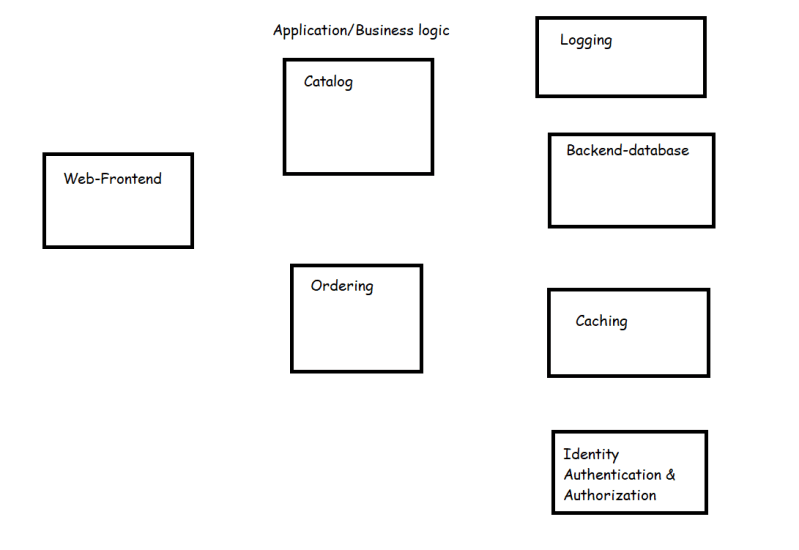
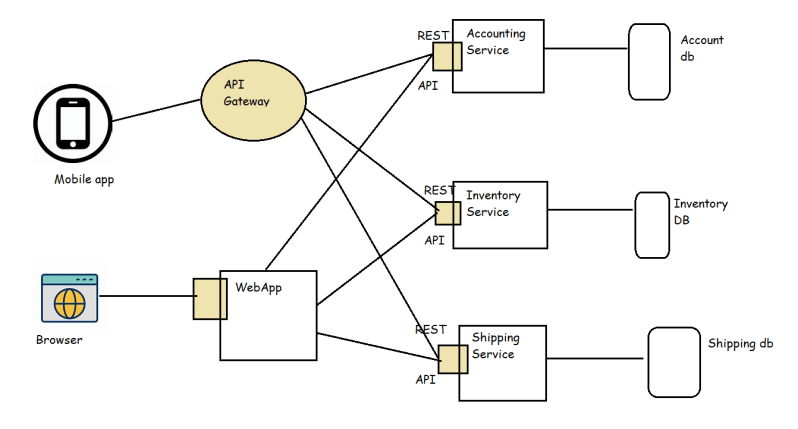
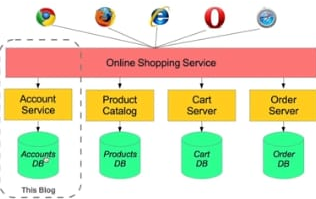
### Docker Compose

* This is a tool for defining a running multi container Docker application
* All the modern applications will have the some/all of the following components 
* When all of these components work together, we have a successful application
* Let’s understand microservices
* To bring up our application we need to run multiple containers in a specific order.
* Docker compose is one approach to bring up multiple containers of your application on a single docker host platform.
* Docker compose is widely used in Developer environments
* Docker stack is one more approach which helps to run multiple containers of your application on multi nodes.

#### Microservices

* This is an architectural pattern that structures and application as a collection of services that are
  + Highly maintable and testable
  + Loosely coupled
  + Independently deployable
  + Organized around business capabilities
* Simple microservice architecture 
* Now we can deploy each service in a container/vm Now this gives us the following benifits
  + We can have parallel developments for each service
  + Reusability of services can be done across applications
  + We can scale the individual services when there is load.
* The best way to run individual services is in containers and to manage the whole application deploymets we use orchestration platforms such as
  + Kubernetes
  + Docker Swarm
  + OpenShift

Docker compose: Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.



