Continuous Integration and Deployment

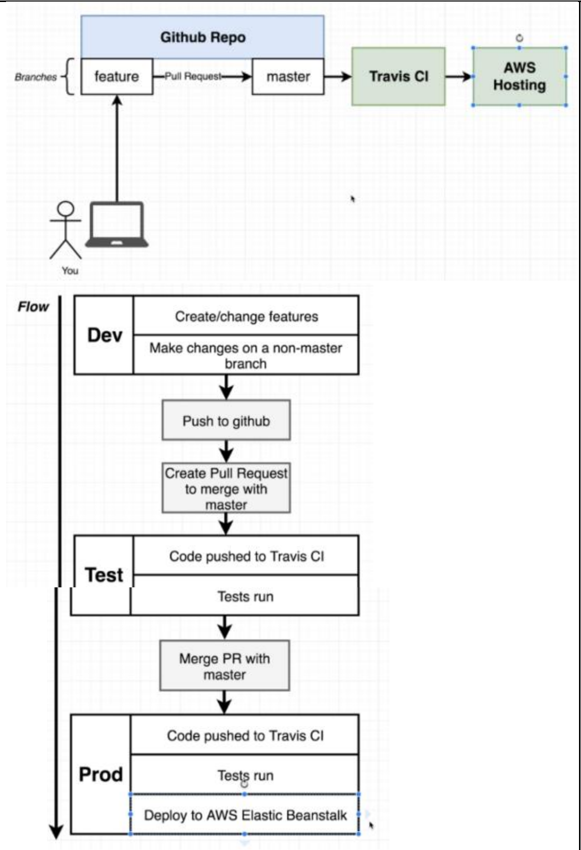
Gitops

https://www.gitops.tech/

Jenkis, Circle, Travis,

One container application

Development and production containers



It usually makes sense to create two containers for a project. One is created specifically for the local development environment and one for production.

For example in a node app you typically have three different steps

- npm run start - starts a development server
- npm run test - runs tests associated with project
- npm run build - builds a production version of the application

The production web service container's startup command would be the npm run build. It will contain the source code and this command will build it so that you have a running web service.

The development web service would have the npm run start as its startup command that runs locally a development web server. You can also have a second development container the startup command of which is the npm run test or you can just have the web service container and execute the npm run test process inside it. Its a matter of choice.

For running two dev containers you can use docker compose. The downside is that you can't have your terminal attached to the npm run test process to give interactive commands to it. The reason is that the primary command of the container is the npm. Then this npm process starts another process, the tests process. So you can attach your terminal to the containers primary command, the npm, but you can't pass the interactive commands to another process inside the container.

For the development process it makes sense to use docker volumes which are actually shared folders between your local machine and the container, so that you can update your source code without re building the development containers.

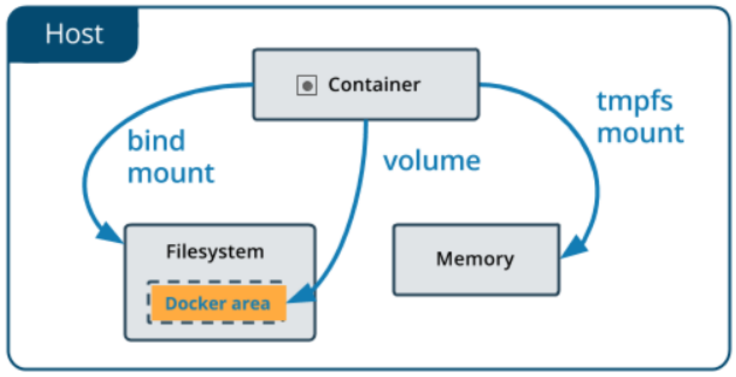
You can have a Dockerfile.dev file responsible for building your development container. You can then choose that specific Dockerfile with the command

```
> docker build -f Dockerfile.dev .
```

Managing data in docker

By default all files created inside a container are stored on a writable container layer. This means that:

- The data doesn't persist when that container no longer exists, and it can be difficult to get the data out of the container if another process needs it.
- A container's writable layer is tightly coupled to the host machine where the container is running. You can't easily move the data somewhere else.
- Writing into a container's writable layer requires a storage driver to manage the filesystem. The storage driver provides a union filesystem, using the Linux kernel. This extra abstraction reduces performance as compared to using data volumes, which write directly to the host filesystem.



Docker has two options for containers to store files on the host machine, so that the files are persisted even after the container stops: **volumes**, and **bind mounts.** Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

Tmpfs_mount is for storing the data in host's memory.

The VOLUME command will specify a mount point in the container. This mount point will be mapped to a location on the host that is either specified when the container is created or if not specified chosen automatically from a directory created in */var/lib/docker/volumes* (these are called anonymous volumes*)*. If the directory chosen as the mount point contains any files then these files will be copied into this volume. Data in the volume, persists because it is stored on the host machine.

> docker container run -v my-volume:/data image_tag

In this example, /data is the mount point in the container and my-volume is the volume on the local host. If my-volume does not exist when this command is run then it is created on the local host.

> dock container run -v /my-folder-on-container/ the first field is omitted. This is a mount point to an anonymous volume.

notice that you have to externally delete anonymous volumes if you want to clear your host machine.

Remove volumes

A Docker data volume persists after a container is deleted. There are two types of volumes to consider:

- Named volumes have a specific source from outside the container, for example awesome:/bar.
- Anonymous volumes have no specific source so when the container is deleted, instruct the Docker Engine daemon to remove them.

Docker volumes

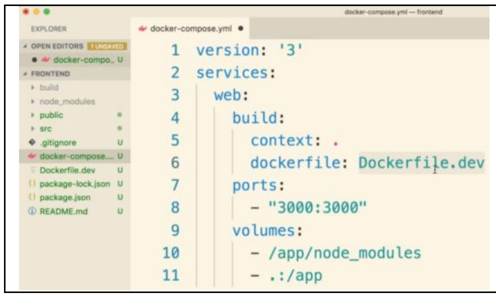
You can create references of files and folders from inside the container to files and folders that exist outside of it in your local machine. Any time the container tries to access something in the mapped directory is going to reach out of the container to the reference directory. This is used to map the source code files so that you don't have to rerun the container every time you make a change to the source code. Docker volumes are created either by the CLI using the -v flag in the run command directly or they can be defined in the docker-compose.yml file.

With the CLI

> docker run -v /app/unref_folder/ -v $(pwd):/app <container_id>

With this command we set up two volume mounts for the container, one for each argument of the -v flag. the : separates the local folder from the container folder. So we map local_path:container_path. The first argument to the -v flag which doesn't use : means don't map this folder which exists in the container folder all the rest contents of which are mapped to outside the container

With the docker-compose file



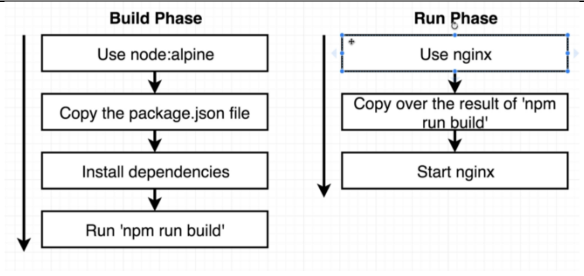
```
version: '3'
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile.dev
    ports:
      - "3000:3000"
    volumes:
      - /app/node_modules
      - .:/app
```

The working directory referred to with a dot, is the directory within which the docker-compose.yml file is, since all commands written in it use paths relative to the docker-compose file.

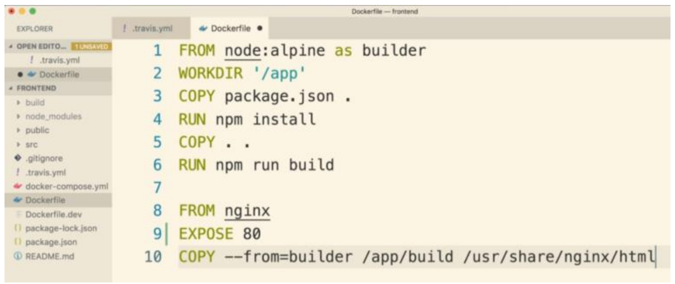
Important

To mount local Windows folders as Docker volumes, those folders first need to be shared and mounted on the VM that is running Docker. By default, C:\Users is shared in VirtualBox, so mounting volumes from that location will work without any configuration.

Multi step build



This Dockerfile can exist in the same directory with the Dockerfile.dev file.



```
FROM node:alpine as builder
WORKDIR '/app'
COPY package.json .
RUN npm install
COPY . .
RUN npm run build

FROM nginx
EXPOSE 80
COPY --from=builder /app/build /usr/share/nginx/html
```

In the production container we need to install a proper web server like nginx instead of the development server. We can manually install it as a dependency of our project. But the thing is that there is a public nginx image that we can use instead. The problem though is that in order to build the node app we use another base image (node:alpine) and we can't use a second one. The solution is a two-step build process.

In the first step we build the production files (in this case an index.html and an app.js). These files created by the build process are the only files that we actually need in the production container and a web server to serve them. So there is a second step in which we use the nginx base image and we just copy these files from the first step. The production container is the result of this second step.

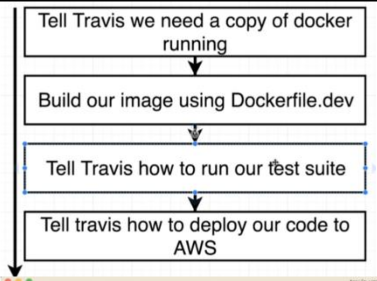
Every file of the first step will be ignored, will not exist in the resulting container of the second step. All that happens in the first phase is temporary. Each step begins with a FROM instruction. With the COPY instruction you explicitly declare which files from the first temporary container you want to use in this container. The --from=builder defines from which container to copy. There is no need to explicitly define a primary command for the ngnix container since it is done automatically by this base image. The destination folder is the folder which nginx uses to serve static files from, so we put them there.

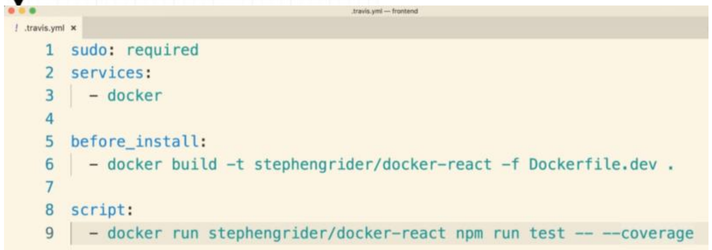
Travis CI

Travis CI is a web service to which our commits are automatically pushed to and trigger some jobs. These jobs can be: running some user defined tests on the code or automatically deploy it to a hosting provider. Typically Travis is used for automated testing or deployment or both.

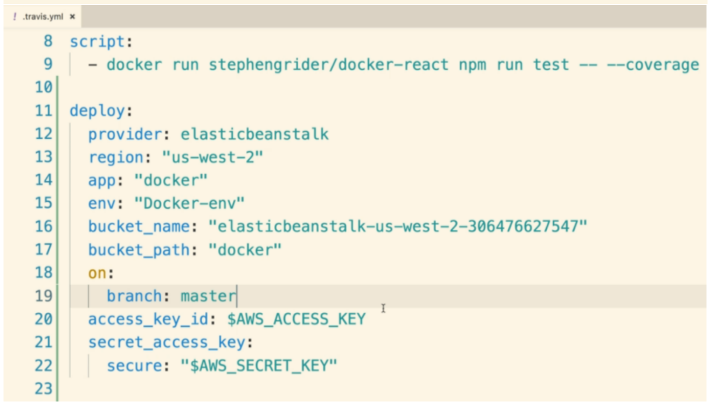
First you have to setup Travis to watch a repository. Once you do that every time you push to that repo travis will pull that code and execute some actions that you define in a .travis.yml file. You place this file in the same folder with your project where it is tracked by git. When you push your commit to github Travis will pull the code and read this file so will know what to do.

.travis.yml file


Tell Travis we need a copy of docker running
Build our image using Dockerfile.dev
Tell Travis how to run our test suite
Tell travis how to deploy our code to AWS



```yaml
sudo: required
services:
  - docker

before_install:
  - docker build -t stephengrider/docker-react -f Dockerfile.dev .

script:
  - docker run stephengrider/docker-react npm run test -- --coverage
```



```yaml
script:
  - docker run stephengrider/docker-react npm run test -- --coverage

deploy:
  provider: elasticbeanstalk
  region: "us-west-2"
  app: "docker"
  env: "Docker-env"
  bucket_name: "elasticbeanstalk-us-west-2-306476627547"
  bucket_path: "docker"
  on:
    branch: master
  access_key_id: $AWS_ACCESS_KEY
  secret_access_key:
    secure: "$AWS_SECRET_KEY"
```

This is what this travis file contains.

sudo: required (the actions we make with docker require superuser permissions)

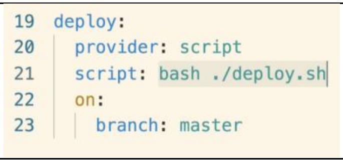
Install docker

If a test returns a status code anything other than zero then it is considered a fail by Travis.
-- --coverage is an npm run test flag which makes the test run to not hold the terminal (for interactive use) but return 0. This is necessary since Travis will wait for the test to return.

You don't seem to define a particular branch for the test process (only for the deploy process). This means that if you have a feature branch even if you push to this branch and not the master, the tests will still run in the new code. <u>Actually Travis fakes a merge to the master and runs the tests</u>.
<u>Notice that since there is integration between Travis and Github if you make a pull request from feature to master in the pull request page you will be able to see the results of the Travis tests on the feature branch</u>.

Travis will get your code, zip it in one file and upload it to an S3 bucket. Once that is done it will say to elastic beanstalk to deploy an application from this file on that bucket. EB will download the file and will build the image and run the container.

Notice that you don't have to write a script on how Travis will deploy your code to EB, because Travis has build in support for EB. You just define provider: elasticbeanstalk.



```yaml
deploy:
  provider: script
  script: bash ./deploy.sh
  on:
    branch: master
```

There is no such option for deploying to kubernetes though (in 2018) so you have to write the deployment script your self. You do this by defining provider: script and then defining the script file.

Any time you make push commits to your connected github repository, travis will run the tests but will only deploy if there is a push to the master branch (as the on instruction dictates).

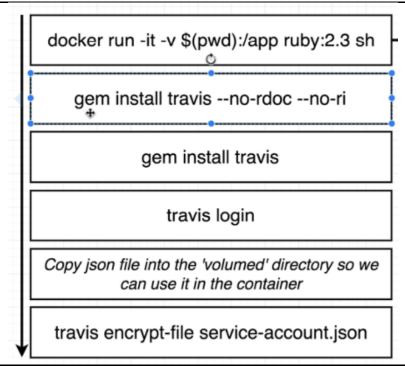
Handling credentials

Whenever you want to enable Travis to programmatically manage a cloud provider service you need to create a user in that service in the provider, take the keys for this user and add them to Travis. If the key is just a plain text string you can set it up as an environment variable in Travis. If it is a whole file then you have to encrypt locally, upload it to travis and have it decrypt it when it wants to access the provider's service. The encryption and uploading are done by a Travis CLI that you install locally on your machine.

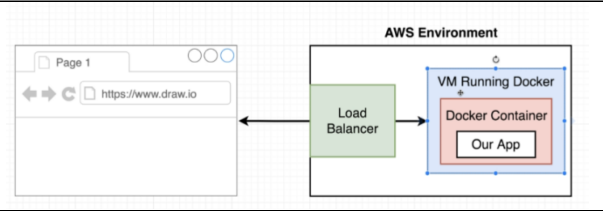
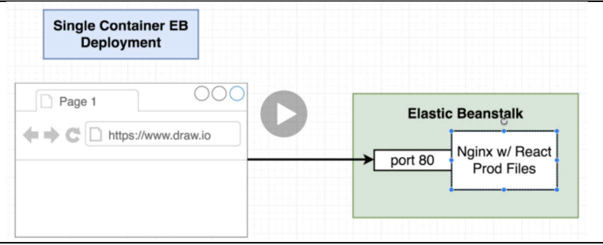
Travis specific environment variables

In aws IAM we create a user for Travis and give him permissions to deploy to ELB. You create a key for programmatic access to AWS. Notice that you don't write this key inside the travis file since it is part of your repo and will be visible to others. You can set it up as an Environment variable in Travis. You can do this through the Travis GUI. You can also define env variables in the .travis.yml file.

Travis CLI

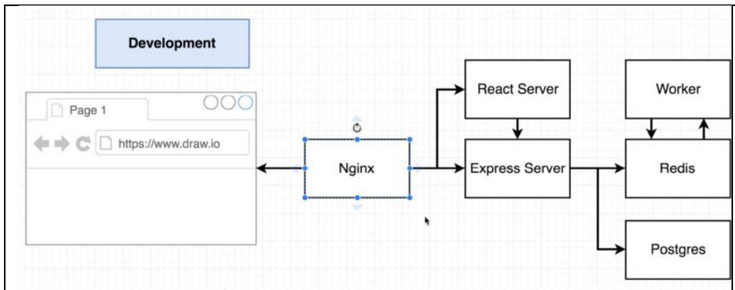
 `docker run -it -v $(pwd):/app ruby:2.3 sh`
`gem install travis --no-rdoc --no-ri`
`gem install travis`
`travis login`
Copy json file into the 'volumed' directory so we can use it in the container
`travis encrypt-file service-account.json`
	Travis CLI requires ruby to run. Since it might be cumbersome to install ruby on windows you can just create a docker container from a public ruby image and do the encryption and uploading from the container. This is another demonstration of the usefulness of containers. Notice that if you have docker, all you need to run a docker ruby image, create a docker volume for the container and run a terminal on it is one command.

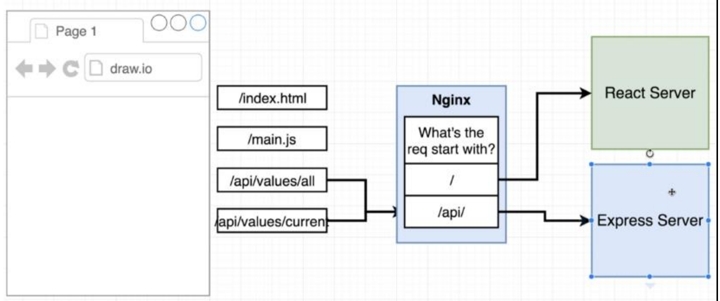
AWS EB

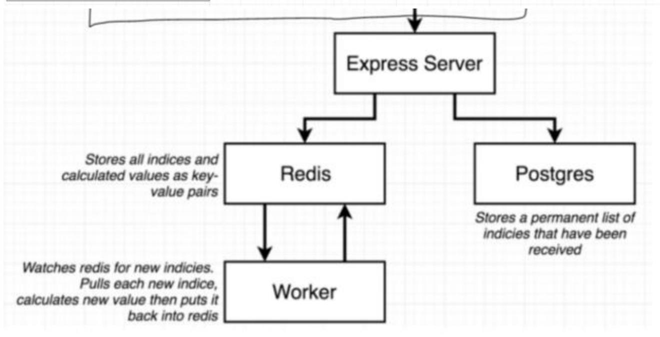
 AWS Environment Page 1 https://www.draw.io Load Balancer VM Running Docker Docker Container Our App	Before writing the deploy part of the travis file you should create a Beanstalk app in AWS. It will automatically create the infrastructure for you. This is the infrastructure that Elastic Beanstalk will set up. Notice that if there is a lot of traffic ELB can automatically scale out and spin up new vms with your container.
 Single Container EB Deployment Page 1 https://www.draw.io Elastic Beanstalk port 80 Nginx w/ React Prod Files	The application assets are served by an nginx server which is part of the container and its sole purpose is to serve those static files. Notice that when we have more than one containers we would have another nginx server that is responsible for routing

Multi container application

- ❖ Development



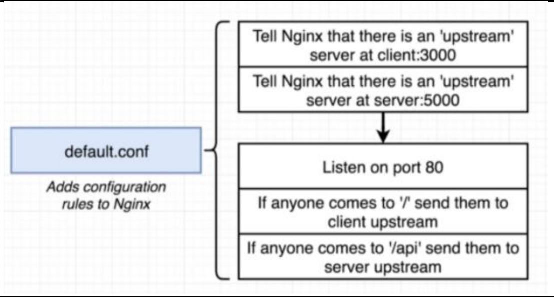



Express Server
Stores all indices and calculated values as key-value pairs
Redis
Postgres
Stores a permanent list of indicies that have been received
Watches redis for new indicies. Pulls each new indice, calculates new value then puts it back into redis
Worker

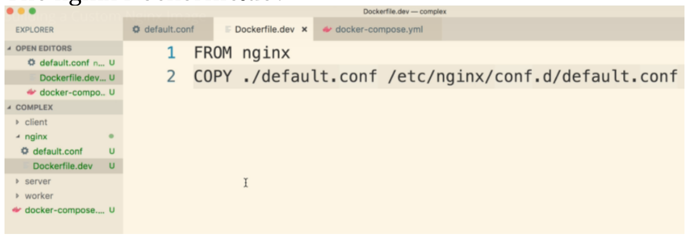
We have five containers defined in the services section in the docker-compose file:

- Postgres (uses a public image)
- Redis (public image)
- Nginx (public image custom config) routes traffic to react server or express server.
- React server (build) He names it client. It is the development react server, the server that serves the static react application code during development (index.html and index.js). in production this would be an nginx server different from the one that does the routing.
- Server (build from Dockerfile) this is the express server that serves the api requests coming form the react application.
- Worker (build)

Nginx configuration is described in a default.conf file.

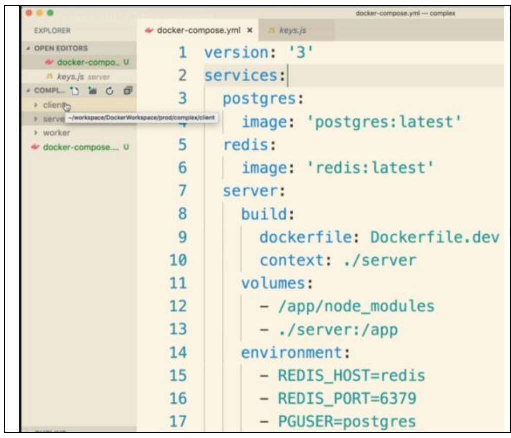

default.conf
Adds configuration rules to Nginx
Tell Nginx that there is an 'upstream' server at client:3000
Tell Nginx that there is an 'upstream' server at server:5000
Listen on port 80
If anyone comes to '/' send them to client upstream
If anyone comes to '/api' send them to server upstream

The nginx Dockerfile.dev



```
FROM nginx
COPY ./default.conf /etc/nginx/conf.d/default.conf
```

Environment Variables on containers



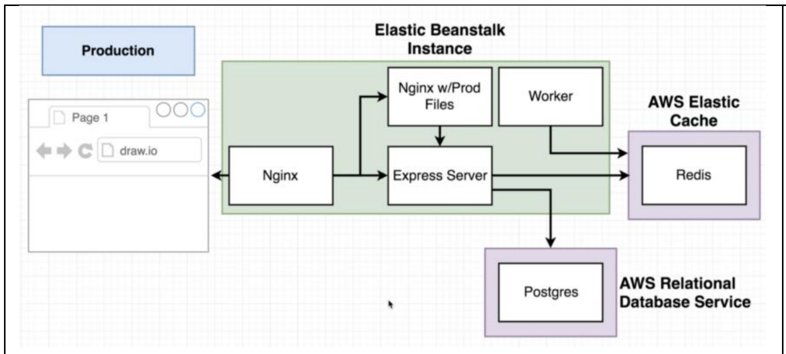
```
version: '3'
services:
  postgres:
    image: 'postgres:latest'
  redis:
    image: 'redis:latest'
  server:
    build:
      dockerfile: Dockerfile.dev
      context: ./server
    volumes:
      - /app/node_modules
      - ./server:/app
    environment:
      - REDIS_HOST=redis
      - REDIS_PORT=6379
      - PGUSER=postgres
```

Our project will most definitely use some environment variables to read some secret keys and passwords etc. These environment variables are defined by us in the container, usually through the docker-compose file. The environment variables defined in the docker compose file are applied on runtime, which means after the build phase. When the container is created the env vars are created in it.

Since the env vars are container specific they are defined within the specific service in the docker-compose file. When you want to refer to another container (docker compose service) you refer to it by its name, so redis_host=redis where redis is the name of the redis container.

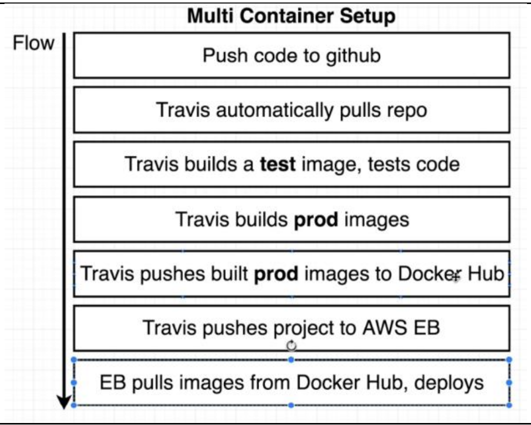
❖ Production

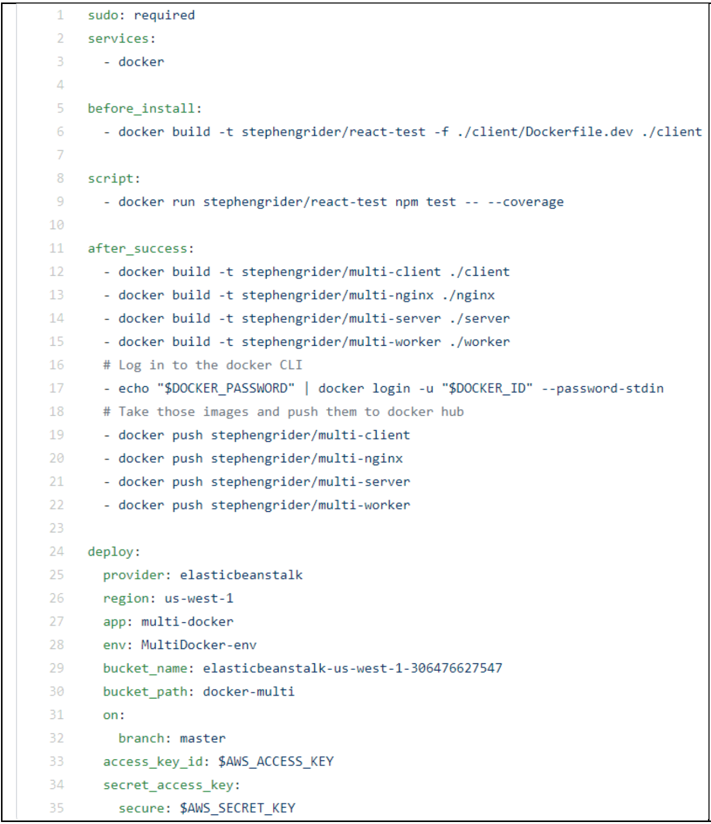
The architecture

 Production Elastic Beanstalk Instance Page 1 draw.io Nginx Nginx w/Prod Files Worker Express Server AWS Elastic Cache Redis Postgres AWS Relational Database Service	Notice that in production we don't use our own containers for Redis and Postgresql. We use the managed AWS services instead which offer a lot of nice features. The other containers belong to an EB instance which has its own security groups etc. Created automatically by EB. We have to manually configure though the communication between the EB instance and the external database services. We do this by using security groups. All these services are in the same VPC.

You can set up the env variables for your EB instance from AWS gui. They are added to all containers of the EB instance.

The deployment process

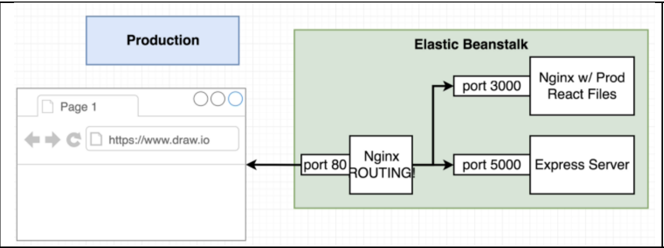
 Multi Container Setup Flow Push code to github Travis automatically pulls repo Travis builds a test image, tests code Travis builds prod images Travis pushes built prod images to Docker Hub Travis pushes project to AWS EB EB pulls images from Docker Hub, deploys	In the previous example with one container Elastic Beanstalk (EB) was building the image from a Dockerfile which is not ideal. It is better to have EB use an already build image and just run it. Travis will build the production images from our Dockerfiles. Another difference is that we upload the production images to our account in Docker Hub. AWS is integrated with docker hub so it is very easy to pull from there and run them. Travis zips the whole repo and pushes it to an S3 bucket, from which EB gets it, but actually EB only needs the EB configuration file (Dockerrun.aws.json) which contains all the necessary information about which images to download from the docker hub etc.



```yaml
sudo: required
services:
  - docker

before_install:
  - docker build -t stephengrider/react-test -f ./client/Dockerfile.dev ./client

script:
  - docker run stephengrider/react-test npm test -- --coverage

after_success:
  - docker build -t stephengrider/multi-client ./client
  - docker build -t stephengrider/multi-nginx ./nginx
  - docker build -t stephengrider/multi-server ./server
  - docker build -t stephengrider/multi-worker ./worker
  # Log in to the docker CLI
  - echo "$DOCKER_PASSWORD" | docker login -u "$DOCKER_ID" --password-stdin
  # Take those images and push them to docker hub
  - docker push stephengrider/multi-client
  - docker push stephengrider/multi-nginx
  - docker push stephengrider/multi-server
  - docker push stephengrider/multi-worker

deploy:
  provider: elasticbeanstalk
  region: us-west-1
  app: multi-docker
  env: MultiDocker-env
  bucket_name: elasticbeanstalk-us-west-1-306476627547
  bucket_path: docker-multi
  on:
    branch: master
  access_key_id: $AWS_ACCESS_KEY
  secret_access_key:
    secure: $AWS_SECRET_KEY
```

The .travis.yml production file

You must add AWS access keys and Docker Hub keys as env variables in Travis so that Travis has programmatic access to these services.



We could use a single nginx container to do both the routing and the static assets serving but having two separate services is better in terms of scalability.

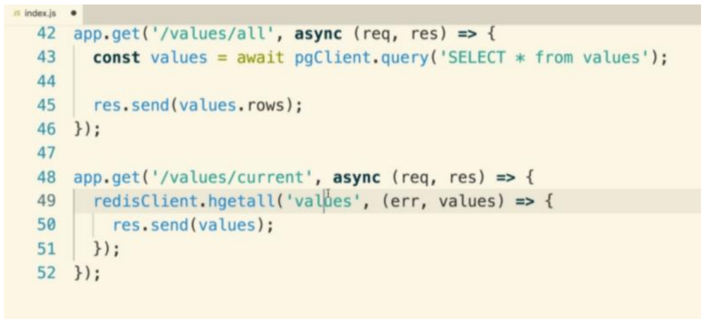
Dockerrun.aws.json file

In the previous example where we had only one container with one Dockerfile, when we pushed that code to EB it automatically built and run that container. In the case in which our repo has many containers we must do some configuration to EB so that it knows what to do with our repo (when it is pushed in EB by Travis). this configuration is described in a Dockerrun.aws.json file. It is the equivalent of docker-compose but for AWS EB deployment. EB in the background builds the containers using another AWS service called elastic container service (ECS). in that service containers are described as Task definitions. So in the dockerrun.aws file we define Task definitions.

Misc

If we don't explicitly define a CMD instruction inside a Dockerfile the base image's primary command will be used.

Promises vs callback


```
app.get('/values/all', async (req, res) => {
  const values = await pgClient.query('SELECT * from values');

  res.send(values.rows);
});

app.get('/values/current', async (req, res) => {
  redisClient.hgetall('values', (err, values) => {
    res.send(values);
  });
});
```

Misc

Spinnaker (from Netflix) is an open source, multi-cloud continuous delivery platform for releasing software changes with high velocity and confidence. Spinnaker is an open source software used for multi-cloud continuous delivery platform for releasing software changes with high velocity and confidence. Spinnaker is tested by hundreds of teams over millions of deployments and it has proved so much successful that it is supported almost all the major cloud platforms such as AWS, Google cloud, Oracle cloud, Kubernetes, Microsoft Azure and Cloud Foundry. It is one of the successful open source deployment tool out there.

Kubernetes

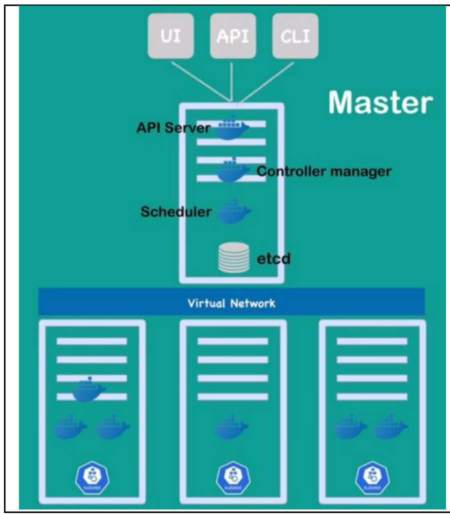
Intro

https://www.youtube.com/watch?v=aSrqRSk43lY 10 min intro

Kubernetes is a container orchestration tool, a tool for managing containers. Such tools offer high availability, scalability and disaster recovery (backup and restore). Kubernetes gives you a means to do and manage deployments, an easy way to scale your deployments and also gives you monitoring capabilities. A deployment is constantly managed, so if a container fails there is an auto heal process which can spin a new one up. For scaling you don't have to create a new vm to put your new container into. Kubernetes can automatically decide where to put it in the existing cluster optimizing resource usage according to the configuration of the kubernetes scheduler. Then there is the services functionality which defines how a service with many running containers can be accessed (using a load balancer for example). It offers a lot of benefits like automated health checks (if a container is down it will do whatever it can to spin a new one up) and Rolling restarts and deployments (when you are deploying a new service there is a copy of the previous version that serves until the new version is deployed), there are extensions that automatically manage the SSL encryption etc. so its not only useful for large scale applications but for smaller ones too.

Kubernetes cluster architecture

A Kubernetes cluster consists of a set of nodes, which all run containerized applications. Of those, a small number are running applications that manage the cluster. They are referred to as master nodes, also collectively known as the **control plane**. Kubernetes cluster consists of many nodes. Each node is a virtual or physical machine. Each cluster has at least one master node. All the other nodes are the worker nodes.



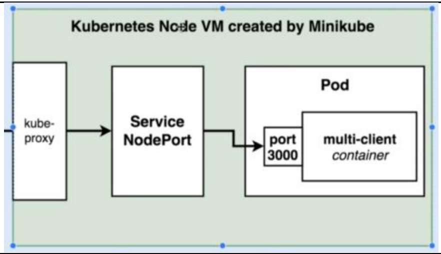
Master node components (services)
API Server which is the entry point to the cluster. Any ui, api or cli speak to this process. Another one is the
Controller manager which keeps track of what is happening in the cluster whether a container died and needs to be restarted etc. Another one is the
Scheduler which manages the deployment of the containers based on workload and the available cluster resources. It decides on which worker node the next container will be scheduled to based on containers workload and the available worker nodes resources.
Another one is the **etcd** key value store which holds the current state of the entire kubernetes cluster, the configuration and status of every node and every container inside the nodes. Backup and restore is achieved with the etcd snapshots from which you can recover the entire clusters state. (etcd is a distributed key value store that lets you store and share data across a distributed cluster of machines. Kubernetes uses etcd to store data about your cluster and share it across the Kubernetes control plane).

Another important component of the cluster is the **Virtual Network** that spans all the cluster nodes essentially making the cluster a combined unified machine with all the resources of the individual nodes combined. Worker nodes usually are more powerful than the master node. Usually there are more than one master nodes in a cluster for redundancy.

Worker node components

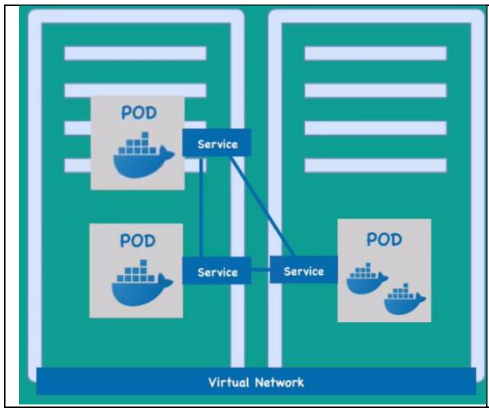
Kubelet. Each worker node runs a kubernetes process called Kubelet. Kubelet is a process that enables the communication between the nodes of the cluster and also can be used to execute other processes in the node like running an application process.

Kube-proxy

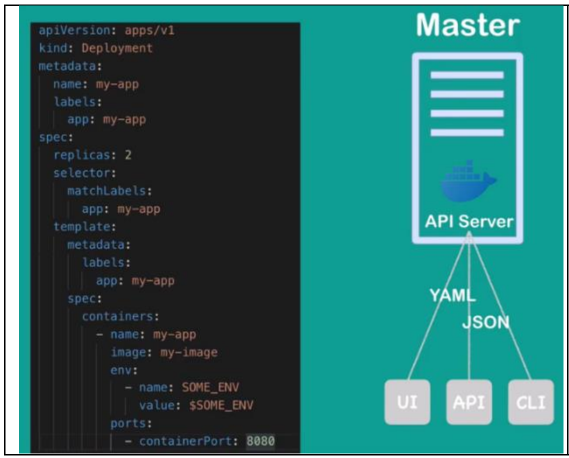


Every node has a **kube-proxy** process that is the gateway of the node, an interface between the node and the outside world. This process will parse every request coming to the node and will route it to the appropriate service of the node. You might have many services in a single node.

