

CAB432 Study Guide | 2022 Semester 2

Associate Professor Jim Hogan | Notes for CAB432 at the Queensland University of Technology

Table of Contents

- [CAB432: Cloud Computing](#)
 - [Week 1](#): Introduction
 - [Week 2](#): Docker
 - [Week 3](#): Cloud Applications, REST, and Node
 - [Week 4](#): Node and Express
 - [Week 5](#):
 - [Week 6](#):
 - [Week 7](#):
 - [Week 8](#):
 - [Week 9](#):
 - [Week 10](#):
 - [Week 11](#):
 - [Week 12](#):
 - [Week 13](#):
-

CAB432: Cloud Computing

Cloud Computing is among the most important developments in the IT industry in recent years, and one which has received enormous attention. Cloud is a natural progression from earlier trends in service and infrastructure outsourcing and virtualisation, but is distinguished by its elasticity and scale: service and infrastructure provisioning may change rapidly in response to variations in demand, allowing clients to cater for unexpected spikes in load without tying up capital in expensive and potentially underutilised assets. Cloud services and technologies are becoming increasingly diverse and sophisticated, moving rapidly from the original 'bare metal' offerings and providing a rich set of options and APIs. This unit provides a technically oriented introduction to Cloud Computing,

giving you experience in developing modern cloud applications and deploying them to the public clouds of the major vendors.

Week 1: Introduction

What is Cloud Computing?

Cloud computing is, at its core, IT infrastructures and services that can be turned on when needed and are payed only when being used. These services can be scaled as needed and automatically adjust themselves based on the current load. Modern cloud computing operates as a global scale providing access to everyone, everywhere at anytime.

So what business case would there be for cloud computing?

- 1. Only pay for what you use
- 2. Don't have to spend the money, time and resources to build server rooms and run them
- 3. Capacity is managed through elastic provisioning
- 4. Most major vendors provide unlimited capacity based on how much you pay

Public, Private, and Hybrid Clouds

Public Cloud	Private Cloud	Hybrid Cloud
Services offered by the major vendors	Services hosted and managed by large companies or government	Services based on a mix of public and private clouds
Available globally on a commercial basis	Same elastic service model but on a smaller scale	Often used to manage regulatory requirements and client concerns over location of sensitive data sets
"Limited only by the size of your credit card"	Usually more limited service offerings	Government services, Major corporate's

Cloud Pre-requisites

Virtualisation	Elasticity	Scale
----------------	------------	-------

Virtualisation	Elasticity	Scale
Costly to maintain lots of small machines	Measure load and automatically scale as needed	Cloud data centres are huge
Better to maintain lots of virtual machines (VM) in the one place	Scale out when people are waiting. Scale in when we have too many machines	Major vendors have nodes around the world and their own undersea cables

Virtualisation

Virtualisation Layers
Application(s)
Guest OS (many)
Hypervisor
Host Operating System - Linux, Windows, UNIX...
Physical Hardware

Types of Cloud Services

Infrastructure as a Service (IaaS)	Platform as a Service (PaaS)	Software as a Service (SaaS)
Virtual Machine + OS	Not just the OS	Application hosting in the cloud
Pick your size	Pre-configured software stack on each VM	Subscription-based
Storage - entity storage, SQL, NoSQL, archive	Managed provisioning and scaling	All hosting and management is done for you at a fee
Pick your DB		Salesforce is usually used

Major Providers

Microsoft Azure

- Provide a mix of XaaS services but mainly provide IaaS and PaaS

- Their focus areas consist of big data and machine learning
- Azure SQL and SQL Server are two instances of services
- Cognitive Services and Azure ML are key offerings

Google Cloud Platform

- Provide a mix of IaaS and PaaS
- Their focus area is IaaS at a large scale
- PaaS remains through Servlet based AppEngine
- Have application areas in search tech, data and machine learning

Amazon Web Services

- Provide a mix of XaaS services
- Huge range of elastic application services
- Have made major improvement in machine learning services in recent years
- Developing a focus on IoT

Week 2: Docker

Docker

Docker is a popular container as a service application that has command-line tools to run containers, create images and do much more. Docker can also be run inside of a Linux OS that is in itself being virtualised by a VM.

