InfoQ Dev Summit Munique (15 e 16 de outubro): Descubra como a DKB, a Delivery Hero e a Zalando estão resolvendo os desafios do desenvolvimento.

Apresentando o Sock Shop: um aplicativo de referência nativo da nuvem

Principais conclusões

- A Sock Shop começou como um aplicativo de demonstração simples, mas depois provou ser bastante útil e acabou se tornando um aplicativo de referência nativo em nuvem e totalmente em contêiner.
- O aplicativo foi desenvolvido usando uma combinação de Go, Java com Spring e Node.js. Ele conta com um pipeline completo de integração e entrega contínua até nosso ambiente de "preparação", um cluster Kubernetes executado na AWS.
- APIs que utilizam a especificação Open API. A equipe utilizou uma ferra especificações do Swagger e valida a API em nosso serviço em contêiner
- A Sock Shop possui implantações para Docker (autônomo), Docker Swa Nomad da Hashicorp e AWS ECS.
- Durante o trabalho, a fusão de dois serviços em um único foi muito mai é adiar a decisão de dividir os serviços se ela não estiver muito clara.

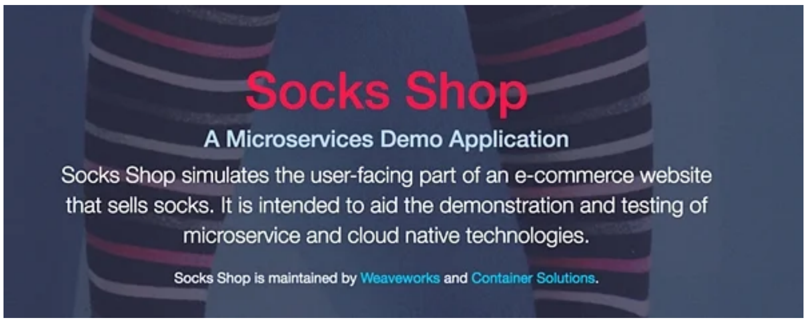
Texto original

APIs using the Open API specification. The team u Dredd , which takes our swagger specs and then v our containerized service.

Classificar esta tradução
O seu feedback vai ser usado para ajudar a melhorar o Google Trad

O mundo dos microsserviços, contêineres e frameworks de orquestração é, no mínimo, um espaço turbulento. Aparentemente, surge uma nova ferramenta, serviço ou produto a cada semana, com novos conceitos, técnicas e jargões. O termo Cloud Native surgiu para abranger essa nova maneira de construir aplicativos com uma mentalidade de "nuvem em primeiro lugar". No entanto, tentar definir exatamente o que Cloud Native significa pode ser um exercício inútil, visto que o espaço que ele tenta definir está em constante mudança. Um desenvolvimento recente que vimos foi a criação da Cloud Native Computing Foundation. Subordinada à Linux Foundation, é uma organização sem fins lucrativos com foco em tecnologias comuns e software de código aberto.

This feels like a step in the right direction, but it does not yet help us in defining what a cloud native application should be. We can point to some technologies (containers, orchestration frameworks) and best practices (ephemeral, scalable, flexible), however it often helps to have a concrete example to reference. This is exactly what we have been building …


Socks Shop
A Microservices Demo Application
Socks Shop simulates the user-facing part of an e-commerce website that sells socks. It is intended to aid the demonstration and testing of microservice and cloud native technologies.
Socks Shop is maintained by Weaveworks and Container Solutions.

This is the Sock Shop. A real life, fully containerized, microservice architecture, cloud native application.

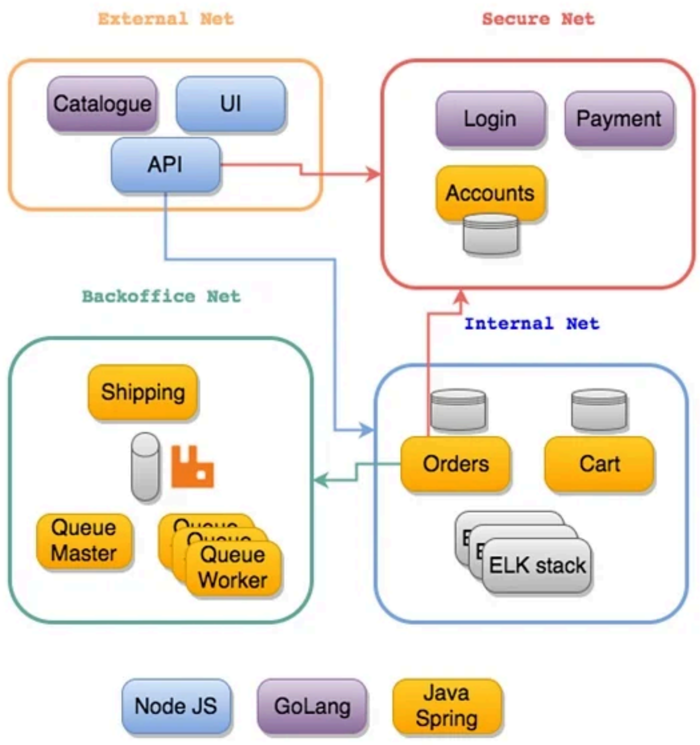
This project began as a small demo application for DockerCon to show off some new services developed by Weaveworks. As a company which focuses on building products and tooling for microservice and container based applications, they needed an actual application on which to demo their services. In two weeks, we built an “aggressively microserviced” application, throwing in a variety of technologies, programming languages, and datastores.