Docker is installed in every node



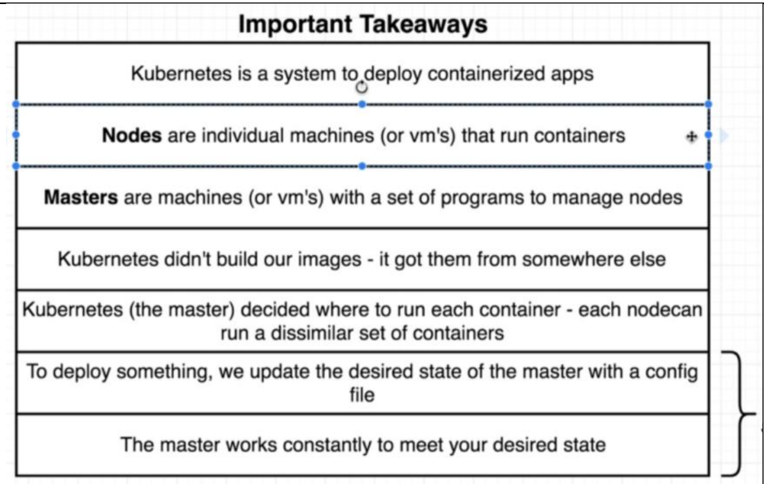
The smallest basic unit in kubernetes is the Pods. A **pod** is an abstraction layer over a container, it wraps up and manages a container so if the container dies it will spin up another one. A Node can have multiple pods. Kubernetes interacts with the Pods not with the containers themselves. A pod can contain more than one containers but usually you have one container per pod. Each pod is assigned its own internal IP in the kubernetes virtual network. But pods are ephemeral structures. They can die and restarted again with a new IP. So it wouldn't be very convenient to have to track all these changes manually. This is where the services component of kubernetes come into play. A **service** is attached to a pod and sits in front of it and is always on. A service offers two main benefits, a static IP address and load balancing functionality. So even if a pod dies and is restarted, the service with its static IP address is still there always available.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  labels:
    app: my-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: my-image
          env:
            - name: SOME_ENV
              value: $SOME_ENV
          ports:
            - containerPort: 8080
```

Any interaction with a kubernetes cluster is achieved through the API server process of the master node. A common cli tool for this job is **kubectl**. The configuration messages the API server accepts, are either in yaml or in json format. Configuration is a declarative process described in a yaml file. This yaml file configures a Deployment which is a kubernetes component. A **deployment** is actually a template for creating pods. The configuration defines a "should be" state for the cluster. If at any point in time the actual state of the cluster, the "is" state, doesn't match the should state, the controller process of the master node identifies it and tries to modify the is state accordingly.

Important takeaways



Important Takeaways

- Kubernetes is a system to deploy containerized apps
- **Nodes** are individual machines (or vm's) that run containers
- **Masters** are machines (or vm's) with a set of programs to manage nodes
- Kubernetes didn't build our images - it got them from somewhere else
- Kubernetes (the master) decided where to run each container - each nodecan run a dissimilar set of containers
- To deploy something, we update the desired state of the master with a config file
- The master works constantly to meet your desired state

Each node has docker installed. When you define an image in the configuration these nodes can connect to the docker hub and pull the defined image. Notice that a node doesn't create an image, it gets a ready to run image from a hub.

By default kubernetes will automatically decide which pods to spin up in which nodes. You can change that if you want.

Kubernetes offers a declarative style of deployment. You just have to declare what you want, what is the desired configuration and kubernetes will try to always follow these instructions. Notice that kubernetes allows you to use an imperative approach if you want (you check the current state and decide what to do)

If at any point you want to modify your deployment, you just have to update the desired state in a configuration file and send it to the master node of the kubernetes cluster through kubectl.

Dev and deploy in a nutshell

In general the creation of the cluster is independent from the creation of your services within the cluster. They are two different processes. If you create your cluster with terraform (for example an EKS in AWS), then you can have a distinct repo just for that. Each of your microservices is one distinct repo with its own dockerfiles and its kubernetes deployment file. When the cluster is ready, you can start deploying your services on your cluster by applying the deployment file of a service.

Ci/cd with Kubernetes

There is a Kubernetes cluster running. The ci/cd pipeline for the service you work on, would look something like this: You have a git repo for you service. Your repo contains your dockerfiles and your kubernetes deployment files (a production deployment with the production image and a test deployment with the test image). You push a commit to your repo. Jenkis builds the test image and uploads it an image hub. Then it applies the testing deployment file to your testing environment (being an individual cluster or a namespace within a cluster that has test and prod namespaces). New pods are created with the new test image. The tests run. If they are successful somehow Jenkins gets notified and builds the production image. It uploads it to a hub. Then it applies the production deployment to the production cluster. New pods are created with the new production image.

Gitops

https://www.gitops.tech/

It is a generic CI/CD solution, built with Kubernetes integration specifically in mind. It demands that your infrastructure is defined as code. You have one specific repo for your infrastructure code and one repo per service as usual. It ensures that the cluster is in the same state as your infrastructure repo. It offers a push approach (similar with the one I described above) and a pull approach where we install an observer in the cluster who watches the container registry and the environment (deployment) repo. When there are changes in the repo it updates the cluster to match the new configuration (and vice versa if the cluster changes for some reason it updates it to match the repo).

Local development

Hot reloading of new code? For development you can have a local cluster created with minikube for example. You make changes to your code (without committing yet) you

Multiple environments (test, stage, prod)

two options:

1. Use a K8s cluster for each environment
2. Use only one K8s cluster and keep them in different namespaces.

If your requirements are small, you can get away with using a single cluster. You can easily have one large monolithic cluster that handles all of the workloads for your organization.

(option 1) Seems the safest options since it minimizes the risks of potential human mistake and machine failures, that could put the production environment in danger. However, this comes with the cost of more master machines and also the cost of more infrastructure management.

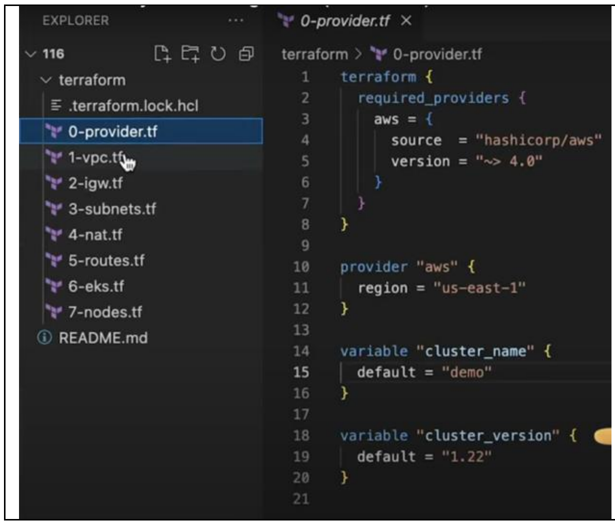
(option 2) Looks like it simplifies infrastructure and deployment management because there is one single cluster but it raises a few questions like: How does one make sure that a human mistake might impact the production environment? How does one make sure that a high load in the staging environment won't cause a loss of performance in the production environment?

Have in mind that you can have one branch per environment. When you push in the staging branch the deployment happens to the staging environment. If you push to the master, it happens on the production environment.

Have in mind qovery (https://www.qovery.com/) with which you can easily create a staging environment in AWS.

Create an EKS cluster and connect with it API gateway with terraform

https://www.youtube.com/watch?v=4cuI4RIq4Hs a nice 5 minutes video



```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

variable "cluster_name" {
  default = "demo"
}

variable "cluster_version" {
  default = "1.22"
}
```

With these 8 files, he configures an EKS cluster from scratch. VPC, internet gateway, subnets, routes and finally an eks cluster and some nodes for the cluster. When the cluster is up and running you can start deploying your services (applying their deployment files).

0. He
1. We create a cluster with the 4 subnets.
2. We create and assign a node to the cluster.

Then we can start deploying our services.

Kubernetes objects

A Kubernetes object is a "record of intent"--once you create the object, the Kubernetes system will constantly work to ensure that object exists.

Whenever you send a yaml configuration file to a kubernetes cluster (through kubectl) you actually describe some kubernetes objects. There are various types of objects each with its own purpose and functionality: Pod, Service, Deployment, StatefulSet, ReplicaController, componentStatus, configMap, Event, Namespace, Endpoints, Controller (A controller kubernetes object is any object that constantly tries to make a desired state a reality). Notice that you can define custom Kubernetes Objects. Many 3rd party kubernetes applications define custom objects that you can configure and use.

--- (separator)

Each object can have a distinct configuration file. Notice though, that we can combine the configuration of objects in one file separating the definitions with three dashes --- its a matter of preference.

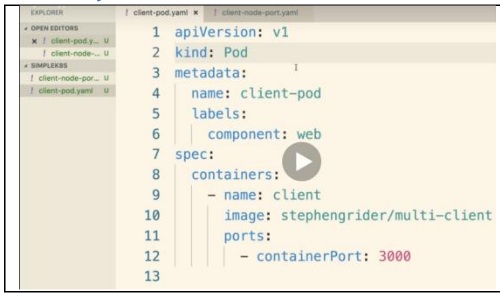
By creating an object, you're effectively telling the Kubernetes system what you want your cluster's workload to look like; this is your cluster's desired state. Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster.

To work with Kubernetes objects--whether to create, modify, or delete them--you'll need to use the Kubernetes API. When you use the kubectl command-line interface, for example, the CLI makes the necessary Kubernetes API calls for you. You can also use the Kubernetes API directly in your own programs using one of the Client Libraries.

spec (desired state) and status (is state)

Every object configuration usually uses at least these two fields. The spec defines the desired state and the status the is state. The status field is handled (updated) by kubernetes

Pod object

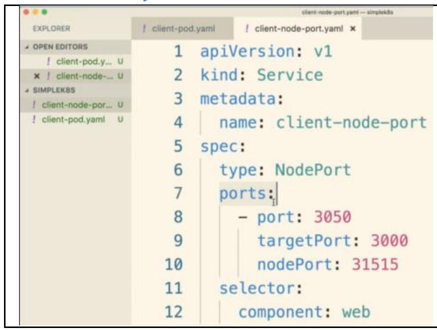
 ```yaml apiVersion: v1 kind: Pod metadata: name: client-pod labels: component: web spec: containers: - name: client image: stephengrider/multi-client ports: - containerPort: 3000 ```	A wrapper of container. It can have more than one containers although usually it has only one. spec: containers: name: We can refer to a container in a pod with its name as it is defined in its configuration file. containerPort: the port to which the container listens to.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: client-pod
  labels:
    component: web
spec:
  containers:
    - name: client
      image: stephengrider/multi-client
      ports:
        - containerPort: 3000
```

Update a Pod in place

The metadata.name and the kind can identify uniquely a specific object. When we update the config file with a new image for an existing object, kubernetes checks the kind and name to decide if it needs to spin up new pods or update the existing ones. So if you need to update some objects you have to keep their name the same otherwise it will just create new ones for the new names (and stop the previous ones). You can only update specific properties of an existing pod object. These limitations are overcome with the use of Deployment object all configurations of which can be updated.

Service object



```yaml
apiVersion: v1
kind: Service
metadata:
  name: client-node-port
spec:
  type: NodePort
  ports:
    - port: 3050
      targetPort: 3000
      nodePort: 31515
  selector:
    component: web
```

In kubernetes we have to explicitly configure all the networking stuff, ports etc. A service object can be one of four subtypes: ClusterIP, NodePort, LoadBalancer, Ingress.

The ports instruction. The <port> instruction defines the port that this service listens to for messages from other Pods of the cluster. The <nodePort> is the port that this service listens to for messages from sources other than cluster's nodes, for example from the browser. The <TargetPort> is to which port of the connected pod to forward all these incoming requests.

We define a selector of my_key:my_value (component:web in this example) for the service object. In the metadata instruction of the Pod object we define a label component: web. This links the two objects together. This Service will be connected to this Pod. The service looks for every pod of its node that match its selector.

Exposing Service objects outside the cluster

Kubernetes offers three well defined mechanisms for this. The two most basic ones. are NodePort and LoadBalancer, which cover up to L4 (TCP/UDP). For higher level functionality (TLS termination, HTTP path based routing, etc.) you can use an Ingress Controller.

- NodePort

A service object of NodePort type is used to expose a container to the outside world, usually only for development.

- ClusterIP

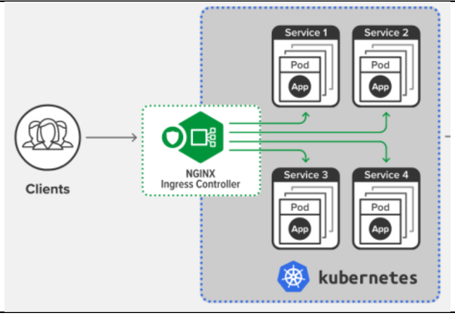
Exposes a set of pods to other objects in the cluster. You need to define a port and a targetPort. The host name for services running in Pods behind a ClusterIP service, is the name of the ClusterIP as it is defined in its configuration file.

- LoadBalancer

a way of getting traffic into a cluster. A load balancer service can be used instead of a clusterIP in front of a set of pods. But is only suitable for one set of pods not for multiple sets (same with the clusterIP). in addition to that a LB object automatically configures the load balancer service of the host provider on which the cluster is running, for example if it is in AWS it will configure an aws LB to route traffic to the LB service of the cluster.

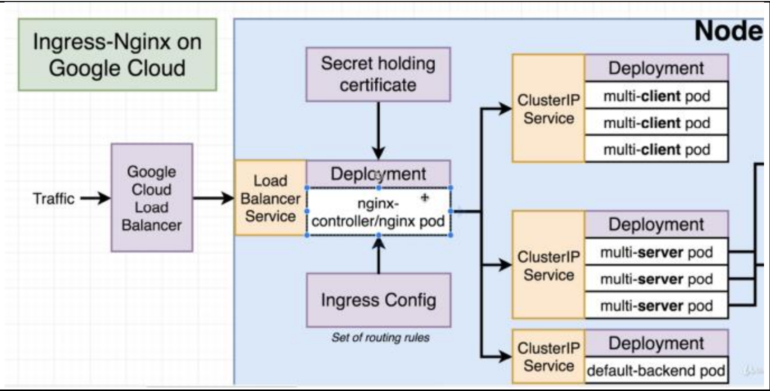
- Ingress (or Ingress Controller)

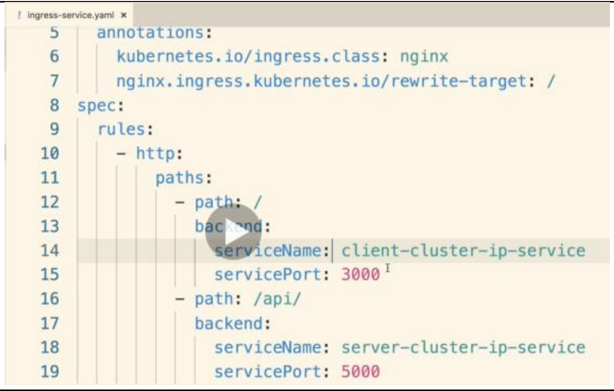
Exposes a set of service objects to the outside world. (On the contrary the cluster ip and load balancer expose a set of pods).



When we have a Service that needs to be exposed outside the cluster, we create an Ingress for it. there are popular public implementations for an Ingress, for example NginX Ingress Controller (Ingress-NGINX).

It creates an input (Ingress) to a cluster by configuring an nginx server. it creates a deployment with an nginx pod. The deployment contains both an nginx service and an nginx controller. A controller kubernetes object is any object that constantly tries to make a desired state a reality. In front of the deployment it creates a loadBalancer service and connects it to a provider specific load balancer. You create a configuration file that defines the Ingress and the ingress deployment tries to always match this desired state. You configure your routing rules. You use the ingress-nginx implementation instead of just an nginx pod manually created by you because it contains a lot of configuration details to properly work as an Ingress to a kubernetes cluster. For example when it routes traffic to a specific service it bypasses the ClusterIP service and routes the request directly to the Pod itself. This is useful for sticky sessions for example where you want some requests to be routed to specific Pods. The created LB listens to port 80 and 443 by default. (The default-backend-pod deployment contains a service responsible for cluster health checks. It can also be part of one of your other deployments). Notice that the easiest way to install Ingress-Nginx to your production environment is using Helm.





```
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
    - http:
        paths:
          - path: /
            backend:
              serviceName: client-cluster-ip-service
              servicePort: 3000
          - path: /api/
            backend:
              serviceName: server-cluster-ip-service
              servicePort: 5000
```

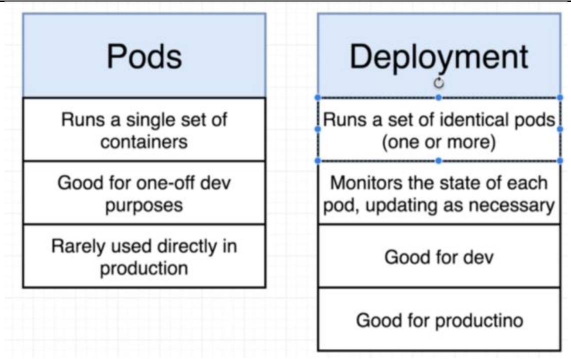
The secret holding the TLS certificate for HTTPS is created by another 3rd party kubernetes application.

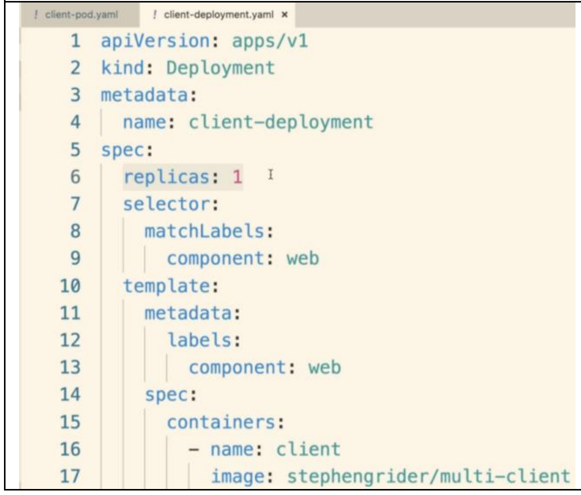
Ingress controller vs load balancer

While ingresses and load balancers have a lot of overlap in functionality, they behave differently. The main difference is ingresses are native objects inside the cluster that can route to multiple services, while load balancers are external to the cluster and only route to a single service.

Deployment object

Usually you don't use pods directly. Instead you define a deployment which maintains the pods.

 Pods Runs a single set of containers Good for one-off dev purposes Rarely used directly in production Deployment Runs a set of identical pods (one or more) Monitors the state of each pod, updating as necessary Good for dev Good for productino	A deployment object maintains a set of identical pods ensuring that they have the correct config and that the right number exists. We define the managed pods with a template.


`apiVersion: apps/v1`
`kind: Deployment`
`metadata:`
`  name: client-deployment`
`spec:`
`  replicas: 1`
`  selector:`
`    matchLabels:`
`      component: web`
`  template:`
`    metadata:`
`      labels:`
`        component: web`
`    spec:`
`      containers:`
`        - name: client`
`          image: stephengrider/multi-client`
	Every pod that will be created by that deployment will have a label of component:web and a container of that image.
	
	Replicas
	How many pods of that configuration to create
	
	Selector
	This deployment will only manage pods with the label of component: web. When you describe the configuration of the containers in the template you assign labels to these containers. You might create some pods and assign another label to them, so that this deployment won't manage them. Why?
	
	This deployment must also have a Service object connected with it which is defined in a separate configuration file.

Update image version

Make a deployment to recreate the pods with the latest version of an image

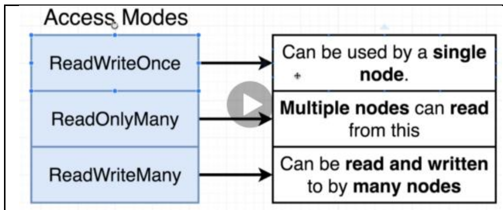
The image has changed but its name remains the same.

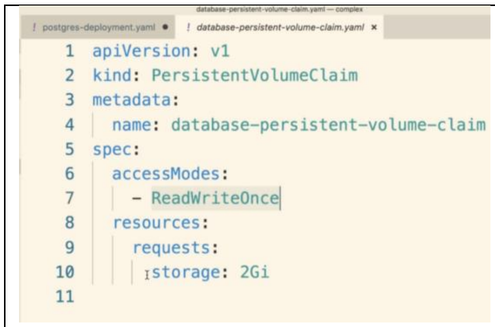
Data storage objects

There are three types of objects, **Volume**, **Persistent Volume**, **Persistent Volume Claim**.

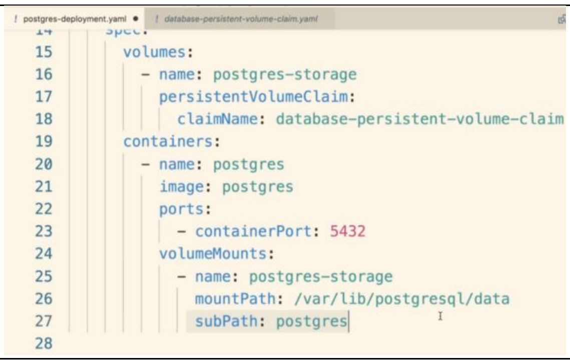
- Volume is for storing data in pod level. It is a volume inside the Pod. This means that isn't affected by containers death but is affected by Pod death. So it's not a persistent volume.
- A Persistent Volume exists outside of the pod so it is not affected by pods death.
- A Persistent Volume Claim is like a statement that exists inside your cluster. This statement informs all the objects inside the cluster about what Persistent Volumes are available for this cluster. An object that wants to use a persistent storage of the specs described in the volume claim, can claim it. There are statically provisioned persistent volumes which are volumes that already exist and dynamically provisioned volumes that will be created on the fly when they are claimed. On your local machine persistent volumes are created in the host machine hard drive. In production there are specific services for storing persistent kubernetes volumes like AWS Block Store (these are referred to as Storage Classes). There is a default option so if you don't specify explicitly it will use that

For each volume object you have to create a configuration file.

 Access Modes ReadWriteOnce Can be used by a single node. ReadOnlyMany Multiple nodes can read from this ReadWriteMany Can be read and written to by many nodes	
The volume claim object	How it is claimed by another object



```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: database-persistent-volume-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```



```yaml
      volumes:
        - name: postgres-storage
          persistentVolumeClaim:
            claimName: database-persistent-volume-claim
      containers:
        - name: postgres
          image: postgres
          ports:
            - containerPort: 5432
          volumeMounts:
            - name: postgres-storage
              mountPath: /var/lib/postgresql/data
              subPath: postgres
```

In this example the default storage class is used since it is not explicitly defined

- mountPath is where this external persistent volume will be mounted inside the container's filesystem. So whenever sth is written there it will be actually written in the external storage. For example in case we have a postgres we will mount it to the path that the db writes its data.

- Subpath: it will create a folder inside the mountPath in the external persistent volume. Postgres needs a subpath.

Get information about where kubernetes stores its persistent volumes.

```
> kubectl get storageclass
```

```
> kubectl describe storageclass
```

```
> kubectl get pv
```

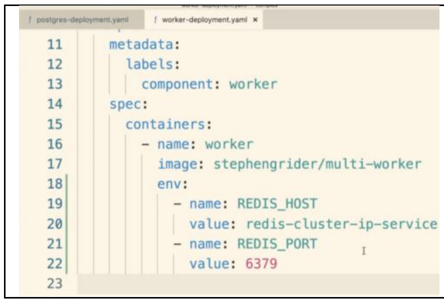
Get the persistent volumes

```
> kubectl get pvs
```

Get the persistent volume claims

Secret object

Environment variables for pods

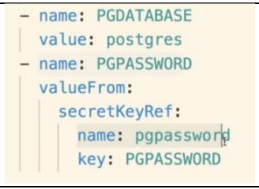


```yaml
    metadata:
      labels:
        component: worker
    spec:
      containers:
        - name: worker
          image: stephengrider/multi-worker
          env:
            - name: REDIS_HOST
              value: redis-cluster-ip-service
            - name: REDIS_PORT
              value: 6379
```

If a service running in a container inside a pod needs access to some environment variables we can define these variables in the pod configuration file in the container setting.

When you want to define passwords for your services you don't want to store them as plain text inside your object's configuration files. What you can do instead is use a special kubernetes object called Secret.

Secret, securely stores a piece of information in the cluster. Type of secret objects: generic (key-value pair), docker-registry and tls. You can save many key-value pairs in a single secret object. You can create it using a configuration file or either through a one off imperative command. The command enables you not to have to deal with a plain text file that you want to keep secret. You just have to run this command both in dev and in prod environments so that the object is created. Then you have to point to it any services that want to access it.

	To use a secret key-pair as the value for an env variable we use the <valueFrom> instruction. The name of the <secretekeyRef> refers to the name of the secret object.

```
- name: PGDATABASE
  value: postgres
- name: PGPASSWORD
  valueFrom:
    secretKeyRef:
      name: pgpassword
      key: PGPASSWORD
```

Namespace object

In Kubernetes, namespaces provides a mechanism for isolating groups of resources within a single cluster.

You can assign namespaces to different sets of resources (different sets of objects). this way you can refer to the whole set whenever necessary. For example you can create a rolebinding (permissions) for a specific namespace. There are some default namespaces created by kubernetes in a cluster.

For example “staging” namespace. You can create a deployment file that targets the staging namespace of your cluster.

Context

A context in Kubernetes is a group of access parameters (for kubctl cli). Each context contains a Kubernetes cluster, a user, and a namespace. Contexts are stored in the kubeconfig file.

Instead of manually defining with which namespace of your cluster you want to interact

>kubectl get pods -n dev

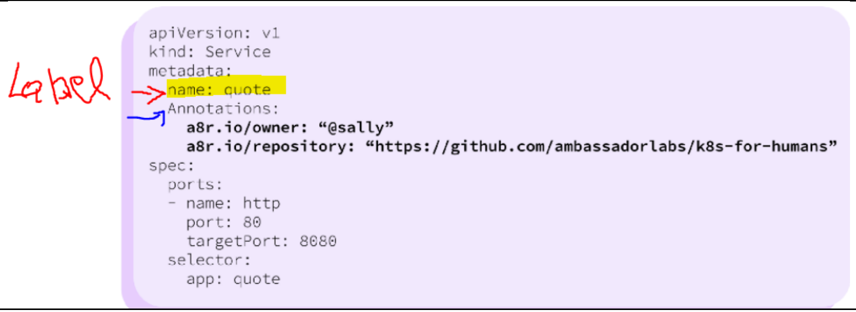
you can create a context, define the cluster, the user and neamaspace=dev and set the current context to this context and then just do

>kubectl get pods

Labels and Annotations

Both labels and annotations are ways to attach metadata to objects in Kubernetes. labels are for kubernetes, annotations are for other applications and humans.

- Labels can be used to select objects and to find collections of objects that satisfy certain conditions. Selectors are used to query labels. To ensure efficient queries, labels are constrained by RFC 1123. RFC 1123, among other constraints, restricts labels to a maximum 63 character length.
- In contrast, annotations are not used to identify and select objects. UseKubernetes annotations to attach arbitrary non-identifying metadata to objects. They are not needed for queries, so they have no constraints on characters. Clients such as tools and libraries can retrieve this metadata. The metadata in an annotation can be small or large, structured or unstructured, and can include characters not permitted by labels.

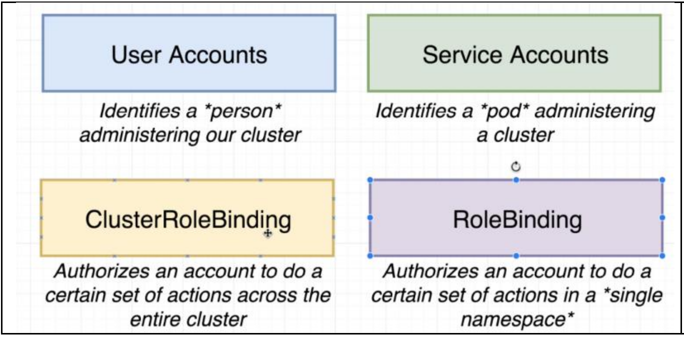
```
apiVersion: v1
kind: Service
metadata:
  name: quote
  Annotations:
    a8r.io/owner: “@sally”
    a8r.io/repository: “https://github.com/ambassadorlabs/k8s-for-humans”
spec:
  ports:
  - name: http
    port: 80
    targetPort: 8080
  selector:
    app: quote
```

ConfigMap object

A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.

A ConfigMap allows you to decouple environment-specific configuration from your container images, so that your applications are easily portable.

Role Based Access Control (RBAC)


User Accounts
Identifies a *person* administering our cluster
Service Accounts
Identifies a *pod* administering a cluster
ClusterRoleBinding
Authorizes an account to do a certain set of actions across the entire cluster
RoleBinding
Authorizes an account to do a certain set of actions in a *single namespace*

Limits who can access and modify objects in our cluster. So there are kubernetes applications (that you install on your kubernetes cluster) that can modify your configurations etc. To do so they need to have permissions to do so. These are defined with the RBAC.

Service accounts are for kubernetes applications. We assign role bindings to user accounts and service accounts.

Service discovery

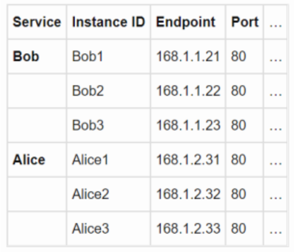
The concept in a nutshell

You build a microservice. You build a client library for it so that other services can easily communicate with it. The client library contains the endpoint of it's service. But it is not a fully qualified url with ip and port. It is something generic like this "http://my-service-name".

The client library uses some other services on the background, to decide to which exact service instance to route the request. This service discovery and load balancing process can be handled by Kubernetes or (if you don't use Kubernetes) by a combination of services like Eureka, Zuul and Ribbon.

Service discovery in non-Kubernetes context

In the Eureka, Zuul and Ribbon case, the client library calls the service like this "ribbon.sendRequest("http://bob/hello", ..)". Behind the scenes, Ribbon would look up which instances are associated to service bob, choose an instance and send the request there. Ribbon pools Eureka to learn the set of available instances, and implements richer logic on top (load balancing policies, retries etc.). Just have in mind that Ribbon has been deprecated in favor of Spring Cloud LoadBalancer. What Ribbon does for inter service requests (discovery and load balancing), Zuul, an API gateway service, does for external client requests. It polls Eureka to get available instances and applies richer logic on top. Zull also does additional api gateway things (SSL termination, rate limiting etc.). All microservices register their service name, IP, port, health endpoint, and other metadata to Eureka as soon as they come up, and keep refreshing the same information with regular heartbeats as long while they are available. When heartbeats stop, Eureka evicts the instance from the registry after a timeout. Each microservice can have a local copy of the registry in it's own cache.



Service	Instance ID	Endpoint	Port	...
Bob	Bob1	168.1.1.21	80	...
	Bob2	168.1.1.22	80	...
	Bob3	168.1.1.23	80	...
Alice	Alice1	168.1.2.31	80	...
	Alice2	168.1.2.32	80	...
	Alice3	168.1.2.33	80	...

An example of Eureka's registry

Service discovery models:

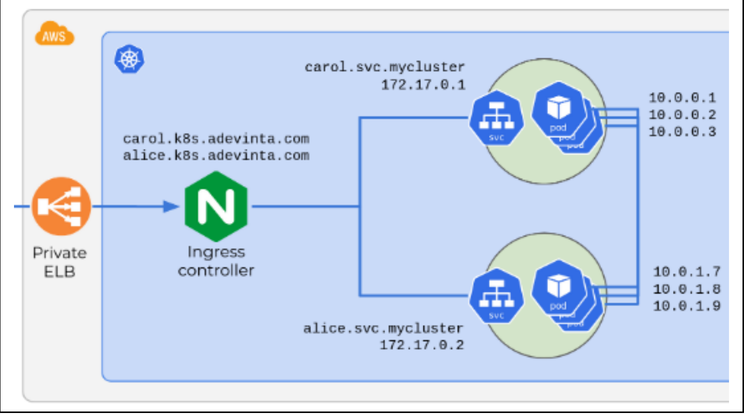
- Client-side service discovery (Eureka)

The client service communicates with the discovery service to get the location of a service. It gets the response with the location. It then communicates with the service in this location

- Server-side service discovery (aws solution)

The client service communicates with the discovery service which gets the location of the service and it communicates with the service too. it gets the response and sends it back to the client service.

Service discovery in Kubernetes
In Kubernetes you don't need Eureka and Zuul. Kubernetes is able to deal with both service discovery and load balancing on its own, although using very different approaches.



To deploy microservices like Bob and Alice we use ordinary Deployment objects that generate a number of Pods running the microservice. For each Deployment we also create a Service that defines how to reach those Pods.

Each Service object gets a private DNS record like carol.svc.mycluster that resolves to a virtual IP (172.17.0.1 in the diagram). When any Pod talks to carol.svc.mycluster Kubernetes will do its magic and route traffic to healthy Pods running Carol.

Internal K8s services communication (like Ribbon)
So in the client library of your microservice the endpoint is the private DNS record of the Service object of a deployment. You can also use a service mesh for extra functionalities in the service to service communication.

External communication (Similar to Zuul)
When we have a Service that needs to be exposed outside the cluster, we create an Ingress for it, for example NginX Ingress Controller (or even better an API gateway). It will map HTTP requests with a Host header carol.k8s.adevinta.com to the service's private DNS record, carol.svc.mycluster, which resolves to the virtual IP, etc.

API gateway, Ingress Controller and Service Mesh
An API gateway is an Ingress controller with rich functionality. They are both responsible for handling external traffic to your cluster, but generally the API Gateway implements many functionalities (authentication/authorization, TLS termination, throttling, caching etc.). An API gateway can be either inside or outside of your Kubernetes cluster. This means that you can connect AWS gateway (with private link) so that it routes traffic to your Kubernetes cluster. Or you can use a public implementation for a native Kubernetes API gateway like Kong (or the new native API gateway from kubernetes).

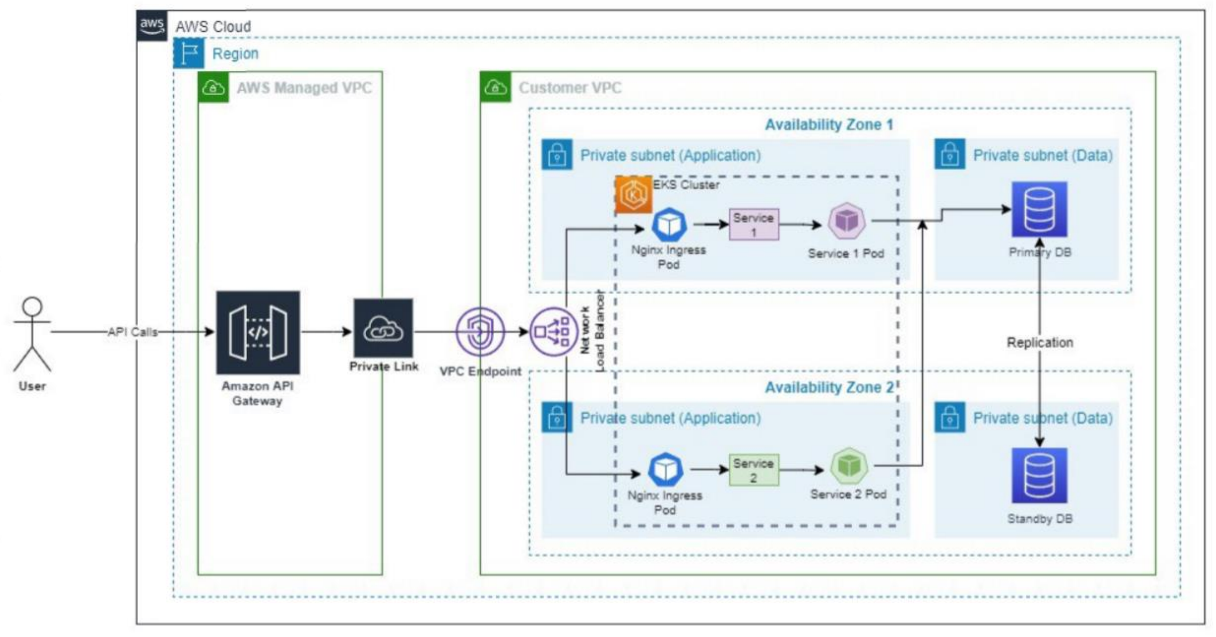
The service mesh (like Istio) on the other hand, is mostly used to handle traffic within your cluster (internal services communication). Typical uses of service meshes include: monitoring and observing requests between apps, securing the connection between, services using encryption (mutual TLS), improving resiliency with circuit breakers, retries, etc. Since service meshes are deployed alongside your apps, they benefit from: low latency and high bandwidth, unlikely to be targeted for misuse by bad actors

- Internal API gateway

you want it inside, then it's better to use a public implementation like Kong. Usually, these implementation are functionality rich ingress controllers. **Kong** has plugins for authentication, certificate management, Grpc control. (Kubernetes Gateway API. This is a new resource, introduced in 2021 by Kubernetes. It is the evolve of the Ingress. In beta in 2021.) have in mind that most popular implementations like Kong, are developing service mesh solutions too (KongMesh).

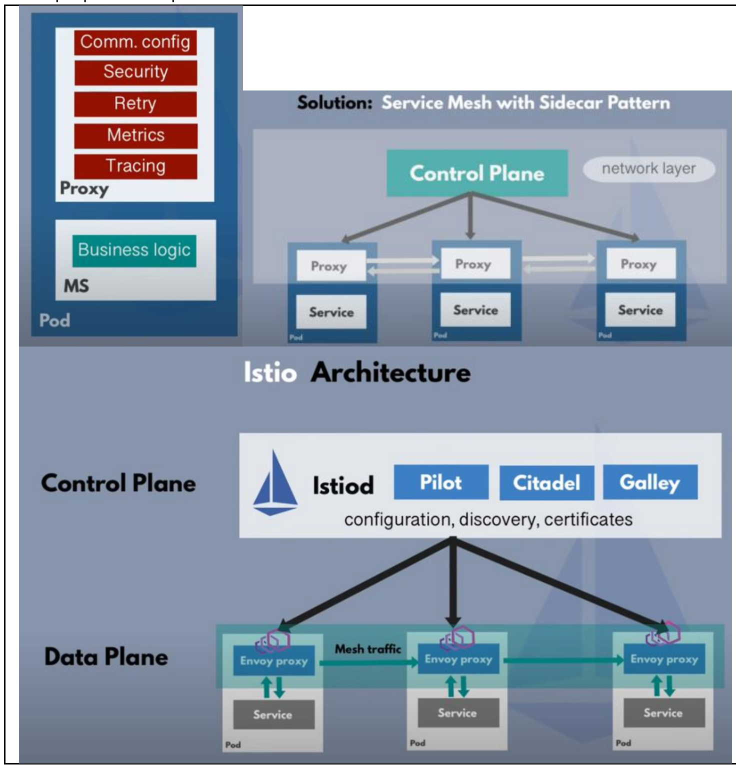
- External API gateway

You can use AWS API Gateway to expose their microservices running in Kubernetes. API Gateway private integrations let you expose services running an EKS cluster to clients outside of your VPC using Network Load Balancers (NLB) and Application Load Balancers (ALB). Currently, customers that use API Gateway to expose their private microservices running in EKS manage their API Gateway configuration separately from their Kubernetes resource definitions. For example, many customers use an infrastructure-as-code tool, like CloudFormation or Terraform, to create API Gateway resources and Helm or a GitOps tool to manage their Kubernetes cluster configuration.
This post will use AWS Controller for Kubernetes (ACK) to create and manage API Gateway resources. ACK is a community-driven project that lets you manage AWS services using the Kubernetes API and tools you are already familiar with, like kubectl. Using ACK, you can create and update AWS service resources, like an S3 bucket or API Gateway API, the same way you create and update a Kubernetes deployment, service, or pod.



Service mesh

One popular implementation of a service mesh is Istio.

 Comm. config Security Retry Metrics Tracing Proxy Business logic MS Pod Solution: Service Mesh with Sidecar Pattern Control Plane network layer Proxy Service Istio Architecture Control Plane Istiod Pilot Citadel Galley configuration, discovery, certificates Data Plane Envoy proxy Mesh traffic Service Pod	A service mesh implementation installs a proxy service in front of your service inside of your service pod. This proxy is responsible for handling any communication between your service (MS from microservice in the picture) and any other service of the cluster. Instead of writing all the communication logic in every service, you focus on the service functionality only, and delegate all the communication stuff to the proxy. Service discovery configuration, internal encryption (with mutual TLS for example), retry logic, even metrics and tracing configuration with the tools of your choice (like Prometheus and Zipkin respectively) are implemented by this proxy service. It's also easy to setup traffic splitting for each service (canary deployment) where the service mesh routes let's say 10% of the traffic to the new version of a service and the rest to the old one. There are open source implementations for such a proxy service like Envoy proxy that Istio uses.

Serverless

- One option is to trigger serverless functions outside of your cluster (for example aws lambdas using lambda controller pod inside the cluster)
- Another option is to use internal functions that run inside the cluster (for example fission)

Calling aws Lambdas from kubernetes cluster

AWS controllers for Kubernetes (ACK)

AWS Controllers for Kubernetes (ACK) lets you define and use AWS service resources directly from Kubernetes. With ACK, you can take advantage of AWS-managed services for your Kubernetes applications without needing to define resources outside of the cluster or run services that provide supporting capabilities like databases or message queues within the cluster.

https://aws.amazon.com/blogs/compute/deploying-aws-lambda-functions-using-aws-controllers-for-kubernetes-ack/

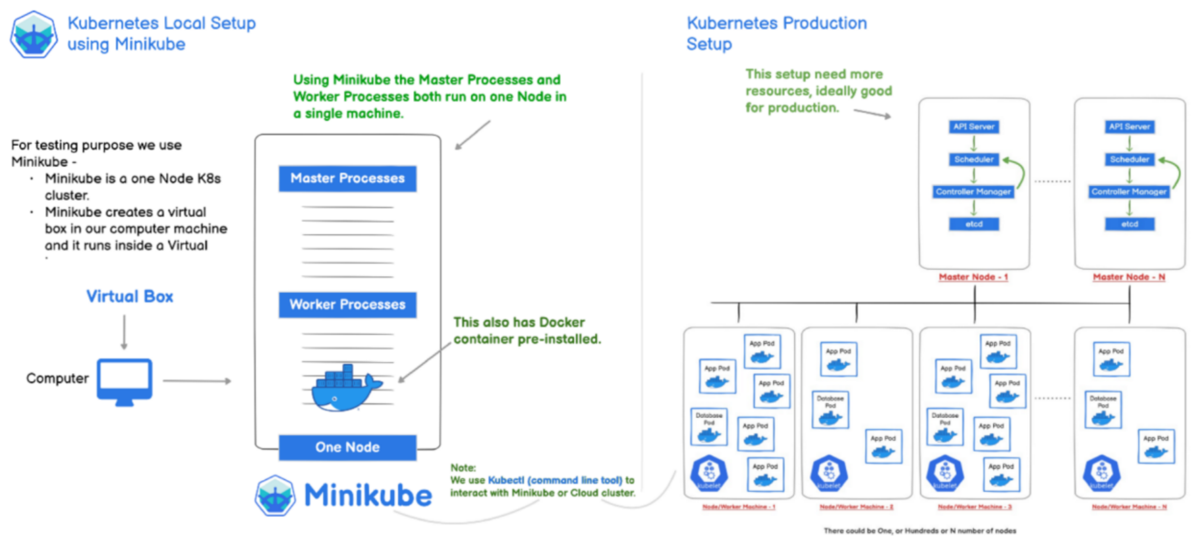
You have to define an AWS lambda controller pod in your cluster. This is a custom kubernetes resource created by aws. Any of your pods can communicate with this controller, to spin up a lambda function. This lambda controller gets the zip file that contains the lambda function code from s3 where it is stored, and spins a new lambda function in aws infrastructure for it. you define the proper permissions so that it can do it.

