Microservices Deployment with Containers – Part 1

This lesson covers microservices deployment with containers in detail.

We'll cover the following

- A social platform with microservices architecture
- Deploying different services with containers

We know that *containers* are lightweight, provide a consistent environment across different platforms, have less startup time in comparison to *VMs*, facilitate better resource monitoring, etc.

Now let's understand why the container technology suits best for the microservices deployment, with the help of a simple example.

A social platform with microservices architecture

We'll take the example of a social networking service like Facebook. It has several different features that enable the platform's users to interact with each other better, such as:

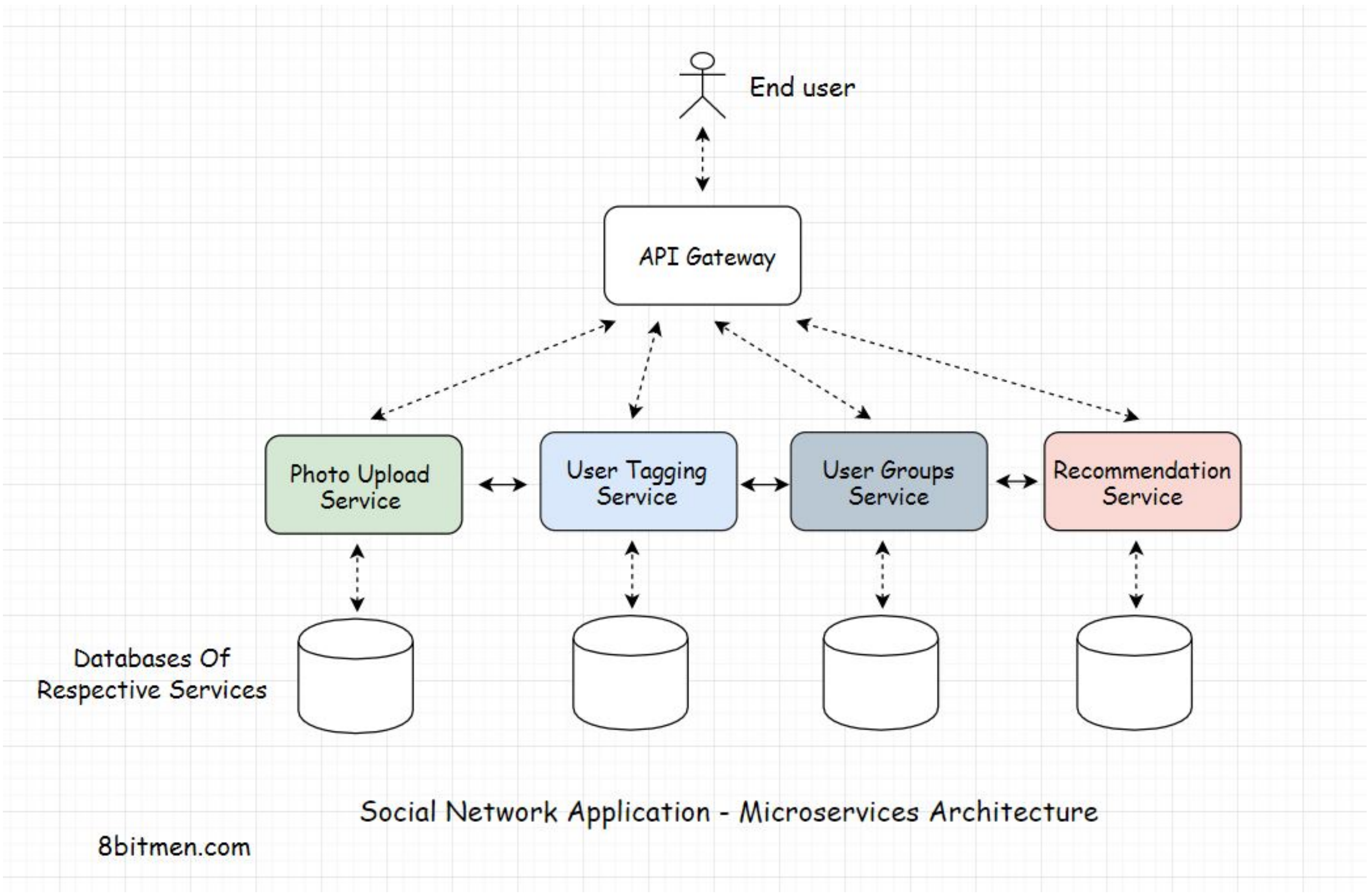
- The users can upload their photos on the site and share it with their friends.
- They can also tag their friends, places they visit, and so on.
- They can share their opinions with the world by writing a post.
- They can get along with users on the platform who have similar interests by creating dedicated groups.
- Users get recommendations on places to visit, new friends, books, movies, etc., based on their interests and browsing behaviors.

These are some of the key features that our social network should have to make it big.

Every feature will be a separate microservice.

If you are thinking why? Why not code all the features as a monolith? Why make *things complex?*

I’ve discussed the concepts like when to choose microservice architecture for your application and so on in another course. You can check it out here.



Deploying different services with containers

Once we are done writing code and our microservices are ready, the next step is picking the right technology to deploy our code. In this use case, we’ll leverage the container technology to deploy our microservices.

Why deploy in containers and not directly on VMs?

We’ve already discussed the differences between the *VMs* and *containers* in the previous lesson. We know that running containers is cost-effective, lightweight, & consumes fewer resources. It’s also easier to manage and scale the system than running the services directly on VMs.

> ***Some stats***: Uber runs over 4000 microservices.

According to an *InfoQ* article, posted back in 2014, Google starts over two billion containers per week, three-thousand per second. You can imagine the container count at Google today.

Imagine running all that code in *VMs* and the resources it would require.