After its initial run, we saw benefits in keeping the project alive. It turned out to be quite useful, both as a testbed for container and microservice focused tools, as well as reference application for what a Cloud Native system should look like. Over the course of the next few months, we worked to convert this demo application to be production ready. We overhauled each service, swapping out datastores where necessary, merging services where there was high coupling, as well as adding standard features such as logging and monitoring. We additionally built a full continuous integration and delivery pipeline through to our "staging" environment, a Kubernetes cluster running on AWS.

Microservices

Cloud Native applications must be flexible. We need quick, independent, deployments, which can allow us to make the right decisions and the right the changes at the right time. This enables Agile organizations. Change should be anticipated and encouraged rather than avoided. Cloud Native is about independence of individual services. Each piece of our system cannot depend on others, so as to avoid cascading failures. And finally, Cloud Native should be technology agnostic. Choosing a programming language for one piece of our system, or going with a relational database for a specific set of data should not affect the rest of our system.

Clearly, this is not something we can achieve with a monolithic architecture. While this is not the necessary architecture in every case, Cloud Native applications should be built on top of microservices. With the Sock Shop, we aggressively "microserviced" the application.



This gave us a lot of benefits; we were able to leverage polyglot persistence, i.e. different datastores for different services. We were also free to choose the language for each service. The real benefits come in development velocity. As this is an open source, community project, there are lots of developers involved for short periods. With a proper segregation of the services, it is quite easy for someone to come in and quickly fix a bug or add a feature to one of the services, without needing to understand the entire architecture.

Containers

The widespread adoption of microservices has been in large part due to certain technologies which have made it much easier for us have applications split out into small, independent pieces. Specifically, container technology. It provides us with lightweight packaging and tooling.

When microservices go too far
The claim is made that starting from scratch from microservices can actually be more difficult than converting an existing monolith application. One of the reasons given is that with a greenfield project, the bounded contexts between services is not always clear. If you split your services along the wrong boundaries, you end up with too much inter service traffic and this can be difficult to rectify. We ran into this issue early on with our Sock Shop application. As

we were aggressively splitting out services, we had a log-in service and a customer service. As we moved forward it became clear that they were too highly coupled and should in fact be a single service. Merging them into a single service was much trickier than anticipated. The advice here is delay the decision to split services if it is not extremely clear.

In addition to increasing adoption of microservices, containers have also been at the core of the shift of applications from on premise to the cloud. Be it the applications themselves which are running in containers, or the infrastructure on which they are running, such as Google Cloud, which has been running in containers for more than ten years. Containers are the building blocks of the cloud, and are a crucial part of Cloud Native applications.

Within the Sock Shop, we adopted containers across the landscape. More than just running our services in docker containers, we looked at how to take advantage of containers throughout the integration and deployment pipeline.

One example was with our builds. We want to have consistent, reproducible builds, but we also don't want to end up with bloated images for our services by including unnecessary tooling. In order to satisfy both of these requirements, we create separate images for building and running our services. The build image includes the necessary tools, and gives us consistent reproducible build artefacts on any environment. Then we copy the built binary (or jar file) into a minimal deployment image for actually running our services. (Example)

Another place we leveraged containers was for testing, which in microservices land is quite different. Our services are quite reasonable in size, so testing the internal functionality can be done easily. However, our services have many more external dependencies, which are notoriously tricky for testing.

A major focus we made on testing was our services APIs. Since these are the contracts through which our services will communicate with each other. We documented all of our APIs using the Open API specification (i.e. Swagger) which has a lot of tooling built around it. We used a tool called Dredd , which takes our swagger specs and then validates the API on our containerized service. As we hook this into our CI pipeline, we get continuous validation of the APIs on all of our services. We run all of these in containers, which gives us the repeatable behavior and is agnostic to the platform in which it is running.