Fission
Fission is a framework for serverless functions on Kubernetes. Write short-lived functions in any language, and map them to HTTP requests (or other event triggers).

We're built on Kubernetes because we think any non-trivial app will use a combination of serverless functions and more conventional microservices, and Kubernetes is a great framework to bring these together seamlessly. Building on Kubernetes also means that anything you do for operations on your Kubernetes cluster — such as monitoring or log aggregation — also helps with ops on your Fission deployment.

100msec cold start

Minikube for local dev environment


Kubernetes Local Setup using Minikube
Using Minikube the Master Processes and Worker Processes both run on one Node in a single machine.
For testing purpose we use Minikube -
Minikube is a one Node K8s cluster.
Minikube creates a virtual box in our computer machine and it runs inside a Virtual
Virtual Box
Computer
Master Processes
Worker Processes
This also has Docker container pre-installed.
One Node
Minikube
Note: We use Kubectl (command line tool) to interact with Minikube or Cloud cluster.
Kubernetes Production Setup
This setup need more resources, ideally good for production.
API Server
Scheduler
Controller Manager
etcd
Master Node - 1
Master Node - N
App Pod
Database Pod
kubelet
Node/Worker Machine - 1
Node/Worker Machine - 2
Node/Worker Machine - 3
Node/Worker Machine - N
There could be One, or Hundreds or N number of nodes

There are kubernetes managed services that can be used to deploy your kubernetes project. For example AWS Amazon Elastic Container Service for kubernetes (EKS) or google kubernetes engine. In development though, you use a tool called **minikube** that creates a kubernetes cluster in a vm in your local machine. Minikube runs a single-node Kubernetes cluster inside a Virtual Machine (VM) on your laptop for users looking to try out Kubernetes or develop with it day-to-day. Minikube also offers a dashboard web app for managing your cluster.

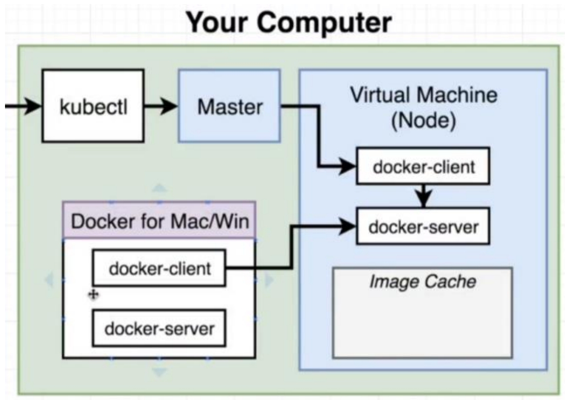
Minikube creates one VM which represents one node. So during development the whole cluster lives inside a single node.

In your local machine you have docker installed and then you have docker installed within the VM minikube created. Any time you type a docker command in the terminal the docker command will read a set of environment variables to decide which copy of docker server to connect to. By default it will look in the global environment variables.

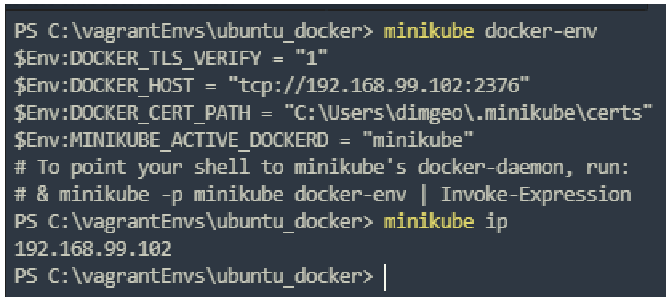
But you can evaluate a set of env variables for a specific session in a terminal and docker will use this set. This is what you can do with the eval command. Then docker will use the docker server on the minikube's vm (the node). then docker ps will list all containers running in the kubernetes node.

Or as I saw you can ssh into you're the vm created by minikube and use the docker installed in it directly.

Maybe this picture is false. Master is also inside the vm. Kubectl is the only thing that runs in the host machine.



> eval $(minikube docker-env)



```
PS C:\vagrantEnvs\ubuntu_docker> minikube docker-env
$Env:DOCKER_TLS_VERIFY = "1"
$Env:DOCKER_HOST = "tcp://192.168.99.102:2376"
$Env:DOCKER_CERT_PATH = "C:\Users\dimgeo\.minikube\certs"
$Env:MINIKUBE_ACTIVE_DOCKERD = "minikube"
# To point your shell to minikube's docker-daemon, run:
# & minikube -p minikube docker-env | Invoke-Expression
PS C:\vagrantEnvs\ubuntu_docker> minikube ip
192.168.99.102
PS C:\vagrantEnvs\ubuntu_docker>
```

Doing so you can use all docker cli commands for kubernetes containers for debugging purposes, manually kill containers to test auto heal, delete cached images etc. For example you could do *docker logs <container-id>* or *docker exec -it <container-id> sh* have in mind though that many of these commands can be executed on individual containers directly from kubectl.

> kubectl get pods (see the names of containers)

> kubectl logs <container-name>

> kubectl exec -it <name> sh

Minikube dashboard

> minikube dashboard

It is a service running in the VM created by minikube. It contains a lot information about your cluster.

> minikube ip

Notice that minikube will assign a static IP to the created virtual machine so in order to connect to it from a browser for example you have to identify it and use it.

Kubectl cli for kubernetes

- Apply a configuration

> kubectl apply -f <conf_file>

Change the current configuration of the cluster. Also used to create a new cluster. For applying two files we have to run it two times. Notice that the controller process of the master node will watch the "should be" and "is" states of the defined configuration and ensure that they match. If you make a > docker ps, get the id of a docker container running in a pod and kill it > docker kill <id> you will notice that it will be restarted in the kubernetes cluster. You can confirm that by running *kubectl get* and checking the number of restarts and the time it is alive.

```
> kubectl apply -f <folder-name>
```

- Delete a configuration

```
> kubectl delete -f <conf_file>
```

```
> kubectl delete <object-type> <object-name>
```

- Update a configuration (same names new images)

Kubernetes updates are Rolling updates

Users expect applications to be available all the time and developers are expected to deploy new versions of them several times a day. In Kubernetes this is done with rolling updates. Rolling updates allow Deployments' update to take place with zero downtime by incrementally updating Pods instances with new ones. The new Pods will be scheduled on Nodes with available resources.

```
> kubectl rollout restart  <resource>
```

Imperatively update to new image version

Before the support of rollout restart a common workaround for updating the pods to the new images would be to tag the images with a version (username/image_name:version) and then use an imperative command in kubectl to explicitly tell the master node to update specific pods with this image. (In the deployment pipeline you can use a script to automate this process defining the appropriate versioning tags in Travis and modifying the kubectl command).

```
> kubectl set image <object_type/object_name> <container_name>=<new_image_tag>
```

For example: kubectl set image deployment/my_depl_name my_container_in_my_deployment=username/img-name:v5

Using this command you can rollback to a previous version. You just have versions of your image in your image hub.

- Create imperatively a kubernetes object

```
> kubectl create <object-type> <object specific fields>
```

- Show information about an object or type of objects

```
> kubectl get <pods or services or etc.> -o wide
```

With the flags will give us extra info for example the ip of the individual pods.

```
> kubectl describe >object_kind> <object_name>
```

Autoscaling

Auto backup and restore

Helm package manager

Have in mind Helm which is a package manager for kubernetes applications. Using it we can install third party kubernetes objects in our kubernetes cluster. There are various helm hubs with many kubernetes ready applications like mongodb, elastic search etc. A **package** is a kubernetes yaml configuration file describing the application. (Helm 3 removes the use of Tiller).

Some kubernetes packages

Cert Manager It automatically handles the TLS certificate

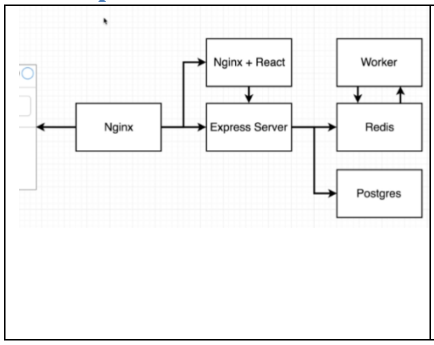
Ingress-nginx. It creates an input (Ingress) to a cluster by configuring an nginx server

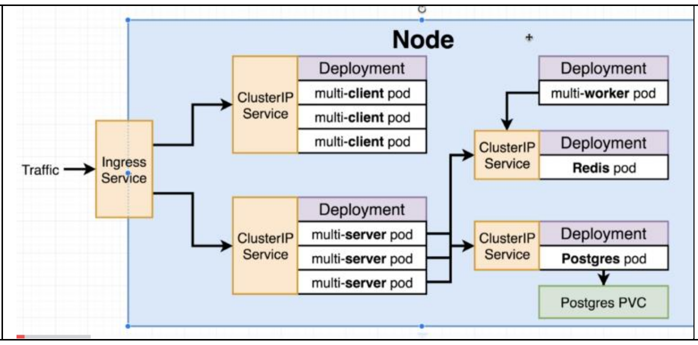
Skaffold

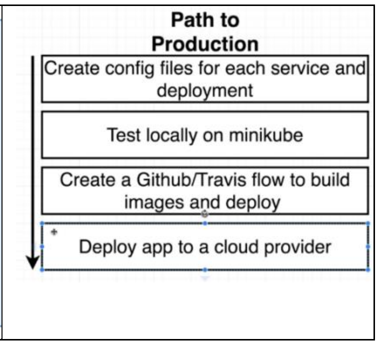
not a helm package, its a tool for auto updating the source code from our host machine in the container during development

Multi container application example

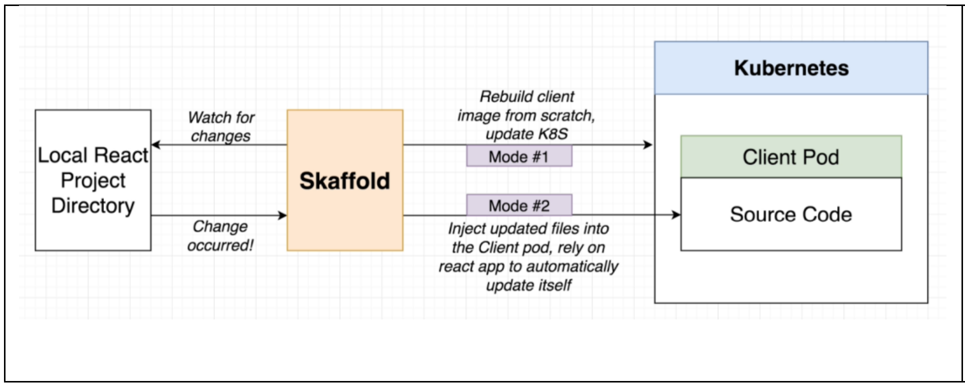
Development






Path to Production
Create config files for each service and deployment
Test locally on minikube
Create a Github/Travis flow to build images and deploy
Deploy app to a cloud provider

Previously we had used docker-compose for development and AWS EB for deployment. Now we will use kubernetes for development and deployment (to AWS). so no need for the compose and aws config files. For this project we will need 11 config files one for each object. We can place them in a folder inside our project.


Local React Project Directory
Watch for changes
Change occurred!
Skaffold
Rebuild client image from scratch, update K8S
Mode #1
Mode #2
Inject updated files into the Client pod, rely on react app to automatically update itself
Kubernetes
Client Pod
Source Code

Previously we used docker volumes to update the source code and have it be reflected inside the container. This was used with docker-compose. Now that we don't use docker-compose but kubernetes we use another tool called **Skaffold**. Its a CLI. You configure skaffold with a yaml file. You actually define which source files and which pod (or deployment or whatever a kubernetes config file defines) to monitor.

It works in one of two possible modes. The first one is the typical one where the image is rebuild. For the second one your project needs to be configured accordingly using nodemon for node js for example. With nodemon whenever there is a change in the project files inside the container nodemon will restart the server so you will be able to see the changes immediately. Skaffold will just copy the changes from your local machine to inside the container. Nodemon will see that and restart the server.

`> skaffold dev` starts skaffold. It starts your defined kubernetes deployments and watch for changes in the defined source code.
> Ctrl+C closes skaffold. Have in mind that it automatically closes all the defined deployments too so you don't have to do it manually.

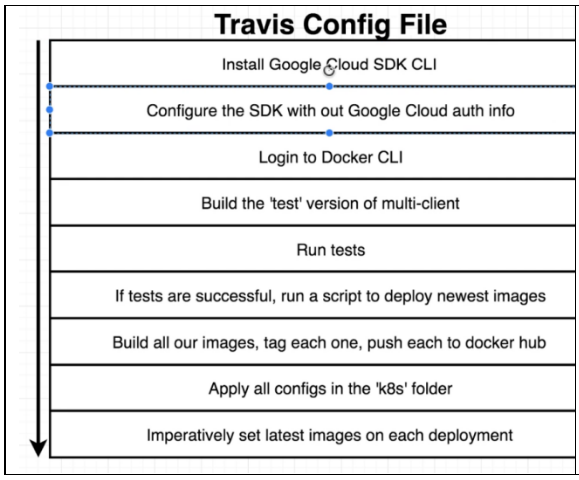
In development environment we can create our own pods for redis and postgresql. The problem with postgresql is that it needs to store data permanently but the container filesystem is ephemeral. If the container crashes the data is lost. In order to overcome this we can use an external Volume on our host machine. Every time posgresql service writes data it will write the data to this volume instead of the ephemeral container filesystem.

Postgres PVC is a Persistent Volume Claim. It needs to be defined so that it can be claimed by the postgresql object.

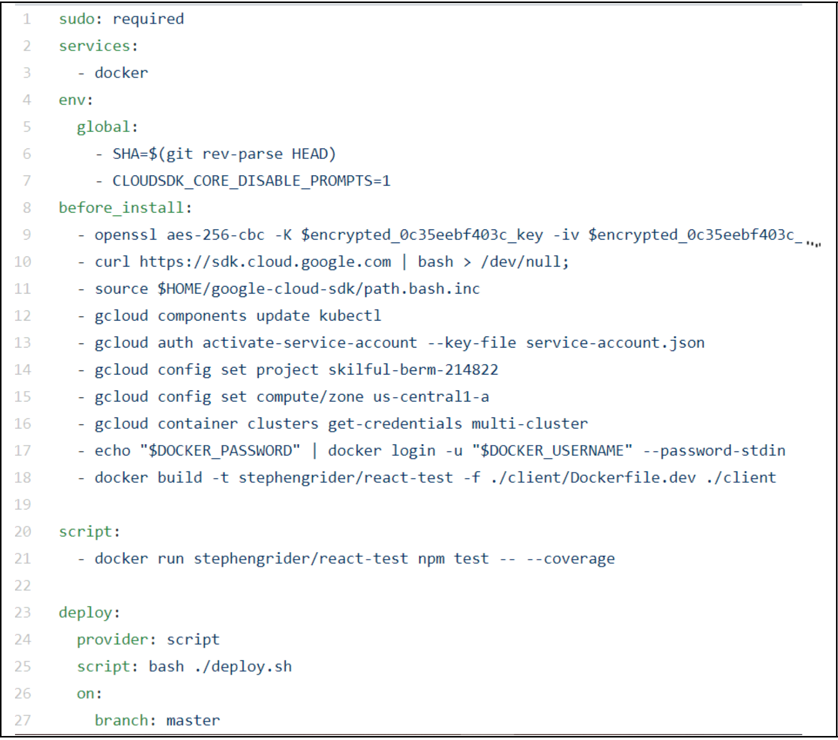
Notice that you have to change the default password for the default posgtresql user by defining it in the pod config file.

Production

He used three nodes in the cluster

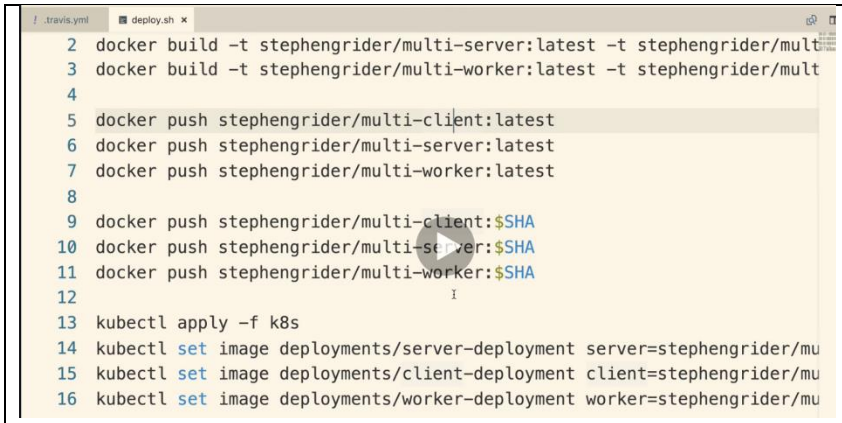

Travis Config File
Install Google Cloud SDK CLI
Configure the SDK with out Google Cloud auth info
Login to Docker CLI
Build the 'test' version of multi-client
Run tests
If tests are successful, run a script to deploy newest images
Build all our images, tag each one, push each to docker hub
Apply all configs in the 'k8s' folder
Imperatively set latest images on each deployment

Travis will build our images, push them to docker hub and deploy our application to a kubernetes cluster.
The google cloud SDK CLI is a tool which we will install our Travis environment so that Travis is able to manage our google cloud project.
Probably the last step would be replaced by the rollout restart command



```
sudo: required
services:
  - docker
env:
  global:
    - SHA=$(git rev-parse HEAD)
    - CLOUDSDK_CORE_DISABLE_PROMPTS=1
before_install:
  - openssl aes-256-cbc -K $encrypted_0c35eebf403c_key -iv $encrypted_0c35eebf403c_
  - curl https://sdk.cloud.google.com | bash > /dev/null;
  - source $HOME/google-cloud-sdk/path.bash.inc
  - gcloud components update kubectl
  - gcloud auth activate-service-account --key-file service-account.json
  - gcloud config set project skilful-berm-214822
  - gcloud config set compute/zone us-central1-a
  - gcloud container clusters get-credentials multi-cluster
  - echo "$DOCKER_PASSWORD" | docker login -u "$DOCKER_USERNAME" --password-stdin
  - docker build -t stephengrider/react-test -f ./client/Dockerfile.dev ./client

script:
  - docker run stephengrider/react-test npm test -- --coverage

deploy:
  provider: script
  script: bash ./deploy.sh
  on:
    branch: master
```

The .travis.yml file



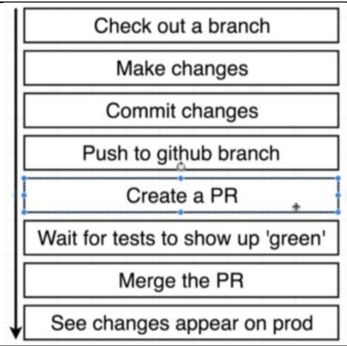
```
docker build -t stephengrider/multi-server:latest -t stephengrider/mult
docker build -t stephengrider/multi-worker:latest -t stephengrider/mult

docker push stephengrider/multi-client:latest
docker push stephengrider/multi-server:latest
docker push stephengrider/multi-worker:latest

docker push stephengrider/multi-client:$SHA
docker push stephengrider/multi-server:$SHA
docker push stephengrider/multi-worker:$SHA

kubectl apply -f k8s
kubectl set image deployments/server-deployment server=stephengrider/mu
kubectl set image deployments/client-deployment client=stephengrider/mu
kubectl set image deployments/worker-deployment worker=stephengrider/mu
```

The deploy script
In the travis file we define a deployment script since in 2018 it didn't had build in support for kubernetes. This is the deployment script. It builds the images pushes to the docker hub and sends a command to kubernetes to reapply a certain configuration.

Notice that we use the GIT_SHA as a version for the images. It is the SHA of a git commit. This process can easily be automated. You define an env variable in the travis yml file which is the GIT SHA like so SHA=$(git rev-parse HEAD). Then you can use this env variable in your deploy script.

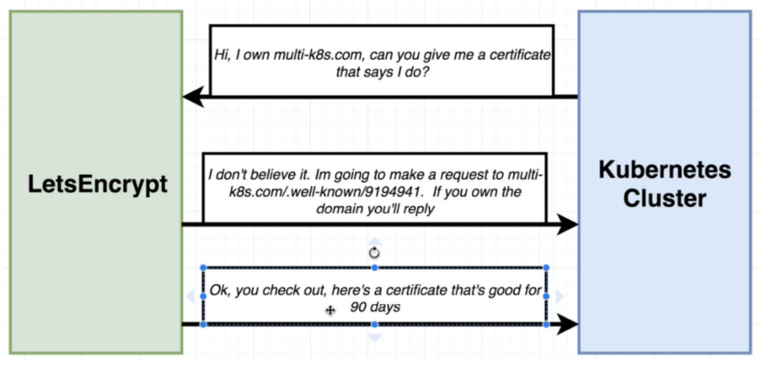
We do this SHA versioning because we need to have a version in the images not just the "latest" because kubernetes will not detect any changes. You can probably avoid this using the new rollout restart command.

Changing the source code



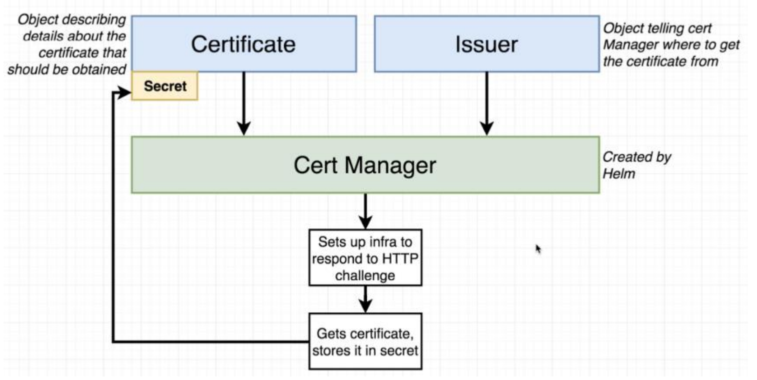
Push to devel in github
Create pull request to master

TLS certificate for https


LetsEncrypt
Kubernetes Cluster
Hi, I own multi-k8s.com, can you give me a certificate that says I do?
I don't believe it. Im going to make a request to multi-k8s.com/.well-known/9194941. If you own the domain you'll reply
Ok, you check out, here's a certificate that's good for 90 days

Let's encrypt is a popular certificates provider. The http process for getting a certificate is as follows. You declare to the issuer that you own a domain. To prove it you have to set up in your domain a specific random url path given to you by let's encrypt. Let's encrypt will make a request in that url and if it gets the proper reply it will issue a certificate valid for 90 days.

There are kubernetes implementations that handle this process. You just have to install and configure them. You can install them using helm. **Cert Manager** is such a service


Object describing details about the certificate that should be obtained
Certificate
Secret
Issuer
Object telling cert Manager where to get the certificate from
Cert Manager
Created by Helm
Sets up infra to respond to HTTP challenge
Gets certificate, stores it in secret

Cert Manager creates a Cert Manager deployment which receives two custom objects (issuer and certificate defined by Cert Manager) configured by us with some config files. The issuer defines the certificate issuer and the certificate contains the certificate details. You can create more than one issuers. When these objects are created it initiates the process. It finally stores the resulting TLS certificate in a secret. You create the objects deploy them wait for a few minutes for the process of getting the certificate to complete. You then have to reconfigure your ingress service to use that certificate and route https traffic. Cert manager will automatically repeat the process before the certificate expires.

Misc

Openshift

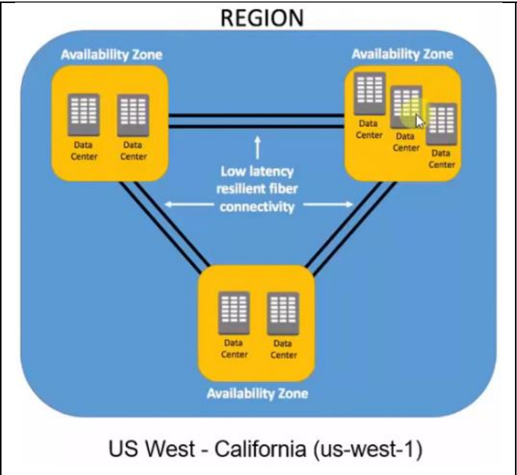
Have in mind Openshift from redhat, which is a platform built on top of Kubernetes and contains additional services that run on top of the Kubernetes cluster. Services like the gateway, the load balancer, logging and metrics with Grafana on top of them to make them more explorable etc.

AWS

Overview

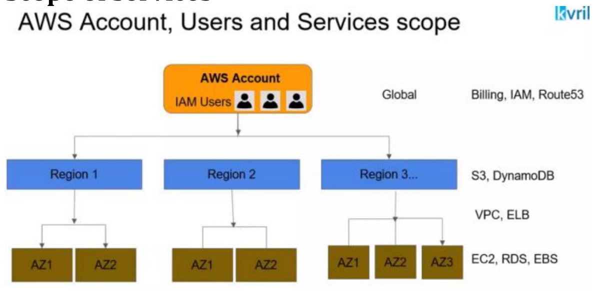
https://aws.amazon.com/prescriptive-guidance/?apg-all-cards.sort-by=item.additionalFields.sortText&apg-all-cards.sort-order=desc&awsf.apg-new-filter=*all&awsf.apg-content-type-filter=*all&awsf.apg-code-filter=*all&awsf.apg-category-filter=*all&awsf.apg-rtype-filter=*all&awsf.apg-isv-filter=*all&awsf.apg-product-filter=*all&awsf.apg-env-filter=*all
Guidelines from aws. Quite good.

Regions and Availability zones



There are Regions, Access Zones AWS. A region is a large geographic area served separately by a group of datacenters and is separated in availability zones (AZ) where each zone consists of some datacenters. Different AWS services work in different scopes. For example when you setup an S3 bucket you choose a region. Us-east-1a the a is a specific AZ. AZ us-east-2a is different for each user. There are also b and c. One user's a could be other users b etc. This is to randomize the distribution of resources among AZs. Deploy your instances in multiple AZs so that if one goes down the others continue to serve.

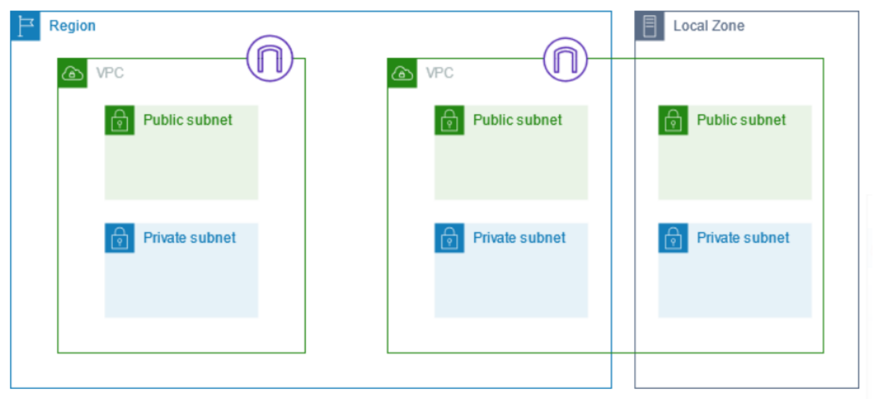
Scope of services



Edge locations

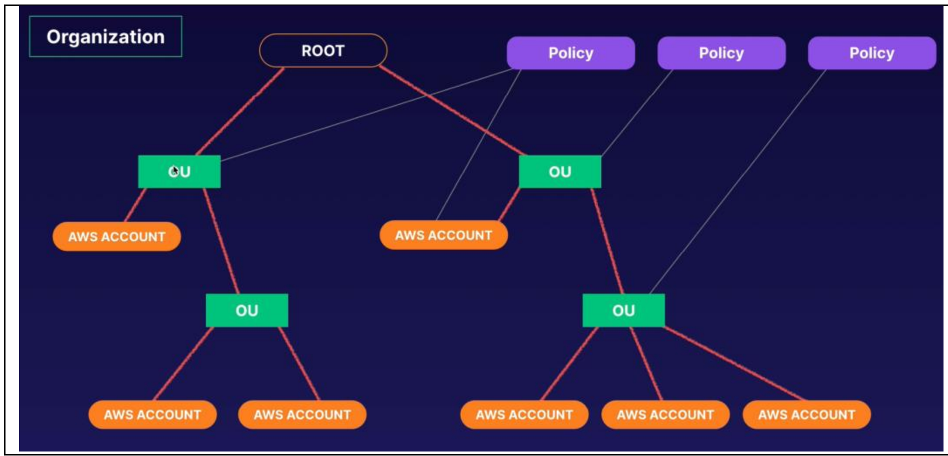
Edge locations are AWS data centers designed to deliver services with the lowest latency possible. Amazon has dozens of these data centers spread across the world. They're closer to users than Regions or Availability Zones, often in major cities, so responses can be fast and snappy. A subset of services for which latency really matters use edge locations, including: CloudFront, Route 53, WAF (Web Application Firewall) and AWS Shield.

Local zones

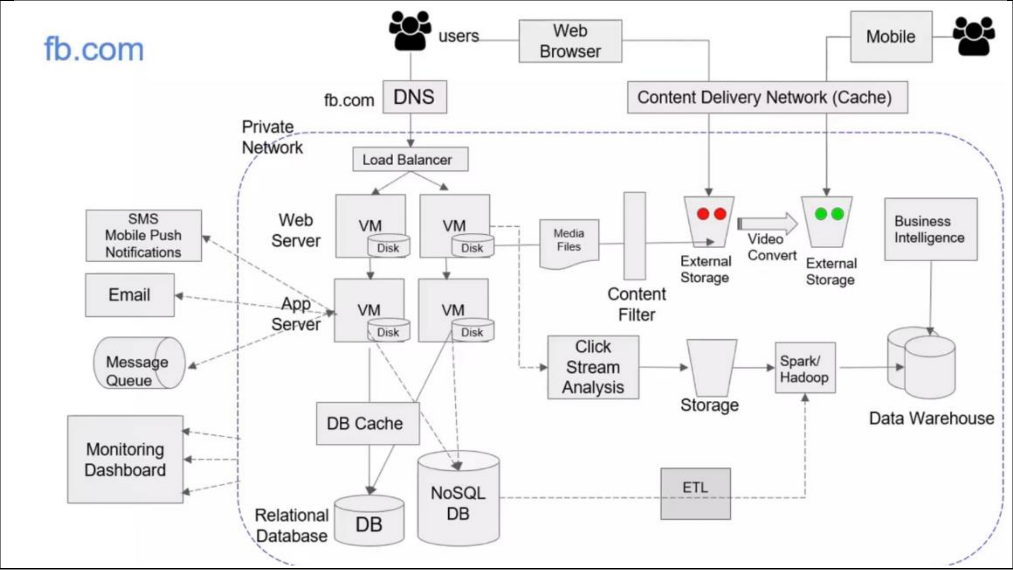
 Region VPC Public subnet Private subnet VPC Public subnet Private subnet Local Zone Public subnet Private subnet	A Local zone is an AWS infrastructure deployment that places compute, storage, database, and other select services closer to your end users. A Local Zone enables your end users to run applications that require single-digit millisecond latencies. Contrary to edge locations, they are not only for caching, CDN, and DNS resolution. They are datacenters that support many aws resources. Greece will be the site of one of 11 AWS Local Zones that the company is planning to establish in Europe.

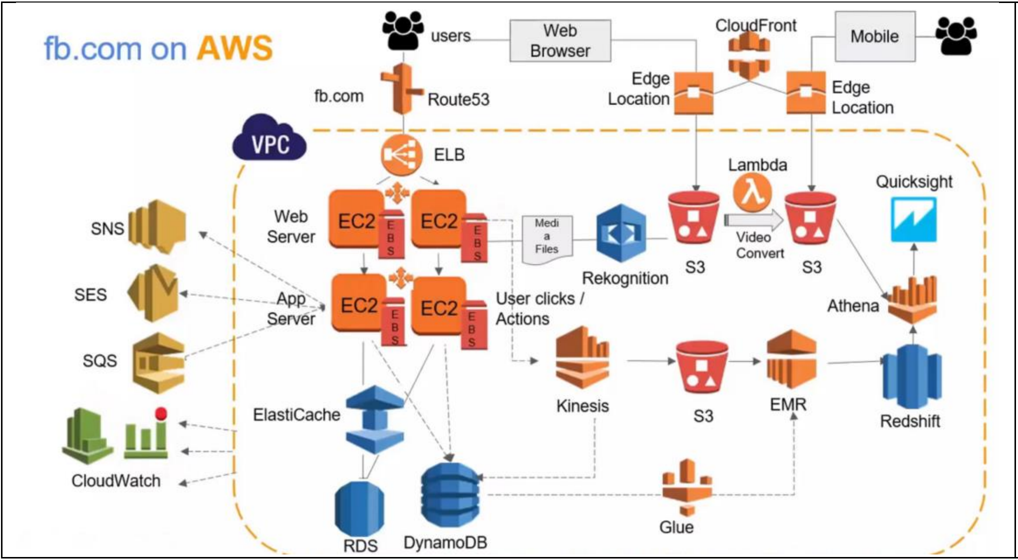
AWS organizations

It is an account management service that enables you to consolidate multiple aws accounts into an organization.

 Organization ROOT Policy Policy Policy OU OU AWS ACCOUNT AWS ACCOUNT OU OU AWS ACCOUNT AWS ACCOUNT AWS ACCOUNT AWS ACCOUNT AWS ACCOUNT	Root account is better to be used only for billing, not for other resources. OU - organization units

Architecture examples

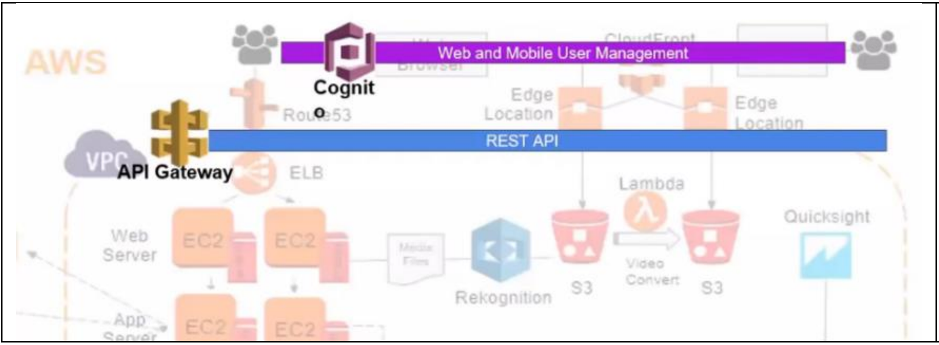
 fb.com users Web Browser Mobile fb.com DNS Content Delivery Network (Cache) Private Network Load Balancer SMS Mobile Push Notifications Web Server VM Disk VM Disk Media Files Content Filter External Storage Video Convert External Storage Business Intelligence Email App Server VM Disk VM Disk Message Queue Click Stream Analysis Storage Spark/ Hadoop Data Warehouse DB Cache Monitoring Dashboard NoSQL DB ETL Relational Database DB	An example of an architecture for a social network and how it could be deployed in AWS. Some static files might need to be converted to a mobile friendly format and be stored and served separately.



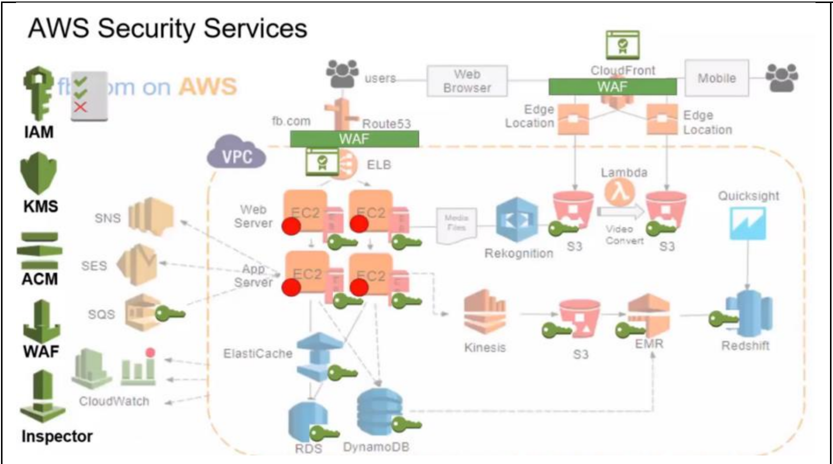
The VPC doesn't contain all these services but it is drawn like this for convenience.

In EC2s you can run your VMs or docker containers. You can setup autoscaling for your EC2s, so that when load is increased more EC2s are spawn off. Have in mind that an EC2 instance must have permissions to access an S3 bucket.

One way to implement the video conversion to mobile friendly format is to set up a FIFO queue and have some workers (EC2s) listen to that queue and whenever a video is uploaded they get it, convert it and store it in the respective S3. another way is to write a lambda function that is triggered whenever a new video is uploaded and does the same process. This way you don't have to maintain the workers.

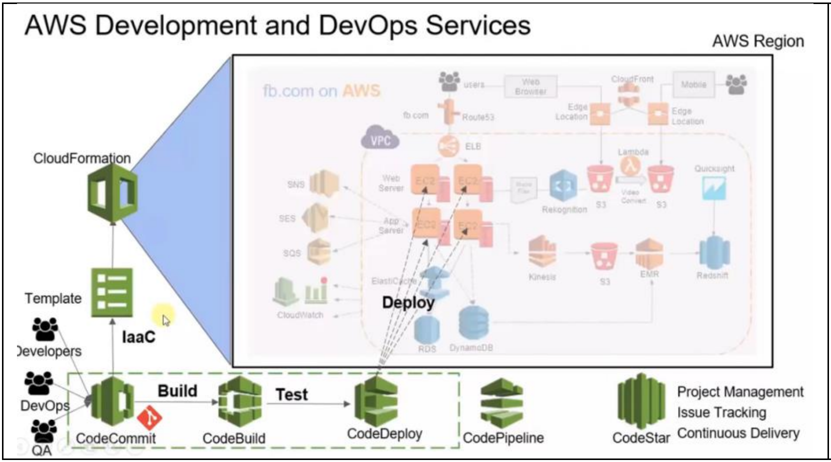


AWS security services



- KMS store the keys used for encryption
- ACM (Amazon Certificate manager) digital certificates usually deployed in load balancers or in the CDN
- **WAF (Web application firewalls**) application firewalls, prevent DDOS, SQL injections, cross site scripting etc. Usually deployed in front of the API gateways or the load balancer and CDN. It lets you monitor the HTTP and HTTPS requests that are forwarded to cloudfront, load balancer or API gateway. Web app firewalls are at application level (level 7) this means that you have access to higher level info like querystrings etc based on which you can create rules. Block IPs, countries etc.
- Inspector scans your machines for known vulnerabilities (in terms of compliance)

AWS development and Devops services



CloudFormation infrastructure as a code. It takes a json template and creates the defined infrastructure from scratch. There are many public templates from various publishers available in Quick Start, you can go and use them.