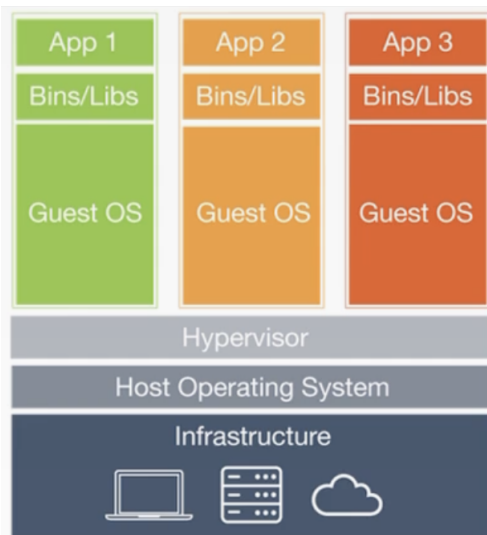
There are many reasons to use docker such as:

- Much more efficient use of "bare-metal" hardware resources
- Many more containers than virtual machines
- More efficient use of CPU cycles
- There is no need to maintain an entire operating system stack
- Easier management and orchestration
- Rebooting containers is much faster and easier than rebooting a VM

There are also however a few cons:

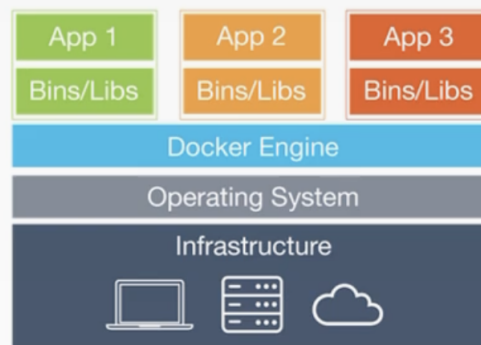
- Virtual machines allow for better isolation
- Containers may have port collisions

- Within containers there is no access OS-level daemons
- There is an assumption that guest software is compatible with Linux



Virtual Machines

Each virtual machines includes the application, the necessary binaries and libraries and an entire guest operating system - all of which may be tens of GBs in size.



Containers

Containers include the application and all of its dependencies, but share the kernel with other containers. They run as an isolated process in userspace on the host operating system. They're also not tied to any specific infrastructure – Docker containers run on any computer, on any infrastructure and in any cloud.

Containers

A container is an instance of an application(s), whether running, stopped or finished. These containers are installed on top of some "base image" with each container being isolated from other containers and from the guest OS (although they can interact with these throughout well-defined interfaces).

Images

An image is a "pre-canned" software stack loaded from some repository, typically dockerhub.com. An image acts as a blueprint for containers and follows a cookie-cutter model where as many identical images can be deployed as we want.

