Design architecture of Containerized FineractCN

Problem Statement:

Presently Fineract CN is in the process being deployed on a Virtual Machine. The disadvantages of this deployment include:

- **Scalability**: A microservice will not scale horizontally as workload increases.
- **Availability**: There is no tool to make sure at least one instance of a microservice is always available in the case where the other instance(s) fail.
- **Reliability**: There are no health checks to make sure that the system is not only up and running, but can also process a request.
- **Efficiency**: The deployment is not efficient enough. Reason being one instance can be overloaded with many requests while there is still memory and CPU resources available.

Solution:

A good solution to this problem is to containerized the system using container technology. Here is a list of possible container technologies. From this list, Docker and Docker Swarm will be used. The reasons for choosing this solution (containerizing Fineract CN) and how this solution solves the problem is explained here.

Overview of design architecture:

Fineract CN uses four external tools: Spring Cloud Eureka, Apache ActiveMQ, Apache Cassandra and MariaDB and thirteen domain microservices.

- The microservices and tools each reside in separate docker containers.
- The database images use an external volume to persist data even after the database's container goes down.
- Docker swarm is used to orchestrate the microservices.
- The Docker CLI contains the docker-compose file used by docker swarm, Jenkins (for continuous integration) and Jfrog (to manage the artefacts)
- The orchestrator scales the docker containers horizontally and vertically as indicated in the docker-compose file
- The black arrow indicates the order in which the containers start.
- First, we start the external tools (Eureka, ActiveMQ, Cassandra and MariaDB) and perform health check to make sure they are ready to receive request.
- Then we start the domain microservices in order of the number of dependents each microservice has. That is, the microservice with the highest number of dependents starts first then the next highest, and so on. Hence we have the order, Provisioner, Identity, Rhythm, Organization ...
- Health checks are done to make sure the domain microservices connect and can successfully communicate with the external tools.

Containerization and benefits of containerizing Fineract CN.

What are containers?

Containers offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually run. This decoupling allows container-based applications to be deployed easily and consistently, regardless of whether the target environment is a private data center, the public cloud, or even a developer’s personal laptop.

Benefits of containerizing Fineract CN using containers, case study: Docker
• **Cost saving:** One advantage of using container technologies like Docker, is cost saving. Docker drives down cost while raising profit by dramatically reducing infrastructure resources. The nature of Docker is to have fewer resources than an application running on a traditional virtual machine. It will be cheaper to run lightweight Fineract CN images in a production environment than running heavy virtual machines containing the microservices. For development we could put all the images in one container, which will require less resources.

• **Standardization and productivity:** Docker containers ensure consistency across different environments, i.e., from development to production environments. Standardizing service infrastructure across the entire pipeline allows every developer contributing to Fineract CN to work on an environment that is as close as possible to the production environment. By doing this, contributors are more equipped to efficiently analyze and fix bugs within Fineract. This reduces the amount of time wasted on defects and installation troubles, and increases the amount of time available for feature development.

• **Continuous Integration Efficiency:** Docker enables you to build a container image and use that same image across every step of the deployment process by providing a consistent development, build, test and production environment. The length of time it takes from build to production can be speed up notably. A user with enough resources can run Fineract CN such that each microservice maps to a single image and, can use the same images both in his/her development and production environment.

• **Multi-Cloud Platforms:** One of Docker’s greatest benefits is portability. Over last few years, all major cloud computing providers, including Amazon Web Services (AWS) and Google Compute Platform (GCP), have embraced Docker's availability and added individual support. Docker containers can run inside an Amazon EC2 instance, Google Compute Engine instance, Rackspace server or VirtualBox, provided that the host OS supports Docker. If this is the case, Fineract CN containers running on an Amazon EC2 instance can easily be ported between environments, for example to VirtualBox, achieving similar consistency and functionality. Also, Docker works very well with other cloud providers like Microsoft Azure, and OpenStack, and can be used with various configuration managers like Chef, Puppet, and Ansible, etc.