Then he describes the CI/CD pipeline where a developer does a commit which is automatically built and deployed. (The IaaC is the infrastructure template I guess)

EC2

Root device volume

By default an EC2 instance has a Root device volume (/dev/xvda) where the OS is installed. They can be encrypted too. You can now encrypt the root device volume when you create the EC2 instance. This was not possible. You have to take a snapshot of the volume, make a copy of it, encrypt the copy and create an image from the encrypted copy. Then deploy that image to an EC2 instance.

Ec2 instances might sit on top of a hypervisor.

Tags

Tags is a way to add some information to the EC2 instance and volumes so that you know for example to which department they belong etc. An EC2 instance can inherit its name from its tag plus a counter.

EBS

You can add more volumes to your EC2 instance of type EBS. EBSes are virtual hard drives. They are in the same AZ as their EC2 instances so that latency between EC2 and its EBS is small. They are automatically replicated within their AZ. If you add additional storage to an EBS instance you have to run a command on it that repartitions the disk in order to see the extra space.

Instance store volumes

You can choose to attach to your EC2 instance an instance store instead of an EBS for the root device volume. Instance store is ephemeral. If the instance reboots you lose all data stored on it.

Security groups

- Instance level firewall.
- For opening ports (for Inbound and outbound traffic)
- Effects are immediate

Security groups are virtual firewalls using which you can restrict all communication but some ports for example. Each instance has its own security group. They are distinct objects which you can assign EC2 instances to. For example you can have a webserver security group with open only the ports 22 for SSH and 80 for HTTP. An EC2 instance with a web server can be in one security group and an RDS instance in another one, you have to set up an inbound rule that opens access in db security group on port the db listens to, specifically for the security group of the web server EC2. you specify the security group by its security group ID. You can add more than one sec groups to an instance. The effects are immediate. All inbound traffic is blocked by default. You open ports with security groups. Notice that if you allow Http in, automatically an outbound rule is created to allow outbound traffic for it. This means that security groups are stateful.

You can block traffic form specific ports or ips for the associated ec2 instance. But it's more convenient to do it in the ACL (subnet level firewall) which works on subnet level so it will block it for all ec2 instances of the subnet.

Snapshots and Images (AMIs)

Take snapshot of a volume -> create AMI image from it -> deploy the image

One usage is when you want to change the AZ of an EC2 volume (which is an EBS instance). Snapshots are like a photograph of a volume. Notice that snapshots exist on S3. Volumes exist on EBS. You can create snapshots of your volumes, then create an image (an AMI) from a snapshot and deploy that image to an EC2 instance in another availability zone.

Bootstrap script

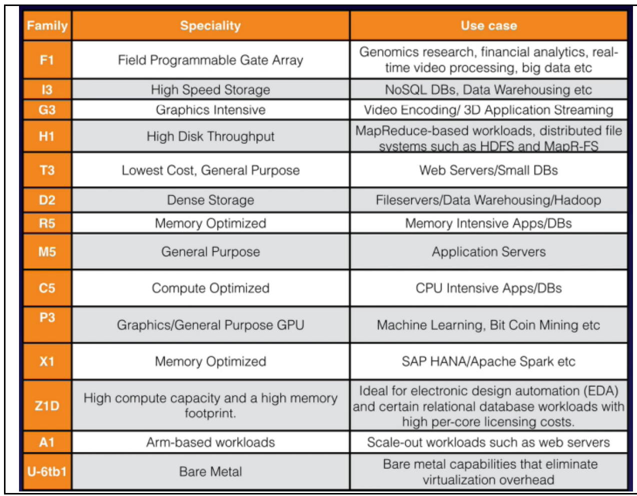
It is a script that will be executed when the EC2 instance is launched. Using it you can install things on your instance for example.

Placement groups

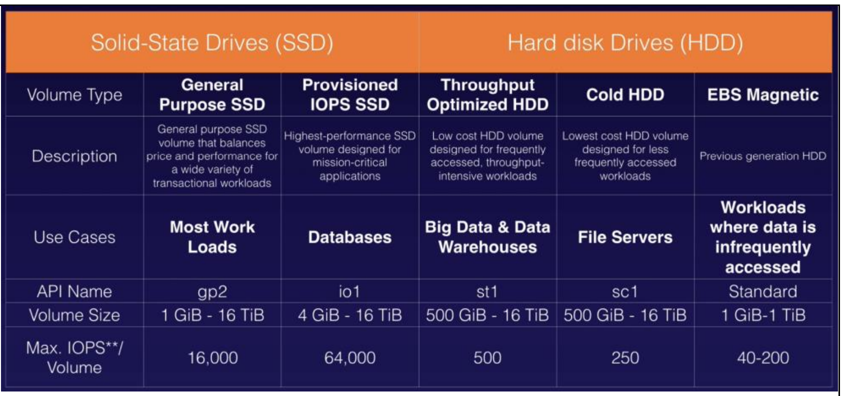
There are three main groups. The spread placement group: Each EC2 instance is in its own rack so it is a more resilient placement. Clustered group: all very close each other.

There are three types of network cards to choose from. The default are virtual cards

Some Ec2 options



Family	Speciality	Use case
F1	Field Programmable Gate Array	Genomics research, financial analytics, real-time video processing, big data etc
I3	High Speed Storage	NoSQL DBs, Data Warehousing etc
G3	Graphics Intensive	Video Encoding/ 3D Application Streaming
H1	High Disk Throughput	MapReduce-based workloads, distributed file systems such as HDFS and MapR-FS
T3	Lowest Cost, General Purpose	Web Servers/Small DBs
D2	Dense Storage	Fileservers/Data Warehousing/Hadoop
R5	Memory Optimized	Memory Intensive Apps/DBs
M5	General Purpose	Application Servers
C5	Compute Optimized	CPU Intensive Apps/DBs
P3	Graphics/General Purpose GPU	Machine Learning, Bit Coin Mining etc
X1	Memory Optimized	SAP HANA/Apache Spark etc
Z1D	High compute capacity and a high memory footprint.	Ideal for electronic design automation (EDA) and certain relational database workloads with high per-core licensing costs.
A1	Arm-based workloads	Scale-out workloads such as web servers
U-6tb1	Bare Metal	Bare metal capabilities that eliminate virtualization overhead



	Solid-State Drives (SSD)		Hard disk Drives (HDD)		
Volume Type	**General Purpose SSD**	**Provisioned IOPS SSD**	**Throughput Optimized HDD**	**Cold HDD**	**EBS Magnetic**
Description	General purpose SSD volume that balances price and performance for a wide variety of transactional workloads	Highest-performance SSD volume designed for mission-critical applications	Low cost HDD volume designed for frequently accessed, throughput-intensive workloads	Lowest cost HDD volume designed for less frequently accessed workloads	Previous generation HDD
Use Cases	**Most Work Loads**	**Databases**	**Big Data & Data Warehouses**	**File Servers**	**Workloads where data is infrequently accessed**
API Name	gp2	io1	st1	sc1	Standard
Volume Size	1 GiB - 16 TiB	4 GiB - 16 TiB	500 GiB - 16 TiB	500 GiB - 16 TiB	1 GiB-1 TiB
Max. IOPS**/ Volume	16,000	64,000	500	250	40-200

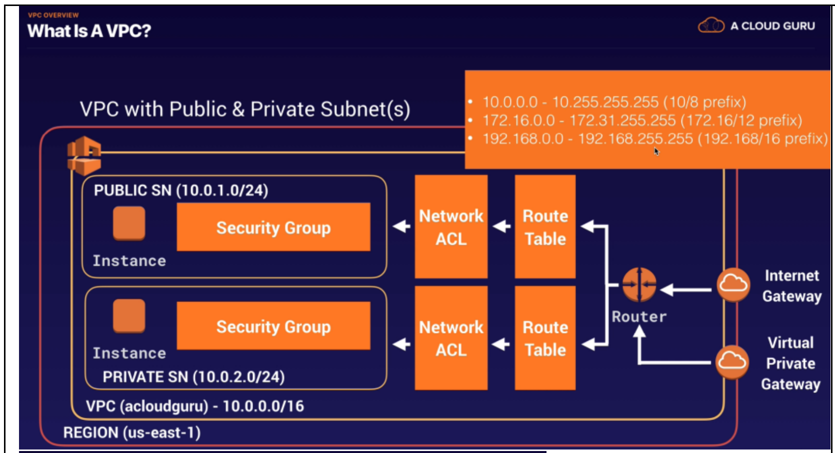
VPC

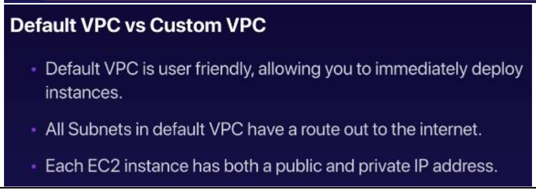
Intro

- It is per region (but you can peer it across regions)
- It has a CIDR block (a set of addresses)
- It can span AZ and local zones
- It can have subnets each with its own subset of CIDR block (CIDR blocks of the subnets cannot overlap).

A virtual private cloud is a logical datacenter in AWS. It lets you provision a logically isolated section of the AWS cloud where you can launch AWS resources in a virtual network that you define. You have complete control over your virtual network, including selection of your own IP address range, creation of subnets and configuration of route tables and network gateways.

Apart from the custom VPCs that you configure there is also a default VPC. By default AWS creates a default VPC and everything you do is done with it. So, when you create an EC2 instance, it is created within the default VPC. Each region has its own default VPC. If you select another region from the AWS gui, you will see the VPCs you have created in that region.





Default VPC vs Custom VPC

- Default VPC is user friendly, allowing you to immediately deploy instances.
- All Subnets in default VPC have a route out to the internet.
- Each EC2 instance has both a public and private IP address.

A VPC example. It has two subnets one public (connected to the internet) and one private. There are two gateways in, one for each subnet. Each request on them goes to a router which routes it accordingly. There is a route table and a network access list (ACL) for each subnet. An ACL is a stateless firewall which means that you have to define both inbound and outbound rules. If you define an inbound rule for a port, you have to create a respective outbound rule too. Security groups instead are stateful. With ACLs you can block specific ports or IPs. The only way to access your private instance from your public one, is through an SSH from the public to the private. In this setup the public instance is called a bastion host.
In general IANA defines three different set of IP ranges that are reserved for private networks.

When you create a VPC a route table an ACL and a security group is created automatically. Then we must create subnets. Then one Internet gateway for our VPC. Then route tables, one for each subnet. Then we can create EC2 instances in our subnets. When you create it there are options to connect it with a specific vpc and a specific subnet of it.

Subnets

A subnet is a range of IP addresses in your VPC. You can launch AWS resources into a specified subnet. Use a public subnet for resources that must be connected to the internet, and a private subnet for resources that won't be connected to the internet. To protect the AWS resources in each subnet, you can use multiple layers of security, including security groups and network access control lists (ACL).

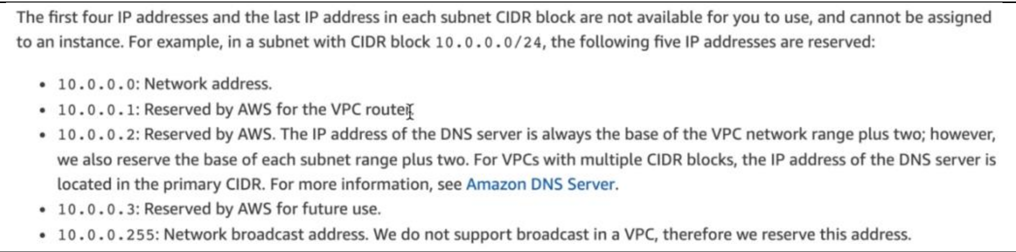
A subnet is a range of IP addresses in your VPC. You can launch AWS resources, such as EC2 instances, into a specific subnet. When you create a subnet, you specify the IPv4 CIDR block for the subnet, which is a subset of the VPC CIDR block. Each subnet must reside entirely within one Availability Zone and cannot span zones.

The allowed block size for a subnet is between a /28 netmask and /16 netmask.

If you create more than one subnet in a VPC, the CIDR blocks of the subnets cannot overlap. For example, if you create a VPC with CIDR block 10.0.0.0/24, it supports 256 IP addresses. You can break this CIDR block into two subnets, each supporting 128 IP addresses. One subnet uses CIDR block 10.0.0.0/25 (for addresses 10.0.0.0 - 10.0.0.127) and the other uses CIDR block 10.0.0.128/25 (for addresses 10.0.0.128 - 10.0.0.255).

There are tools available on the internet to help you calculate and create IPv4 subnet CIDR blocks. You can find tools that suit your needs by searching for terms such as 'subnet calculator' or 'CIDR calculator'.

The first four IP addresses and the last IP address in each subnet CIDR block are not available for your use, and they cannot be assigned to a resource, such as an EC2 instance.

 The first four IP addresses and the last IP address in each subnet CIDR block are not available for you to use, and cannot be assigned to an instance. For example, in a subnet with CIDR block `10.0.0.0/24`, the following five IP addresses are reserved: • `10.0.0.0`: Network address. • `10.0.0.1`: Reserved by AWS for the VPC router. • `10.0.0.2`: Reserved by AWS. The IP address of the DNS server is always the base of the VPC network range plus two; however, we also reserve the base of each subnet range plus two. For VPCs with multiple CIDR blocks, the IP address of the DNS server is located in the primary CIDR. For more information, see Amazon DNS Server. • `10.0.0.3`: Reserved by AWS for future use. • `10.0.0.255`: Network broadcast address. We do not support broadcast in a VPC, therefore we reserve this address.	Have in mind that AWS reserves some IP addresses of your subnet range and assign them to specific resources by default.

A subnet CIDR reservation is a range of IPv4 or IPv6 addresses that you set aside so that AWS can't assign them to your network interfaces.

(Elastic) Network interfaces

An elastic network interface is a logical networking component in a VPC that represents a virtual network card. You can create and configure network interfaces and attach them to instances in the same Availability Zone. AWS ENIs are virtual network cards attached to EC2 instances that help facilitate network connectivity for instances. Having two or more of AWS Network Interface connected to an instance permits it to communicate on two separate subnets.

Each instance has a default network interface, called the primary network interface. You cannot detach a primary network interface from an instance. You can create and attach additional network interfaces. The maximum number of network interfaces that you can use varies by instance type.

Route tables

- Each subnet has a route table. (many subnets can have the same Route table)

A VPC has at least one route table. One route table of a VPC is the main table which simply means that every subnet created for that VPC would be associated with the main route table. Its good practice to have your main table private, not open to the internet and explicitly assign any subnet you want to be public, to a specific open route table. In a route table you define Routes.

A route has a destination and a target. To make one open to the internet you define as Destination 0.0.0.0/0 and Target the internet gateway of your VPC. Destination: you are in the subnet, and you want to reach a specific destination. To do so you must go through the target. Repeat the same with ::/0 as destination for IPv6. you can think of it like this: you are in your subnet and you want a way out of it to the internet. If for example you ping the IPv4 address of some external web server then your requests destination is an IPv4 address (it is caught by the route/rule in the route table) and its target is the internet gateway which will forward it to the internet.

ACL (access control list)

- Subnet level firewall. Each subnet has one ACL (many subnets can have the same ACL)

They are stateless, you have to define both inbound and outbound rules. When you create a VPC a default ACL is created. Every subnet must be connected to an ACL. An ACL can be connected with many subnets. Traffic reach to ACL before it reaches the security groups. Rules are evaluated in order so you must have a deny rule before an allow rule. You have to open ephemeral ports in the outbound rules. You can do more things than security groups, for example block specific IP addresses.

Internet gateways

A gateway is a network "node" that connects two different networks that use different protocols for communicating. In the most basic terms, an Internet gateway is where data stops on its way to or from other networks.

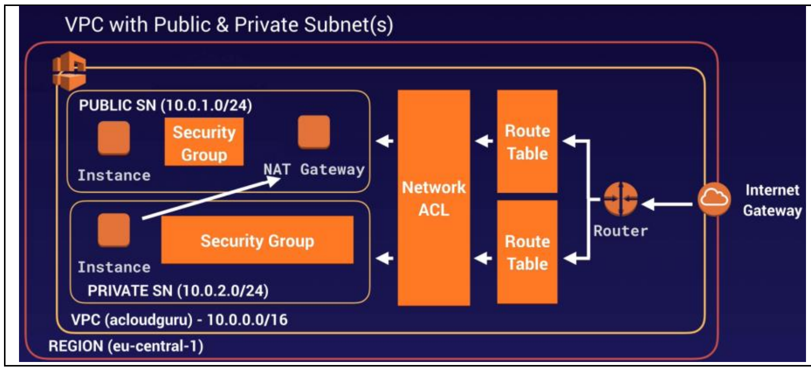
VPC endpoints

They are virtual devices. You can connect from your VPC to other AWS services without going through the internet. Traffic does not leave the Amazon network. For example if you want to communicate with an S3 bucket from your private instance, you normally have to go through the internet using the NAT gateway. What you can do instead is communicate directly from your private instance through a VPC endpoint to your S3 bucket without leaving the amazon internal network.

There are two categories, interface endpoints and vpc gateways.

NAT instances and NAT gateways

- It enables you to enable instances in the private subnet to initiate outbound IPv4 traffic to the Internet or other AWS services
- prevent the instances from receiving inbound traffic initiated by someone on the Internet



You can use a network address translation (NAT) instance in a public subnet in your VPC to enable instances in the private subnet to initiate outbound IPv4 traffic to the Internet or other AWS services, but prevent the instances from receiving inbound traffic initiated by someone on the Internet. Private instances send traffic to the NAT and NAT sends it to the internet. You actually create a route rule in the private subnets route table that has a destination of 0.0.0.0/0 and a target the NAT instance you have created.

A NAT instance is an EC2 instance created from a NAT image. It must be in the public subnet. If you have lots of private EC2 instances all routing traffic to a NAT instance then it might turn out to be a bottleneck so it is not the preferred solution. The preferred one is a NAT Gateway which is a AWS service and is redundant inside the availability zone. It can handle a lot more traffic reliably and with redundancy. A NAT instance (because is a EC2 instance) must be behind a security group. There is no such requirement for a NAT gateway.

Ephemeral ports

An ephemeral port is a random high port the client software picks to communicate to a known service port. For example, if I ssh from my machine to a server the connection would look like: 192.168.1.102:37852 ---> 192.168.1.105:22

22 is the standard SSH port I'm connecting to on the remote machine; 37852 is the ephemeral port used on my local machine. After completion of the communication session the ephemeral ports become available for reuse.

Elastic IP addresses

An Elastic IP address is a static, public IPv4 address designed for dynamic cloud computing. You can associate an Elastic IP address with any instance or network interface for any VPC in your account. With an Elastic IP address, you can mask the failure of an instance by rapidly remapping the address to another instance in your VPC. Note that the advantage of associating the Elastic IP address with the network interface instead of directly with the instance is that you can move all the attributes of the network interface from one instance to another in a single step

Flow logs

A service that stores all traffic from and to your VPC. Actually you can configure it to one of three levels. At VPC level, to subnet level all the way down to network interface level (of particular instances as I understand).

Have in mind that there are some things not logged to flow logs. For example the traffic generated when your instances contact the amazon DNS server, or traffic for windows license activation, DHCP traffic.

Bastions

A bastion host is a special-purpose computer on a network specifically designed and configured to withstand attacks. The computer generally hosts a single application, for example a proxy server, and all other services are removed or limited to reduce the threat to the computer.

In an Amazon Web Services (AWS) context, a bastion host is defined as "a server whose purpose is to provide access to a private network from an external network, such as the Internet. They are instances that sit within your public subnet and are typically accessed using SSH or RDP. Once remote connectivity has been established with the bastion host, it then acts as a 'jump' server, allowing you to use SSH or RDP to log in to other instances (within private subnets) deeper within your VPC. When properly configured through the use of security groups and Network ACLs (NACLs), the bastion essentially acts as a bridge to your private instances via the internet. There are community AMIs for bastion hosts. To SSH to your private instances you first SSH to your bastion host and from there you ssh to your private instance.

VPC peering

VPC peering. The concept is that you connect two VPCs and the instance of the first one see the instances of the second one as if they were in the same VPC.

You can peer any two VPCs in different Regions, as long as they have distinct, non-overlapping CIDR blocks.

You can peer with a virtual private cloud (VPC) in another AWS account by using AWS::EC2::VPCPeeringConnection. This creates a networking connection between two VPCs that enables you to route traffic between them so they can communicate as if they were within the same network.

RDS

RDS run on VMs to which you don't have access to for security reasons. You can't login to them. AWS has the responsibility to maintain the OS and the DB on them. There are two main settings:

- Multi-AZ

On multi AZ if your db in the first AZ fails AWS will automatically point the DNS record that your EC2s use to connect to the database to the IP of the second database in the other Availability Zone. When your production db is written to, this write will automatically be synchronized to the stand by db (synchronous replication). this option enhances reliability, not performance. For performance improvement you need read replicas instead of multi AZ.

How RDS multi availability zone model works. In an Amazon RDS Multi-AZ deployment, Amazon RDS automatically creates a primary database (DB) instance and synchronously replicates the data to an instance in a different AZ. When it detects a failure, Amazon RDS automatically fails over to a standby instance without manual intervention.

- Read Replicas

In this case you have a primary database and every write on it is replicated to a replica database. If the primary fails you have to manually create a new DNS entry that points to the IP of the replica and make your EC2 instances use that. The benefit is that for read requests, you can make half of your EC2 instances use the primary db and the other half the replica. you can have as many as 5 replicas. The production db is asynchronously replicated to multiple copies (asynchronous replication). each replica has its own DNS endpoint. You can have read replicas that have multi-AZ deployment to enhance their reliability. You can have read replicas of read replicas. In order to be able to create read replicas you need to enable backups first.

You can have both multi-AZ and read replicas at the same time.

Backups

Automated backups once per day, but stores your transactions for the day and reapplies it to the last backup. Stored in s3. the other option is manual snapshots. Notice that whenever you restore a backup the new database will be a new RDS instance with a new DNS endpoint.

Encryption at rest (as it is called) is supported for the db read replicas, snapshots and backups. It is done using the Key Management Service (KMS).

DynamoDB

(No SQL db)

Stored on SSD, across 3 distinct data centers. There are two modes of operation. Eventual consistent reads and strongly consistent reads. DynamoDB needs at maximum 1 sec to apply a write to all nodes of the db. If you are ok with reading a write 1 second later and any read before that gives the previous state, then you use the first option. Otherwise you use the second option which will return the updated value even if it is only written in one node (reflects all writes that received a successful response prior to the read)

Redshift (warehouse service)

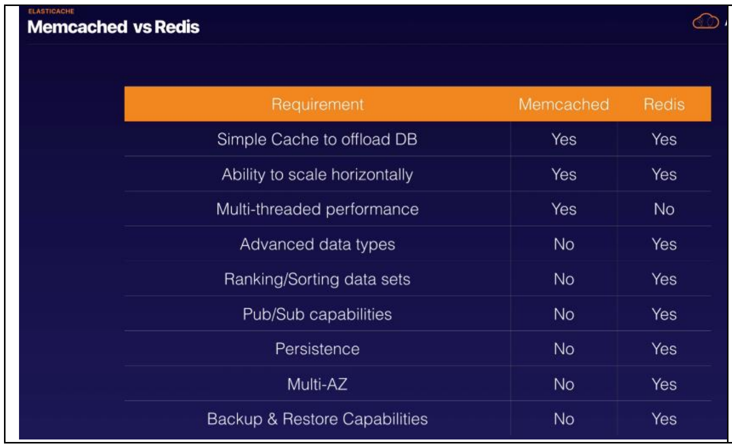
$1k/TB/year

Single node or multi node. In multi mode you have a leader node that manages client connections and receives queries and compute nodes that store data and perform queries and computations. Up to 128 nodes. You are charged for the compute nodes hours. AWS will always try to have three copies of your data, the original and replica on the compute nodes and a backup in S3. snapshots are asynchronously replicated to S3 in another region for disaster recovery.

OLTP vs OLAP

Online transaction processing vs online analytical processing. OLTP queries are the kind of queries performed on a typical SQL database for example retrieve the order with id 12345. OLAP transactions are different. For example you create a query asking for the net profit of the EMEA region. This is translated to many queries and calculation on the results. Executing these queries on your relational db would be very expensive and would affect its overall performance. This is the reason for storing your data in data warehouses too, which are built for storing large amounts of data and there are tools that perform OLAP queries on them. Typically you have a relational database for your web service which contains only the necessary data your web application needs and another data warehouse solution for all types of data both your relational db data and additional data like users clicks etc. Then for business intelligence you can query the data warehouse instead of the relational db, allowing the later to perform well for the web application users.

ElastiCache



ELASTICACHE

Memcached vs Redis

Requirement	Memcached	Redis
Simple Cache to offload DB	Yes	Yes
Ability to scale horizontally	Yes	Yes
Multi-threaded performance	Yes	No
Advanced data types	No	Yes
Ranking/Sorting data sets	No	Yes
Pub/Sub capabilities	No	Yes
Persistence	No	Yes
Multi-AZ	No	Yes
Backup & Restore Capabilities	No	Yes

Uses one of two in memory caching engines, redis or memcached. Redis is multi AZ, it supports backups and restore

Available only from within the VPC

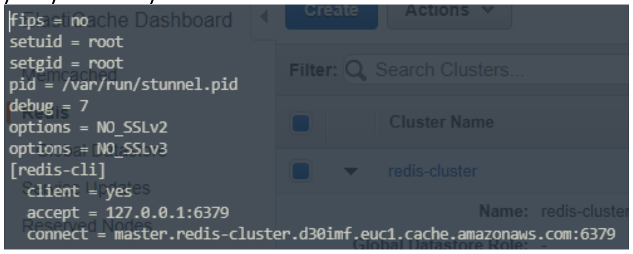
By default you can only access it by ec2 instances running in the same vpc with the elastic cache cluster. Even if the security group has port 6379 open for all sources (0.0.0.0) it means all sources from inside the vpc! To connect from outside you need to do it through a VPN. I found that the easiest way connect to your redis cluster from within your ec2 instance is to use docker to download redis container and then run the redis-cli command on it. (redis-cli is not installed in the ec2 node, so you would have to install it. but using a container seems easier).

- ➢ docker run -d redis
- ➢ docker exec -it e0c061a5700bfa400f8f24b redis-cli -h host -p port

Connecting to it with encryption in transit enabled

If you have encryption in transit enabled in your elasticache cluster you can't connect to it with redis-cli since it can;t handle SSL/TLS connections. To do it you have to use stunnel

/etc/stunnel/redis-cli.conf



```
fips = no
setuid = root
setgid = root
pid = /var/run/stunnel.pid
debug = 7
options = NO_SSLv2
options = NO_SSLv3
[redis-cli]
  client = yes
  accept = 127.0.0.1:6379
  connect = master.redis-cluster.d30imf.euc1.cache.amazonaws.com:6379
```

In stunnel you define a redis cli group where you specify the remote redis server. Notice that in the accept instruction you define localhost. This means that you connect to redis by connecting to your localhost (and stunnel forwards the connection using encryption).

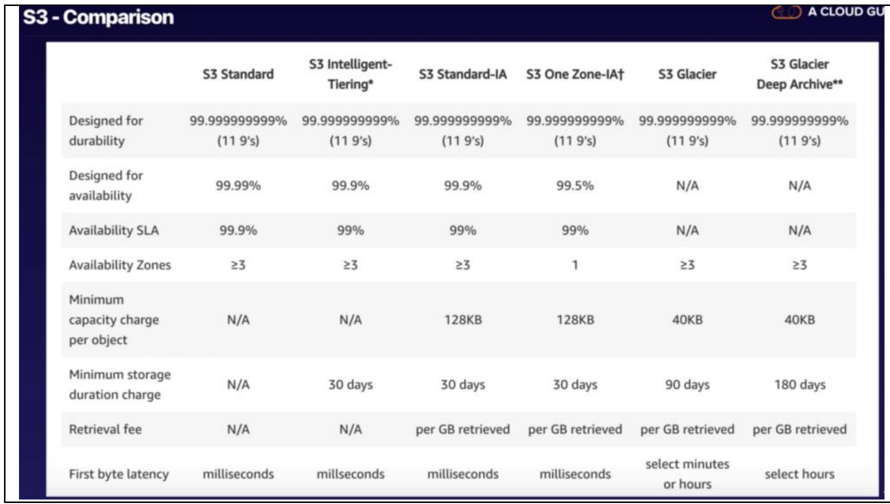
> redis-cli -h localhost -p 6379 -a psw ping

With > sudo netstat -tulnp | grep -i stunnel you can see the stunnels you have defined.

S3

Block based storage vs object based storage

S3 is an object storage. An object has a key (can be the file name), a value (the sequence of bytes), a version id (you can upload the same file again and it will be stored with a different version prefix), metadata about the data and subresources (access control lists and torrent) ACL define permissions. You can't store a db or an OS. You need block based storage for that. Versioning makes you use more space since all versions are stored.



S3 - Comparison

	S3 Standard	S3 Intelligent-Tiering*	S3 Standard-IA	S3 One Zone-IA†	S3 Glacier	S3 Glacier Deep Archive**
Designed for durability	99.999999999% (11 9's)	99.999999999% (11 9's)	99.999999999% (11 9's)	99.999999999% (11 9's)	99.999999999% (11 9's)	99.999999999% (11 9's)
Designed for availability	99.99%	99.9%	99.9%	99.5%	N/A	N/A
Availability SLA	99.9%	99%	99%	99%	N/A	N/A
Availability Zones	≥3	≥3	≥3	1	≥3	≥3
Minimum capacity charge per object	N/A	N/A	128KB	128KB	40KB	40KB
Minimum storage duration charge	N/A	30 days	30 days	30 days	90 days	180 days
Retrieval fee	N/A	N/A	per GB retrieved	per GB retrieved	per GB retrieved	per GB retrieved
First byte latency	milliseconds	millseconds	milliseconds	milliseconds	select minutes or hours	select hours

S3 storage is extremely durable (multiple AZ), designed to sustain the loss of 2 facilities concurrently.
S3 is a universal namespace. Names must be unique globally. Buckets are like folders. When you upload successfully you get a Http 200 response.
Supports MFA (multi factor authentication) delete

S3 has different storage Tiers based on your needs. Glacier tiers is for long storage but retrieval is slow (eg. 12 hours)

Encryption on S3

There is encryption in transit with SSL/TLS and at rest. Encryption at rest can be either server side (files are encrypted by AWS) or client side (the user encrypts the files and then uploads them to S3)

Access to the S3 API is governed by an Access Key ID and a Secret Access Key. The access key identifies your S3 user account while the secret key is akin to a password and should be kept secret.

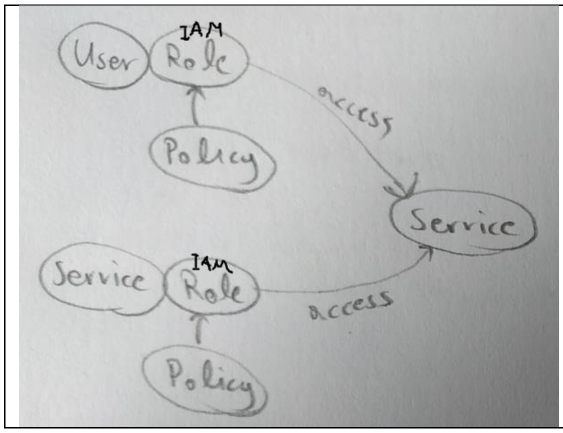
Once uploaded your application can reference its assets by copying their public URLs (such as http://s3.amazonaws.com/bucketname/filename) and pasting them directly into your app's views or HTML files. These files will now be served directly from S3, freeing up your application to serve only dynamic requests.

Cognito

IAM

There are two different types of users in AWS. You are either the account owner (root user) or you are an AWS Identity and Access Management (IAM) user. The root user is created when the AWS account is created. IAM users are created by the root user or an IAM administrator for the account.

Identity Access Management

 User IAM Role Policy access Service Service IAM Role access Policy	There are four main objects. **Users**, **groups**, **policies**, **roles** Policies are json docs with the specific permissions. You assign policies to groups and roles. Roles are assigned to aws services (to be able to access other aws services for example Allow EC2 instance to access an S3 bucket). The root account is the user with email the one you used to register to AWS. It is the super user for AWS. All users and other stuff that you create in IAM are global, applied to all regions. (you don't create users for specific regions)

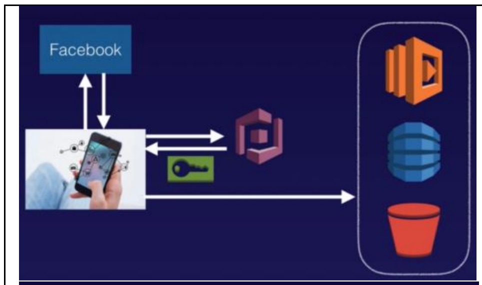
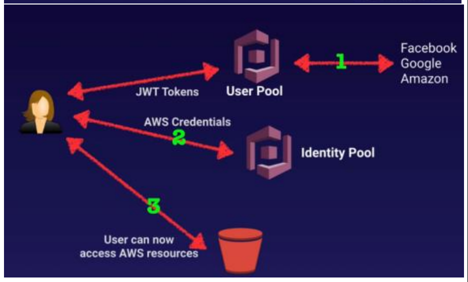
Cognito

https://www.youtube.com/watch?v=QEGo6ZoN-ao excellent 20 mins

it is an authentication and authorization service so that you don't have to build your own. you don't have to set up storage, compute it provides them transparently. It is also a highly available service by default. It has two main concepts. User pools and identity pools. The first one handles authentication and the second one authorization. You can configure it to be the authentication service or you can set up 3rd party auth services like google etc. it offers a lot of triggers (post signup etc.) so that you can write custom code on them executed in lambdas. For example you can assign custom tags to users like admin, teacher, student etc. Then using the Identity pool, you can set up IAM roles for each tag. This way you can restrict access to specific aws services per user tag. When a user is authenticated he receives a token (maybe jwt) that he sends in each subsequent request. It is integrated with AWS API gateway, so when an authenticated request comes in with a specific cognito token (jwt for example), gateway checks with cognito to get the authorization details and block or allow access to specific services.

Aws role restriction vs application role restriction

Notice the difference between aws role restriction and application specific role restriction (for example in Django rest). In the first case you restrict access to entire aws services. In the later case you can restrict access in application specific logic. The user can have access to the application but not all parts of it for example.

  Facebook Google Amazon 1 JWT Tokens User Pool AWS Credentials 2 Identity Pool 3 User can now access AWS resources	It offers social authentication for your application out of the box, without you having to write code for it. It acts as an identity broker between the identity provider and AWS. Successful authentication generates a JWT. A user logs in with facebook. Fb sends him a token, he sends it to cognito which sends him back a key (that maps to an IAM role) and provides access to your application's AWS services Cognito keeps also track of the association between user identity and the different devices they sign in from. For example the user does some action that triggers a notification to him. Cognito will use SNS push notifications to send it to his laptop, phone.

High Availability (HA)

Load balancers

There are three types. The application LB (ALB), the network LB (NLB) and the classic or elastic LB (ELB). The application load balancer works on layer 7, so it has access to your application details which means that it is very versatile since you can define load balancing rules based on your application nuances. The network LB operates in layer 4, but is extremely efficient, it can route millions of requests. The classic LB is a legacy one only used now for simple cases like a round robin.

Have in mind that the load balancer has its own internal IP and since it will route the traffic to your EC2 your web server will record the load balancers IP and not your real user IP. For this reason you can access the x-forwarded-for http header to learn the user's IP. You can have internal load balancer (inside private subnets). load balancers can use the health checks on your instances and stop transferring load to a dead instance. ALBs and ELBs don't have a fixed IP so what you get is a DNS name for them.

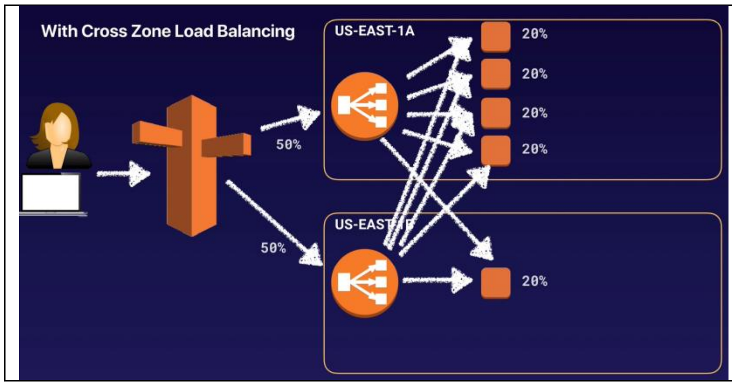
With an application load balancer you can create sophisticated rules like if then statements. It uses target groups

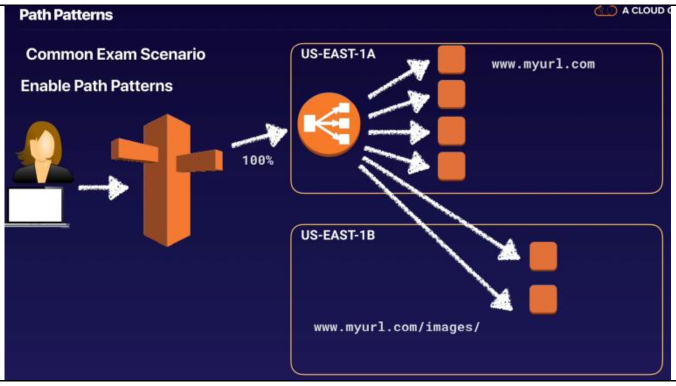
With a network load balancer you can have a fixed ip address for your load balancer or ultra high performance.

Target groups

You can assign your instances to target groups and have a load balancer target requests to the target group using target groups settings.

Have in mind that you can enable Sticky sessions on your LB so that users stick to the same EC2 instance during a session. There is also cross zone load balancing and path based routing options for the ALBs.

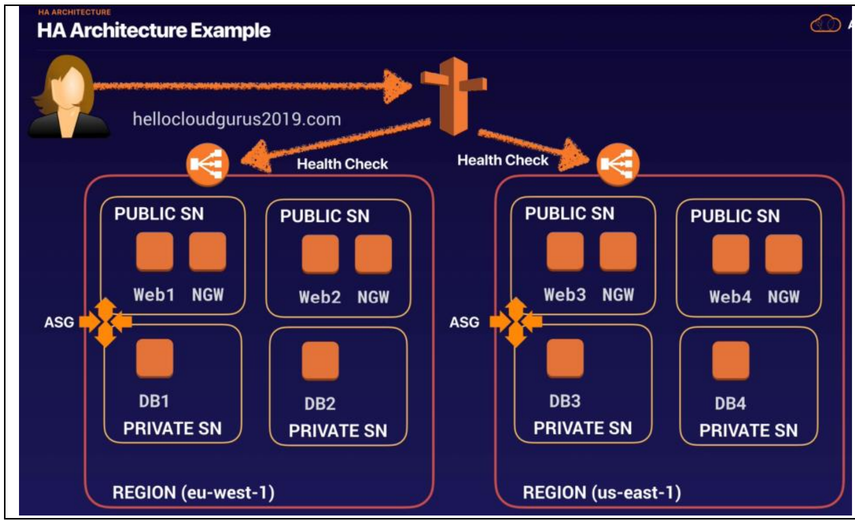




Autoscaling

You can define an autoscaling policy for your architecture where if a metric reaches a threshold AWS automatically creates more instances (and stops them using another metric?). to create an autoscaling system, you first need to create a launch configuration where you just define the settings for the instances you want to create, without actually creating them. (you can also create an AMI image (Amazon Machine Image) from one EC2 instance and use it as the base image for the launch configuration). Then you define an autoscaling group (ASG) for that launch configuration. You define the min and max number of active instances and the metric and threshold. If an instance dies for a reason autoscaling will detect that and launch a new instance based on your configuration settings. In general its good practice to use an autoscaling group whenever you have a fleet of identical instances like workers in a queue for example.