For container technology, we can pick *Docker* to run our services. To deploy one single microservice, we will first create a *Docker image* for it; the image will contain the application code, dependencies, configuration, libraries, and everything required to run that particular service. This step is known as *containerizing the application*.

Once the image is created, we will deploy that image on the machine from the command line tool. Mounting images and running them as containers is done by the *container engine*.

A running instance of a container image is called a container.

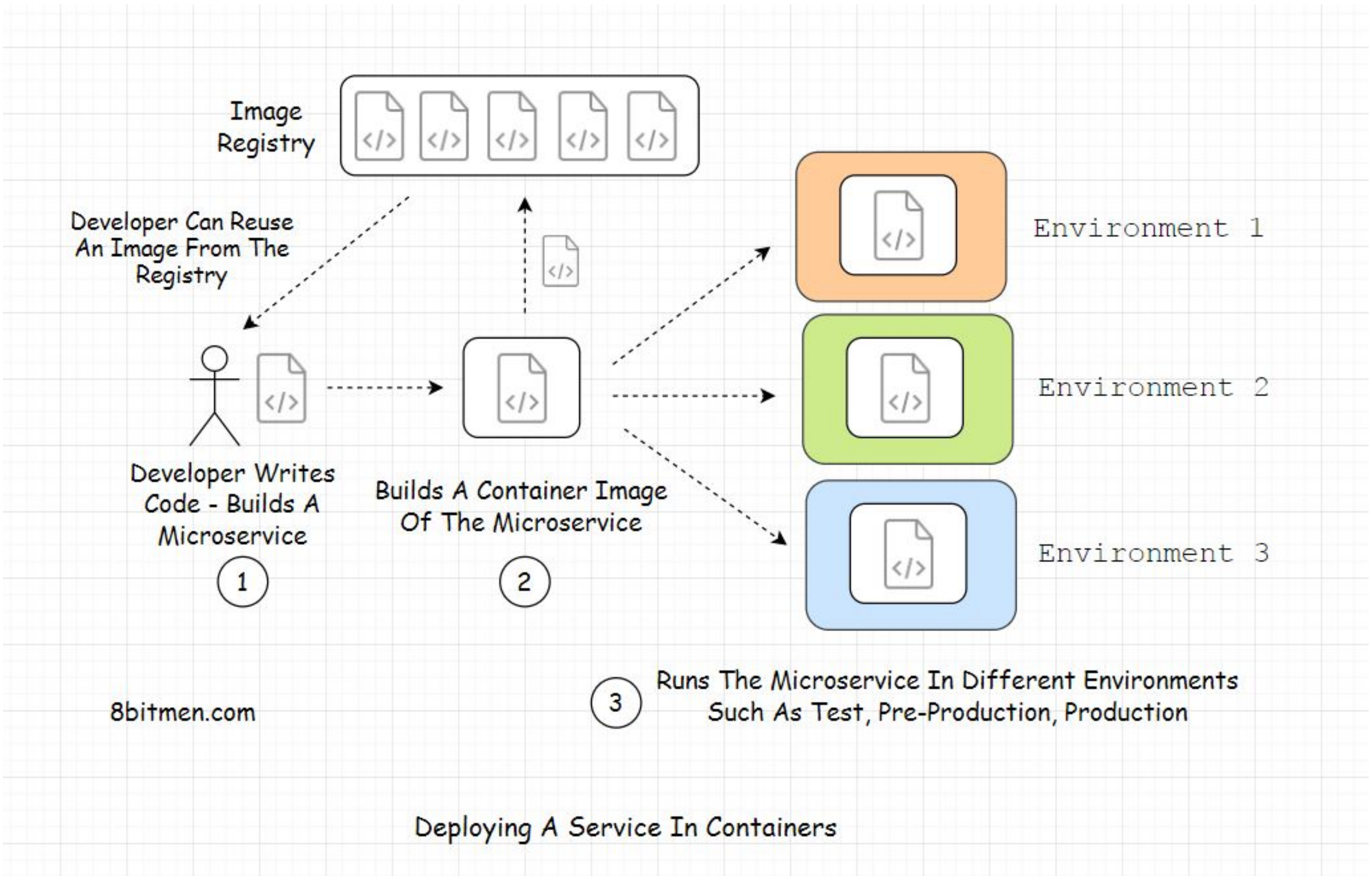


Image Registry
Developer Can Reuse An Image From The Registry
Environment 1
Environment 2
Environment 3
Developer Writes Code - Builds A Microservice
1
Builds A Container Image Of The Microservice
2
3
Runs The Microservice In Different Environments Such As Test, Pre-Production, Production
8bitmen.com
Deploying A Service In Containers

The container ecosystem also contains image registries that can be both public and private, just like the *GitHub* repositories. These registries contain container images as templates that can be reused with or without modifications at a later point in time.

Revisiting the why containers and not *VMs* discussion, containers save engineering teams a solid amount of time by letting them off the hook for making the machine ready to run a particular service built with a certain technology stack.

ready to run a particular service built with a certain technology stack.

When built using microservices, the entire system isn’t built using one particular technology. All the different services are written using different technology stacks. Speaking of our social network application, we can write different modules using different technologies such as *C++, Java, RoR, Go, Python*, and so on.

If it wasn’t for container technology, the operations team would have to ascertain that the machines are ready with respect to compatibility with a particular technology before the application is deployed.

Now, when the number of microservices and their instances reaches the hundreds of thousands, this can become a huge pain.

Additionally, a microservice that runs on a certain technology cannot be deployed on the machine prepped up for any other technology without making serious modifications to the environment.

However, with containers, we don’t have to worry about all this stuff as they keep the environment isolated from the application code.

We’ll continue this discussion in the next lesson.