• **Rapid and continuous deployment:** Docker manages to reduce deployment to seconds. This is due to the fact that it creates a container for every process and does not boot an OS. Data can be created and destroyed without worry that the cost to bring it up again would be higher than affordable. So, on a production environment containing Fineract CN, we can easily take down an unhealthy container and bring a new one up in a matter of seconds.

• **Efficiency:** Containers also provide the benefit of abstraction. Because developers no longer think in terms of machines, Fineract CN docker images can be colocated on different machines without affecting the microservices themselves. This means that tasks from multiple users can be packed tightly onto fewer machines.

Efficiency can be measured by the ratio of the useful work performed by a machine or process to the total amount of energy spent doing so. Most of the tools available for deploying traditional applications are not efficient enough, any idle CPU time is money wasted. Therefore, it is the system administrator’s responsibilities to keep utilization at acceptable levels, which requires ongoing management. This is where containers and container orchestration technologies come in. Container orchestration technologies provides tools that automate the distribution of Fineract CN microservices across a cluster of machines, ensuring higher levels of utilization than are possible with traditional tooling.

Therefore we can deploy Fineract CN such that it is resilient and always available just by declaring the required configurations in the specification file.

• **Faster deployments with lowered risk:** Some contributors have development machines that are different from production. When the dev team is ready to move the project into the production environment, they are often met with a production server that is different and possibly incompatible with the new or updated application. Containers provide isolation such that Fineract CN can run on different environments. Container Orchestration technologies will manage Fineract CN during deployment by controlling down time or rolling back to the previous version of the application when the current version faces failures.

• **Easy scaling of applications and clusters:** With container and container orchestration technologies, scaling is just a matter of changing a number in a configuration file. The orchestration technology takes care of the rest. For scaling to be successful, the resources should be available on the cluster to consume this replication.

• **Self-Healing Systems:** Container orchestration technologies like Kubernetes provide a self-healing system. When it receives a desired state configuration, it does not simply take actions to make the current state match the desired state a single time. It continuously takes actions to ensure that the current state matches the desired state. This means that not only will this technology initialize Fineract CN, but it will guard it against any failures that might destabilize the system and affect reliability.

---

**Shortcomings of containerizing Fineract CN using containers, case study: Docker**

• **Weaker isolation:** Virtual machine provided a higher level of isolation since the system’s hardware resources are all virtualized and presented to the virtual machine through a hypervisor. This implies intrusion into a VM is isolated only to that VM.

Containers are weaker because they share the OS kernel and other system components. As a result, flaws and attacks have a much greater potential to affect the underlying OS and over into other containers, potentially propagating malicious activity far beyond the original event.

• **Potential of sprawl:** Managing the lifecycle of a container is absolutely critical. Containers can be spun up and replicated at a very fast rate. This is an important benefit of containers, but it’s also possible to consume a lot of computing resources without truly realizing it. It’s not bad if the Fineract CN microservice containers are spun down or deleted when they’re no longer needed. But the costs to scale up the containers, and then forgetting to scale it back later, can impose significant (and unnecessary) cloud computing costs.

• **Working with stateful applications is complicated:** Containers are not suitable for stateful applications like database. By design, all of the data inside a container disappears forever when the container shuts down, unless you save it out of the container. There are ways to save data persistently in Docker, such as Docker Data Volumes, but this is arguably challenging and still has yet to be addressed in a seamless way.
**Struggling with dependencies:** VMs are extremely self-contained, each VM includes its operating system (OS) and drivers. As a result, VMs can be migrated to any other system as long as a suitable hypervisor is available. By comparison, containers run on top of the host’s OS, sharing much of the underlying OS kernel along with many libraries and binaries. Placing dependencies on containers that can limit portability between servers. For example, Linux containers under Docker cannot run on current versions of Windows Server.