High Availability architectures



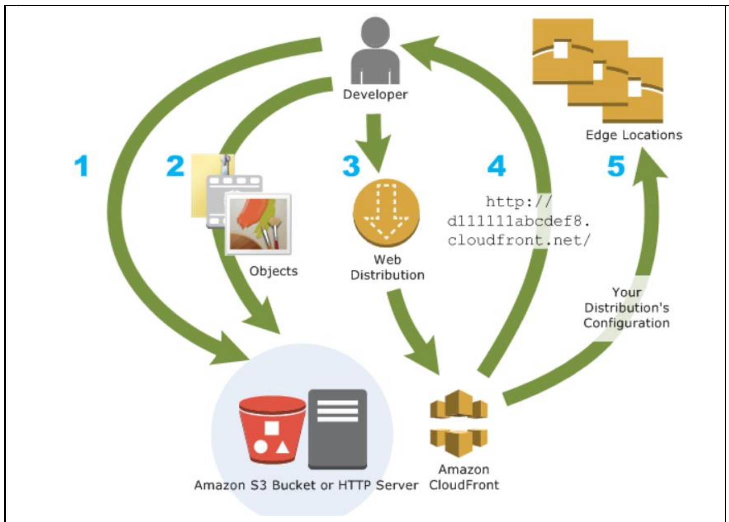
This is an example of a HA architecture. You have a duplicate system in two regions while each subnet is on its own AZ within a region. You have also autoscaling group (ASG) configured. You can recover both from a region fail and from an AZ fail.

Have in mind simian army a service of Netflix. One functionality is to create a "monkey" that destroys one AZ, a gorilla that does more damage etc. This way they test their HA.

Cloudfront

Amazon CloudFront is a web service that speeds up distribution of your static and dynamic web content, such as .html, .css, .js, and image files, to your users. CloudFront delivers your content through a worldwide network of data centers called **edge locations**. When a user requests content that you're serving with CloudFront, the request is routed to the edge location that provides the lowest latency (time delay), so that content is delivered with the best possible performance.

- If the content is already in the edge location with the lowest latency, CloudFront delivers it immediately.
- If the content is not in that edge location, CloudFront retrieves it from an origin that you've defined—such as an Amazon S3 bucket, a MediaPackage channel, or an HTTP server (for example, a web server) that you have identified as the source for the definitive version of your content.

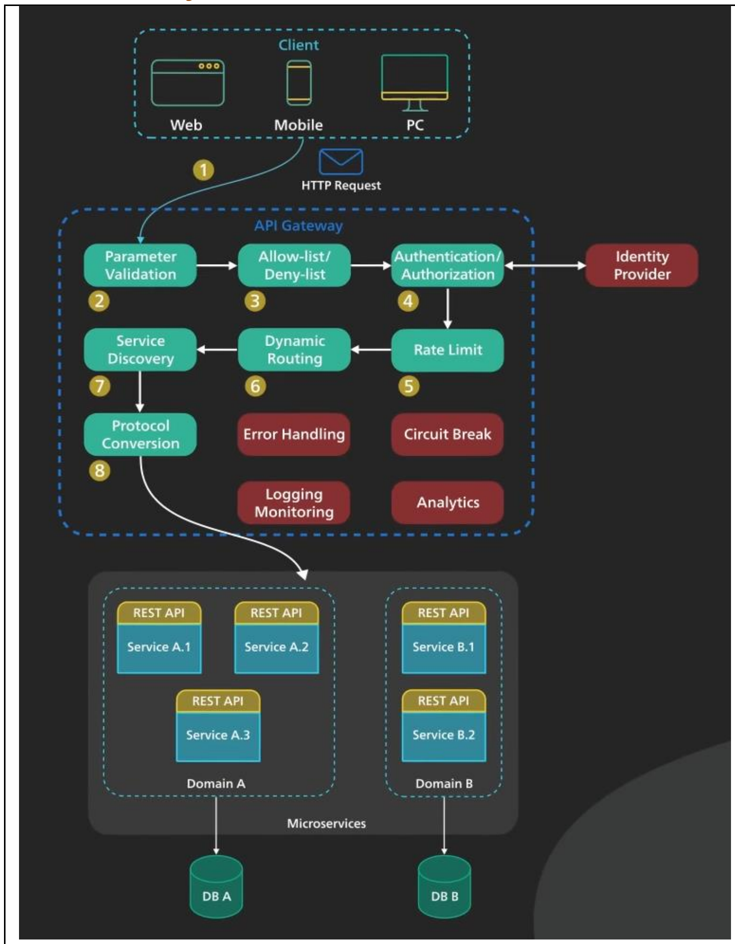

Developer
Edge Locations
1
2
3
4
5
Objects
Web Distribution
http://d111111abcdef8.cloudfront.net/
Your Distribution's Configuration
Amazon S3 Bucket or HTTP Server
Amazon CloudFront

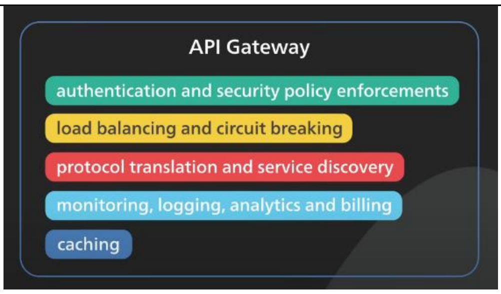
You create **a CloudFront distribution** to tell CloudFront where you want content to be delivered from, and the details about how to track and manage content delivery (me: something like a configuration file). Then CloudFront uses computers—edge servers—that are close to your viewers to deliver that content quickly when someone wants to see it or use it.

As you develop your website or application, you use the domain name that CloudFront provides for your URLs. For example, if CloudFront returns d111111abcdef8.cloudfront.net as the domain name for your distribution, the URL for logo.jpg in your Amazon S3 bucket (or in the root directory on an HTTP server) is https://d111111abcdef8.cloudfront.net/logo.jpg.

Or you can set up CloudFront to use your own domain name with your distribution. In that case, the URL might be https://www.example.com/logo.jpg.

API Gateway





API Gateway

- authentication and security policy enforcements
- load balancing and circuit breaking
- protocol translation and service discovery
- monitoring, logging, analytics and billing
- caching

2. It validates the http request
3. It checks the ip address or other http headers against its allow/deny list. At this step It can apply rate limiting too, against ip addresses for example. (but not based on users, user roles etc. this information is received later)
4. It passes the request to an identity provider for authorization and authentication. It receives an authenticated session back with the scope of what the request is allowed to do.
5. Rate limit based on authenticated session
6. And 7. With a help of a service discovery component it locates the appropriate service to handle the request by path matching.
8. It transforms the http request to the appropriate protocol the service expects. For example it could be grpc. When the response comes back it transforms it back to the public facing protocol.

It can track errors. It can implement a circuit braking functionality to protect services from overloading. It can be deployed in multiple regions to improve availability (independently from the backend services that can be just in one region if we don't need them in multiple)

Service registry and discovery service. Microservices register to this service and they can discover the "location" of other services.

API gateway vs load balancer

In a region, the request goes first to the api gateway, then it points to an api that might sit behind a load balancer. If you have many regions you have DNS load balancing.

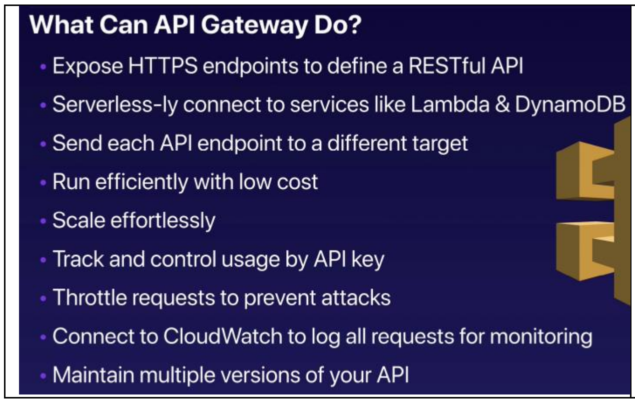
API gateway predominately does API management and provides various other key features such as IAM (Identity and Access Management), Rate limiting, circuit breakers. Hence, it mainly eliminates the need to implement API-specific code for functionalities such as security, caching, throttling, and monitoring for each of the microservice. Microservices typically expose the REST APIs for use in front ends, other microservices and 3rd party apps with help of API gateway.

Can API gateway replace load balancer?

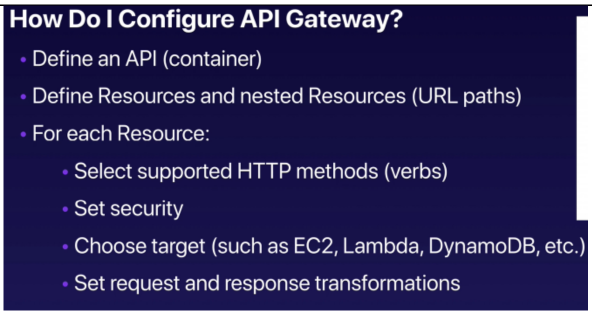
TL;DR: yes, API Gateway can replace what a Load Balancer would usually provide, with a simpler interface and many more features on top of it. The downside is that it doesn't come cheap.

Api gateway and custom django api

Me: You can create your microservices with Django rest framework and deploy them on ec2 instances (or on containers infrastructure like ECS, EKS etc.) and put them behind api gateway. Then in the api gateway you define the paths for each service (all requests to *sales/api* are routed to sales service, *inventory/api* to inventory service etc.) and the individual services handle the fine-grained routing like sales/api/sale/1 etc. you can have load balancer between the api gateway and your services. This way you route *sales/api* to the sales load balancer and the load balancer distributes the requests among several sales microservices.


What Can API Gateway Do?

- Expose HTTPS endpoints to define a RESTful API
- Serverless-ly connect to services like Lambda & DynamoDB
- Send each API endpoint to a different target
- Run efficiently with low cost
- Scale effortlessly
- Track and control usage by API key
- Throttle requests to prevent attacks
- Connect to CloudWatch to log all requests for monitoring
- Maintain multiple versions of your API


How Do I Configure API Gateway?

- Define an API (container)
- Define Resources and nested Resources (URL paths)
- For each Resource:
  - Select supported HTTP methods (verbs)
  - Set security
  - Choose target (such as EC2, Lambda, DynamoDB, etc.)
  - Set request and response transformations

You can enable API caching (caching responses)

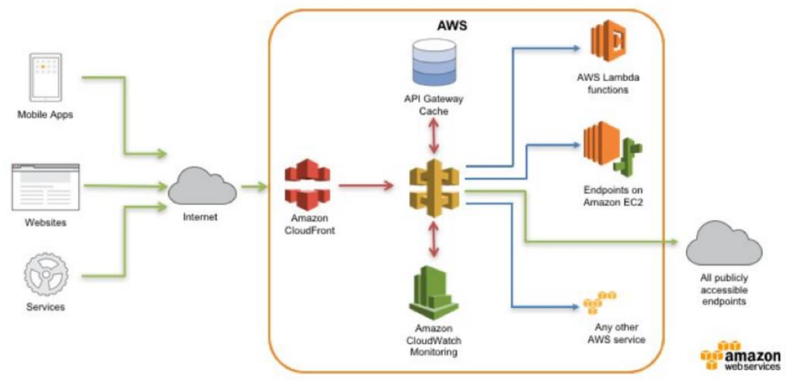
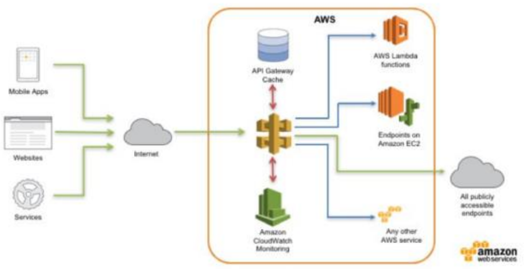
CDN and API gateway

CDN in front of API gateway, this is a common pattern for API Protection in AWS.

In general there are two API endpoint "setups":

1. Regional API endpoint
2. Edge optimized endpoint

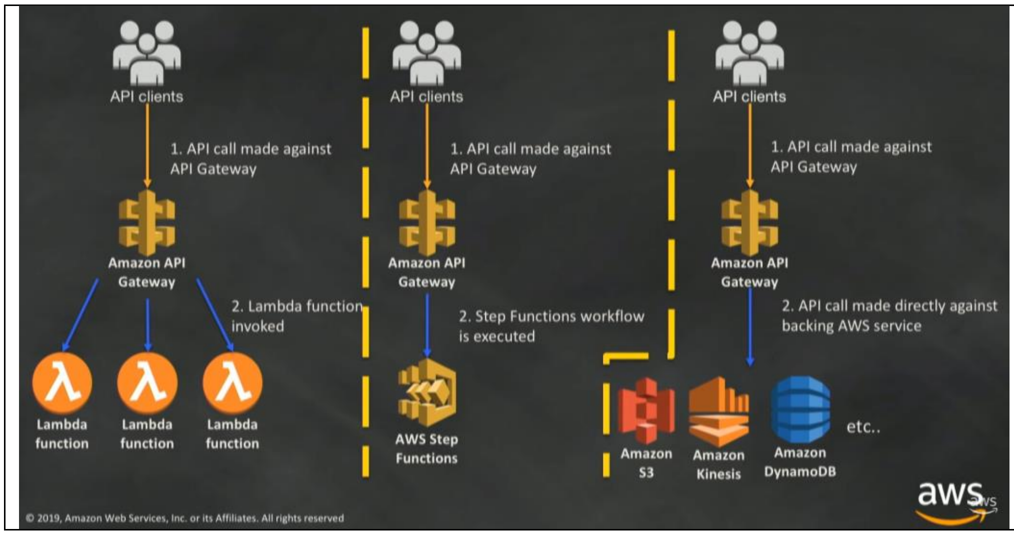
Here is a good article by AWS in setting up CloudFront and WAF with AWS API Gateway, though the same principals apply to any API Gateway: https://aws.amazon.com/blogs/compute/protecting-your-api-using-amazon-api-gateway-and-aws-waf-part-i/

Edge optimized endpoint	Regional API endpoint Cloudfront is missing
 AWS Mobile Apps Websites Services Internet Amazon CloudFront API Gateway Cache Amazon CloudWatch Monitoring AWS Lambda functions Endpoints on Amazon EC2 Any other AWS service All publicly accessible endpoints amazon web services	 AWS Mobile Apps Websites Services Internet API Gateway Cache Amazon CloudWatch Monitoring AWS Lambda functions Endpoints on Amazon EC2 Any other AWS service All publicly accessible endpoints amazon web services

For the regional API endpoint, your customers access your API from the same Region in which your REST API is deployed. This helps you to reduce request latency (me: because your users are in the same region with your API service) and particularly allows you to add your own content delivery network, as needed.

In the edge-optimized API endpoint where your API clients access your API through a CloudFront distribution created and managed by API Gateway.

Examples

 API clients 1. API call made against API Gateway Amazon API Gateway 2. Lambda function invoked Lambda function Lambda function Lambda function API clients 1. API call made against API Gateway Amazon API Gateway 2. Step Functions workflow is executed AWS Step Functions API clients 1. API call made against API Gateway Amazon API Gateway 2. API call made directly against backing AWS service Amazon S3 Amazon Kinesis Amazon DynamoDB etc.. © 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved aws	

Api gateway endpoints

- Edge optimized endpoint

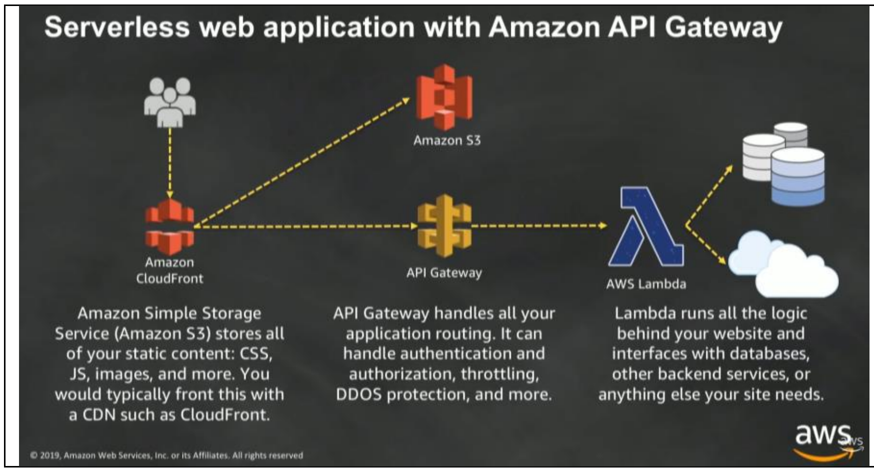
It automatically integrates api gateway behind cloudfront CDN

- Regional endpoint

You set up an api gateway in one region, and you can manually set up a cdn in front of it.

- Private endpoint

Only for services inside your VPC



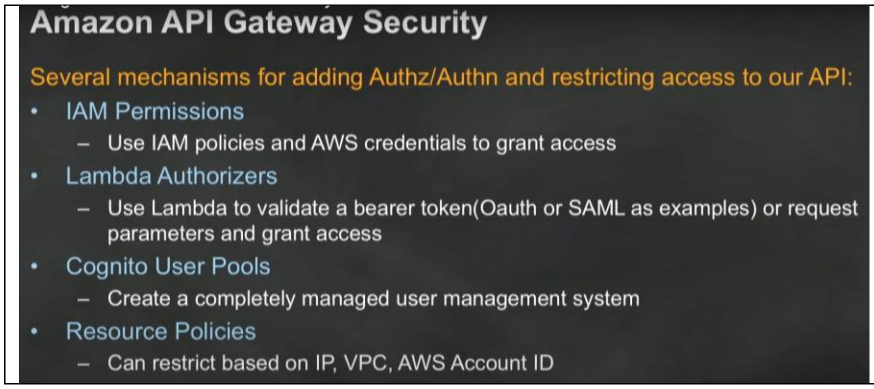
Serverless web application with Amazon API Gateway

Amazon Simple Storage Service (Amazon S3) stores all of your static content: CSS, JS, images, and more. You would typically front this with a CDN such as CloudFront.

API Gateway handles all your application routing. It can handle authentication and authorization, throttling, DDOS protection, and more.

Lambda runs all the logic behind your website and interfaces with databases, other backend services, or anything else your site needs.

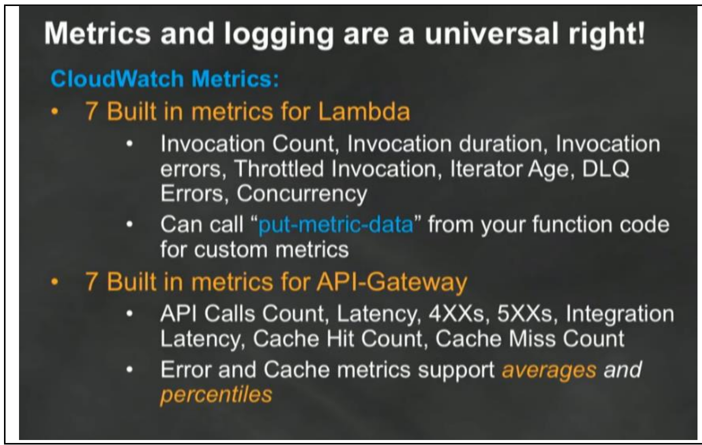
© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved



Amazon API Gateway Security

Several mechanisms for adding Authz/Authn and restricting access to our API:

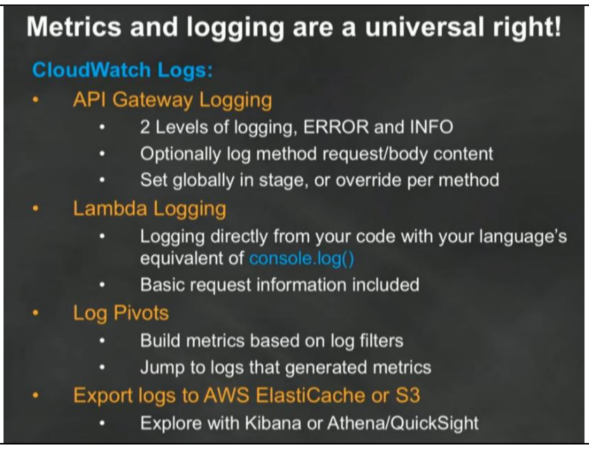
- IAM Permissions
  - Use IAM policies and AWS credentials to grant access
- Lambda Authorizers
  - Use Lambda to validate a bearer token(Oauth or SAML as examples) or request parameters and grant access
- Cognito User Pools
  - Create a completely managed user management system
- Resource Policies
  - Can restrict based on IP, VPC, AWS Account ID



Metrics and logging are a universal right!

CloudWatch Metrics:

- 7 Built in metrics for Lambda
  - Invocation Count, Invocation duration, Invocation errors, Throttled Invocation, Iterator Age, DLQ Errors, Concurrency
  - Can call "put-metric-data" from your function code for custom metrics
- 7 Built in metrics for API-Gateway
  - API Calls Count, Latency, 4XXs, 5XXs, Integration Latency, Cache Hit Count, Cache Miss Count
  - Error and Cache metrics support *averages and percentiles*

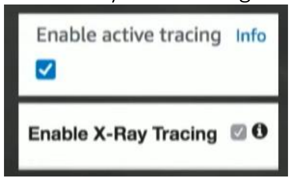


Metrics and logging are a universal right!

CloudWatch Logs:

- API Gateway Logging
  - 2 Levels of logging, ERROR and INFO
  - Optionally log method request/body content
  - Set globally in stage, or override per method
- Lambda Logging
  - Logging directly from your code with your language's equivalent of console.log()
  - Basic request information included
- Log Pivots
  - Build metrics based on log filters
  - Jump to logs that generated metrics
- Export logs to AWS ElastiCache or S3
  - Explore with Kibana or Athena/QuickSight

AWS X-Ray is also integrated with api gateway.



Enable active tracing Info

☑

Enable X-Ray Tracing ☑

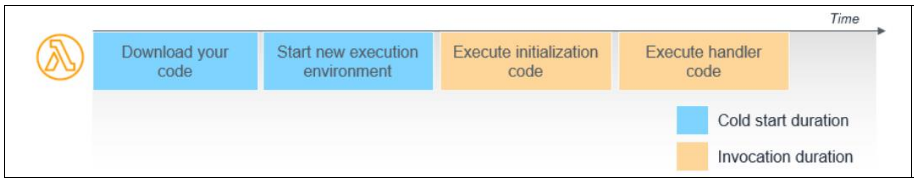
Lambdas

1. Be warned, lambdas have a timeout of 15min max, so anything longer than that should not be serverless

2. Usually, a cold start takes 3-5 seconds, but when Lambda is deployed inside of VPC, it takes up to 10-15 seconds. When AWS Lambda connects to storage services like DocumentDB or RDS, this is the only option.
3. There is a concurrency limit for how many concurrent lambdas you can spawn in a region. By default, it is 1000. You can change it. if the limit is reached any new invocation would be queued and will be executed when there is free room.

You can invoke Lambda functions directly using the Lambda console, a function URL HTTP(S) endpoint, the Lambda API, an AWS SDK, the AWS Command Line Interface (AWS CLI), and AWS toolkits.

Cold and warm starts



When the Lambda service receives a request to run a function via the Lambda API, the service first prepares an execution environment. During this step, the service downloads the code for the function, which is stored in an internal Amazon S3 bucket (or in Amazon Elastic Container Registry if the function uses container packaging). It then creates an environment with the memory, runtime, and configuration specified. Once complete, Lambda runs any initialization code outside of the event handler before finally running the handler code.

In this diagram, the first two steps of setting up the environment and the code are frequently referred to as a "**cold start**". You are not charged for the time it takes for Lambda to prepare the function but it does add latency to the overall invocation duration.

After the execution completes, the execution environment is frozen. To improve resource management and performance, the Lambda service retains the execution environment for a non-deterministic period of time. During this time, if another request arrives for the same function, the service may reuse the environment. This second request typically finishes more quickly, since the execution environment already exists and it's not necessary to download the code and run the initialization code. This is called a "**warm start**".

There are open source warming libraries that ping your lambdas but even this doesn't guarantees warm start. This approach might be adequate for development and test environments, or low-traffic or low-priority workloads.

You can call a lambda from another lambda. The first one has to have a "invoke other lambda function" role.

It is an AWS service that let's you deploy functions without having to worry about creating a runtime environment for the code to execute in. AWS takes care of provisioning and managing the servers that you use to run the code. When you call that function amazon will scale up a runtime environment, execute your function and then scale down the runtime environment.

You upload your code which can be executed by specific triggers or it can be scheduled jobs. Your uploaded code is now called a lambda function. It functions as a service.

You can use lambda in the following ways:

1. As an event driven compute service where lambda runs your code in response to events.
2. As a compute service to run your code in response to HTTP requests using Amazon API Gateway (or API calls made using AWS sdks)

Benefits

- NO SERVERS TO MANAGE

AWS Lambda automatically runs your code without requiring you to provision or manage servers. Just write the code and upload it to Lambda.

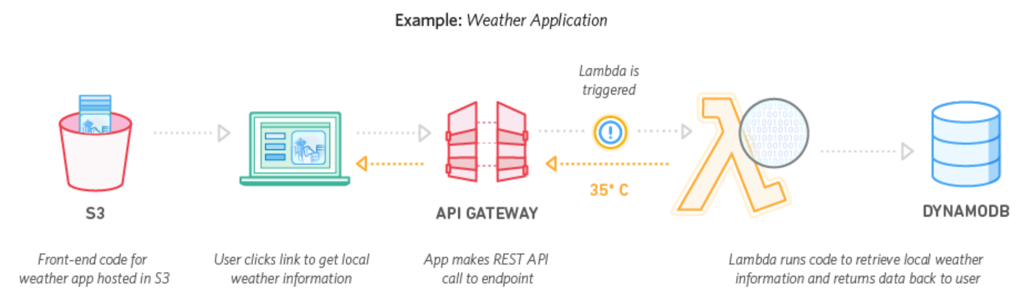
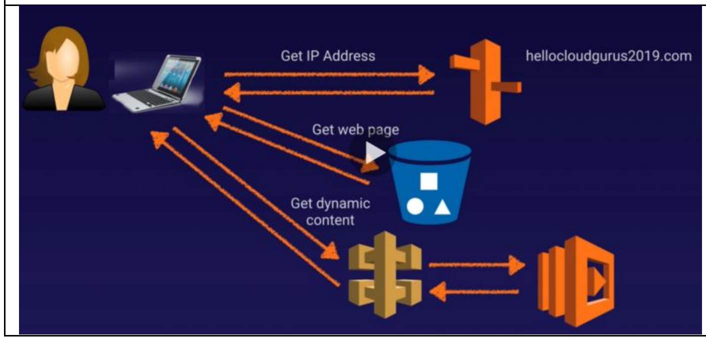
- CONTINUOUS SCALING

AWS Lambda automatically scales your application by running code in response to each trigger. Your code runs in parallel and processes each trigger individually, scaling precisely with the size of the workload. For one million triggers you would have one million lambda functions running in parallel.

- SUBSECOND METERING

With AWS Lambda, you are charged for every 100ms your code executes and the number of times your code is triggered. You don't pay anything when your code isn't running. Very cheap in relation with traditional EC2 instances.

Two very similar examples for a serverless Web Application using lambda

 Example: Weather Application Lambda is triggered 35° C S3 API GATEWAY DYNAMODB Front-end code for weather app hosted in S3 User clicks link to get local weather information App makes REST API call to endpoint Lambda runs code to retrieve local weather information and returns data back to user	Notice that the web site in these examples is a static web site, stored in s3. All dynamic content is retrieved by lambda functions.
 Get IP Address hellocloudgurus2019.com Get web page Get dynamic content	

Serveless definition

A serverless service is one for which you don't have to manage a server at all. Anything that has to do with the server and the underlying OS etc. Must be out of your thinking. So an RDS database despite the fact that it is a managed service, there is still the component of the underlying os, where AWS has to maintain it and update it etc so you might have downtimes as a consequence. So it is not serverless.

Other Services

AWS Command Line

You enable programmatic access to an EC2 instance. You create an access key pair (not the ssh keys). You SSH to it, specify the access key and then you can use the aws cli. All commands start with *>aws name_of_service command* for example: >aws s3 ls

Notice that the access keys are stored in your EC2 instance in a .aws folder. But you can use roles instead of access keys and that is more secure since you don't have to store anything in your EC2. you create a role with admin rights attach it to your EC2 and done.

```
> aws s3 cp local_file s3://s3_bucket_name
```

Cloudwatch

Monitor your AWS resources and your applications running in AWS. Monitors performance. Its not free. You activate it when you create an EC2 instance. You can set up alerts on metrics, so you are notified. You can send your logs to cloudwatch.

AWS Amplify

Easily host and serve your react app (or other static app) through CDN. You connect your repo with source code and amplify will build it, and deploy it. (Auto ci/cd).

"Deploy and host your app using either Amplify Console or Amazon CloudFront/S3. The Amplify Console offers fully managed hosting with features such as instant cache invalidation and atomic deploys. For more control with setting up a CDN and hosting buckets, use CloudFront and S3." The Amplify CLI provides you the option to manage the hosting of your static website using Amazon S3 and Amazon Cloudfront directly as well.

https://docs.amplify.aws/cli/hosting/hosting/

You can also do this manually. One of the cool features associated with using AWS S3 is the ability to set up your bucket as a static web hosting platform.

EFS (Elastic file system)

It is a file storage service for EC2s. While an EBS is only mounted to one EC2 instance an EFS instance can be shared by more than one EC2s. capacity is elastic. You don't specify it in the beginning, for example 500GB. If you add a 1TB file later on, EFS will automatically expand. We have to open NFS port in the EFS instance since EC2s will communicate with EFS with the NFS protocol. You create the EFS instance and then you mount it to your EC2s with a command. It can support thousands of NFS connections.

Route53

- Just because GoDaddy registered your domain doesn't mean they have to manage your DNS. Domain registration is a process by which a registrar such as GoDaddy negotiates with ICANN to register a domain name on your behalf. DNS services are offered by GoDaddy, but you don't have to use them. If you want more flexibility, scalability, etc. you could use another DNS service such as AWS Route53)
- DNS for ECS cluster. There is different process for making it privately available, reachable through DNS from your other VPC resources and a different one for making it publicly available.

With route53 you create hosted zones which contain your records. There are two types of hosted zones, public and private.

1. Public hosted zone: A public hosted zone determines how traffic is routed on the internet.
2. Private hosted zone: A private hosted zone determines how traffic is routed within an Amazon VPC.
   A private hosted zone is a container that holds information about how you want Amazon Route 53 to respond to DNS queries for a domain and its subdomains within one or more VPCs

A highly available and scalable Domain Name System (DNS) web service.

Amazon Route 53 is designed to give developers and businesses an extremely reliable and cost effective way to route end users to Internet applications by translating human readable names like www.example.com into the numeric IP addresses like 192.0.2.1 that computers use to connect to each other. Route 53 effectively connects user requests to infrastructure running in Amazon Web Services (AWS) – such as an Amazon Elastic Compute Cloud (Amazon EC2) instance, an Amazon Elastic Load Balancer, or an Amazon Simple Storage Service (Amazon S3) bucket – and can also be used to route users to infrastructure outside of AWS

DNS is mostly UDP Port 53, but as time progresses, DNS will rely on TCP Port 53 more heavily. DNS has always been designed to use both UDP and TCP port 53 from the start, with UDP being the default, and fall back to using TCP when it is unable to communicate on UDP, typically when the packet size is too large to push through in a single UDP packet.

https://www.red-gate.com/simple-talk/cloud/infrastructure-as-a-service/introduction-to-route-53/

In order to manage domains in Route 53 you can create one or more hosted zones, which is the basic entity. A **Hosted Zone** is similar to a zone file in any other DNS system which consists of a collection of resource record sets. A record set can be a top-level domain, its subdomains, DNS records, MX records, etc. Route 53 allows almost all types of record types in a record set.

When creating hosted zone for your domain, Route 53 will add two record sets by default. One is a SOA (Service Of Authority) record and the second one is a NS record with four virtual name servers. You need to update your registrar with these name servers, so that they should answer DNS queries for your domain. Next you need to create an A-record for your root domain. As A-records can only be pointed to an IP address, you will need create an elastic IP and attach it to an EC2 instance. An alias for the A record means that the A record will not point to an IP but rather to another domain.

Sometimes you may only need to move a subdomain to Route 53. Procedure for this is same as above. First create a hosted zone for your subdomain (say subdomain.example.com). It will add SOA and NS records for the subdomain. Using the name server addresses, create NS records for this subdomain in your registrar account. To use this domain name, you need to create an A-record that will point to some IP addresses. Though it's possible to create a CNAME for a subdomain, it's not possible in this case, as CNAME is not allowed at zone apex.

When you register a URL, a hosted zone is created for it for which you can create record sets. These record sets contain the DNS records. There are various routing policies to choose from. For example the simple routing allows you to define only one record. If you define an A record with multiple IP addresses (let's say you operate three different EC2 instances) then Route53 will randomly route requests to your URL among the defined values.

Some policies allow to take into consideration the health checks. Health checks monitor the health of your endpoints. You can set them up to test them every 30 secs for example. If a health checks fails then Route53 will not route requests to your unhealthy resources.

There is also a traffic flow tool which is a GUI tool within which you can create sophisticated routing policies.

Elastic beanstalk

It is Amazons platform as a service, in the sense that you don't need to manage amazon services yourself. It is a service in which you can just upload your code, for example python code, and amazon will automatically provision the different services that it needs in order for it to run (security groups, s3 buckets, EC2 instances etc.). This way you don't need to know much about AWS. Its much simpler than cloudform.

SQS

A web service that gives you access to a distributed queue system. There is a standard type and a FIFO type. The standard type offers **at least once** delivery but due to its distributed nature occasionally you might have more than one copy of the same message delivered out of order. Ie. It offers *best effort ordering*. Have in mind the visibility timeout. If you can't allow that you might use a FIFO type queue which offers **exactly once** processing. They are limited though to 300 transactions per second.

Long vs short polling

SQS offers long polling.

SWF

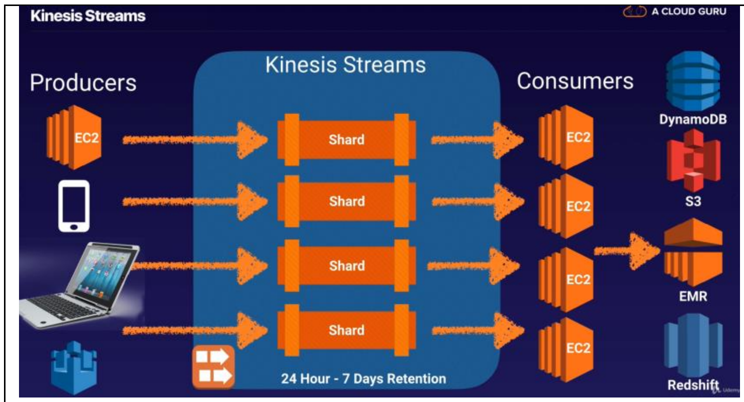
It is a system used for coordinating generic tasks where a task might involve human interaction as well, for example go to the shelf of the store and pick up the stuff to put in a box and mail the box. You have workflow starters, deciders and workers.

SNS

Simple notification service allows you to set up operate and send notifications from the cloud delivered to subscribers or other applications. You can send Push notifications to mobiles, SMS text messages, emails, messages to SQS or to any HTTP endpoint. You can group recipients based on topics. Some recipients would be subscribed to one topic but not to another. SNS stores its messages redundantly in many AZs. SNS is a push system while SQS a pull one, you have to have instances constantly asking for data.

Kinesis

Its a platform on AWS to send your streaming data to. there are three types of kinesis services. Typical streaming data is stock prices or IOT sensor data.



Kinesis streams
Its a service to which all of your data producers send the data. Data is retained there for 1-7 days and is distributed in shards. So data is persisted. Then you have consumers which are EC2 instances that can get the data from kinesis streams analyze them and store the analytics in other services like DynamoDB.

Kinesis Firehose
Your producers send the data to firehose but the data is not persisted there, it has to be analyzed on the fly and then stored to s3, or a warehouse or an elasticsearch cluster or whatever.

Kinesis analytics

This service can be used for on the fly analysis of streaming data. It can work inside both steams and firehose.

Containers in AWS

Introduction

AWS Elastic Beanstalk vs AWS ECS vs AWS Fargate for Containers

AWS Elastic Beanstalk is an application management platform that helps customers easily deploy and scale web applications and services. It keeps the provisioning of building blocks (e.g., EC2, RDS, Elastic Load Balancing, Auto Scaling, CloudWatch), deployment of applications, and health monitoring abstracted from the user so they can just focus on writing code. You simply specify which container images are to be deployed, the CPU and memory requirements, the port mappings, and the container links.

Elastic Beanstalk will automatically handle all the details such as provisioning an Amazon ECS cluster, balancing load, auto-scaling, monitoring, and placing your containers across your cluster. Elastic Beanstalk is ideal if you want to leverage the benefits of containers but just want the simplicity of deploying applications from development to production by uploading a container image. You can work with Amazon ECS directly if you want more fine-grained control for custom application architectures.

ECS gave you great flexibility, but at the expense of managing the underlying clustered hosts. There was also the lighter weight Elastic Beanstalk offering: the clustering platform was managed automatically, but there wasn't as much configuration wiggle room, and was mainly geared toward websites. With ECS Fargate, the underlying clustered hosts and orchestration is handled for you. You can focus completely on your containers and clustering parameters. But, unlike Elastic Beanstalk, you could be more flexible in your configuration.

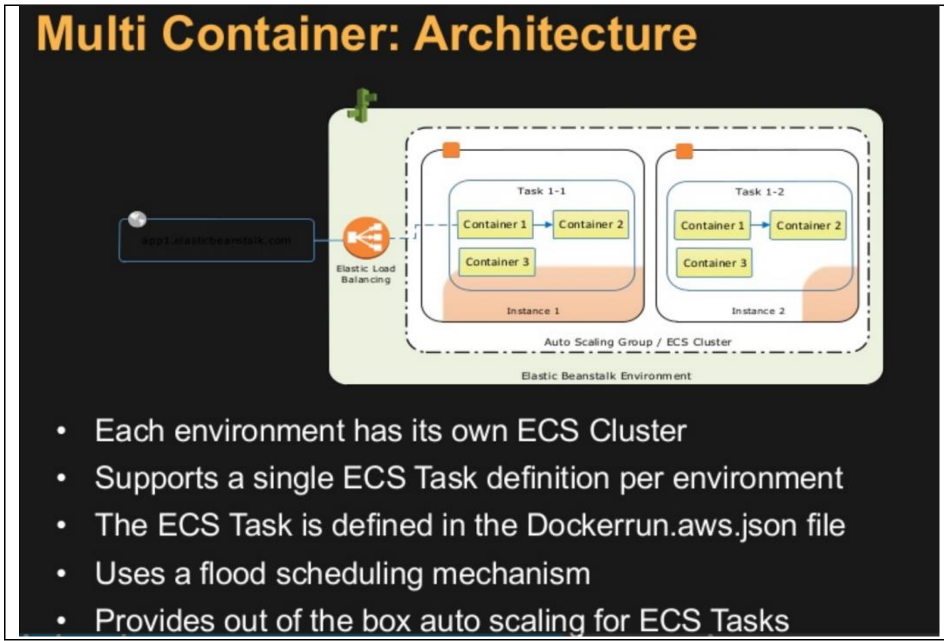
It completely abstracts the underlying infrastructure, and you see each and every one of your containers as a single machine. It is true that you are giving up some cool aspects of real PaaS. Yes you will still have to manually update your container's images, and sometimes you'll have to write your own Docker images. This can be a struggle at first if you don't know the basics of system administration. But, it also means that you can do pretty much anything you can think about and have complete flexibility and freedom in the systems, languages, tools, libraries, and versions that you want to use.

❖ **Kubernetes on AWS**

AWS makes it easy to run Kubernetes. You can choose to manage Kubernetes infrastructure yourself with Amazon EC2 or get an automatically provisioned, managed Kubernetes control plane with Amazon EKS. Either way, you get powerful, community-backed integrations to AWS services like VPC, IAM, and service discovery as well as the security, scalability, and high-availability of AWS.

❖ **Elastic Beanstalk**

In a multi container setup all containers run in a single EC2 instance. The autoscaling works in EC2 instance level so all containers are scaled as a single unit. This might not be convenient in some cases for example you can't just scale the number of workers.



