“**Microservice architectural style** is an approach to developing a single application as a **suite of small services**, each **running in its own process** and **communicating with lightweight mechanisms**, often an HTTP resource API. These services are **built around business capabilities** and **independently deployable** by fully automated deployment machinery. There is a **bare minimum of centralized management** of these services, which may be written in **different programming languages** and use **different data storage technologies**.”

James Lewis and Martin Fowler - <https://www.martinfowler.com/microservices/>

**Benefits of Microservices**

* Independent scaling
* Independent releases and deployments
* Independent development
* Decentralized governance

**Operational Concerns**

* Service replication
* Service registration and discovery
* Service monitoring and logging
* Resiliency
* DevOps

**When to Use Microservices**

Microservices architecture best fits for:

* Two-pizza team / Cross functional
* Applications with high scalability needs
* Projects with high release velocity
* Read: <https://blog.christianposta.com/microservices/the-real-success-story-of-microservices-architectures/>

**Project Environment Setup**

**Postgress Details**

DB instance identifier : udagram-goswami-db

Master username : postgress

Master password : postgress

POSTGRESS\_USERNAME= udagram-goswami-db;

POSTGRESS\_PASSWORD= postgress;

POSTGRESS\_DB=postgres;

POSTGRESS\_HOST=udagramdemo.abc4def.us-east-2.rds.amazonaws.com;

AWS\_REGION=us-east-1;

AWS\_PROFILE=default;

AWS\_BUCKET= udagramgoswamidev1;

JWT\_SECRET=helloworld;