Install Compose on Linux systems

On Linux, you can download the Docker Compose binary from the [Compose repository release page on GitHub](https://github.com/docker/compose/releases). Follow the instructions from the link, which involve running the curl command in your terminal to download the binaries. These step-by-step instructions are also included below.

For alpine, the following dependency packages are needed: py-pip, python3-dev, libffi-dev, openssl-dev, gcc, libc-dev, rust, cargo and make.

Step1: Run this command to download the current stable release of Docker Compose:

**sudo curl -L "https://github.com/docker/compose/releases/download/1.28.5/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose**

Note:To install a different version of Compose, substitute 1.28.5 with the version of Compose you want to use.

If you have problems installing with curl, see [Alternative Install Options](https://docs.docker.com/compose/install/#alternative-install-options) tab above.Apply executable permissions to the binary:

**sudo chmod +x /usr/local/bin/docker-compose**

Note: If the command docker-compose fails after installation, check your path. You can also create a symbolic link to /usr/bin or any other directory in your path.

**sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose**

$ docker-compose --version

docker-compose version 1.28.5, build 1110ad01

be understandable even if you’re not familiar with it.

## Prerequisites

Make sure you have already installed both [Docker Engine](https://docs.docker.com/get-docker/) and [Docker Compose](https://docs.docker.com/compose/install/). You don’t need to install Python or Redis, as both are provided by Docker images.

## Step 1: Setup

Define the application dependencies.

Task1 -Create a directory for the project:

mkdir composetest

cd composetest

Create a file called app.py in your project directory and paste this in:

import time

import redis

from flask import Flask

app = Flask(\_\_name\_\_)

cache = redis.Redis(host='redis', port=6379)

def get\_hit\_count():

retries = 5

while True:

try:

return cache.incr('hits')

except redis.exceptions.ConnectionError as exc:

if retries == 0:

raise exc

retries -= 1

time.sleep(0.5)

@app.route('/')

def hello():

count = get\_hit\_count()

return 'Hello World! I have been seen {} times.\n'.format(count)

TASK-2: Create another file called requirements.txt in your project directory and paste this in: flask redis

## TASK-3: Create a Dockerfile

FROM python:3.7-alpine

WORKDIR /code

ENV FLASK\_APP=app.py

ENV FLASK\_RUN\_HOST=0.0.0.0

RUN apk add --no-cache gcc musl-dev linux-headers

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

EXPOSE 5000

COPY . .

CMD ["flask", "run"]

This tells Docker to:

* Build an image starting with the Python 3.7 image.
* Set the working directory to /code.
* Set environment variables used by the flask command.
* Install gcc and other dependencies
* Copy requirements.txt and install the Python dependencies.
* Add metadata to the image to describe that the container is listening on port 5000
* Copy the current directory . in the project to the workdir . in the image.
* Set the default command for the container to flask run.

For more information on how to write Dockerfiles, see the [Docker user guide](https://docs.docker.com/develop/) and the [Dockerfile reference](https://docs.docker.com/engine/reference/builder/).

## TASK 4: Define services in a Compose file

Create a file called docker-compose.yml in your project directory and paste the following:

version: "3.9"

services:

web:

build: .

ports:

- "5000:5000"