Multi Container: Architecture

- Each environment has its own ECS Cluster
- Supports a single ECS Task definition per environment
- The ECS Task is defined in the Dockerrun.aws.json file
- Uses a flood scheduling mechanism
- Provides out of the box auto scaling for ECS Tasks

You can launch a cluster of multicontainer instances in a single-instance or autoscaling Elastic Beanstalk environment using the Elastic Beanstalk console.

One very important disadvantage is that "it supports a single ECS Task definition per environment". This means that all cluster instances must run the same set of containers which means that they are scaled all together as a single unit.

This is not convenient for having a web container and some workers. You want to scale only your workers and this is not possible.

ECS

the ECS task definition closely resembles a Kubernetes pod while the service definition mimics a Kubernetes deployment.

Launch types

You can run it either in Fargate launch type or on EC2 launch type. In the first case the underlying resources the cluster runs in, are managed by amazon. In the latter you have to manage them.

Task Definitions

To prepare your application to run on Amazon ECS, you create a task definition. The task definition is a text file, in JSON format, that describes one or more containers, up to a maximum of ten, that form your application.

Container Instances

The containers you describe in a task definition live in a container instance. What the container instance actually is, depends on the launch type. In EC2 launch type, the container instance is a an EC2 instance that is running the Amazon ECS container agent and has been registered into a cluster. Tasks using the Fargate launch type are deployed onto infrastructure managed by AWS.

When you set a container instance to DRAINING, Amazon ECS prevents new tasks from being scheduled for placement on the container instance.

Services

After you have your task definitions, you can create services from them to maintain the availability of your desired tasks. To scale your containers, you have to scale your tasks which are scaled by their service.

An Amazon ECS service enables you to run and maintain a specified number of instances of a task definition simultaneously in an Amazon ECS cluster. If any of your tasks should fail or stop for any reason, the Amazon ECS service scheduler launches another instance of your task definition to replace it in order to maintain the desired number of tasks in the service.

Clusters

An Amazon ECS cluster is a logical grouping of tasks or services. If you are running tasks or services that use the EC2 launch type, a cluster is also a grouping of container instances.

Container agent

it is a process running in the container instance. The Amazon ECS container agent allows container instances to connect to your cluster.

Resources and Tags

Amazon ECS resources, including task definitions, clusters, tasks, services, and container instances, are assigned an Amazon Resource Name (ARN) and a unique resource identifier (ID). These resources can be tagged with values that you define, to help you organize and identify them.

Service Discovery for Amazon ECS

Amazon ECS Service Discovery. Service discovery uses AWS Cloud Map API actions to manage HTTP and DNS namespaces for your Amazon ECS services. You can enable service discovery for your containerized services. Amazon ECS creates and manages a registry of service names using the Route 53 Auto Naming API. Names are automatically mapped to a set of DNS records so you can refer to services by an alias, and have this alias automatically resolve to the service's endpoint at runtime. Today (May 2020), service discovery is available for Amazon ECS tasks using AWS Fargate or the EC2 launch type with awsvpc networking mode.

DNS for ECS services

You can make a service reachable by DNS. There is different process for making it privately available, reachable through DNS from your other VPC resources and a different one for making it publicly available. In both cases you can't use a specific IP for your DNS records since the underlying container instances might change during the lifetime of your services. You can make your service privately reachable with AWS Cloud Map that points to the correct IP every time. For making it publicly available the easiest and the proper way is to put it behind a load balancer. Load balancer has a DNS name to which you can point your DNS records.

Your ALB will already have a DNS name with some long abdcef123.us-east-1.amazonaws.com type of hostname. If you're hosting your DNS at Route 53, create an ALIAS record for your domain which points to that long hostname. If your DNS doesn't support ALIAS records then you can't use a CNAME for your root naked domain. What you can do is to use a CNAME for the www subdomain of your domain which points to that same long hostname and use redirection of dns queries made to your naked domain to your www subdomain (this is what I did for heroku-godaddy and also for AWS-godaddy). this way when you visit zakanda.com the registrar will forward you to www.zakanda.com which points to your ALB. If you set up SSL then you forward to www subdomain through https protocol.

Pushing code

I imagine the automated pipeline like this. You commit changes to master on github. This triggers the docker build actions in Travis, creates the images and pushes them to a container hub. Then ECS updates its services with the "enforcedDeployment" option enabled which redeploys the service using the updated images.

For the manual process I would commit the code and then manually build the images from it, push them to a hub and then update the services manually.

Tips

- If you update a task definition you have to update its service and make it use the new task definition.
- If you push an update container image with the same name to your docker registry and you want to update your tasks with it you can just restart the service that controls this task with the option EnforceDeployment on.
- When you create an ECS cluster an auto scaling group is created for it with the desired number of EC2 instances. When you want to close your cluster temporarily you can edit it and set the desired instances to 0. Then when you want to start it again set it to what it was.
- ?? When you update your services (created with the rolling update option) then in order to achieve no downtime your cluster needs to scale up to more instances, deploy the updated services to them and then scale down again. This means that your containers will run in new container instances with new IP addresses. Maybe you must create an Elastic IP address and connect it to your ec2 instance.
- The default user created by ecs in the ec2 instances is "ec2-user". you can ssh into them with that user.
- The env variables defined in a task only have effect when the container runs by the task. They will not be available if you manually run a custom command to your container. To do so you have to specify the needed env variables with the docker run command. The easiest way is to upload an env file to your container instance and point to it with the run command.
- ?? Notice that the health checks performed by the target group of the load balancer, use the internal IPs of the container instances (ec2 instances). This means that you have to add these IPs to the allowed hosts of your django app. Is there any way to avoid this and use for example the private dns name of the service that you want to check?

https://forums.aws.amazon.com/thread.jspa?messageID=423533 maybe a workaround would be to identify the health check requests in the reverse proxy and set a custom host header for them (for example zakanda.com) so that you don't have to hard code the IPs in your code.

	Getstream	Database	Static	branch
State 0 (dev)	Zakanda-dev	Db-dev	Static-dev	Demo
State 1 (prod)	Zakanda last heroku	Db last heroku	Static-last heroku	Master (last heroku)

Internet - ALB - reverse proxy - application server

Internet to ALB is encrypted with SSL certificate that lives in the ALB. All the rest communication is with http. Application Load Balancer does pass "x-forwarded-for/x-forwarded-proto/x-forwarded-port" headers with the request.

The X-Forwarded-Proto (XFP) header is a de-facto standard header for identifying the protocol (HTTP or HTTPS) that a client used to connect to your proxy or load balancer. Your server access logs contain the protocol used between the server and the load balancer, but not the protocol used between the client and the load balancer. To determine the protocol used between the client and the load balancer, the X-Forwarded-Proto request header can be used.

Redirect http to https

http requests can be redirected to https with different ways. One way is with the application server (in django with SECURE_SSL_REDIRECT=True setting). Another one is from the load balancer. With the aws ALB you can set a redirect action. But the first option has some issues. All communications to application server are through http (https only to internet to load balancer). this means that if you redirect all http to https in the application server you will get infinite redirections. So the proper way to redirect https to http is in the load balancer level.

Misc

Beginner friendly projects

1. Deploy and host a production-ready highly available WordPress website on AWS

https://www.aosnote.com/offers/xFzqby9z/checkout (TECHWITHLUCY coupon code). You do it yourself without elastic beanstalk so that you know all the underlying fundamental parts.

https://aws.amazon.com/getting-started/hands-on/build-wordpress-website/

2. Build a Serverless Web Application:

https://aws.amazon.com/getting-started/hands-on/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/

Multiple projects in AWS

There are a few potential approaches you could take:

1. Use a naming convention and tags
2. Isolate projects via separate VPC and IAM groups
3. Completely separate accounts for each app

The third is usually the best option. You can use AWS organizations to handle the multiple accounts and have consolidated billing. From the organizations interface invite a new user to the organization. You can choose to create a new user instead of inviting an existing one. For email you can use a mail like this: dimgeows+something@gmail.com. This is an alias to your existing email.

Roles

Trust relationships and Permission policies. A role has both of them. The Trust relationships are what aws services can be connected with that role and the permissions is what permissions the aws services connected with that role have.

A security group can have a source that is the same security group.

Block based storage vs object based storage

Block storage is the oldest and simplest form of data storage. Block storage stores data in fixed-sized chunks called — you guessed it — 'blocks'. By itself, a block typically only houses a portion of the data. The application makes SCSI calls to find the correct address of the blocks, then organizes them to form the complete file. Because the data is piecemeal, the address is the only identifying part of a block — there is no metadata associated with blocks. This structure leads to faster performance when the application and storage are local, but can lead to more latency when they are farther apart.

Compared to block storage, object storage is much newer. With object storage, data is bundled with customizable metadata tags and a unique identifier to form objects. Objects are stored in a flat address space and there is no limit to the number of objects stored, making it much easier to scale out.

Round Robin

Round robin load balancing is a simple way to distribute client requests across a group of servers. A client request is forwarded to each server in turn. The algorithm instructs the load balancer to go back to the top of the list and repeats again.

Hypervisor

A hypervisor (or virtual machine monitor, VMM) is computer software, firmware or hardware that creates and runs virtual machines. A computer on which a hypervisor runs one or more virtual machines is called a host machine, and each virtual machine is called a guest machine. The hypervisor presents the guest operating systems with a virtual operating platform and manages the execution of the guest operating systems.

DMZ

In computer security, a DMZ or demilitarized zone (sometimes referred to as a perimeter network or screened subnet) is a physical or logical subnetwork that contains and exposes an organization's external-facing services to an untrusted, usually larger, network such as the Internet. The purpose of a DMZ is to add an additional layer of security to an organization's local area network (LAN): an external network node can access only what is exposed in the DMZ, while the rest of the organization's network is firewalled. The DMZ functions as a small, isolated network positioned between the Internet and the private network and, if its design is effective, allows the organization extra time to detect and address breaches before they would further penetrate into the internal networks. (In AWS VPC with a public and a private subnet the public one acts as a DMZ for the private one)

How to automate the restart of web server and other stuff after a EC2 instance failure?

The current product is built in Python/Django, MySQL, PostgreSQL on AWS (**EC2, S3, RDS, ELB Elastic Beanstalk**). You should be very comfortable in this envinronment.

AWS (EC2, EMC, DDB, S3, SQS, RDS, etc)

User uploaded files

There are 2 possible methods:

- Direct upload

This is the preferred approach if you're working with file uploads bigger than 4MB. The idea is to skip the hop to your dyno, making a direct connection from the end user browser to S3. While this reduces the processing required by your application it is a more complex implementation and limits the ability to modify (transform, filter, resize etc…) the file before storing in S3. Most of the articles listed below demonstrate how to perform direct uploads. Large file uploads in single-threaded, non-evented environments (such as Rails) block your application's web dynos and can cause request timeouts and H11, H12 errors.

- Pass-through uploads

Pass-through uploading sends the file from the user to the application on Heroku which then uploads it to S3. Benefits of this approach include the ability to pre-process user-uploads before storing in S3. However, be careful that large files don't tie up your application dynos during the upload process.

Amazon Step functions

You might want a 2nd lambda function to always follow a first and only run if the first succeeds

You may want to execute 2 functions in parallel and feed the combined results into a third

You may want to choose between 2 functions based on the outcome of the first one

You can write a lot of code to orchestrate the interaction of your microservices (which in this case are lambda functions) or you can use amazon step functions to coordinate your workflows. It has embedded try catch finally patterns to manage task timeouts, retries and failures.

Monitoring (Observability)

Intro

You might need to use an APM service for sure. But you might also use a log streaming serbice to keep track of your logs. And an additional error tracking service for context around application errors. The Grafana blog post "Logs and Metrics and Graphs, Oh My!" provides more details about the differences between logs and metrics.

➢ APM services
  - Prometheus+Grafana (powerful, free, but steep learning curve)
  - New relic (powerful and free plan)
  - Dynatrace (amazing but expensive)
  - Instana
  - Datadog

Have in mind that from k8s you get some cpu and memory data information out of the box.

➢ Log streaming/aggregation services

Loki (free, like Prometheus but for logs), OpenSearch, coralogix, elasticsearch

➢ Application Error monitoring service

Sentry (you can connect sentry with Grafana to see everything in a single dashboard)

Notice that you can monitor your application errors with Prometheus directly using Django-prometheus for example that adds a middleware and sends data to a specific url.

Prometheus+Grafana vs New Relic vs Dynatrace

TLDR; Dynatrace One is, in my fairly well informed/experienced opinion, is the Bently or Aston Martin of monitoring solutions. Sure, you can invest a lot of time and effort in customizing the heck out of the Prometheus/Grafana open source stack and get amazing results from that (and even that's become way easier to do in the past year or so), but I personally see that solution as a sort of "kit car" that's only as good as the people you have setting it up and maintaining it. New Relic is like the Camry of monitoring solutions; affordable (even without the big client discounts), sufficient for 85% of the things you'd typically throw at it, supports some deeper extension (i.e. there's plenty of "aftermarket tuning" that can be done). You can mostly bridge the gap between New Relic and Dynatrace or Prometheus if you're willing to accept the limitations (samples instead of everything) and do your homework, but at the end of the day you're not driving this "Camry" off the lot and "taking it to the track". With Dynatrace One you can take it to track day right away and expect to have a more-than-respectable showing.

Instana

Enterprise Observability solution (acquired by IBM)

Instead of configuring and connecting many different open-source tools yourself, you get a complete solution. Very nice and featureful.

Price $75 per host per month, minimum order quantity 10 hosts so minimum of $750/month. A host is a physical or virtual OS instance that you monitor with Instana.

ELK stack

Elasticsearch, Logstash and Kibana. Elk's APM is terrible and half baked. Not a fan at all

Dynatrace

Very Expensive. Dynatrace One was hands down the single best "whole ecosystem" APM/monitoring tool I have ever had the pleasure of using. After some light configuration work, we had automatically generated datacenter health dashboards that allowed us to drill down to the network level and see some (minor) network traffic issues, full visibility of the applications from nuts to scalp (host VM health/status, inter-host networking health/status, inter-app and public facing transaction tracing out of the box, real user monitoring, no transaction sampling - we got every bit of telemetry generated by every transaction, viz on our websocket connections/transactions with some more minor server-side tweaking. It's a beautiful product in comparison to New Relic, which needs more heavy lifting and multiple agents (APM, JS, and Infrastructure) to only come close to Dynatrace One's capabilities (which uses only a single host-level agent). I really fell in love with it, but holy Moses the cost! For what we were implementing, only 50 hosts in our UAT environment, the license we were quoted was upwards of $65K annually.

kube-prometheus-stack

have in mind this for Prometheus and Grafana on Kubernetes. https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack

Monitoring stack

- Cloud Provider (generally AWS)
- Kubernetes (generally using cloud-provider's version, eg: EKS)
- Prometheus

- Prometheus Alert Manager
- Cloud Provider's Monitoring System for Cloud-Provider managed SaaS services (eg: CloudWatch, for watching RDS, ES, etc)
- ElasticSearch + Kibana for log aggregation and log visualization/searching/dashboard
- Grafana for visualization of metrics from CloudWatch + Prometheus + Application (via RDS)

- Amazing Dashboards for Grafana. See: https://github.com/DevOps-Nirvana/Grafana-Dashboards/raw/main/images/kubernetes-nginx-ingress-via-prometheus.png for example
- SNS-to-slack simple terraform module to send all CloudWatch/AWS alerts into Slack
- Prometheus setup to send alerts to Slack, and critical alerts to Pagerduty
- Setup an "dead-mans" alert in Prometheus which always alerts and sends an http request every minute into an AWS Serverless stack which saves transparently into an CloudWatch metric, which has an alert to detect if Prometheus dies.
- Besides good default/best practice alerts for AWS resources which go to slack, create some critical alerts which send to PagerDuty.

Once you've done this once, 99% of it can be done via Terraform.
I've rinse and re-used this stack with about 20 companies now. Pure beauty. You can setup alerting schedules and such in PagerDuty to do shifts. You can enable devs to monitor and maintain things by inviting them into Slack and giving them access to Grafana/Kibana. Generally on top of that above I CI/CD 99% of everything with either Gitlab or Github. Or whatever the client wants, but I prefer those two.

-- ----

Aws Cloudwatch

By default, AWS gives you visibility into metrics like CPU load logs, network latency, request volume, etc., but not EC2 memory usage. For other metrics like EC2 memory usage, you'll have to install and configure a CloudWatch agent on the instance. CloudWatch contains a lot of different metrics to choose from.

An agent is simply a background task that collects specified metrics on an EC2 instance and sends them to CloudWatch. If you're familiar with AWS, you may already know that for one service to access another, roles and permissions are needed.

Aws X-Ray

Monitor your application traces and get a view of your connected services. Very suitable for monitoring microservices. You can easily enable it when using aws api gateway in front of your microservices. X-Ray provides an end-to-end view of requests as they travel through your application and shows a map of your application's underlying components.

Aws cloudwatch synthetics

It monitors your users experience. For example, how much time does it take to load a page? Is a link broken? Are there any page load errors? It monitors your endpoints. It uses the concept of canaries. A canary is a configurable script that follows the same routes and perform the same actions as a customer. You can run scheduled canaries.

Aws cloudwatch ServiceLens

Tie all the monitoring tools in one place. Monitor the health performance and availability of your application.

1. Infrastructure monitoring
2. Transaction monitoring (uses traces to understand dependencies between your resources and identify faults in your services)
3. End user monitoring (uses canaries)

3rd party APM tools (application performance monitoring)

while Amazon Cloudwatch can provide detailed cloud insights, it doesn't have the robust application monitoring features offered by New Relic. Businesses looking specifically for an application performance monitoring tool may find more success with New Relic.

I see Sentry and New Relic as having quite different purposes.

- I use Sentry for debugging errors in production
- I use NewRelic for figuring out performance issues.

New relic

Application monitoring

Server monitoring (resource usage, cpu, ram, disk, network etc.)

Browser monitoring

Mobile monitoring

Availability monitoring

After you install the Python agent, it begins to collect data about your app. You can view the data as charts and tables in the New Relic UI. If you are using Docker, install the agent within each container.

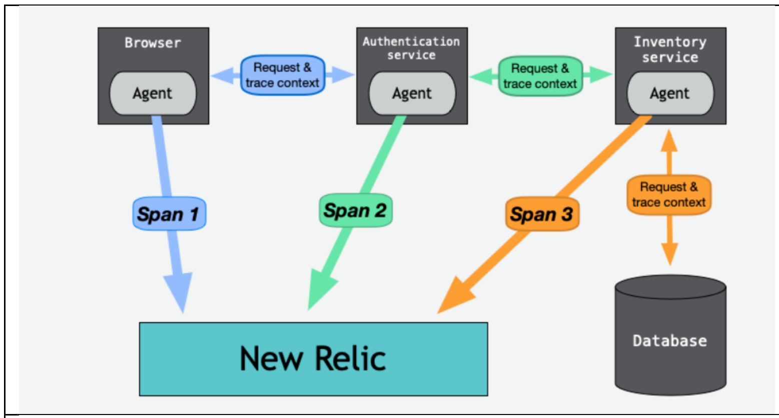
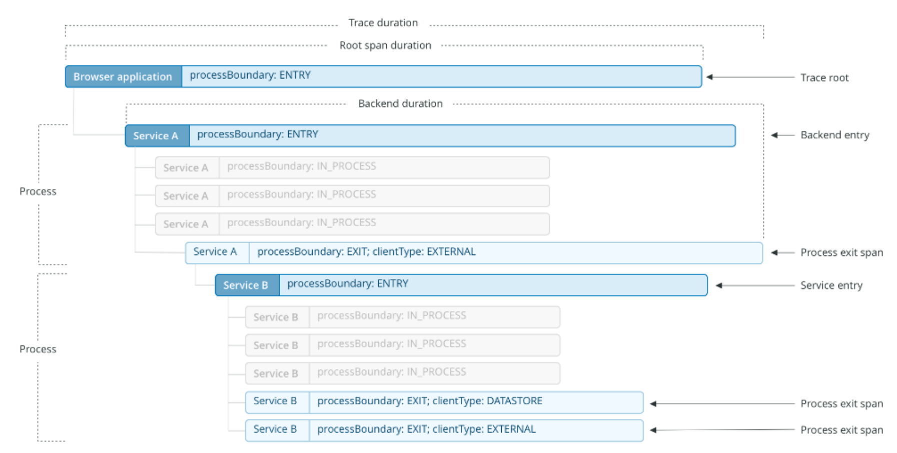
a) Application Stack Trace Details

New Relic offers awesome interface for debugging the server side performance issues. While this running on your servers it keeps track of the entire run time stack and gives a detailed report on issues by thread profiler. For example, it counts up the number of database queries that are running slow and within your particular thread and gives a report on how many different queries are executed and how long they took to executes. This is a out of the box feature to allow you quickly to identify performance issues: which queries take too long, which query is called too many times, what function is running for way too long.

b) Global Real User Tracking

Since this runs on your servers and also as part of js on your webpage, you can track accurate real user page load times. This was way more valuable to our company than the simulated traffic because it allowed us to see exactly how our web pages were performing all across the world. This offers for real user page performance tracking allowed us quickly to see if there were particular users that were getting a worse experience than others.

Distributed traces

	Here is an example of a web transaction where agents measure the time spent in each service. Agents then send that timing information to New Relic as spans, and the spans are combined into one distributed trace.
	A distributed trace has a tree-like structure, with "child" spans that refer to one "parent" span. This diagram shows some important span relationships in a trace:

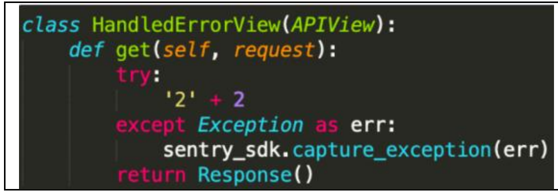
Browser
Agent
Request & trace context
Authentication service
Agent
Request & trace context
Inventory service
Agent
Span 1
Span 2
Span 3
Request & trace context
New Relic
Database

Trace duration
Root span duration
Browser application processBoundary: ENTRY
Trace root
Backend duration
Service A processBoundary: ENTRY
Backend entry
Process
Service A processBoundary: IN_PROCESS
Service A processBoundary: IN_PROCESS
Service A processBoundary: IN_PROCESS
Service A processBoundary: EXIT; clientType: EXTERNAL
Process exit span
Service B processBoundary: ENTRY
Service entry
Process
Service B processBoundary: IN_PROCESS
Service B processBoundary: IN_PROCESS
Service B processBoundary: IN_PROCESS
Service B processBoundary: EXIT; clientType: DATASTORE
Process exit span
Service B processBoundary: EXIT; clientType: EXTERNAL
Process exit span

Sentry

Sentry's SDK hooks into your runtime environment and automatically reports errors, uncaught exceptions, and unhandled rejections as well as other types of errors depending on the platform. Generally, unhandled errors are errors that aren't caught by any except (or try/catch) clause.

Notice that you can also report handled exceptions

```
class HandledErrorView(APIView):
    def get(self, request):
        try:
            '2' + 2
        except Exception as err:
            sentry_sdk.capture_exception(err)
        return Response()
```

Sentry vs logging

Sentry is for errors and crashes. Logging is for tracing events even non errors. Sentry saves context for our errors. Same errors are not stored again. So if you want tracing you need to use your typical logging infrastructure.

Prometheus + Grafana

You have prometheus and graphana microservices. You open graphana and add a data source. You add the Prometheus service endpoint. Notice that you can also monitor postgresql with Prometheus.

Depending on your needs New Relic may end up being free. Their new free tier is pretty generous. If price is your primary concern though you can't get more free than Prometheus & Grafana. There's a fairly steep learning curve but given the popularity of the projects it's well worth it in my opinion.

but what if you are using an old school architecture like a single server and a database, would you still consider Prometheus + Grafana? That's a great question, that I can only partially answer. For monitoring a single server, yeah it'll work—but I'm not certain how much value you'd get out of running your monitoring on the server you're monitoring.

Prometheus

Prometheus is a free software application used for event monitoring and alerting. It records real-time metrics in a time series database built using a HTTP pull model, with flexible queries and real-time alerting.

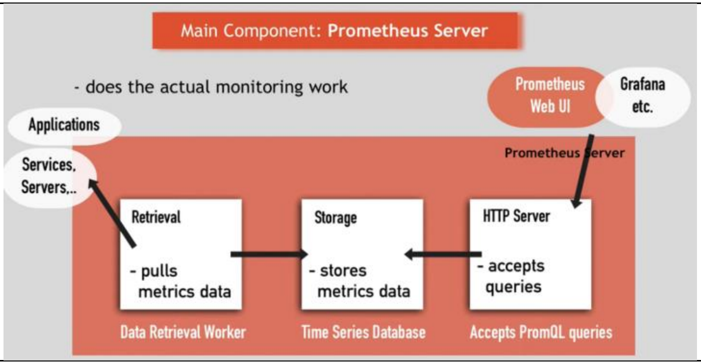
Grafana

Grafana is a multi-platform open source analytics and interactive visualization web application. It provides charts, graphs, and alerts for the web when connected to supported data sources.

Tech stack example

Our stack is Ruby on Rails on the backend with React.js and TypeScript on the frontend, backed by Postgres and Redis databases. We run everything on AWS and Heroku, logging with LogDNA, CI with Github Actions, error tracking with Sentry, monitoring with Datadog, and source code on Github.

Prometheus (metrics)

 Main Component: Prometheus Server - does the actual monitoring work Prometheus Web UI Grafana etc. Applications Services, Servers,... Prometheus Server Retrieval - pulls metrics data Storage - stores metrics data HTTP Server - accepts queries Data Retrieval Worker Time Series Database Accepts PromQL queries	

It uses a pull approach. Your targets (your application, your OS etc.) implement a /metrics endpoint. Prometheus service pulls metrics from this endpoint.

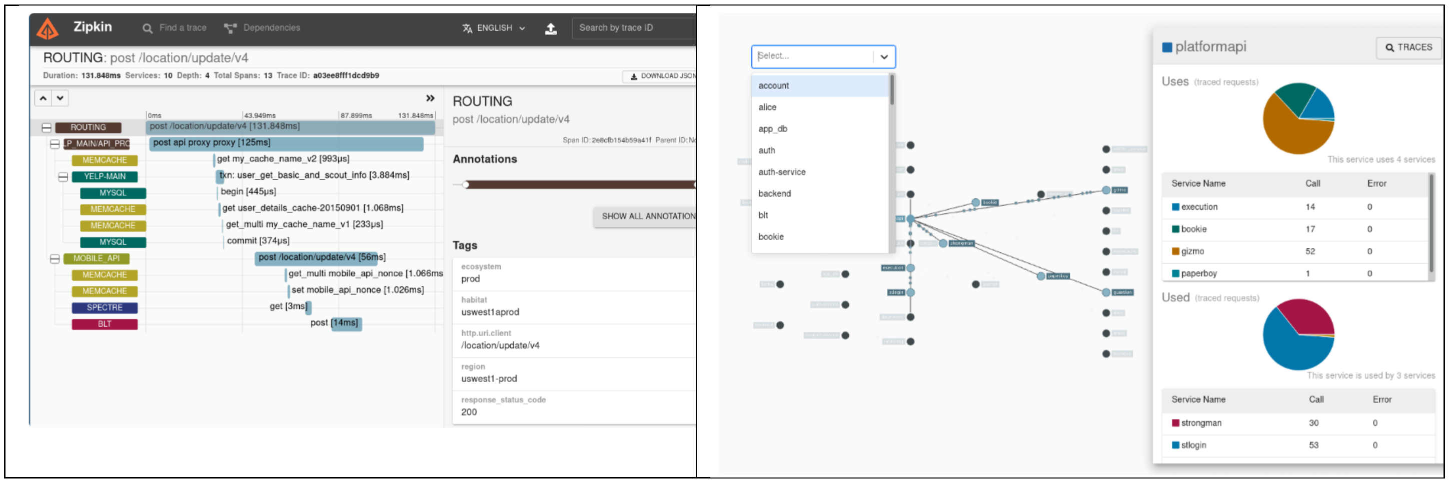
New relic and others, use a push approach where the targets push their metrics to the service. Pull approach offers some advantages.

Some services expose this native Prometheus endpoint and the data in the right format by default. For services that don't we have to implement our own service (or application component) for that. This service is called Prometheus Exporter. It fetches the metrics from the service converts it to the correct format and expose it under a /metrics endpoint. There is a list of exporters for many famous services (databases, messaging systems, linux server, elasticSearch, etc.). these exporters are also available as docker images. So if you want to monitor your postgresql container in Kubernetes, you deploy a side exporter container in the same pod. To monitor your own applications (num of requests, exceptions, how many resources they are using) there are Prometheus client libraries for many languages that implement this component. You choose what metrics to expose to it.

Prometheus has excellent out of the box integration with Kubernetes.

Zipkin (tracing)

Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data.



Elasticsearch (logs store)

Misc

A NOsql json data store, built on top of Apache Lucene

It runs as a cluster, sharding the data to nodes.

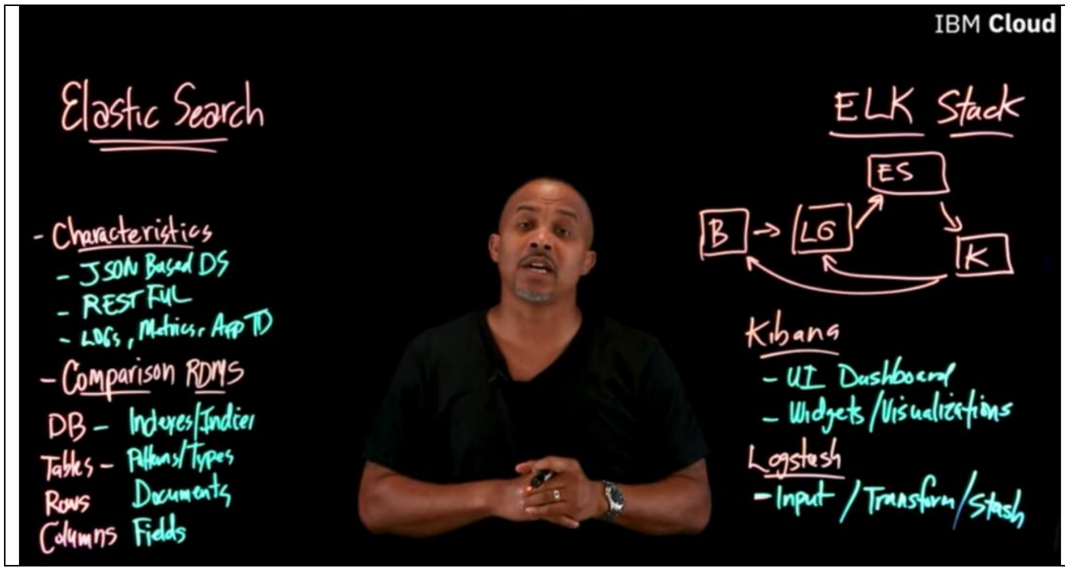
On the CAP theorem, it offers PA, availability and partition tolerance.

A typical use case is to **store there the logs of all your services and analyze them**.

It is the basic component of the ELK stack composed of Elasticsearch, Logstash and Kibana.

Kibana is the UI. Logstash is a service that gets in data and stores it to Elasticsearch (the B here is Beats a logstash related service)

The ELK stack exists as a collection of containers for using it in development on your local machine.

 IBM Cloud Elastic Search - Characteristics - JSON Based DS - REST FuL - Logs, Metrics, App TD - Comparison RDMS DB - Indexes/Indices Tables - Patterns/Types Rows Documents Columns Fields ELK Stack ES B LG K Kibana - UI Dashboard - Widgets/Visualizations Logstash - Input / Transform / Stash	ELK

Thinks that you can do:

- Searching a large number of product descriptions for the best match for a specific phrase (say “chef’s knife”) and returning the best results
- Given the previous example, breaking down the various departments where “chef’s knife” appears (see Faceting later in this book)
- Searching text for words that sound *like* “season”
- Auto-completing a search box based on partially typed words based on previously issued searches while accounting for mis-spellings
- Storing a large quantity of semi-structured (JSON) data in a distributed fashion, with a specified level of redundancy across a cluster of machines

Git

Intro

Version control systems

Local (changes are saved in your local computer)

Centralized (changes are saved in a shared server)

Distributed (changes are saved locally and are pushed to the server. The code exists both in the local machine and the server)

You have to think of git as having 3 areas.

1. The **working area/tree** (you have made a change in a file, and you haven’t staged it yet)
2. The **staging area** or **git index** or **cache** (A file here is marked to be committed)
3. The **object store area** which is the actual repository. (the file is committed and exist in the repository)

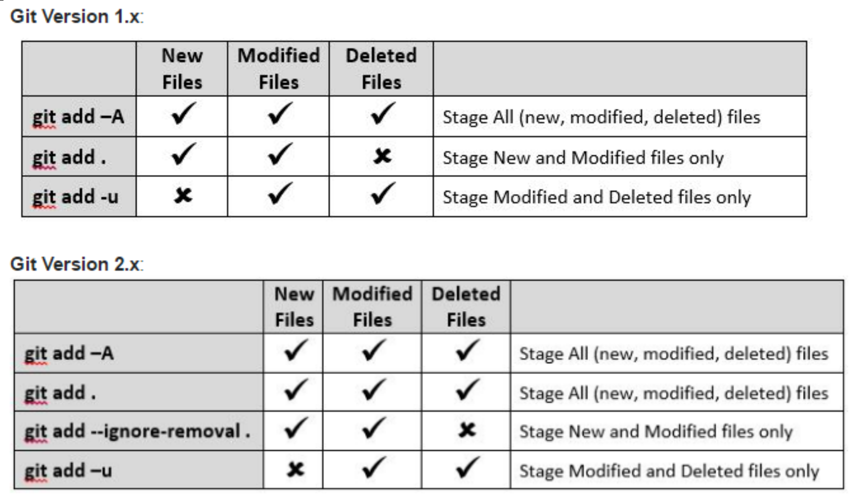
As you work with git you move files between these areas.

Files in Git can be in one of 3 different states

- Modified (modified recently)
- Staged (are marked to be saved)
- Committed (have been saved)

There is also the distinction between untracked and unstaged files

An untracked file is a file that exists in the working tree of a git repository but isn't listed in the index. An unstaged file is a file that is listed in the index, but the contents of the file in the working tree are different to the contents listed in the index



Git Version 1.x:

	New Files	Modified Files	Deleted Files	
git add –A	✓	✓	✓	Stage All (new, modified, deleted) files
git add .	✓	✓	✗	Stage New and Modified files only
git add -u	✗	✓	✓	Stage Modified and Deleted files only

Git Version 2.x:

	New Files	Modified Files	Deleted Files	
git add –A	✓	✓	✓	Stage All (new, modified, deleted) files
git add .	✓	✓	✓	Stage All (new, modified, deleted) files
git add --ignore-removal .	✓	✓	✗	Stage New and Modified files only
git add –u	✗	✓	✓	Stage Modified and Deleted files only

> git add *
means add all files in the current directory, except for files, whose name begin with a dot. This is your shell functionality, actually, Git only receives a list of files. * is not part of git - it's a wildcard interpreted by the shell. * expands to all the files in the current directory. * avoids hidden files (i.e., files that their name begins with a ".")

> git add .
though, has no special meaning in your shell, and Git add the entire directory recursively, which is basically the same, but it also adds files whose name begin with a dot.

Bare repositories

> git init --bare <directory>

The --bare flag creates a repository that doesn't have a working directory, making it impossible to edit files and commit changes in that repository. You would create a bare repository to git push and git pull from. Central repositories should always be created as bare repositories because pushing branches to a non-bare repository has the potential to overwrite any in progress working directory content.

Clone a repo

Create a local copy (clone) of a github repository in your local machine. The remote repo is reffered to as ***origin*** (A remote named origin is created, pointing to the URL you cloned from). For every branch *foo* in the remote repository, a corresponding remote-tracking branch *refs/remotes/origin/foo* is created in your local repository.

Fork a repo

Create a copy of a github repository in your github account (the repo is not on your local machine but on github under your account). Then you can clone your fork in your local machine.

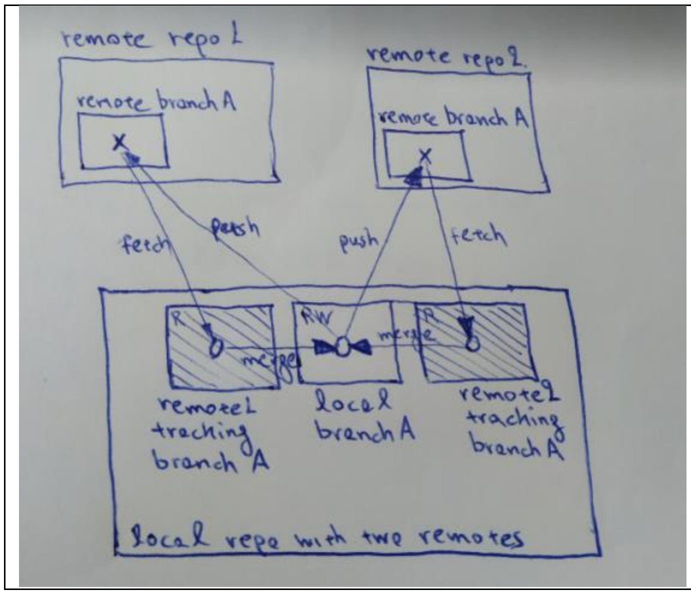
Branches

Local and remote tracking branches

There are two types of branches

- Local branches stored in .git/refs/heads/
- Remote tracking branches stored in .git/refs/remotes/

The local branches of your local repository in your pc and the remote tracking branches which are local branches that have a direct relationship to a remote branch.

 remote repo 1 remote branch A remote repo 2 remote branch A fetch push push fetch R RW R merge merge remote1 tracking branch A local branch A remote2 tracking branch A local repo with two remotes	You can think of remote tracking branches as read-only branches. You can check out a remote branch just like a local one, but this puts you in a detached HEAD state (which means that your HEAD ref is pointing to a ref that is not in sequence with your local history). If you approve the changes a remote branch contains, you can merge it into a local branch with a normal git merge. If you're on a tracking branch and type git push, Git automatically knows which server and branch to push to. Also, running git pull while on one of these branches fetches all the remote references and then automatically merges in the corresponding branch.

What happens is this: The actual branch on a remote repository is tracked by a remote-tracking branch entity that exists locally. The name of a remote tracking branch is [remote repository alias]/[branch name]. The only way to change a remote tracking branch is with fetch or pull, you can't work on a remote tracking branch directly. To work on it you have to create a local branch based on that remote-tracking branch.

git checkout --track -b [branch1] origin/[branch1] Branch branch1 set up to track remote branch refs/remotes/origin/branch1.

Switched to a new branch "branch 1" or simply git checkout -b [branch] origin/[branch].

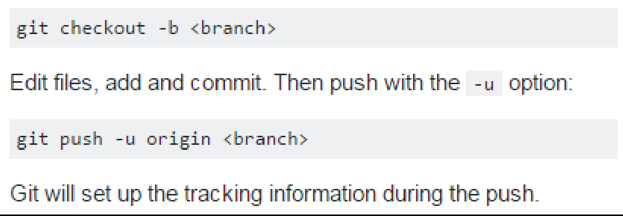
Head

HEAD is a pointer that points to a specific commit. The branch of the commit that the HEAD points to is the current branch. Whenever you create a branch, a pointer called HEAD is also created which points to the required files. No new files are being created.

checkout

Updates files in the working tree to match the version in the index or the specified tree. If no paths are given, git checkout will also update HEAD to set the specified branch as the current branch.

Some Commands

 `git checkout -b <branch>` Edit files, add and commit. Then push with the `-u` option: `git push -u origin <branch>` Git will set up the tracking information during the push.	You create a local branch You add changes and commit In the push action, a new branch will be created in the remote repository. A new remote tracking branch will also be created locally.