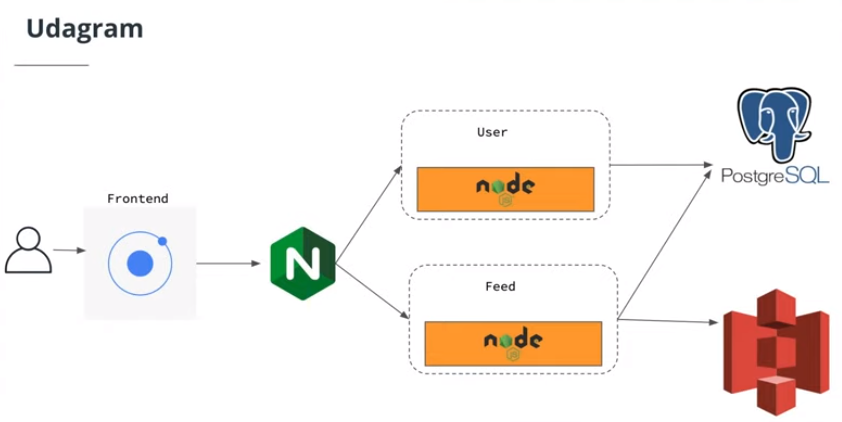
#### **Start the frontend server**

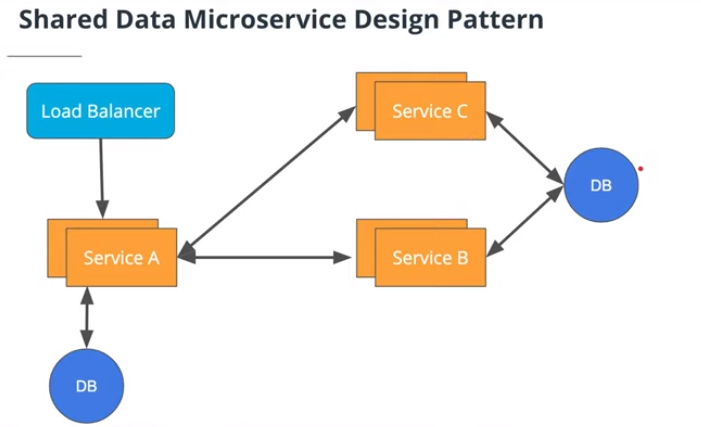
npm install

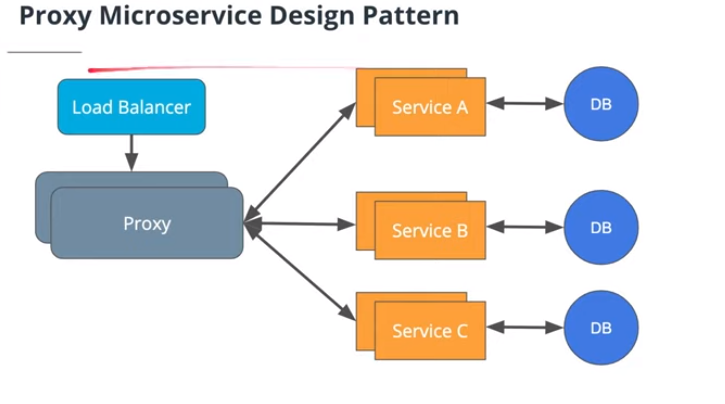
ionic build

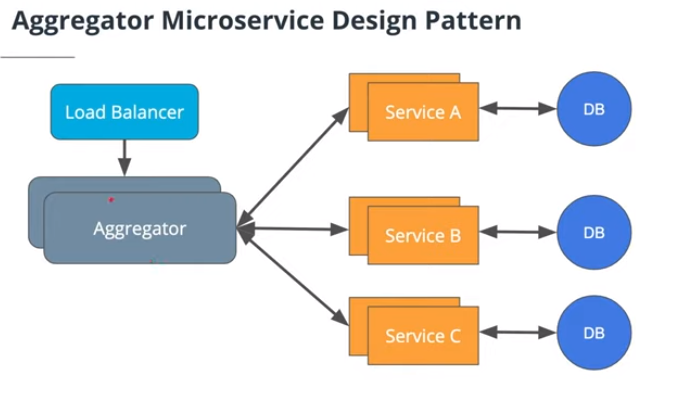
ionic serve

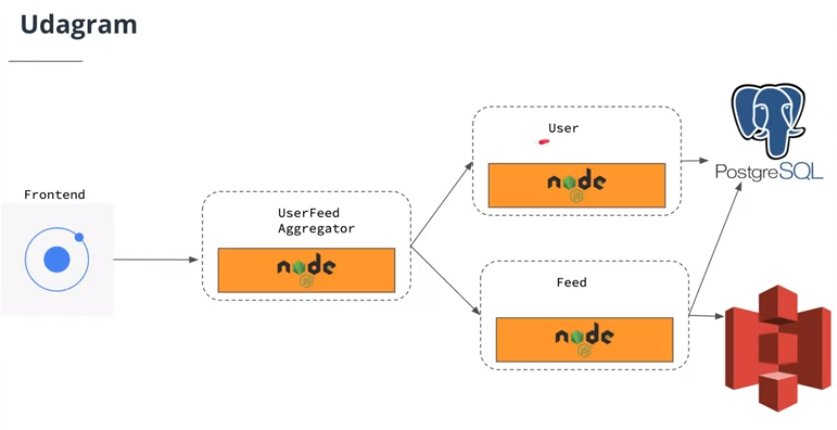
**Udagram Microservices Architecture**

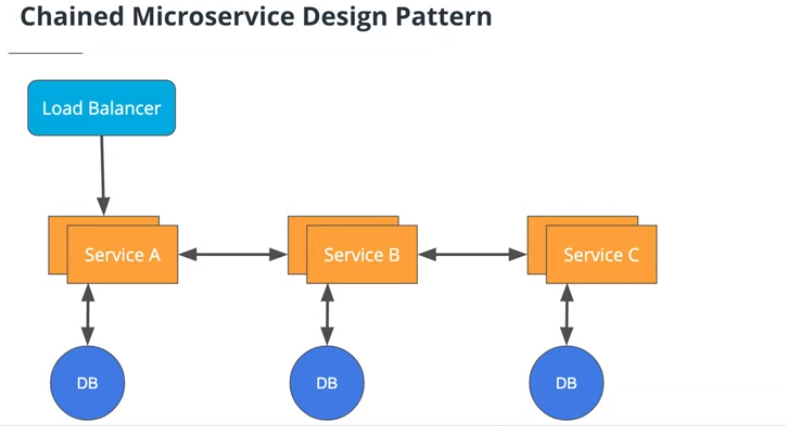
****

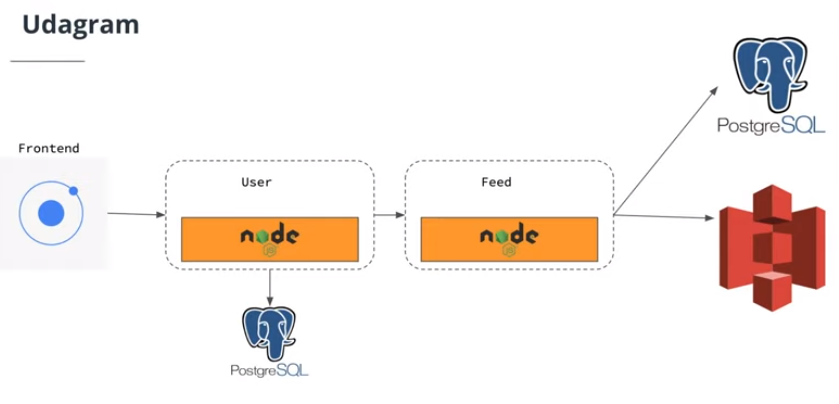
****

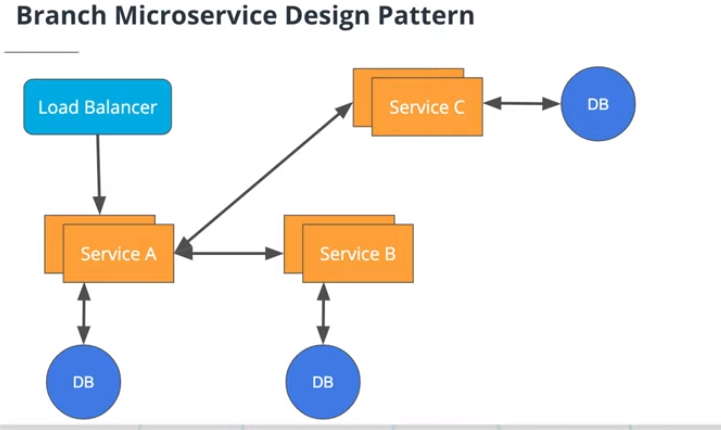
****

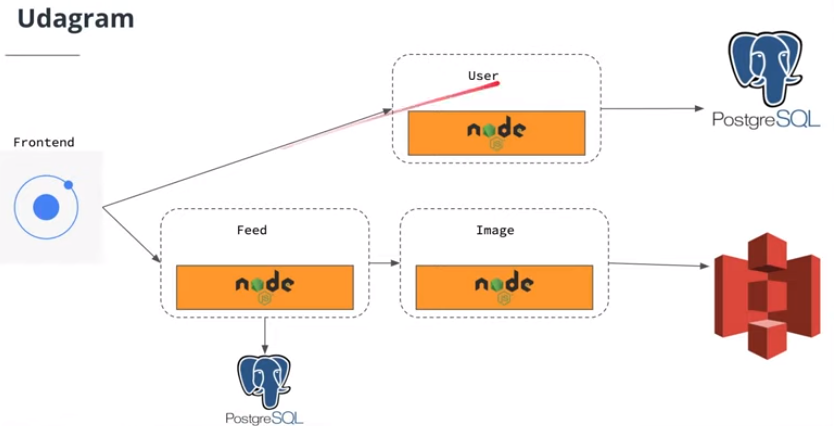
****

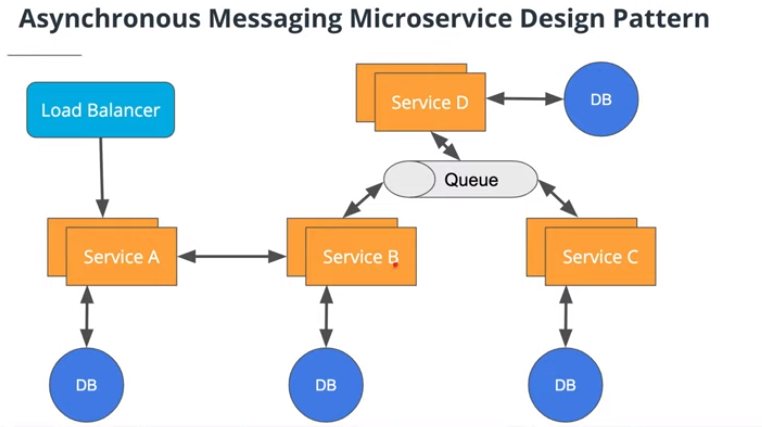
****

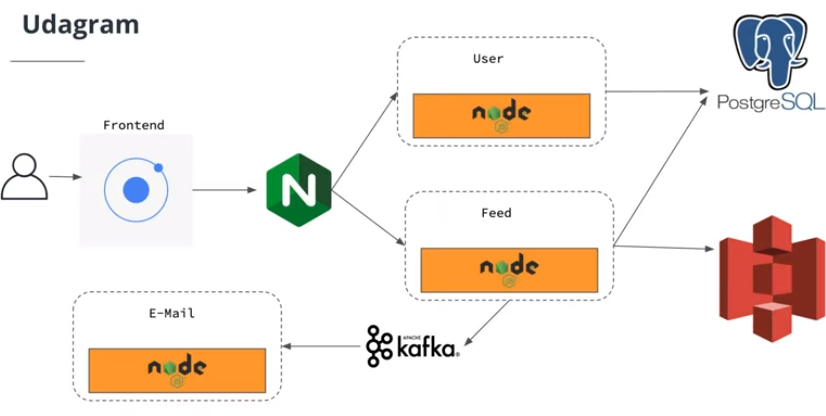
****

****

****

****

****

****

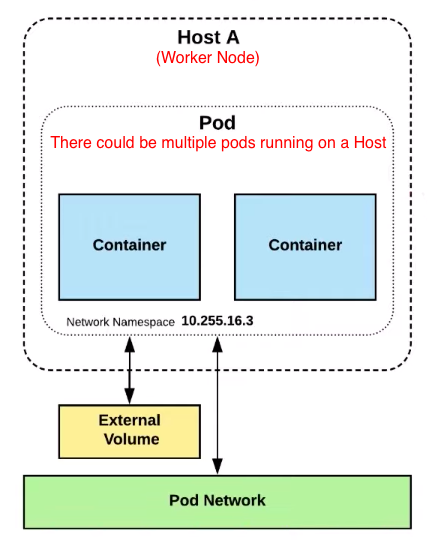
**KUBERNETES**

Container Orchestration : Container Orchestration is a logical process of grouping multiple containers into a single virtual entity. This logical grouping helps manage a group of containers that have a similar set of configuration requirements. There are multiple Container Orchestration tools available in the market, such as **Kubernetes,** Docker Swarm, Amazon ECS, Mesos, and Nomad. In this lesson, we’ll learn more about Kubernetes.

**What is Kubernetes?** Kubernetes (K8s) is an Apache 2.0-licensed open source Container Orchestration tool for effectively managing containerized applications. Kubernetes can automate the deployments, maintaining a logical group of containers, and helps to scale the application services.

## What is a Pod?

A pod is a "logical-grouping" of tightly coupled containers (one or more) that have shared storage, a network, and a standard specification. **The worker node(s) hosts one or more pods at a time.** The image below shows a pod having two containers running in a host.