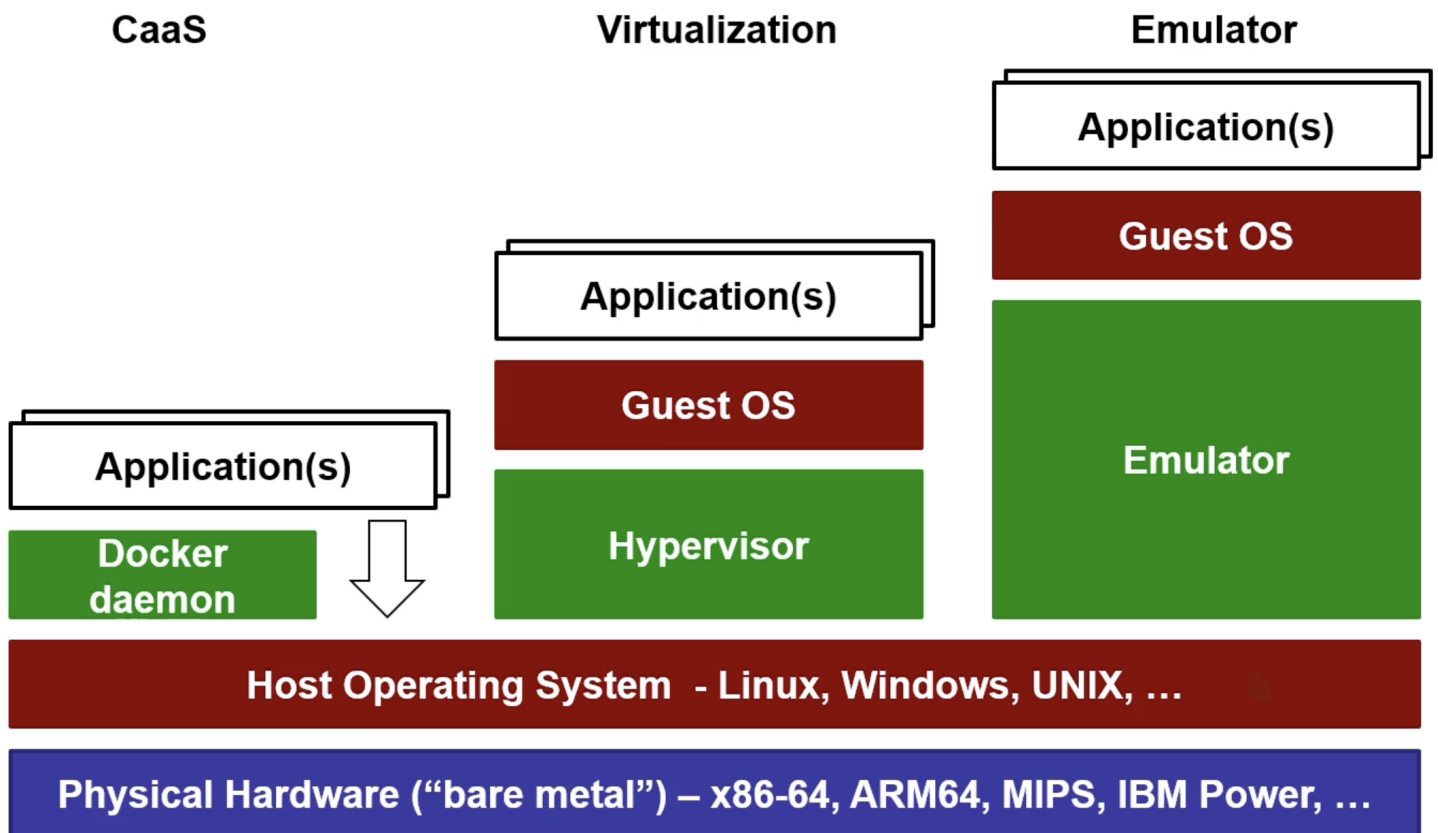
Dockerfile

A docker file is a script used to build a docker image. These typically start with a base OS image and execute a sequence of commands consisting of installation and configuration tasks for the needed stack. A docker compose is a file that allows services and configurations for multi-container applications to be specified.

Application Virtualisation Levels

- Typically layered on top of an operating system. This is usually Linux for a number of reasons.
- Things like hardware, firmware, OS, middleware and standard tools are not virtualised.
- Precise, user-defined set of software and configuration:
 - There is no need to clone a virtual machine leading to a faster and lighter experience
 - Scripts can instantiate a cloned container within a matter of seconds
 - Pre-canned scripts and images are able to be shared with others

CaaS vs Virtualisation vs Emulators



Why use Virtualisation

There are many reasons to use virtualisation such as:

- **Cost:** It is much cheaper to run virtual machines because it allows for fewer physical machines for the same amount of computation.
- **Normalisation:** With normalisation we get:
 - Cookie cutter resources: All the resources are the same
 - To define an instance and machine image
 - Easier management and maintenance of units

- **Software Lifecycle:** The ability to deploy many identical machine images and update them all the same
- **Isolation:** Using isolation we can have many users on the same hardware with hard separation of virtualised resources
- **Instance Lifecycle:** Each instance we create can be:
 - Scaled up: Quickly replace the current resources with more powerful ones
 - Scale out: Quickly start up additional, identical instances
 - Continuity: Quick fail-over to another resource