When you create a new branch the original one is automatically named master.

git branch (list all local branches. The one with * is the current branch, the active one, the one that we are working with)

git branch –v (list all the local branches along with their last commit (the commits message)) git branch –r (List all remote tracking branches)

git branch –a (list all)

git branch –m newName (renames the current branch)

git branch BranchName (Create a new branch starting at the same point in history as the current branch)

git checkout BranchName (make it the current branch, Switch from the current branch to the indicated branch)

git checkout –b NewBranchName StartPoint (Create a new branch and switch to it from the current branch. Note that the StartPoint refers to a revision number (or the first 6 characters of such) or an appropriate tag)

git merge BranchName (Merge the specified branch into the current branch and auto-commit the results)

git merge BranchName --no-commit

git branch - -merged (shows the merged branches, the ones that have been merged to another branch)

git branch - -no-merged

git reset - -hard (Undo everything since the last commit)

Syncing

The git remote family of commands let you create, view, and delete connections to other repositories. Such a connection is just a convenient name that can be used as a reference for the remote repository. These commands are essentially an interface for managing a list of remote entries that are stored in the repository's ./.git/config file. The remote repositories can be accessed either through http or ssh protocols. The first is for read only the second for read and write. The remote repositories are called simply "remotes" in git context.

> git remote

lists the remote connections. With the -v flag shows the url of each remote repository.

The following commands modify the connections of a repository by modifying the .git/config file

> git remote add <name> <url>

Adds a record for a remote named <name> at the repository url <url>. with -f flag will fetch immediately

> git remote rm <name>

> git remote rename <old_name> <new_name>

> git remote get-url <name>

> git remote show <name>

Shows information

>git remote prune <name>

Deletes the local branches for <name> that are not present on the remote repository. With --dry-run flag to show the branches without pruning them

In git there is no real time synchronization between your local and remote repository pair. You need to manually pull central repository (upstream) commits into your local repository or manually push your local commits back up to the central repository.

<name> origin

When you clone a repository with git clone, it automatically creates a remote connection called “origin” pointing back to the cloned repository.

Adding teammates repositories in your remotes

Apart from having a connection to the central repo you can create a connection to a teammate’s repo with git remote add. So that it is possible to collaborate outside of the central repository. This can be very useful for small teams working on a large project.

Fetch

git fetch grabs changes from remote repository and puts it in your repository's object database. It also fetches branches from remote repository and stores them as remote-tracking branches.

The git fetch command downloads contents (commits, files, and refs) from a remote repository into your local repository without updating your local repo's working state, leaving your current work intact.

Example

For example lets assume there is a central repo origin which the local repository has been cloned from using the git clone command. Let us also assume an additional remote repository named coworkers_repo that contains a feature_branch which we want to add to our local repo in order to work on it.

```
> git remote add coworkers_repo git@bitbucket.org:coworker/coworkers_repo.git
```

We created a reference to this new remote repo

```
> git fetch coworkers feature_branch
```

We create a remote_tracking_branch locally and download the contents of this remote branch

```
> git checkout coworkers/feature_branch
```

Optionally we can checkout this new remote tracking branch to check its content. Have in mind that we can’t make and commit changes directly here. We have first to create a local branch for that remote tracking branch

```
> git checkout -b local_feature_branch
```

We create a local branch named “local_feature_branch” which is in the same state with the remote tracking branch. Now we can start adding commits to it.

```
> git fetch --all
```

Fetch all registered remotes and their branches

```
> git fetch <remote>
```

Fetch all branches of the remote repository with aliased locally with <name>

```
> git fetch <remote> <branch>
```

git pull is the more aggressive alternative, it will download the remote content for the active local branch and immediately execute git merge to create a merge commit for the new remote content.

Push

The git push command is used to upload local repository content to a remote repository. Notice that git push will only merge to the remote repository if the merge is a fast forward merge. In any other case it will raise an error. The only other option is to force a pull in which case it will overwrite any later commits of the remote repository that don't exist in the local one. This must only be done is there are no other developer that has used these commits. Have also in mind that pushing will overwrite any in in progress working directory content (you don't have that issue if the remote repository is bare)

Example

```
> git checkout master
> git fetch origin master
> git rebase -i origin/master                # Squash commits, fix up commit messages etc.
> git push origin master
```

Since we made sure the local master was up-to-date, this should result in a fast-forward merge, and git push should not complain about any of the non-fast-forward issues discussed above.

Delete a branch (local and remote)

```
> git branch -D branch_name
> git push origin :branch_name  (a branch name prefixed with a colon to git push deletes the remote branch)
```

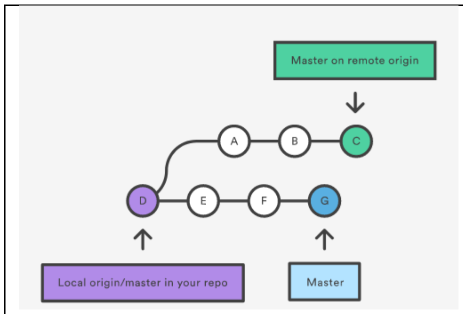
```
> git push <remote> <branch>

> git push <remote> --all

> git push <remote> --tags
```

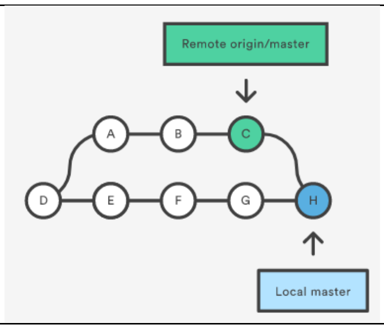
Tags are not automatically pushed when you push a branch or use the --all option. The --tags flag sends all of your local tags to the remote repository.

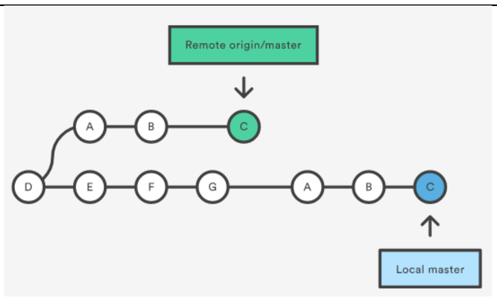
Pull

The git pull command is actually a combination git fetch followed by git merge. A new merge commit will be-created and HEAD updated to point at the new commit. Have in mind the >git pull --rebase. Many developers prefer rebasing over merging, since it's like saying, "I want to put my changes on top of what everybody else has done."

The remote master has progressed Your local master has progressed too	This is a typical pull that will do a fetch and a three way merge	This is a pull with the rebase flag

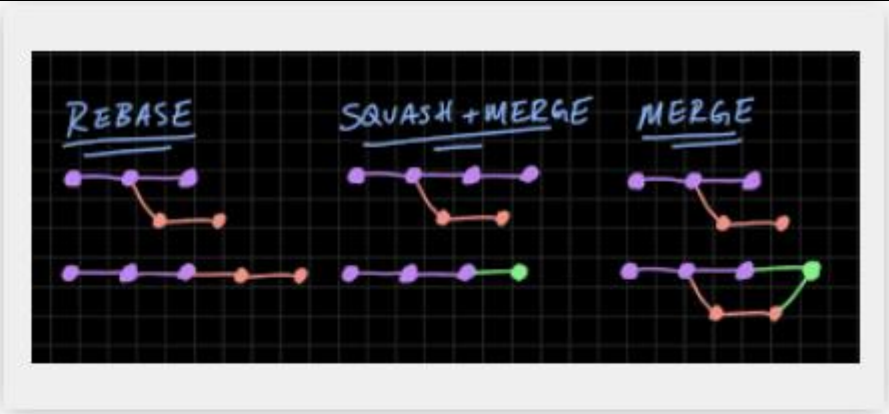






Merge

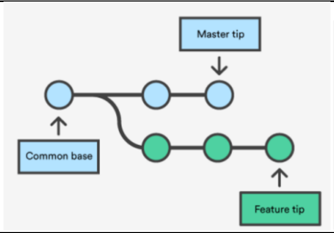
Merging is Git's way of putting a forked history back together again.

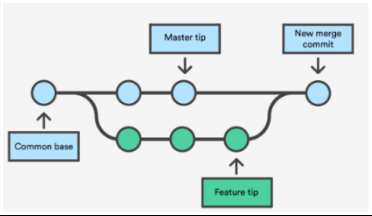


Squash, Merge, or Rebase?
matt-rickard.com

Git fetch upstream
Get merge upstream/master

- Typical merge (three way merge)

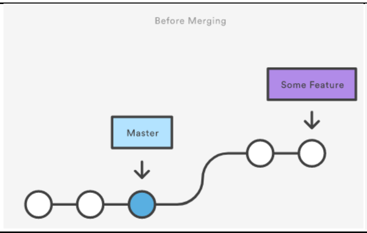


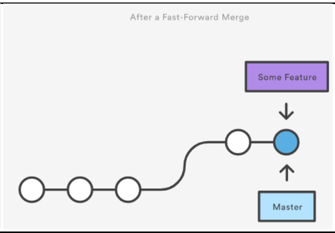


Merge commits are unique against other commits in the fact that they have two parent commits

The name three way merge, comes from the fact that Git uses three commits to generate the merge commit: the two branch tips and their common ancestor.

- Fast forward merge





But if the two branches, the one that will receive the merge (receiving branch) and the one that is to be merged to the receiving one (target branch) form a linear history then there is no need for a typical merge commit that has two parents. You just have a fast forward merge

Wherever possible for example for small features of bug fixes, you should prefer fast forward merges even if the receiving branch has changed. You just have to rebase the target branch to the receiving branch. Then you can do a fast forward merge.

Prepare to merge

- Checkout the receiving branch
- Fetch latest remote commits (so that your local branches that track the receiving branch and the merging branch are up to date)
- Merge (git merge <branch name>)

Merge conflicts

If the two branches you're trying to merge (three way merge) both changed the same part of the same file, Git won't be able to figure out which version to use. When such a situation occurs, it stops right before the merge commit so that you can resolve the conflicts manually.

Merge conflicts may occur if competing changes are made to the same line of a file or when a file is deleted that another person is attempting to edit.

If there are any merge conflicts in any git operation, the operation is not applied and you get a warning that notifies you about the conflicts. So you can resolve them and reapply the operation.

Once you've identified conflicting sections, you can go in and fix up the merge to your liking. When you're ready to finish the merge, all you have to do is run git add on the conflicted file(s) to tell Git they're resolved. Then, you run a normal git commit to generate the merge commit.

Resolve competing changes

Git modifies the file to mark the conflicts, so you must edit the file in a text editor and modify it to keep what you want and delete the git marks. Then you add the changes and commit them. This commit is the one that resolves the conflict.

`If you have questions, please`
`<<<<<<< HEAD`
`open an issue`
`=======`
`ask your question in IRC.`
`>>>>>>> branch-a`
	For example: edit the file, the <<<<<<< HEAD is the changes in the base branch and ====== separates the changes of the other branch. After that the other branch changes follow and they end with >>>>>>> branch name.
	So you manually edit what you want and delete the git marks (<<<<< etc.)

You modify the conflicted files

You git add them

You git commit (to generate the merge commit)

Resolve deleted file

You can either select to keep the file by staging it with git add filename or remove it git rm filename. Then commit the changes. This commit resolves the conflict.

git merge --abort can run only after merge conflicts to revert the merge operation and try to establish the pre merge condition.

git merge --continue can run only after merge conflicts

squash merge

Squash merging is a merge option that allows you to condense the Git history of topic branches when you complete a pull request. Instead of each commit on the topic branch being added to the history of the default branch, a squash merge adds all the file changes to a single new commit on the default branch.

Rebase

From a content perspective, rebasing is changing the base of your branch from one commit to another making it appear as if you'd created your branch from a different commit. Internally, Git accomplishes this by creating new commits and applying them to the specified base. So public branches the commits of which might have been used by other developers must not be rebased. The primary reason for rebasing is to maintain a linear project history. For example, consider a situation where the master branch has progressed since you started working on a feature branch. You want to get the latest updates to the master branch in your feature branch, but you want to keep your branch's history clean so it appears as if you've been working off the latest master branch. This gives the later benefit of a clean merge of your feature branch back into the master branch.

If there is a merge conflict during rebase, you fix it, you add the fix (without committing) and you do a git rebase --continue

```
> git rebase <base>
```

Have in mind that <base> can be any kind of commit reference (for example an ID, a branch name, a tag, or a relative reference to HEAD).

`git checkout mybranch` Make current the branch you want to rebase

`git rebase master` rebase the current branch with the base commit of the master branch

`git checkout master`

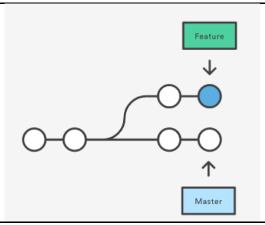
`git merge mybranch` merge the rebased branch to master and auto-commit the result

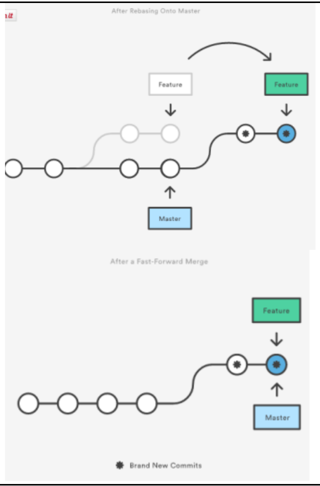
You have checked out the feature branch and git rebase origin/master will merge in the requested branch (origin/master in this case) and apply the commits that you have made locally to the top of the history without creating a merge commit (assuming there were no conflicts). No need to afraid of merge conflicts while rebasing:

1. If you frequently commit your ongoing work (which you should) and rebase, the probability of having unmanageable merge conflicts goes way down.
2. If you run into conflicts while rebasing, fix them, and `git rebase --continue`, the rebase will continue to go off without a hitch (there will be a merge commit, but once again it will contain useful information about where the conflict was and how it was resolved).
3. If the conflicts are too bad and you need to bail out and attempt a normal fast-forward merge, you can easily do so with `git rebase --abort` (leaving you where you were before attempting the rebase).

Rebase moves a branch to a new master commit, a new base commit. Rebase itself has 2 main modes: "**manual**" and "**interactive**" mode. You can call it also a fast forward merging.

Notice: you should never rebase commits once they've been pushed to a public repository. You should use rebase only for cleaning up your local work but never to rebase commits that have already been published, because rebase deletes the commits of the branch that is to be rebased and creates new commits. If the original commits are public then other people might have used them so you must not change them because this will mess with other people's histories.

	In this situation you have 2 options. Merge or Rebase and merge (fast forward merge)

 After Rebasing Onto Master Feature Feature Master After a Fast-Forward Merge Feature Master Brand New Commits	Rebase and merge Rebasing is the process of moving a branch to a new base commit. The primary reason for rebasing is to maintain a linear project history (to avoid forked history). From a content perspective, rebasing really is just moving a branch from one commit to another. But internally, Git accomplishes this by creating new commits and applying them to the specified base—it's literally rewriting your project history. It's very important to understand that, even though the branch looks the same, it's composed of entirely new commits. Rebasing is like saying, "I want to base my changes on what everybody has already done. Rebasing is a common way to integrate upstream changes into your local repository. Don't Rebase Public History. A word of caution: Only do this on commits that haven't been pushed to an external repository. If others have based work off of the commits that you're going to delete, plenty of conflicts can occur. Just don't rewrite your history if it's been shared with others.

Interactive rebasing

You rebase but you can control each individual commit of your branch before it is "rebased". You can squash some commits, delete others etc. This way you can see that many projects have a clear clean history because the devs have used interactive rebase to clean the insignificant commits when they were merging.

```
> git rebase --interactive <base>
```

This opens an editor where you can enter commands (described below) for each commit to be rebased. These commands determine how individual commits will be transferred to the new base. You can also reorder the commit listing to change the order of the commits themselves. Once you've specified commands for each commit in the rebase, Git will begin playing back commits applying the rebase commands.

Squash and Stash

Squash commits

Get rid of some commits and keep only the important ones when you push to a repository.

This is a great way to group certain changes together before sharing them with others. It actually melds (blends) the commit that has been marked as squash, with its previous commit. If you have 4 commits that you want to "merge" to one, then you squash the latest 3 commits. Each one will be melded to its previous and you will end up with one commit. Its something very common in interactive rebasing.

https://ariejan.net/2011/07/05/git-squash-your-latests-commits-into-one/ This is an easy example.

git commit –amend

It lets you combine staged changes with the previous commit instead of committing it as an entirely new snapshot. To Git, it will look like a brand new commit. Combine the staged changes with the previous commit and replace the previous commit with the resulting snapshot. Running this when there is nothing staged lets you edit the previous commit's message without altering its snapshot. Don't Amend Public Commits.

Stash

Often, when you've been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don't want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the git stash command.

Stashing takes the dirty state of your working directory — that is, your modified tracked files and staged changes — and saves it on a stack of unfinished changes that you can reapply at any time.

`git status` you might have some staged changes (changes that have been added) and unstaged changes.

`git stash` now if you run git status there would be nothing shown. At this point, you can easily switch branches and do work elsewhere; your changes are stored on your stack. To see which stashes you've stored, you can use git stash list

`git stash list`

```
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

`git stash apply stash@{2}` You can apply any stash of the available ones. If you don't specify one, then the last one will be used

`git stash apply --index` without --index the staged files (ones that you had added) are not restaged. --index does also that.

`git stash pop` apply the stash and delete it afterwards

Having a clean working directory and applying it on the same branch aren't necessary to successfully apply a stash. You can save a stash on one branch, switch to another branch later, and try to reapply the changes. You can also have modified and uncommitted files in your working directory when you apply a stash — Git gives you merge conflicts if anything no longer applies cleanly.

Collaboration models

1. You fork a public repo
2. You clone your forked repo to your machine
   - Origin is the forked repo in your github
   - Upstream is the original public repo (you must add this remote manually I think)

Pull requests

Suppose you want to add something to a public repository:

The normal Github PR process is to fork the original repo, clone it to your machine, make a local branch off master, make some commits on that branch, push that branch to your forked repo in Github, and then make a PR to the original repo from that remote branch of the forked repo. (Have in mind that if you push new commits to your forked repo branch they will be reflected in the PR. The PR will be updated because it tracks your forked repo branch from which it was created).

The original repo admin can then merge this PR to the master branch of his repo. Notice that if there are conflicts (shown in github before merging the PR) the admin (or you) can resolve the conflicts via the github page (or via the command line). After resolving the admin (or you) creates a merge commit (from the github page) on your forked repo. So your PR has been updated. You PR has an option to give the admins of the original write access to it (to your forked repo branch essentially). Then the admin can choose to merge your update PR to original master. Have in mind that there are a lot of ways to resolve conflicts. For example you might do a rebase via your terminal. Then push the changes. The PR will be updated and the admin might choose to merge it to original master.

Have in mind that you need to keep your local repo updated with the lastest changes in the original repo. So you need the original repo as a remote named "upstream" by convention. Then you can pull from that repo to keep your local repo updated.

https://gist.github.com/Chaser324/ce0505fbed06b947d962 a good short step by step description

https://www.youtube.com/playlist?list=PLwNuX3xB_tv8Gf2sELkXSqCCWcUq-AbrG a short list for how to contribute to Ivy (a specific open source project)

https://docs.github.com/en/pull-requests/collaborating-with-pull-requests

- You fork the public repository. This creates a copy under your account
- You clone the forked repository in your local machine
- You create a new branch locally
- You make the modifications on the source code, add the files and create commit
- You push your branch to origin (your forked repo). It does a fast forward merge and you have a remote feature branch.
- In github, you visit your forked repo you go to the feature branch and manually create a Pull request by pressing the corresponding button. Github knows the base repo of your fork so it knows where to apply the Pull request.
- In github, the owner of the public repo sees a new Pull request in his account. He can decide to merge it to the public repo by pressing a merge button. He can give his feedback. The proper to merge the pull request, is for the owner to create a remote for the forked repo and fetch the new branch, then marge it on master locally. If everything works fine then he can merge the pull request from github gui or by pushing to master.

Pull requests in Github

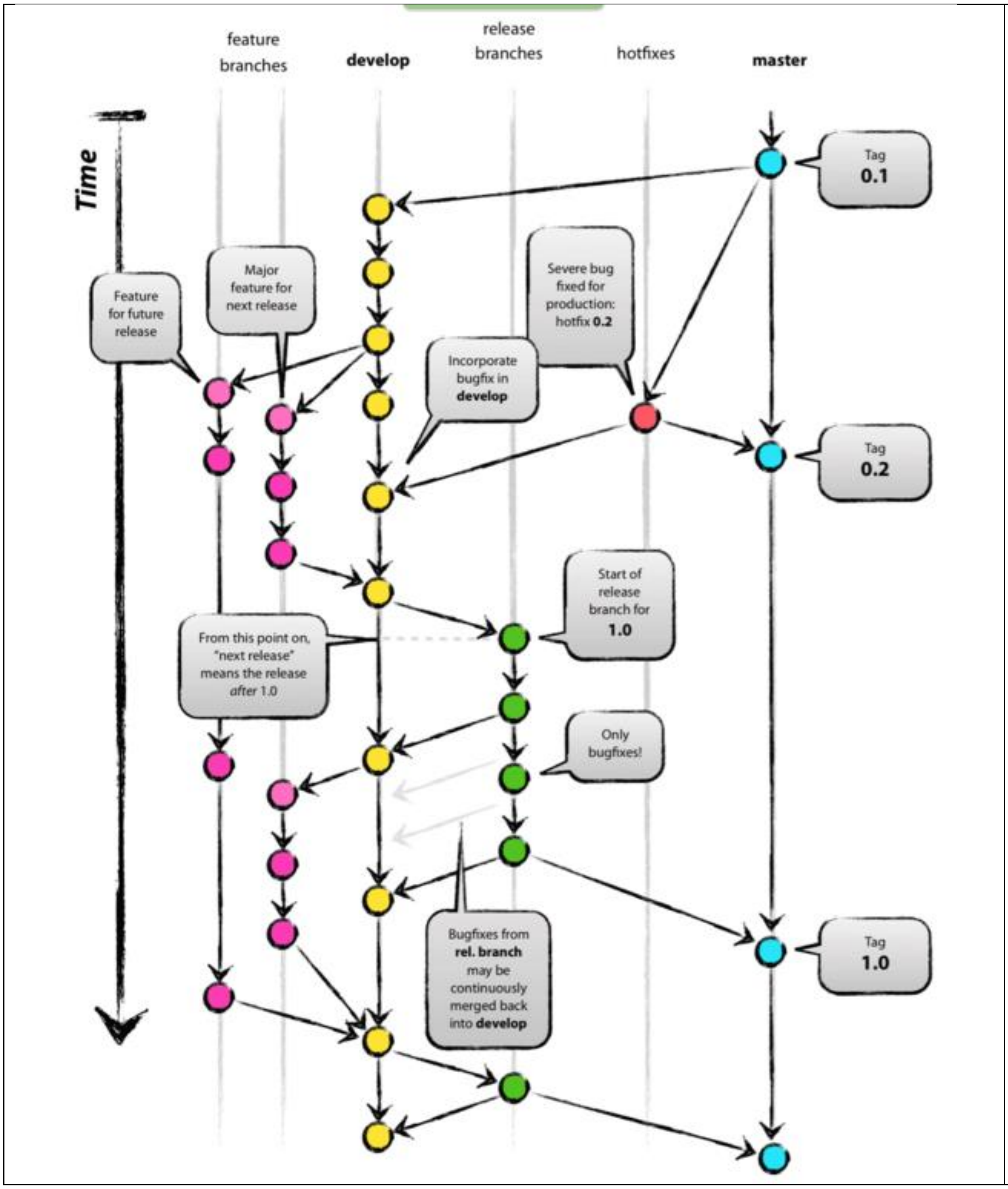
A repo can have issues. The issues have an issue number and a title. You make a new branch. You make a new commit on this branch to fix an existing issue. You create a pull request for that fix. In the name of the pull request you can refer to the issue that this solves by #issue number. You can also write the title of the issue like: ***issue_title*, fixes *#issue_number.*** If the pull request is accepted and the branch is merged with the master branch, you can delete your branch. Notice that the referring issue automatically switched to status closed.

Each repo has a default branch a base branch. Any pull requests and commits are against this default base branch. You can define another default branch if you want.

Notice that in general when a pull request has been accepted after the discussion and extra changes etc, then you must not merge it directly to the master branch. A good advice is to deploy the new branch in production and let your users use it for a while. If there are problems then you can very easily re-deploy your master branch. (Probably this is preferred from merging and then in case of problem, deploying a previous commit of the master since the master branch is adviced to be always deployable)

Nvie branching model

http://nvie.com/posts/a-successful-git-branching-model/


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Severe bug fixed for production: hotfix 0.2
Major feature for next release
Feature for future release
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Collaboration Models

Fork & Pull Model

The Fork & Pull Model lets anyone fork an existing repository and push changes to their personal fork without requiring access be granted to the source repository. When you have pushed a new branch in your fork github repo, you can go to your branch page and press the Create Pull Request button (You can send pull requests from any branch or commit but **we recommend that you use a topic branch** (a new branch for a specific topic) **whenever you change something**. That way, you can push follow-up commits if you need to update the pull request). It is advicable that before pressing the create pull request press the ***Compare & review*** button to see a review of the pull request that you will create. Your repo is connected with the original repo, so this will create pull request to the original repo. **After you send a pull request, any commit that's pushed to your branch will be**

automatically added to your pull request, which is useful if you're making additional changes. The changes must then be pulled into the source repository by the project maintainer or anyone with push access to the repository. After the merge the maintainer can optionally delete the brach.

This model reduces the amount of friction for new contributors and is popular with open source projects because it allows people to work independently without upfront coordination. [...] Pull requests are especially useful in the Fork & Pull Model because they provide a way to notify project maintainers about changes in your fork.

Keep your fork in sync with the original (upstream)

Fork a repository (create a copy of a github repository in your github account -the repo is not on your local machine but on github under your account). Then you can clone your fork in your local machine. This way your fork will be refered to as origin in your local machine. You can see this if you type ***git remote –v***. But if you want to **keep your fork synced with the original repository** you have to clone the original repository too, to your local machine and refer to it as upstream. (You have to have remotes both for your fork repo (***origin***) and for the original repo (***upstream***)). To do that you use the command: ***git remote add upstream original_repo_url***. The original_repo_url is the clone url that is found in any github repo. Now if you do git remote –v you wil see the origin and the upstream remotes. (Notice that there are 2 entries for each remote, one for fetch and one for push. I don't know why). Then the process is this: You fetch the upstream to your local machine. You checkout to your local fork master branch. You merge the local upstream master (that has been updated with fetch) with your local fork master (If your local branch didn't have any unique commits, Git will instead perform a "fast-forward). Now your local fork is synchronised with the original remote repo. In order to sync also your github you must push your local fork master to your fork on github. Then your github fork repo will be in sync with the original repo.

When you push some changes to your fork, you need to make pull requests to the original repo.

Fork and use example

If you are not familiar with pip and virtualenv first, please read the post about using pip and virtualenv first.

For example:

- You installed django-userena as such: pip install django-userena
- First you should uninstall it: pip uninstall django-userena
- Then go on the app's github page
- Click on the *fork* button
- This will make you a repository with a copy of django-userena
- Install it as such: pip install -e git+git@github.com:your-username/django-userena.git#egg=django-userena
- Then you can edit the code in yourenv/src/django-userena
- Push your commits

Shared repository model

You clone the original repository directly (you don't fork the repo and then clone the fork, you clone the original directly). In this case the original repo is referred to as ***origin***. Then if you want to change something you create a new branch in your local repo (your local folder with the .git folder in it.This folder has been created by the cloning). You commit on your new local branch. You push your new branch to the original remote repo with ***git push origin my-branch_name.*** This will, by default, create a my-branch_name branch on the origin repository on github. The previous command is the same as: *git push origin my-branch_name: my-branch_name*. The push doesn't automatically create a pull request. Then in order to create pull request you go to the remote repo, you select your branch and you press the ***Pull Request*** button (you go to your branch you want to be added to the repo and you press the pull request in the branches page). Then you have to press the ***Send Pull request*** button. A pull request is created. Then the reviewer can go to the pull request page and press the ***Merge*** button and the branche's commit will be added to the master branch. Notice that **in case that the file(s) that this pull request modifies was/were updated in the meantime** the reviewer will not be allowed to Merge in order for conflicts to be avoided. So in order to fix this, you have to download the new changes of the remote to your local repo by using ***git fetch origin***. This will add the new commits of the remote master branch that you didn't took into consideration to your local master branch. Then you check out your new branch (so that you are in your new branch now) and then you merge the master branch of the remote to your local new branch by doing ***git merge origin/master.*** This will not do the merging since there are conflicts (the files that you changed in your branch have changed aslo in the master branch before you merge your branch to it). You do this in order to see these conflicts in the report message and fix them. You can see the conflicts, and update your files (add the master changes) and commit again to your local new branch. Then you push again the new branch. Now there will be no problem with the Merging if no changes have been made to the master since your fetching.

Protected branches

Protected branches ensure that collaborators on your repository cannot make irrevocable changes to branches. If you own a repository with multiple collaborators who create branches and open pull requests, you may need to enforce branch protections to keep your project and pull requests organized and safe. Protected branches block several features of Git on a branch that a repository administrator chooses to protect. A protected branch:

- Can't be force pushed
- Can't be deleted
- Can't have changes merged into them until required status checks pass

Misc

Connect a local repository with existing files to a remote one

1. Create a repository in github, bitbucket etc.
2. Create a repository in your pc that includes your files with git init
3. Run the command git remote add origin https://username@bitbucket.org/dimyG/your_repo.git to add the remote repo to your existing local repo as origin
4. Commit your changes and push to remote.

All the settings files in the .idea directory should be put under version control except the workspace.xml, which stores your local preferences. The workspace.xml file should be marked as ignored by VCS.

For Django you run git init in the level in which the manage.py is.

`git push -u origin new-feature`

This command pushes new-feature to the central repository (origin), and the -u flag adds it as a remote tracking branch.
Git does not support empty file directories, so you will have to create a file inside an empty directory.

Log files and version control

I wouldn't store them in version control, at least in the same project as the source code for the deliverable product. To me, it just doesn't make much sense to combine what's effectively logging data in a directory structure for a deliverable product. The repository used for the project shouldn't contain anything outside of what is needed to build and deploy/deliver the system at hand (including instructions for building and deploying).

I would recommend storing them in the same place or using the same methodology as you do your other documents, such as test result history, design documents, requirements specifications, and so on. I don't know what technology you use, but uploading them to a **SharePoint server** would work, as would uploading them to a shared data server and putting the path to them (and perhaps summaries of them) on a wiki page.

merge local branch and do the same in github

I created a local branch. Then I pushed that local branch to github. So github now has two branches.

Now I want to merge the two branches together both locally and on github.

This is probably a dumb question but somehow I'm not seeing it in the documentation.

You merge the branch locally. Then you push the merged branch to github. Github will now contain the merge. Then delete your local and remote copy of the feature branch.

Collaboration approach

http://stackoverflow.com/questions/5601931/best-and-safest-way-to-merge-a-git-branch-into-master