[](https://classroom.udacity.com/nanodegrees/nd9990/parts/96fffeca-63e0-4bfc-92a6-a869b5b64b9e/modules/8c55d5a1-ae41-4313-ab37-86b1f35b9ada/lessons/e03717be-332d-4a2e-8576-69f7aae7726e/concepts/1fba225c-b8e8-4b2a-a90e-5b36a6f87cc6)

The *set of containers within a pod* have the following essential characteristics:

* share the same namespace (IP address and ports), storage, and network.
* can communicate within the set using localhost
* behaves like a single entity.
* will always run on a single host node (**co-located**) until the service that they run is terminated. Then, it frees up the resources of the node.
* will always be scheduled together to run on a host node as a single entity (**co-scheduled**). If a container is shut down/added/removed, then the pod has to "restart". Here, the "pod restart" means to restart the environment the containers run in.
* uses Docker as the container runtime
* run a single instance of the containerized application. Multiple instances of the application (horizontal-scaling) can be created by running multiple pods, one for each application instance.

## A **Controller** helps to manage multiple pods each running an individual instance of the application.

## The Role of Controller

Assume there are multiple pods, each running an individual instance of the application. Such a set of identical pods is called **ReplicaSet**. The ReplicaSet (of pods) ensures the high-availability of the services hosted inside them. ReplicaSets are created and managed by Controller.

The Controller specifies the necessary attributes and state of Pods and ReplicaSets in a .yaml configuration file, which is called **Deployment**. This configuration file provides declarative updates to manage Pods and ReplicaSets. The Controller can manage the situations, such as when the host (worker-node) fails, or the pod scheduling is interrupted, using the "Deployment" configuration file. In such cases, the Controller automatically replaces the pod by scheduling an identical replacement on a different node.

## Pod Templates

The Controller uses another .yaml configuration file called "Pod Template". It contains the pod specifications such as name, count of replicas, containers to run, port, and many other details. Once a pod or ReplicaSet is created by using the configuration file, they will not be affected if we change some parts of the configuration file later. If needed, the running pods have to be updated manually or re-deployed using the updated file.

## Deployments

Controller specifies the necessary attributes and state of Pods and ReplicaSets in a .yaml configuration file, which is called **Deployment**. This configuration file provides declarative updates to create and manage Pods / ReplicaSets. We define a Deployment to:

* create new Pods or ReplicaSets
* delete the existing Deployments, thereby releasing the compute resources occupied by them

A Deployment contains the details about the containers that would comprise the Pods / ReplicaSets.