IaaS Challenges

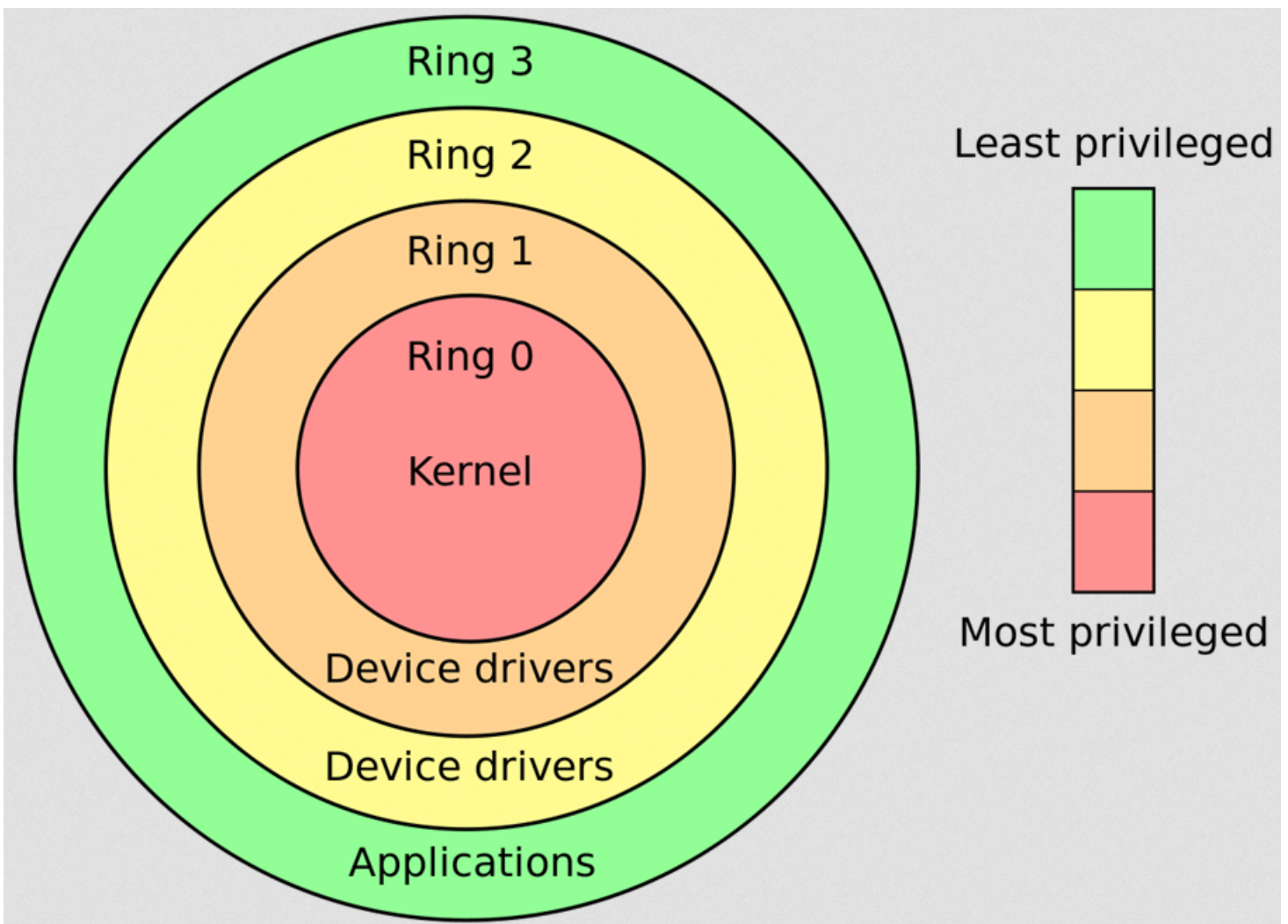
Infrastructure as a service has a few challenges that come along with it:

- A heavyweight and complex hypervisor
- Performance and resource consumption
- Virtualisation of an entire guest OS. This means virtualising things like the kernel, device drivers, the filesystem, the network protocol stack, middleware, applications and many more.

Hardware Virtualisation Terminology

- **Host OS:** The host operating system is the non-virtualised hardware resources. This is the "bare metal" running the hypervisor/VMM (typically your physical device).
- **Guest OS:** The guest operating system is the operating system running inside of virtualised hardware.
- **Hypervisor:** Hypervisor is the software that runs the virtual machine itself. There are 2 types:
 - Type 1: This runs directly on the hardware (like an operating system)
 - Type 2: This runs inside of an operating system