We created a testing Docker image, which contains our test framework and the Dredd tooling. The container reads in an API (Swagger) Specification, and executes API calls against our actual services, performing a full validation of our API spec. Our testing frameworks are ephemeral, created and subsequently removed for each test run, and agnostic to the environment where they are run.

Orchestration

The third pillar on which we build Cloud Native systems is orchestration. Once we have our individual services built, tested, and deployed, the next obstacle is managing our various containers, to maintain a stable yet flexible running environment. As soon as you have more than a few containers, manual orchestration becomes unmanageable; we need automation. The main pieces of any application are the containers (running our services), servers, storage, and networking. From this set of initial elements, we have a desired state: running application(s), discoverable services, fault tolerance, resource efficiency. And ideally, this should all be done without any manual intervention.

Luckily there are many tools and platforms that aspire to take care of this for us. With Sock Shop, rather than choose a specific orchestrator to deploy to, we decided to deploy our application on as many of these platforms as possible. The rationale being that as a reference application, it should not be tied to a specific platform, to be truly 'cloud native' it should function on any orchestrator. An additional benefit of this approach is that it allowed us to evaluate and compare the different tools. At the time of writing, Sock Shop has deployments for Docker (stand alone), Docker Swarm, Kubernetes, Mesos/Marathon (or DC/OS), Hashicorp's Nomad, and AWS ECS, all of which is documented here.

For the most part, the application did in fact 'just work' on these platforms. We simply had to adjust the config files based on the syntax for each tool. Because all of our pieces were containerized, deployment was straightforward. The main caveat here was networking. We use WeaveNet as a plugin for our container networking and DNS needs. In certain cases this just worked, but more often we ran into clashes with existing DNS/Networking pieces built into the tool. Container Networking Interface is/will be the way forward, however it is still in an early phase, and not fully integrated with all tooling yet.

Recap of Cloud Native

There are three main pillars for Cloud Native applications: microservices, containers, and (dynamic) orchestration. While there are often other technologies involved, focusing on having these three pieces will go along way in building a Cloud Native architecture. They are also complementary technologies. As we have seen with the Sock Shop, container technology makes building, testing and deploying microservices much easier. And adding an orchestration platform allows us to confidently run our application at scale. If we were to manually manage the deployments, restarts, scaling and scheduling of each service we would quickly become overwhelmed, and the benefits of having a Cloud Native application would be outweighed by the effort required to keep it running.

Some other pieces which are important for building Cloud Native apps include Continuous Delivery pipelines, comprehensive monitoring and logging, as well as tracing. All of which we have included in the Sock Shop.

Next steps for Sock Shop

Now that the application is in a stable state and we have the necessary automation and testing in place, one of the interesting use cases is swapping out certain tooling/languages/tech to see what unexpected challenges this creates, and how smooth the process is. One of the oft touted benefits of microservices architecture, and I would also argue it should be goal of Cloud Native applications as well, is modularity. Due to low coupling between the various pieces and infrastructure of our application, it should be trivial to swap any part with something equivalent.

One interesting example of this, is when a colleague of mine re-wrote one of our services in .NET. Aside from some challenges with writing the actual services, mostly due to the minimal state of .NET core (turns out that .NET Core on Linux is very bare bones at the moment), swapping out our original Orders service (written in Java) with the .NET version was seamless.

As a slightly more drastic example, I would like to try replacing the actual container runtime, currently Docker, with something else, like rkt. This would go a long way to show where we stand in terms of “cloud nativeness” and validating the Open Container Initiative.

O objetivo do Sock Shop é ser uma aplicação de referência para Cloud Native e também servir como um ambiente de testes para tecnologias, padrões e ferramentas relacionadas. É um projeto totalmente de propriedade e conduzido pela comunidade. Eu recomendo que todos analisem o código, testem a aplicação, usem-na para testar ferramentas e novas tecnologias e contribuam por meio de PRs.

Sobre o autor

Ian Crosby
é um desenvolvedor de software, entusiasta e defensor de longa data. Ele começou sua carreira desenvolvendo sistemas de defesa militar e, desde então, busca usar seus poderes para o bem. Em sua função atual como Engenheiro Sênior na Container Solutions em Amsterdã, ele auxilia empresas a migrar para o novo cenário nativo da nuvem, ao mesmo tempo em que trabalha com parceiros para desenvolver as ferramentas que as levarão até lá.