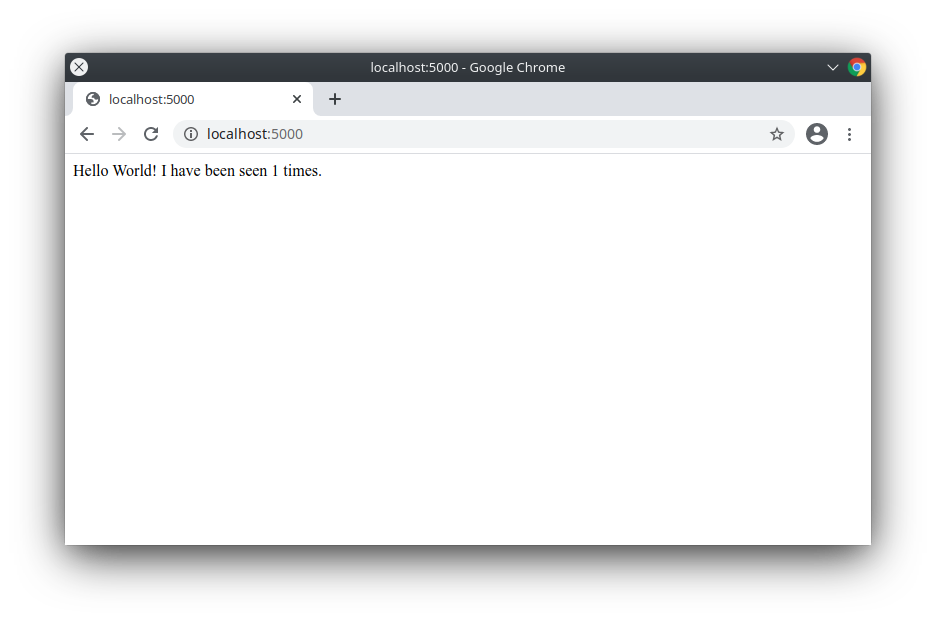
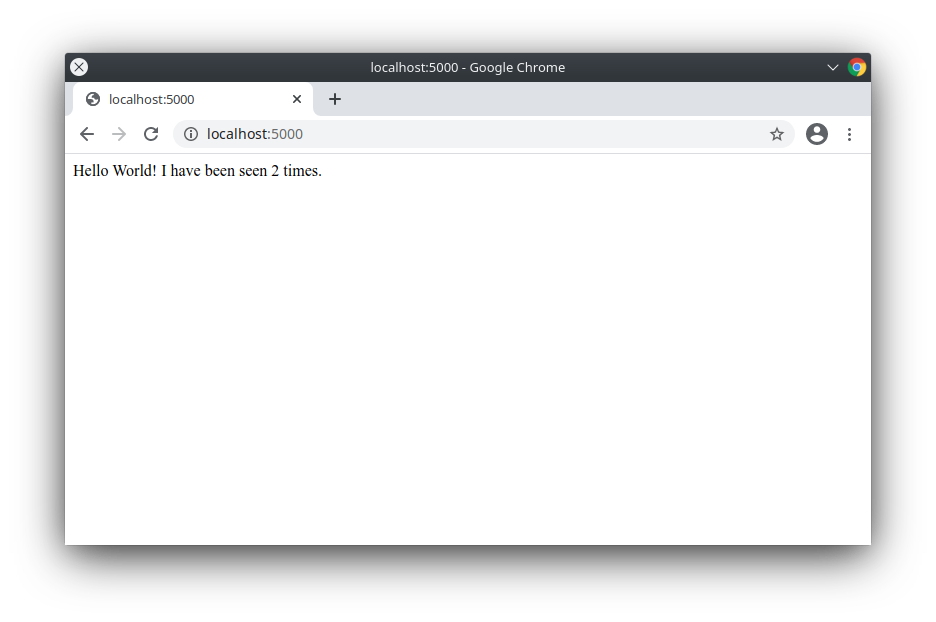
redis:

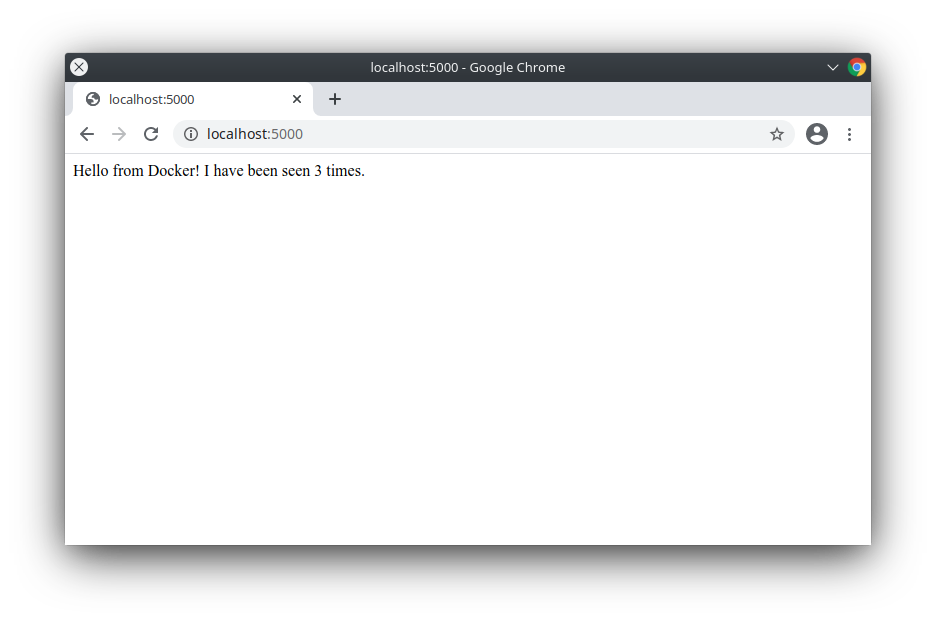
image: "redis:alpine"

## TASK5: Build and run your app with Compose

docker-compose up

Verify by using http:host-name:5000

Refresh the page. The number should increment.



## TASK6: Experiment with some other commands

docker-compose up -d

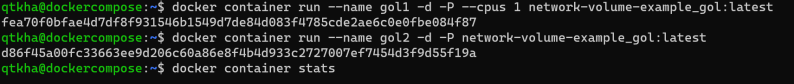
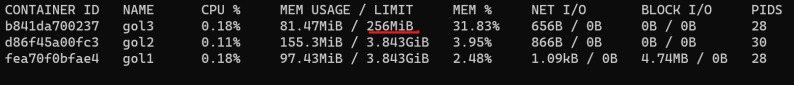
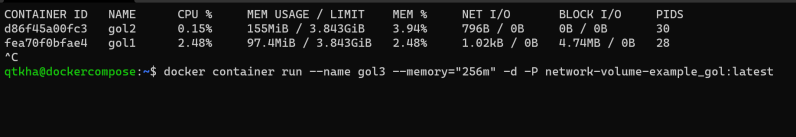
docker-compose ps

docker-compose stop

#### Run container in Production Environments

* We generally run Production environment on multiple servers
* So if we run our application which is microservices in the production environment, We have to deal with
  + Multiple containers
  + Ensure that the containers are running
  + We need to scale the containers depending on demand
  + We need to persist the volumes
* A distributed systems cluster is a group of machines that are virtually or geographically separted and they work together to provide the same service or applications to clients
* A distributed system can help
  + There is no single point of failure (SPOF)
  + This design supports scalability
  + The data is shared b/w all machines
  + The design is readable
* Consensus is the task of getting all the systems ina group agree on some specific values based on votes. Docker swarm uses raft consensus algorithm [Refer Here](https://raft.github.io/)
* The configuration or state of the system will be stored in the database on all managers. In the case of docker swarm this database is etcd
* We need to manage networks for multiple nodes and we already know about overlay
* Docker Swarm supports loadbalancing to interact with services externally as well as internally
* How can we deploy our application i.e. new microservice version in container with out downtime?
* Docker has a swarm mode which is running docker containers on multiple nodes with cluster features
  + Manager & Worker nodes
  + Multi node networking
  + Load balancing
  + Deployments
  + Scalability
* To be very frank industry has majorly adopted kubernetes as the orchestration technology for running contaniers in production.
* To run multiple applications on docker swarm which is
  + Service => Task
  + Docker has developed a docker stacks or you can use docker command line

### Running Docker Containers

* With CPU Limits 
* With memory limits 

For limiting cpu, ram for docker container (Link: https://docs.docker.com/config/containers/resource\_constraints/)

* Docker container allows us to change few configurations while the container is running 