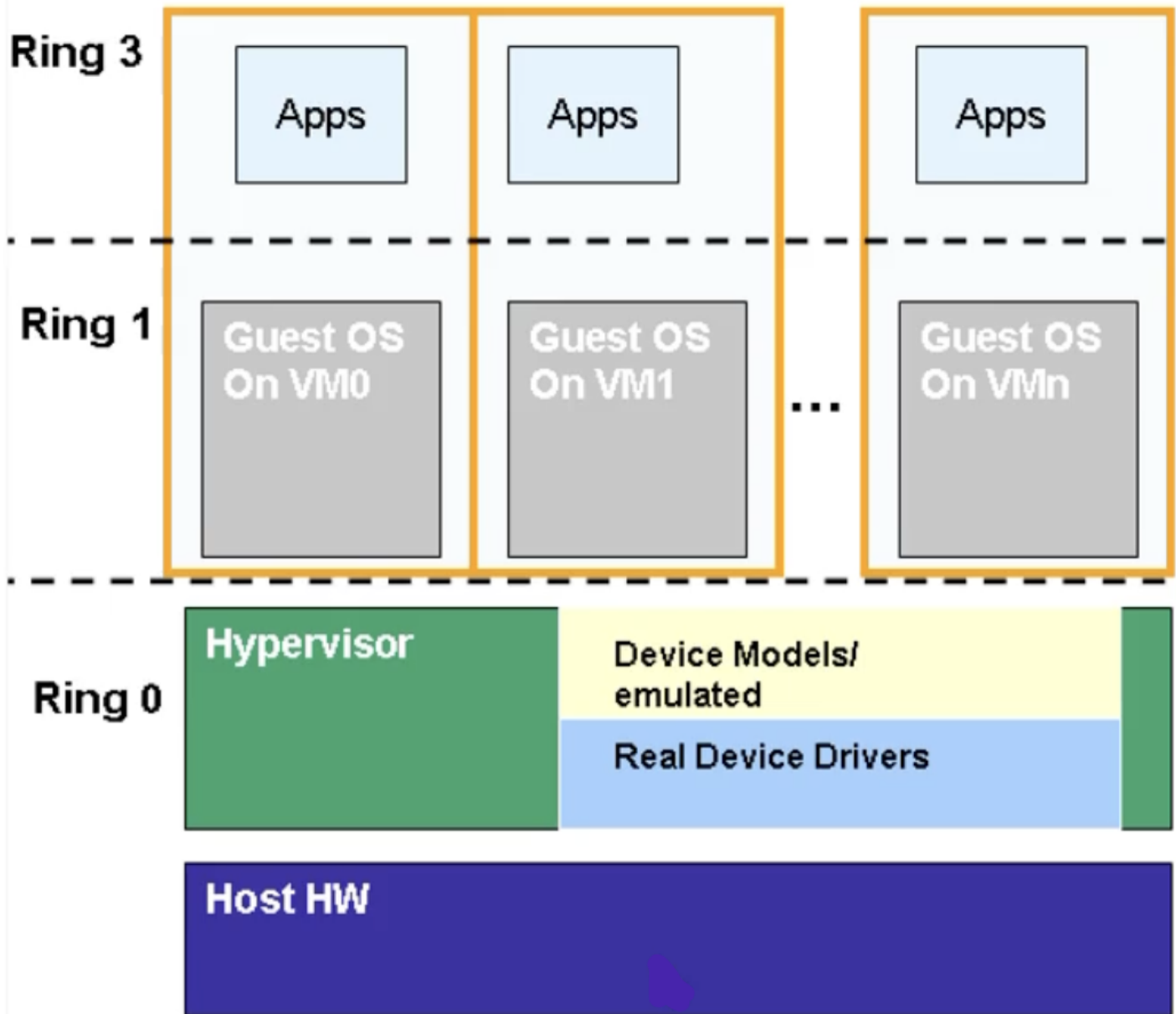
CPU Protection Layers



- **Kernal mode (ring 0):** Privileged instructions such as interrupts or memory management. These instructions are generally reserved for the OS kernel with applications not normally being able to see them.
- **User mode (ring 3):** Non-privileged or "safe" instructions. Application software will ask the kernel to perform privileged instructions on it's behalf. In this mode the Guest OS is given higher privileges via the hypervisor.

Hardware Assisted Virtualisation Basics

Hypervisor Architecture



Hardware Emulation

Hardware emulators are full or partial software implementations of the hardware which use binary translation to convert some instruction to another e.g. ARM to x86, PowerPC to x86. One of the major pros to this is that we can emulate other instruction sets however, this comes at a huge hit to the speed as these binary translations are relatively slow depending on the complexity of the VMM.