```
git checkout master

git pull origin master

git merge test

git push origin master
```

Multiple branches and merge conflicts

You can have 2 branches apart from the master branch. You make changes to the same file in both branches. You can merge one branch with the master. But if you go to merge also the other one afterwards git will not do it. You will get a warning that there is conflict. You can resolve this by deleting the 2nd branch entirely. Or you can use another git tool called mergetool (you need to install it). It opens a window it shows the conflicts you select which one you want to keep. Notice that there will be no conflict if you change different files in the 2 branches.

Reverting

Whenever you want to delete one previous commit, instead of completely deleting it (which might cause integrity errors), you create a new commit that undoes whatever the commit you want to delete does. So it is like an undo but keeping the information that the undo has happened.

You make active the branch that you want

You make a change and commit it. You have created a new commit.

Then you decide that you want to undo it.

git revert HEAD (it will open a text editor in which you can type the commit message since the reverting is to create a new commit that automatically undoes the last commit by creating a new one)

git bisect Check it

Resetting

It is the act of actually deleting a commit. Not use it in general. It is bad practice. It is permanent you can't undo.

git reset (reset the staging area to macth the last commit. The working directory will remain unchanged)

git reset aCommitId (reset the staging area to match the defined commit. The working directory will remain unchanged)

git reset - -hard (reset the satging area and the working area to match the last commit)

git reset - -hard aCommitId (reset the satging area and the working area to match the defined commit)

Cleaning

Delete the untracked files from the working directory. It is permanent you can't undo.

Git clean –n (shows which files are going to be deleted)

SSH communications

You can use different github accounts from one computer using SSH keys.

ssh-keygen –t rsa –C github_account_email (this generates a pubic rss key on your local machine. Then you can login to your github account and add this ssh key. When you do that, you could use ssh to connect to github without defining passwords) You can create a ssh key in your local machine for as many github accounts as you want and then you can connect to them from your machine. So actually you can use ssh for the communication between git on your local machine and github server instead of https which is the default method and we used so far. For example you now use ssh commands to connect to github in the command git remote add origin git@github:dimuG/myTestRep01 you use ssh instead of https (instead of a url).

In fact, when you create a repository on GitHub, you have a choice of automatically including a license file, which determines how you want your project to be shared with others.

There is alos the concept of collaborators. You can define a user to be a collaborator in your github repo. This user will have all the rights to change the repo. A user who is not a collaborator must make a pull reguest.

Tags

You define the version

A commit can have a tag that characterizes it. They appear as releases in github in the releases menu.

- In the commit message you mention what exactly you changed and
- in the tag you usually define a version.

git tag (list all the current tags that your repository use)

git tag –a v0.2 –m 'version 0.2' (you create a tag for the last commit that you made. You name it v0.2 and you define a message for this specifc tag)

git show v0.2 (it lists all the details about this tag)

git tag –a v0.5 8decff (give a tag to a specific commit by using the 6 first letters (the partial checksum) of the commit's hash value that you can take from git log - -pretty=oneline. It opens a file in the text editor and you can define your message for this tag in the first line of the file without #. It is not a comment)

git push originNewName v0.5 (it pushes the specific tag to your remote rep on github. Then you can see this tag in the "**Releases**" tab of github. You will see a v0.5 release.)

git push originNewName - -tags (it pushes all the tags of this rep to github)

So the concept is that first you define the directory of your project as a git repository. Then you select which files you want to stage and then you commit them. Then you can make a change to a file and save it with the same name. There is no need to define a different name for versioning. Git will do this. After the change you stage this file and then you can commit it.

Branching

Create a new local branch and track with it a remote branch

You have a new branch and then you do:

git push -u <remote_repo_alias> <local branch name> for example git push -u origin payments_01

-u is a shortcut for the option --set-upstream which sets the default remote branch for the current local branch. Any future git pull command (with the current local branch checked-out), will attempt to bring in commits from the <remote-branch> into the current local branch.

Examples

git branch fix20 (fix20 is the name of the new branch)

git checkout fix20

or

git checkout –b fix20 (that does what the previous two do. Now whatever we do is done in the new branch)

vim myfile.txt (make changes)

git commit –a –m 'change 01'

git checkout master (switch back to our master branch)

git push origin fix20 (push the new branch to our remote repo in github, which has the local alias origin)

git merge fix20 (merges fix20 with the current branch)

git branch –d fix20 (delete the branch. Since we merged it there is no need to keep it). With **git branch –D fix20** you can delete an unmerged branch

We deleted it from our local repo but it is still on github. So we have to delete from there also.

git push origin :fix20 (deletes the fix20 branch) Don't we need to merge fix20 with master also on github? Maybe not, we can just push our merged master to github. Maybe we created the fix20 on github just to have an identical history locally and on github.

git checkout -- path/to/a_file If you made changes to a file that you want to "undo", this command will restore the file to the state it is in the index.

If you want to pull a branch from someone elses github repo to your local repo then you need to use the command **git checkout –b fix30 origin/fix30**. First you create locally a new branch named fix30 and then you get the files of the fix30 branch of the origin remote repo.

Forked history

When you end up with forked hostiry you must "heal" it using merge or rebase.

Associate a local repo with a remote repo and push local data to it

You create an account on Github, and then you can create repositories that you connect with your local repositories and then you can push and pull data (push and pull from the local point of view). Push local data from the git terminal on your local computer to the remote github repository. Pull remote github repository data from the git terminal to your local git repository.

You have to login to github and create a repository. You will get a url that points to that repo. You can then push an existing local repository of your local machine to the github repository with the command:

git remote add <RemoteName> <RemoteURL> (connect the defined repository on github with the local repository that you are in. The remote repo will be refered to locally, with a local alias named RemoteName. NOTE: As used here, **RemoteName represents a local alias (or nickname) for your remote repository. The name of the remote repository on the server does not necessarily have to be the same as your local alias.)** This command creates a remote in the /remotes/ folder of the git repo. **Remotes are simply an alias that store the url of repositories**. You can see what url belongs to each remote by using **git remote –v**

Create a repository in github.

You get a url that points to that repository

git commit –m ‘initial project version’ (to commit all changes to your local repository)

git remote add origin https://github.com/dimyG/testRep01.git (origin is the local alias for the defined remote repo named “testRep01”)

git push origin master (you push the master branch of your local repository to the remote testRep01 repository the local alias of which is origin (to its master branch). If the branch does not exist in the server it will be created. You need to give password for your github account)

git remote show RemoteName (Print information about the specified remote repo to the console window)

Example

since you ‘connected” your local git repository to the remote one, if you commit something in your local rep it will be commited also to the remote one? No you have to push it manually.

create a file in your local repo, stage it and commit it to your local rep. Then push it to your remote rep.

create file

git add myfile

git commit –m ‘comment’

git push (Update the remote server with commits for all existing branches common to both the local system and the server. Branches on the local system which have never been pushed to the server are not shared)

git push RemoteName BranchName (Updates the remote server with commits for the specific branch. This command is required to push a new branch from the local repo to the server if the new branch does not exist on the server.)

git remote rename origin originNewName (change the name of the remote rep as it appears in your local machine)

you can use also the url directly:

git push git@github.com:git/git.git master   instead of

git push origin branchname

“Private collaborators may fork any private repository you’ve added them to without their own paid plan. Their forks do not count against your private repository quota.”

Notice:

You have a branch locally. You push it to the server (git push origin my_branch). Then someone makes changes to the branch in the server. You want to pull it. You can’t just do git pull origin. You have to tell git “which of your local branches is associated with the remote branch (that is also locally) that is connected with the branch on the server”. You do this with

git branch --set-upstream my_branch origin/my_branch

But instead of doing this every time, you can use the command

git push –u origin my_branch

when you first push your branch to the server. This way the my_branch on the server is tracked by the my_branch locally. Also note that even if you forget the -u the first time you push, you can run the push again with that flag and it will start tracking

Or if your current branch is my_branch, you can use the git branch --set-upstream-to origin/my_branch (or the shortcut git branch -u origin/my_branch). It's the same with git branch --set-upstream my_branch origin/my_branch.

Github Forking

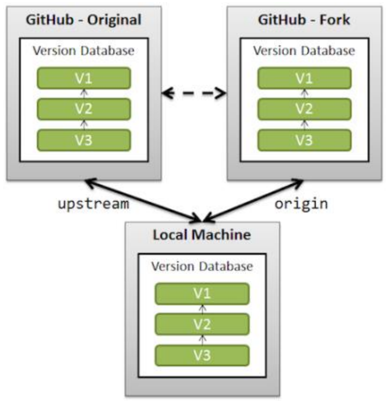
Browse github, find a repository and select **fork**. This will create a copy of the repository on your account on github and the url of this repo will change of course. It will not be the url pointing to the repo of the owner but to your repo. Then you create a new folder in you local machine and you clone (copy) that fork (your remote copy) in your local repo.

mkdir newdir

cd newdir

git clone https://github.com/dimyG/google_pacman (this will create a repository on your local machine)

So you copy the url of this repository from the respective tab and clone it to your local machine in a git repository. (Notice that after you have fork it the url to clone changes to the rep of your account). **The fork is a clone made in the server side**, in github. One more difference between fork and clone, is that when you fork a repo you can contribute the changes to the original repo of the user from which you forked. If you just clone in your local machine and push it to your github account, you can't contribute to the original repo unless if you are declared as a contributor explicitly. The original repo is called "**upstream**" while the forked repo in your github account is called "**origin**". So when you make a fork the code becomes yours. You can clone it to your local machine and make any changes you want. Then you can send a pull request to the upstream.



Notice that if you want to keep a link with the original repo (also called upstream), you need to add a remote, referring that original repo. When you make a fork and at the same time you want to keep track of the original repo you need to add another remote to your local repo:

git remote add upstream your_forked_url (this command will assigne the upstream alias to the original repo of the forked repo thah you defined. So upsetram is reference to the original repo).

You will use this remote to fetch from the original repo in order to keep your local copy in sync with the project you want to contribute to.

git fetch upstream

You will use the "origin" remote for pull and push since you can't change the original (upstream) repo.

You will contribute back to the upstream repo by making a pull request.

Some Commands

cd – (go to the previous dir)

cd "/c/Some Folder/Some Other Folder/" (instead of c:\some folder\)

ls [options]

Options:

-1 = List 1 item per line

-r = Reverse the sort order

-a = Show Everything, including hidden items

-d = list only directories

-l = (letter L, lowercase) = Use a long listing format (more info per item, arranged in columns, vertical listing)

mkdir /c/ExistingParentFolder/NewFolderName

mkdir -p /c/NewParentFolder/NewFolderName (the NewParentFolder doesn't exist but it will be created)

touch newFile.txt (create a single (empty) text file in the current directory)

touch newFile_1.txt newFile_2 . . . newFile_n

echo "This text is added to the end of the file" **>> newFile.txt** (Append text to a file. If the file does not exist, one is created)

echo "This text replaces existing text in the file" **> newFile.txt** (Overwrites text in a file. If the file does not exist, one is created)

rm DeleteFileName (remove a file)

- **git rm file.txt** removes the file from the repo but also deletes it from the local file system.
- **git rm --cached file.txt** To remove the file from the repo and not delete it from the local file system use this

rmdir DeleteFolderName (Removes the specified folder if empty. Operation fails if folder is not empty)

rm -rf DeleteFolderName (remove the directory and all its contents)

git config - - global user.name 'dimyG'

git config - - global user.email dimgeows@gmail.com

git config - - list (you see the settings)

git config - -global core.editor "vim" (change the default text editor to whatever one you want)

Staging area or Git index

The **staging area** is also called **git index** or **cache.** The staged files are not actually in the git repository. You have to commit them to get into the repo. So whenever you make a change in a file, you have to put it in the staging area first and then commit it.

Suppose that you create a new README file.

1. Use "**git add README**". This causes **git** to:
    1. Copy the **contents** of the README file to the git object store and store its contents as a blob. The blob is the main object type in git. The "blob" type is just a bunch of bytes that could be anything (for example a text file, source code, executable binary file, or picture, etc.)
    2. Record in the git index: Filename README and the hash of the README blob in the object store (in .git/objetcs)
    3. Mark (remember) the README file in the user's working directory is now "tracked".
2. Use "**git commit**" which takes the files that are referred to in the "git index" and saves them to the git repository.

http://www.gitguys.com/topics/whats-the-deal-with-the-git-index/#prettyPhoto[postimages]/0/ This is a very good diagram to understand the whole process

start tracking a directory (tell git to use this directory as a git repository)

cd /to the dir/   (ls to see the files or ls –a to see also the hidden ones)

git init (this will create a .git folder inside the selected folder and the selected folder will become a git repository). For Django you run git init in the level in which the manage.py is.

git add myfile.py (Stage this file. **You put it to the staging area.** Its ready to be commited)

git add *.py (start tracing all the python files inside this directory. Put them in the staging area)

git add . (stage the entire directory. Actually it adds new or changed files to the staged changes for the next commit, but does not add removals.)

git add -A *(or **--all**)* (Add all new or changed files to the staged changes, including removed items (deletions))

git add -u = Add changes to currently tracked files and removals to the next commit. Does not add new files.

ignore certain files

create a file called **.gitignore** inside the git repository

this file is a txt that contains the files, folder or file types you want git to ignore. These files are not going to be used by git. Probably it means that nothing that git does (versioning etc) will be made for these files. They are just inside the git repository as if it was a regular folder. The .gitignore file should be in your repository, so it should indeed be added and committed in, as git status suggests. It has to be a part of the repository tree, so that changes to it can be merged and so on.

an example of a .gitignore file:

# Generated files

Bin/

Gen/

#compiled dynamic libraries

**.so*

**.dylib*

.dll

#executables

**.exe*

**.out*

git commit –m 'Initial Project Version' (you commit all the files that you are tracking, that are in the staging area. The –m means commit it and define a message that is connected with this commit. **A commit message**. Whenever you commit a tree object is created that contains the blobs of the commit with their hash values and filenames. A commit object will also be created that will contain the tree object, the commit message, the author and also a reference to the previous commit its parent commit.

git commit –help to see all the commit options

git status (you get a list with all the files in the repository and their status. You will have a list with the **untracked files**, **the ones that exist in the working directory but they are not tracked**. The ones that you have defined to be ignored are not listed at all). You see:

- What files have been added to the git index (for example by using the **git add filename** command).
- What files git notices that exist in the working directory, but aren't in the git index.
- What files have different contents between the version in the git index and the version in the working directory.

git-clean - Remove untracked files from the working tree

Open a file that you track (let's say test.py) and make a change.

Then run

git diff (You see the changes that you have done in your tracking files. It has made internally 2 versions of the file.)

then you have to stage the changed file. You see one diff "block" for each file that has changes. You quit by pressing "q"

git add test.py

If you run

git diff –cached (it shows you what hasn't been commited yet. What is in the stage level)

Then you commit all the changes

git commit

This will create a file that will be opened in the bash with your text editor. The file contains information about the change. In this file it is advisable to add your own commit message in the beginning. A title. Then a description. And a bullets list of what you did using * as bullets. Add a comment that this change resolves a bug and define the bug id. Notice that these messages are written without # in front of them. They are not comments. These notes are the notes that are written in this file when you use the –m option. When you use it the file is created but you don't see it. The use of –m is a shortcut.

So you could do:

git commit –a –m 'second change' (the –a skips the manual staging, meaning that you don't have to type explicitly the add command to stage the file that you have changed. It does that automatically. And the –m is a shortcut for the commit message. But it is advisable not to do this skipping all the time because one big point of versioning is to be able to see what changes you have made to be able to track your bugs etc.

git commit - -amend (you can modify the commit message of your last commit)

remove a file from the staging area

git add myfile.txt

git reset HEAD myfile.txt

delete a file with git

git add myfile.txt

git rm –f myfile.txt (you need to force the deletion when a file is staged, you can't just delete it. You have to force it to be deleted)

git rm - -cached myfile.txt (it removes it from the staging area without deleting it from the working area. It is in the untracked files.)

git mv myfile.txt myRenamedFile.txt (rename a file)

git log (It lists all your commit messages by reverse chronological order)

git log - -pretty=oneline (very commony used. It lists all your commit messages but each one in one line. Before it was a txt file that had additional info also)

git log - -pretty=format: %h : %an : %ar : %s (h is the first letters of the hash value of the commit which is something like an id, an is the user, ar is the date, s is the print the first line of the commit message. It shows in each commit in one line with the requested format)

git log –p -2 (show the last 2 commits)

git log –stat

git log - -since:1.weeks (show the commits of the last week)

git log - -since:"2014-04-01"

git log - -author="Derek banas" (show the commits of a specific user)

git log - -before:"2014-04-01"

git commit -m 'initial commit'

git add forgotten_file

git commit --amend

You end up with a single commit – the second commit replaces the results of the first. But, amending doesn't just alter the most recent commit—it replaces it entirely. To Git, it will look like a brand new commit. It's important to keep this in mind when working with public repositories.

Misc

Tags

A tag is used to label and mark a specific commit in the history. It is usually used to mark release points (eg. v1.0, etc.).

You will not be able to checkout the tags if it's not locally in your repository so first, you have to fetch the tags to your local repository.

# --all will fetch all the remotes.

# --tags will fetch all tags as well

$ git fetch --all --tags –prune

You can checkout tags like this

$ git checkout A ...

$ git checkout version 1.0 ...

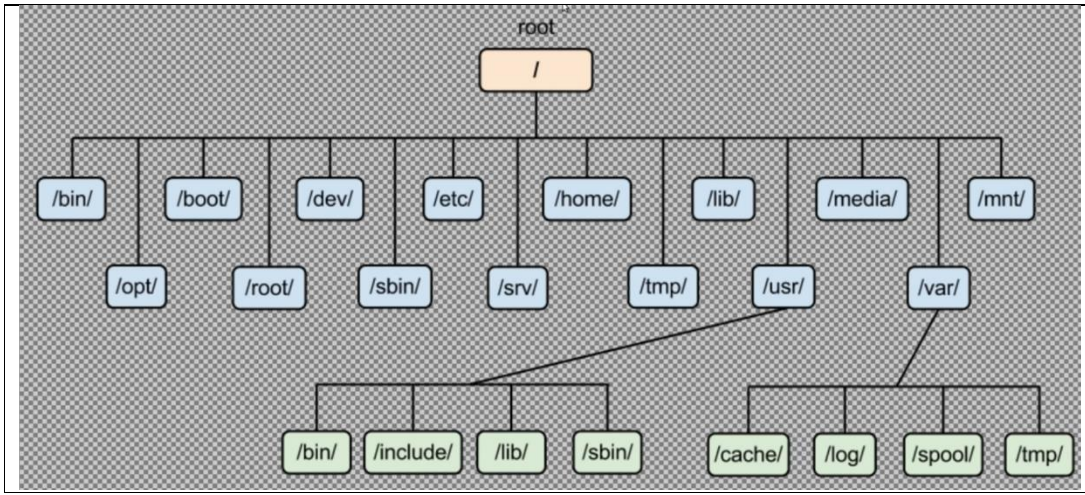
$ git checkout tags/version 1.0 ..

Linux Ubuntu

Filesystem

Linux uses / Windows use \

Capitalization is important. File and file are different files.



The Filesystem hierarchy standard (FHS) defines the structure and layout

Pseudo files are files that don't actually exist on the storage device. It is information which appears as files. This information might be in RAM for example.

Root drive directory is the upper most directory which contains all others. Not to confuse it with the /root directory which is the home folder of the root user.

/bin

Means binaries. This is the location for various installed programs. Ls, cat etc are all installed here

/sbin

These are system binaries. A standard user doesn't have access to without permission

/boot

Contains all files your system needs in order to boot. In other words the boot loaders live here. You don't want to mess with the contents of this folder

/dev

Means devices. Linux follows a standard according to which everything is a file. For example a disk is a file named sda. A partition on that disk would be named sda1, sda2 etc. The keyboard, webcam and any other hardware has a corresponding file that lives in this folder. These are all pseudo files which means that they are not actual files on the system but the kernel transforms information to appear as files.

/etc

Means etc. All system wide configurations (not user specific configurations) are stored here, for example configuration for apt. For example there is an apt folder here which contains a sources.list file with all the repos your system connects to. Libraoffice would not store config here since it would store config for each user and that would be in a different folder.

/lib /lib32 /lib64

Libraries are files the applications use to perform various functions. They are required by the binaries in /bin and /sbin for example.

/media /mnt

mnt means mount. These are the folders where the mounted storage devices are. It could be a usb stick, external drive, network drive, second hard drive. /media is the folder where the storage devices mounted by the system are while /mnt is for drives mounted manually by a user. The user has to select the /mnt to mount them and leave the /media for the OS to manage. So a usb would be in /media/username/devicename/

/opt

Means optional. Manually installed software from vendors reside here usually, but some software packages found in your repo might also be installed here. The instructor uses this folder for placing all of his software projects.

/proc

It contains pseudo files that contain information about system processes and resources. For example every process would have a directory here with information about that process. Every running process has an id (pid). There is a folder here for every pid named as the pid for example if the pid is 2344 the name of the folder that contains information about that process would be /proc/2344/ you can find information about the cpu here too, for example >cat /proc/cpuinfo, or >cat proc/uptime etc.

/root

It is the root user's home folder. Unlike home folders for all other users it is not reside in the /home directory. One reason that it is placed here is that it ensures the root user will always have access to his home folder. The /home folder instead could be placed in another drive which you might not be able to access for some reason (eg. Errors).

/run

Its a tempfs (temporary file system) which means it runs in RAM. Everything that lives here is gone when the system is shut down. Its used for processes that start early in the boot procedure to store runtime information that they use to function.

/snap

The snap packages are stored here. Snap packages are completely self contained applications that run differently than regular packages.

/srv

Means service. Service data is stored. If you run a web or an ftp server, it will store here all files that are to be accessed by the clients. This allows for better security since it is in the root of the drive and also allows you to mount this folder from another hard drive (the actual files are in another hard drive which you mount to your system)

/sys

Means system. It offers a way to interact with the kernel. The files in this directory are stored in RAM. They are not physically written in the disk.

/tmp

Means temporary. Temporary files used by applications during a session are stored here. An example is when you edit a word document the program regularly stores some temporary backups of your file so that it can be restored if there is a crash. These temporary files are stored in /tmp. Usually the folder empties when the system is reboot. There might be some files that are stack here and can't be deleted. If they are big enough and you want to get rid of them you have to log in as root in single user mode and manually delete them.

/usr

Means Unix System Resources. Here applications that are used by a user are installed as opposed to the /bin folder. Any applications that are installed here are considered by the OS as non essential for basic system operation. Programs are installed in the /usr/bin/ or usr/sbin/ or /user/local/bin/ or /usr/local/sbin/. most applications installed from source code will be installed in the local folder. The structure of the /usr folder adheres to the FHS but many applications might not follow it exactly so you might have to search where exactly are installed in the /usr folder.

/var

Means variable. Contains files that are expected to grow large, be variable in size for example log files of various programs . /var/crash/ holds info about processes that have crashed. /var/log/

/home

Each user has a distinct folder inside it. Here users can store their personal files which other users can't access unless they have permissions. The home shortcut leads here. You can mount a user's home folder from a different partition of your hard drive, or from a network storage. Doing so you will be able to re install your whole system (the OS) without losing your files. /home/username/.cache/ apps like the browser use it to store temporary files. /home/username/.config/ for individual application settings. Although some apps store their user specific config directly in the /home/username/ folder. /.icons/ and .themes/ can be used to store your themes and icons for the customization you did. You can back them up and place them again after a re installation. Or if you backup all the config folder then after a re installation you only have to re isnatll the apps and the settings will be the previously used ones.

Mounting

All files accessible in a Unix system are arranged in one big tree, the file hierarchy, rooted at **/**. These files can be spread out over several devices. The **mount** command serves to attach the file system found on some device to the big file tree. Conversely, the **umount** command will detach it again. The file **/etc/fstab** may contain lines describing what devices are usually mounted where, using which options.

> *mount -t type device dir*

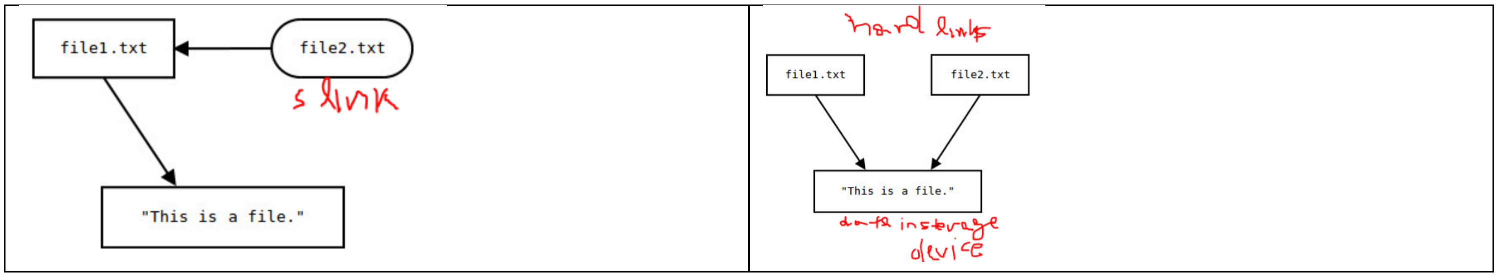
Let's say the CD-ROM device is /dev/cdrom and the chosen mount point is /media/cdrom. The corresponding command is

> *mount /dev/cdrom /media/cdrom*

Mounts may be local or remote. The user or the OS must make the device accessible through the computer's file system. A mount point is a physical location in the partition used as a root filesystem. This is unlike Windows where there is one directory tree per storage component (drive). Meaning that each mount directory has its own root folder which is a distinct drive.

Soft and hard links

A link is an entry in your file system which connects a filename to the actual bytes of data on the disk.



A **hard link** (can refer local files only) is essentially a label or name assigned to a file. Conventionally, we think of a file as consisting of a set of information that has a single name. However, it is possible to create a number of different names that all refer to the same contents using the link command. Hard links cannot refer to files located on different computers linked by NFS, nor can they refer to directories. For all of these reasons, you should consider using a symbolic link, also known as a soft link, instead of a hard link.

A **soft link** (can refer to local and remote files and directories) or Symbolic link is a special kind of file that points to another file, much like a shortcut in Windows or a Macintosh alias. Instead of linking to the data of a file (like hard links), they link to another link. This has some benefits. They can link to directories, or to files on remote computers networked through NFS. > *ln -s source_file.txt mylink.txt* After you've made the symbolic link, you can perform an operation on or execute it, just as you could with the source_file. You can use normal file management commands (e.g., cp, rm) on the symbolic link. A broken symlink, is a symbolic link which points to something that no longer exists.

> ln oldfile.txt newlink.txt.

This will create a new item in your working directory, newlink, which is linked to the contents of oldfile. Now both oldfile and newlink point to the same data on the disk. The new link will show up along with the rest of your filenames when you list them using the ls command. You can use the standard Unix rm command to delete a link. After a link has been removed, the file contents will still exist as long as there is one name referencing the file.

Single user mode?

Pseudo files

Package managers

Installing software

You can install software the traditional way as in windows by going to the web to download a package, but it's usually better to check the Ubuntu Software Center for any program you might want to install. Ubuntu Software Center contains some default **repositories**. Each repository contains information for a software or a collection of software. Ubuntu can download quickly and easily the software contained in a repository. It knows where to download it from. Each program available in the repositories is thoroughly tested and built specifically for each version of Ubuntu. Repositories can be already compiled or can be the source code (defined as ***deb*** and ***deb-src*** respectively inside the sources.list file). To add a repository you can modify the sources.list file.

All stored repositories (or software channels) are listed by the **apt Package Manager** (apt, Advanced Packaging Tool) in ***/etc/apt/sources.list*** or in any file within a folder with the suffix .list before its name ***/etc/apt/sources.list.d/*** By editing these files from the command line, we can add, remove, or temporarily disable software repositories. These files are basically the roadmap for your system to know where it may download programs for installation or upgrade. All your repositories are listed in the sources/list file. Every time you add a repo this file is updated. When you do *apt-get install app0*, apt would install the app0 as it is currently listed in your computer's local software lists, your sources list. Your software lists might contain an old version of the app0 since you might have added this repo some time ago (or they might not contain it at all if you haven't added its repository). So you need to run ***apt-get update*** that updates your local software lists from the internet, with the proper versions for each repository. Then when you run ***apt-get install*** the proper software version will be installed. I also saw referring to it as "updating the package manager cache". It's always a good idea to backup a configuration file like sources.list before you edit it. > *sudo cp /etc/apt/sources.list /etc/apt/sources.list.backup*

Everything in the default repositories is reviewed by the Ubuntu team before it goes out so it's safe. But to get a new version of a program found in a default repository, you have to wait for a new Ubuntu release. So to overcome this, there are the PPAs which is like a repository.

apt-get update

apt-key

apt-get autoremove

apt-get install a_package=2.3.5-3ubuntu1 define a version

apt-get source a_package download and unpack source code of a package to a specific directory

apt-get build-dep a_package_name "install all dependencies for 'a_package_name' so that I can build it". This command searches the repositories and installs the build dependencies for <package_name>. If the package is not in the repositories it will return an error.build-dep is an apt-get command just like install, remove, update, etc.

apt-cache tool is used to search software packages, collects information of packages and also used to search for what available packages are ready for installation on Debian or Ubuntu based systems.

apt-cache pkgnames list all packages

apt-cache policy a_package_name

apt-cache showpkg a_package_name list the dependencies for particular software packages.

PPAs

A PPA, or **Personal Package Archive**, is a repository (collection of software) not included in Ubuntu by default. A PPA can contain one or more programs. As anybody can create a PPA there's no guarantee for quality or security of a PPA. Usually the worry for me is not malicious intent, but conflicting packages.

On the command line you can add a PPA using > *add-apt-repository* command, e.g.:

sudo add-apt-repository *ppa:hotot-team*

sudo apt-get update

sudo apt-get install *hotot*

Install software by compiling it from source (using the make tool)

Packages installed with apt-get are already compiled (for the specific version of Ubuntu), they are executables (they are binary files)

Generally, when a program is distributed as source code (there is no Ubuntu repository or ppa for it so you can't add it in your repository list and apt-get it), the archive containing the code will also include Makefiles. make can read these files and turn the source code into an executable program. The make command runs a series of tasks defined in a Makefile to build the finished program from its source code.

build-essential is a metapackage (contains other packages) it's a package which contains references to numerous packages needed for building software in general. It contains tools like the g++ and gcc compilers (which are the gnu c++ and c compilers), the make tool, etc. for compiling/building software from source. So you can start with (usually C) source files and create executables from them.

```
The general process of making an executable from source code is:

cd ~/Downloads/

wget  Error! Hyperlink reference not valid.

tar -xjf the_tarball_file.tar.bz2

cd to the extracted dir which is the source_dir
```

```
Using checkinstall and auto-apt is the appropriate way of compiling from source

auto-apt run./configure --prefix=/home/myname/apps

make

sudo checkinstall
```

The configure script is responsible for getting ready to build the software on your specific system. Unix programs are often written in C, so we'll usually need a C compiler to build them. In these cases the configure script will establish that your system does indeed have a C compiler, and find out what it's called and where to find it. You can define the location of the installation using the prefix argument. /configure --prefix=/home/myname/apps

If you are in a 64bit linux then by default you will build 64bits software. To build 32 bit software you need some –devel and -32bit libraries. To do so you need to define some options in the /configure script of make: ./configure --build=x86_64-pc-linux-gnu --host=i686-pc-linux-gnu to compile for 32-bit Linux in a 64-bit Linux system.

Another (probably better) way to do it is to set up a 32-bit chroot jail and compile it there. It at least guarantees that you won't mix 32 and 64 bit libraries when compiling but it's an overkill in some cases.

auto-apt If a program tries to access a file known to belong in an uninstalled package, auto-apt will install that package using apt-get. For example: You're compiling a program and, all of a sudden, there's an error because it needs a file you don't have. The program auto-apt asks you to install packages if they're needed, stopping the relevant process and continuing once the package is installed. You use it like this: instead of just using ./configure you write # auto-apt run ./configure

Checkinstall

CheckInstall keeps track of all files installed by a "make install" or equivalent, creates a Slackware, RPM, or Debian package with those files, and adds it to the installed packages database, allowing for easy package removal or distribution. Use > *sudo checkInstall* instead of just running "> *sudo make install*", as that will likely put files all over the filesystem, with no easy way of removing them if things go wrong. If in the future you try to install a package that contains the same file as the software you compiled, you will receive errors and the software you compiled may stop working. CheckInstall will create and install *a deb file* that you can then uninstall using your favorite package manager. The installed package can be removed with synaptic or from the terminal with sudo apt-get remove packagename (or sudo dpkg –r packagename)

dpkg

dpkg is a package manager for Debian-based systems. It can install, remove, and build packages, but unlike other package management systems, it cannot automatically download and install packages or their dependencies.

aptitude and synaptic

apt-get and aptitude are both front ends to dpkg (which is the lowest level package manager). But aptitude is higher level than apt-get and it's newer. There is also another high level manager called synaptic (a graphical front-end to apt).

python

Ubuntu 14.0.4 have repos named python and python3 which refer to versions 2.7.5 and 3.4.0. You can install another version compiling its source code. Ubuntu 12.10+ and Fedora 13+ have a package called python3-pip which will install pip-3.x.

Repository keys

Most repositories don't have an accompanying key. Some of them have a key that you can download. It's not needed to run the software of the repo but it adds the repo in the trusted ones. These keys are the public keys of the repository. Each time you add another apt repository to /etc/apt/sources.list, you'll also have to give apt its key if you want apt to trust it. Once you have obtained the key, you can validate it by checking the key's fingerprint and then signing this public key with your private key. You can then add the key to apt's keyring with apt-key add <key>

gpg (GNU Privacy Guard) is the tool used in secure apt to sign files and check their signatures.

apt-key is a program that is used to manage a keyring (krikos for keys) of gpg keys for secure apt. The keyring is kept in the file /etc/apt/trusted.gpg

Software center

It's a tool with which you can install software using a gui. **Edit > Software Sources**: you can see the installed software. There are Ubuntu Software and Other Software. From Other software you can add a PPA to your system. After you close the Software Sources the Software Center will automatically update so that you can access the new packages from the PPA. You can see the installed PPAs in the **Get Software** list. After an installation you need to update in order to be able to access your new software. This is done automatically by Software Center tool. Once you install new software, updates will come to you through the Ubuntu **Update Manager.**

Packagename-dev

Apart from packages there are also some development packages package_name-dev that usually contain C header files and statically compiled versions of library files (.a extension). Sometimes they contain additional documentation and examples, or even helper applications.

A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that come with your compiler. You request to use a header file in your program by including it with the C preprocessing directive #include,

Shell Commands

Windows

Redirect: myCommand 2>myFile.txt redirects stderr to the file

Stream 0 is stdin

Stream 1 is stdout

Stream 2 is stderr

Stream3-9 are not used

apt-get: It is used to install or update packages from an existing repository.

apt-get install **–y** apache2-utils: the –y means –yes (yes to all install questions)

apt-get –y autoremove

echo: print to std_output. **echo "This is a file." > file1.txt** redirects the string in the file1.txt (creates it if needed). **>>** append to a file.

> Shell redirect. Same with –o or --output (for example echo –o my_file.txt "this is a file.")

curl: curl is a tool to transfer data from or to a server, using one of the supported protocols (DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMB, SMBS, SMTP, SMTPS, TELNET and TFTP). The command is designed to work without user interaction.

Basically you can use curl to download content from the internet. For example if you ran the curl command with the web address set to http://linux.about.com/cs/linux101/g/curl.htm then the linked page will be downloaded. You can redirect it to a file or have it outputted to the std output.

You can use curl to download physical webpages, images, documents and files. For instance to download the latest version of Ubuntu Linux you can simply run the following command:

curl -o ubuntu.iso http://releases.ubuntu.com/16.04.1/ubuntu-16.04.1-desktop-amd64.iso

service *service_name* restart: (eg. a pache2):

tail : prints the last 10 lines of each FILE to standard output.

rm: delete a file or dir, a hard link, The removal process unlinks a filename in a filesystem from data on the storage device, and marks that space as usable by future writes. So the data are still in the storage device but they are inaccessible. But this space can be written (probably if there is n more free space), **rm *** removes all from the working dir, **rm –r a_dir** removes the a_dir directory and all of its contents, **-rf** combines –r (recursively) and –f (force)

shred: completely, unrecoverably erase the data on the disk instead of just rm.

ln: creates links between files. They are hard links or soft links if the **–s** option is given.

cp file direction_folder: copy a file

clear: clears the previous commands of the terminal and its like you have opened a new one starting from the top.

pwd: show the folder that we are currently in

ls: list all that the current folder contains, **-l** use long list format, **--sort=word**, **-t** sort by modification time, **-a** show also hidden objects, **-h** print sizes in human readble format. **ls –l *filename*** prints the detailed stats of the file (access rights etc) **-la**

su: The su command is used to become another user during a login session. Invoked without a username, su defaults to becoming the superuser.

sudo su: run the command su with superuser (as user root) capabilities. sudo su changes the current user to root but environment settings (PATH) would remain same. IMO best practice is to use sudo every time as sudo will log your commands, while sudo su will not (once you are root).

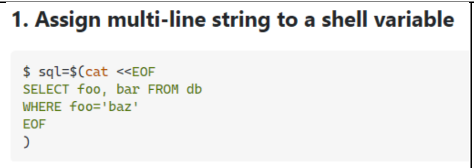
cd / : cd to the top level directory

cat: Display text files on screen. Copy text files. Combine text files. Create new text files

cat */etc/passwd* > */tmp/test.txt* In the above example, the output from cat command is written to /tmp/text.txt file instead of being displayed on the monitor screen. **$ cat > *foo.txt*** create a file **$ cat >> bar.txt** You can append the output to the same file using >> operator. The existing bar.txt file is preserved, and any new text is added to the end of the existing file called bar.txt. To save and exit press the CONTROL and d keys (CTRL+D).

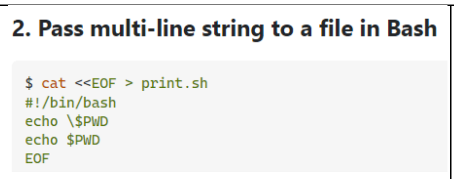
cat <<EOF

for multi line text. For example:



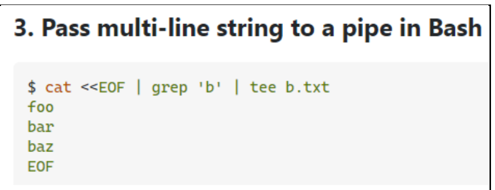
1. Assign multi-line string to a shell variable

```
$ sql=$(cat <<EOF
SELECT foo, bar FROM db
WHERE foo='baz'
EOF
)
```



2. Pass multi-line string to a file in Bash

```
$ cat <<EOF > print.sh
#!/bin/bash
echo \$PWD
echo $PWD
EOF
```



3. Pass multi-line string to a pipe in Bash

```
$ cat <<EOF | grep 'b' | tee b.txt
foo
bar
baz
EOF
```

The b.txt file contains bar and baz lines. The same output is printed to stdout.

chroot: is an operation that changes the apparent root directory for the current running process and their children. A program that is run in such a modified environment cannot access files and commands outside that environmental directory tree. This modified environment is called a “chroot jail”. A chroot environment can be used to create and host a separate virtualized copy of the software system. Usefull for testing (A test environment can be set up in the chroot for software that would otherwise be too risky to deploy on a production system), dependency control (Software can be developed, built and tested in a chroot populated only with its expected dependencies. This can prevent some kinds of linkage skew that can result from developers building projects with different sets of program libraries installed) etc.

chmod: change access permissions **–R** recursive; include objects in subdirectories, **-f** force, **-v** verbose; show objects processed. If a symbolic link is specified, the target object is affected.

Numerical Shorthand

7 rwx, 6 rw-, 5 r-x, 4 r--, 3 –wx, 2 –w-, 1 –x, 0 ---

Another way to use chmod is to provide the permissions you wish to give to the owner, group, and others as a three-digit number. The leftmost digit represents the permissions for the owner. The middle digit represents the permissions for the group members. The rightmost digit represents the permissions for the others.

grep: processes text line by line and prints any lines which match a specified pattern. Grep --color “my_phrase” my_file.html will highlight the occurrences.

lsb_release print linux distribution specific information **–a** (all info)

uptime tells you how long the system has been running. System load averages is the average number of processes that are either in a runnable or uninterruptable state.

df reports the amount of available disk space being used by file systems. **– h** human readable format

You can see all volumes of the machine, including mounted ones.

ifconfig

touch The touch command updates the access and modification times of each FILE to the current system time. If you specify a FILE that does not already exist, touch creates an empty file with that name

wget download files from the Internet. The command recreates the complete directory structure of the site downloaded on your computer's hard drive, and you can store the local copy as a backup or use it for testing purposes **–o mylogfile** Log all messages to logfile instead of standard error. **–O file** documents will be concatenated together and written to file. If file already exists, it will be overwritten. If the file is **-**, the documents will be written to standard output.

-c “command” execute command

- **./ script** runs the script as an executable file, launching a **new shell** to run it
- **source script** reads and executes commands from filename in the **current shell** environment. Same with **. script**

rsync sync one folder with another one (this will sync the folders at the time of the command. If there are changes later you need to run it again)

watch -n 1 'ps -e -o pid,uname,cmd,pmem,pcpu --sort=-pmem,-pcpu | head -15' make a command realtime –n 1 means run it every 1 sec. the command is inside quotes

ping -c3 -I eth0 www.google.com ping a url to see if a specific network adaptor (here the adaptor eth0) can communicate with the internet.

sudo chown -R $USER:$USER . Change the ownership of the the files of the current dir (.) owner would be the $USER defined

>adduser new_username

It creates a new linux user and group (creates the needed directories etc.)

>id username

Shows information about the user

>usermod -aG sudo username

Give him sudo privileges

>sudo chown -R username:groupname /home/username

change the owner of the given folder to be the given user and given group. This is applied to all contents of this folder recursively. chown means change owner.

>ls -la

List all contents of the current folder along with the owner and permission information for each content and show the hidden folders too (the ones starting with .)

Shell programs

Besides **bash** (which stands for Bourne Again SHell, an enhanced version of the original Unix shell program, sh, written by Steve Bourne), there are other shell programs that can be installed in a Linux system. These include: **ksh**, **tcsh** and **zsh.**

Bash Special Characters

http://tldp.org/LDP/abs/html/special-chars.html

< > redirection characters for piping stuff from, and into, files

> Redirects stdout to a file. Creates the file if not present, otherwise overwrites it.

>> Redirects stdout to a file . Creates the file if not present, otherwise appends to it.

< filename Accept input from a file.

| Pass the output of one command to another for further processing. Similar to ">", but more general in effect. connect a command's output with another's input. Useful for chaining commands, scripts, files, and programs together.

; Permits putting two or more commands on the same line

;; Terminator in a case option

. executes a script when it used from the command line. When it is used within a script, loads the file in the script, imports code into the script, appending to the script

In directory names .. means the parent dir, while . means the working dir

/ Separates the components of a filename (as in /home/bozo/projects/Makefile).

$ variable substitution

$$ The $$ variable holds the process ID of the script in which it appears

$? The $? variable holds the exit status of a command, a function, or of the script itself.

~ home directory [tilde]. This corresponds to the $HOME internal variable

~/ is the current user's home directory

~+ current working directory. This corresponds to the $PWD internal variable.

Ctl-D logout from shell

Ctl-A Moves cursor to beginning of line of text (on the command-line).

Ctl-E Moves cursor to end of line of text (on the command-line).

Ctl-L (clear the terminal screen). In a terminal, this has the same effect as the clear command.

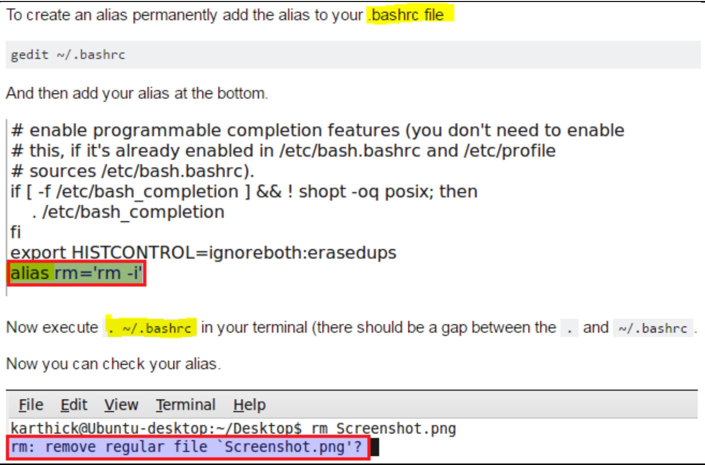
Ctl-O Issues a newline (on the command-line).

Arguments passed to the script from the command line: $0, $1, $2, $3 . . . $0 is the name of the script itself, $1 is the first argument, $2 the second,

./scriptname 1 2 3 4 5 6 7 8 9 10 (call a script with some arguments)

create aliases

For example for opening sublime subl my_file.txt



To create an alias permanently add the alias to your .bashrc file

```
gedit ~/.bashrc
```

And then add your alias at the bottom.

```
# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if [ -f /etc/bash_completion ] && ! shopt -oq posix; then
    . /etc/bash_completion
fi
export HISTCONTROL=ignoreboth:erasedups
alias rm='rm -i'
```

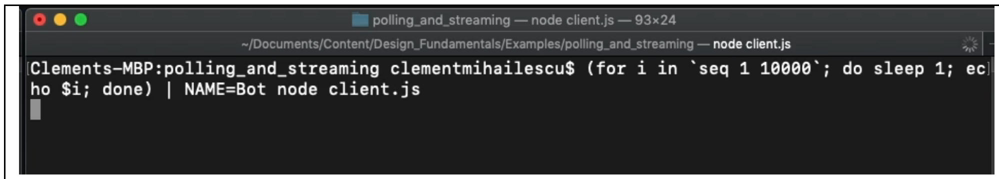
Now execute `. ~/.bashrc` in your terminal (there should be a gap between the `.` and `~/.bashrc`.

Now you can check your alias.

```
karthick@Ubuntu-desktop:~/Desktop$ rm Screenshot.png
rm: remove regular file `Screenshot.png'?
```

There are 2 methods

1. Define your aliases in ~/.bashrc file
2. Create a separate aliases file "reference" it in your bashrc file.



polling_and_streaming — node client.js — 93×24
~/Documents/Content/Design_Fundamentals/Examples/polling_and_streaming — node client.js

```
Clements-MBP:polling_and_streaming clementmihailescu$ (for i in `seq 1 10000`; do sleep 1; ec
ho $i; done) | NAME=Bot node client.js
```

Create a loop, sleep for one sec, and pipe the i in the next command which is a javascript file we wrote. Here this javascript code expects an input from the terminal. This input is the I of the loop. The NAME is parameter of the script. This way we execute this script 1000 times.

Misc

Linux and Unix

Unix is an operating system initially developed in the early 70s. Linux is based on the philosophy behind the UNIX but it doesn't use UNIX code. It is UNIX like

Users and groups

Groups are a collection of users. Assigning users to groups makes it easier to manage permissions. Whenever a user is created, by default, they are added to a new group with the same name as the username. This is called the primary group of the user.

There are human users and software users. Each set has its own id range. So I guess that when you have a web server installed there is a user assigned to it. If a connected client wants to write to the disk then it would only be possible to do it if the webserver user has the necessary rights.

Share folder with windows host from virtual box

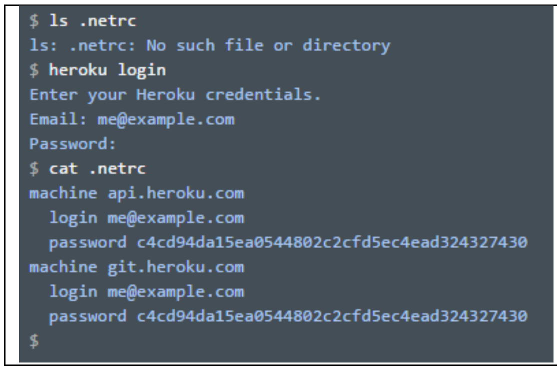
1. go to devices > shared folders and add a new shared folder
2. sudo apt-get install virtualbox-guest-utils
3. Run the command sudo mount -t vboxsf host-folder-name /path/to/your/ubuntu/folder/

An executable within /usr/bin/ is also available in the terminal as command.

~/.netrc

It's a file in which credentials for services are stored and you can use these services without defining the token every time you call them. The file also exists in windows. The netrc format is well-established and well-supported by various network tools on unix.

For example:



```
$ ls .netrc
ls: .netrc: No such file or directory
$ heroku login
Enter your Heroku credentials.
Email: me@example.com
Password:
$ cat .netrc
machine api.heroku.com
  login me@example.com
  password c4cd94da15ea0544802c2cfd5ec4ead324327430
machine git.heroku.com
  login me@example.com
  password c4cd94da15ea0544802c2cfd5ec4ead324327430
$
```

With Heroku credentials stored in this file, other tools such as curl can access the Heroku API with little or no extra work. When using the default HTTP transport, Git uses cURL, and cURL will use the API key stored in .netrc to authenticate with the Heroku HTTP Git service.

Windows

Command line

- type (Cat for windows)

\>type a-file.js it shows the file in the terminal

- Cd to a different drive

g:\>cd **/d** C:\Projects_D\virtualEvns\zakanda_env\zakanda_src\

Misc

Data Serialization

data "serialization" is taking data from the code (Python) and converting it into something that can be sent through the network. For example, converting an object containing data from a database into a JSON object. Converting datetime objects into strings, etc.

Experienced coders tips

1. Prevent unfinished work
2. Enforce coding standards
3. Document chosen patterns
4. Review new patterns early
5. Never expose refactoring (don’t make it an item where a manager can take it out as not priority for example)
6. Assume unexpected changes (give your self extra time when setting deadlines)

Conferences

Qcon

Pycon

djangoCon