expects and what it will return. This would have allowed me to build my frontend from the outside in—working on a service that sends and receives requests, then a module that generates those requests, and finally a simple UI component that passes the request to the module. This approach would have ensured that if I got stuck, it would have been on something necessary for progression. Instead, I wasted time and energy on parts of the application that were non-essential early on, losing time on core infrastructure that I had to build anyway just to get back to the point where I was stuck.

The last point is leveraging CLI tools and scripts more. It doesn't matter if it's Windows PowerShell, Bash, Git, Python, or kubectl. CLI tools are usually GUI-independent, which is especially valuable when working with cloud infrastructure. Being able to launch things from your command line allows for a very quick and efficient way to identify and fix problems within your infrastructure. These issues are not always abundantly clear in a control panel or console, and sifting through logs can sometimes feel like trying to find a needle in a haystack. If every time I was coding, I created a script for any command I ran more than five times—or especially for a set of commands that needed to be run sequentially—I would have saved a lot of time. These tools are often some of the most overlooked assets in a programmer's arsenal.

What to move onto next?

Particularly, my exposure this semester to distributed systems sparked my interest in cloud infrastructure, and that's how I ended up wanting to do a project on DevOps, trying to utilize and learn about these technologies. It initially started off as an attempt to define a blueprint application that people could clone, plug in their own AWS credentials, and have it provision a fully deployed three-tier application with all the necessary resources but no functionality implemented. This would give people a kind of sandbox environment if they wanted to experiment. I felt that a lot of the context around cloud infrastructure is generally not broached for hobbyists or people trying to learn—it's often designed for those who already have a strong foundation using cloud services.

Unfortunately, this project very quickly moved away from that direction. I have an interest in both messaging and authentication systems, as I've worked on small personal projects related to both. I thought this would be a brilliant opportunity to marry them into one project. However, I also began to realize that basing the success of my application on the implementation of something I was new to was quite a risky prospect.

Many of the tools in this space are themselves complex, as they manage abstractions of really intricate software.

For example, you have tools like Terraform to create scripts that create scripts to provision an Elastic Kubernetes Cluster, which itself is an abstraction of Kubernetes. That's just insane.

My next step will be to refactor this application and remove all the state and logic to try and bring it back as an attempt to create a resource for learning. Then, I want to try creating my own Kubernetes cluster to manage my own resources, specifically to gain a better understanding of the internal communication within an EKS cluster.

Conclusion

My experience developing this application taught me a lot about the full development lifecycle, from the conception of an idea to delivering a finished project. I learned the importance of researching the problem domain before attempting solutions, which will help me better architect my future projects. I gained a huge amount of knowledge from using AWS cloud infrastructure, Terraform, and pipelines to automate the testing, building, and deployment of my application.

Although I'll never claim to have it fully defined, I now have a much better understanding of DevOps. I've learned that it's not just about tooling but also about fostering collaboration in the workplace by sharing the responsibility of the application across everyone, rather than segmenting it into isolated parts between teams. Tools are only one part of this—it's also about having a willingness to learn and not being afraid to make mistakes.

Learned pririty assessment for developing codebases for applications and leveraging the command line interfaces and automation scripts to speed up repetitive tasks In my own workflow. Lastly this project gave me the skills to programmatically create networking resources now its given me the interest to configure and create theses resources myself

References

[1] https://veddevopsblog.hashnode.dev/deploying-a-three-tier-application-on-aws-with-kubernetes-a-step-by-step-guide
[2]https://www.researchgate.net/profile/Floris-Erich-2/publication/316879884_A_Qualitative_Study_of_DevOps_Usage_in_Practice/links/5a8250c0a6fdcc6f3ead7b34/A-Qualitative-Study-of-DevOps-Usage-in-Practice.pdf
[3] https://arxiv.org/pdf/1907.10201

Acknowledgements

I want to give a thanks to my supervisor Kevin O'Brian for his guidance.
MK for the support.