Para-virtualisation

Para-virtualisation is a way of modifying the guest OS to make virtualisation easier. VMM does this by implementing an API for the guest OS or by simulating privileged instructions. The obvious pro to this method is that we can avoid the costly binary translations. The downsides to this is that we must modify the guest OS and we can no longer emulate incompatible instructions sets.

Hardware Assisted Virtualisation

Hardware assisted virtualisation is

There are many pros to this such as:

- The guest OS can continue in a privileged kernel mode
- Most ring 0 instructions do not "VMExit" to "ring -1"
- This method is much faster than full hardware virtualisation
- This method is sometimes faster than para-virtualisation and at other times not. It all depends on the circumstances
- The guest OS does not need to be modified

One of the main cons to this method is that it requires hardware support although this is quite common now days.

Week 3: Cloud Applications, REST, and Node

Moving to the Cloud

When moving to the cloud it's important to note that virtualisation means latency. The requests and responses that need to be sent and received are no longer free and instant. It's because of this that in cloud computing we must choose our method of data persistence carefully whether that be slow, long-term storage, fast cached storage or others.

Due to each message having an associated "flagfall" and cost associated we generally try to bundle them together when possible.

Another thing to note is that the interactions through the cloud are stateless and asynchronous. We don't always guarantee an immediate response but we do guarantee a response, clients send messages and then await the returning message from the service.

State and Statelessness

State is the current state of some object such as the values of it's fields. In the context of cloud computing what we mean when we refer to an objects state is how we can maintain that specific objects state between different server requests.

Statelessness can be a confusing concept due to the reason that there may be some state there, it's just that we don't rely on that state being preserved. An object is stateless initially but the state of that object can be recoverable via some other means.

REST

REpresentational State Transfer, or REST for short, is a set of principles that define how web standards, such as HTTP and URIs, are supposed to be used. Using the principles of REST we must:

- Give everything an Id:
 - <https://www.flickr.com/photos/sdufva>
 - <https://www.flickr.com/photos/tags/golden-retrievers>
 - <https://www.example.com/orders/2007/11>
 - <https://www.example.com/products?colour=green>
- Link things together:
 - Global URIs allow most unbounded linkage of resources and attributes
 - URIs can capture state, allowing state transitions to be mapped to link transitions
- Use standard methods:
 - HTTP
 - GET, POST, PUT, DELETE
- Have resources with multiple representations:
 - JSON, XML, Plain Text
- Communicate statelessly:
 - Keep the state in the URI or in the client

GET

A GET request is a HTTP request method used to retrieve data from a data source. An example of a GET request to developer.mozilla.org is:

```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: en
```

After the request is made a response will be transmitted from the host destination, here is an example of a 200 Ok response:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 55743
[More here...]
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>A simple webpage</title>
  </head>
  <body>
    <h1>Welcome</h1>
    <p>Hello, world!</p>
  </body>
</html>
```

POST

A POST request is a HTTP method used to send some data to a data source. An example of a POST request is:

```
POST /users HTTP/1.1
Accept: application/json
Authorization: Basic XXXXXXXXXXXXXXXXXXXX
Content-Type: application/json
Host: localhost
Connection: keep-alive
Content-Length: 984

{
  "name": "Franklin Schafer",
  "purchase": {
    "id": "Bicycle computer",
    [More here...]
  }
  [More here...]
}
```

Much like the GET request a response will be transmitted once the server receives the request.

Cloud Requests

All these requests first come into the Elastic Load Balancer with those requests then being sent on into the Scaling Pool machines.

Elements of the HTTP Session

A HTTP session consists of three phases:

1. The client establishes a TCP connection
2. The client sends a request and then waits for an answer
3. The server processes the requests and sends back an answer containing a status code and the appropriate data

Each request follows the general structure of:

- Method + Parameters: Path to the resource without the domain name
- The HTTP protocol version being used
- Any subsequent headers e.g. Content-Type
- An empty line above the data block

Each request follows the general structure of:

- Status line, HTTP version used + the status code and information
- Additional HTTP headers such as the timestamp, type, data size...
- The data block (with a blank line above it)

Common Response Codes:

Code	Name
100	Continue
200	OK
201	Created
202	Accepted
301	Moved Permanently
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
500	Internal Server Error
501	Not Implemented

Code	Name
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout

Node.js

Node.js is essentially server side JavaScript which was introduced to support rapid server connections and processing. Node.js has many advantages such as:

- It being fast due to it being compiled and being very optimised
- Having support for asynchronous javascript connection processing
- Not having to allocate a new thread for every connection
- It's open-source
- Having very strong support from major vendors such as Google, Microsoft and more

An example of a simple Node.js server:

```
const http = require("http");

const hostname = "127.0.0.1";
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader("Content-Type", "text/plain");
  res.end("Hello World\n");
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Simple Storage Service

Simple storage service can be thought of as lots of very large objects each containing a collection of <key, Object> pairs. This is based on a non-relational, addressable bucket model and was the original AWS service layer.

It's important to remember that data persistence is up to the vendor you're using and that the data utility is up to you.

Week 4: Node and Express

NPM

NPM is an online repository of open-source Node.js projects. It is also a command-line utility that installs, removes, updates and manages installed npm packages.

To install a package locally we type:

```
npm install <package-name>
```

and to install the package globally (not recommended unless you know the utility is globally useful)

```
npm install <package-name> -g
```

These packages will be installed into your projects `node_modules` folder with all package dependencies stored in `package.json` and `package-lock.json`.

Express

Express is an application framework designed for node. To use Express we must first install it using `npm install express -g`. Let's take a look at a very simple hello world express program:

```
const express = require("express");
const app = express();

const hostname = "127.0.0.1";
const port = 3000;

app.get("/", (req, res) => {
  res.send("Hello World!");
});

server.listen(port, () => {
  console.log(`Express app listening at http://${hostname}:${port}/`);
});
```

We can handle routing using the `Router` class.

```
const express = require("express");
const router = express.Router();

router.get("/", (req, res) => {
  res.send("Home page");
});

router.get("/about", (req, res) => {
  res.send("About page");
});

module.exports = router;
```

Express Generator

Express generator will create and setup a lot of the necessary boilerplate for an Express application for us. To use it we must first install it using `npm install express-generator -g`. Once installed we can create an Express application using `express helloworld` in the terminal. The following files will be created inside the project directory:

```
.
├── app.js
├── bin
│   └── www
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.jade
    ├── index.jade
    └── layout.jade
```

Fetch

We can simulate GET and POST requests in Javascript using the `fetch()` function. The fetch request works by passing in the url for the data and an optional parameter containing headed data for the request being sent (i.e. authorisation tokens). Here is a simple GET request using the `fetch()` method:


```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then((response) => response.json())
  .then((json) => console.log(JSON.stringify(json)));
```

Here is a simple POST request using the `fetch()` method:

```
const headerContent = {
  method: "POST",
  body: JSON.stringify({
    title: "foo",
    body: "bar",
    userId: 1,
  }),
  headers: {
    "Content-type": "application/json; charset=UTF-8",
  },
};

fetch("https://jsonplaceholder.typicode.com/posts", headerContent)
  .then((response) => response.json())
  .then((json) => console.log(json));
```

Promises

A promise is a special Javascript object that can hold the future value of an asynchronous operation.

A promise can be in one of three states at any given time:

- Pending: This is the initiate state, it is neither fulfilled or rejected
- Fulfilled: This state is true when the operation has been completed and the promise now contains the retrieved data
- Rejected: This state is true when the operation has failed

Promises work by chaining functions from each other using the keyword `.then()`. We can also use `.catch()` at the end of our chain to throwback any errors we retrieve. For example, let's say we want to print to the console the returned data if, and only if, the fetch request returns a valid response, without having to freeze our application waiting for the API callback. We can use promises for this as such:

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then((response) => {
    if (response.ok) {
      return response.json();
    }

    throw new Error("Failed network response");
  })
  .then((result) => {
    console.log(result);
  })
  .catch((error) => console.log(`Problem with error message: ${error}`));
```

We can also use Async/Await to accomplish the same thing:

```
{
  () => {
    try {
      const response = await fetch('https://jsonplaceholder.typicode.com/posts/1')

      if (!response.ok) {
        throw new Error('Failed network response');
      }

      console.log(await response.json());
    } catch (error) {
      console.log(`Problem with error message: ${error}`);
    }
  }
}
```