

Distributed Systems: Introduction (CO3404)



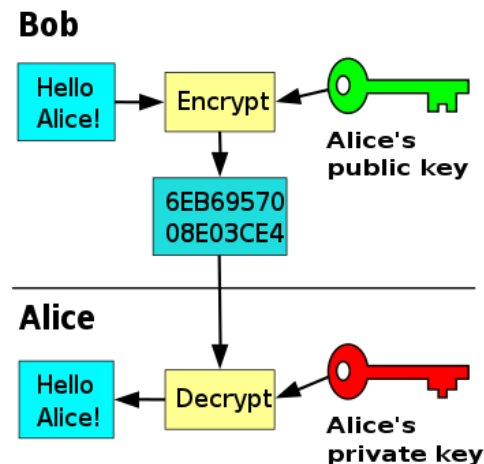
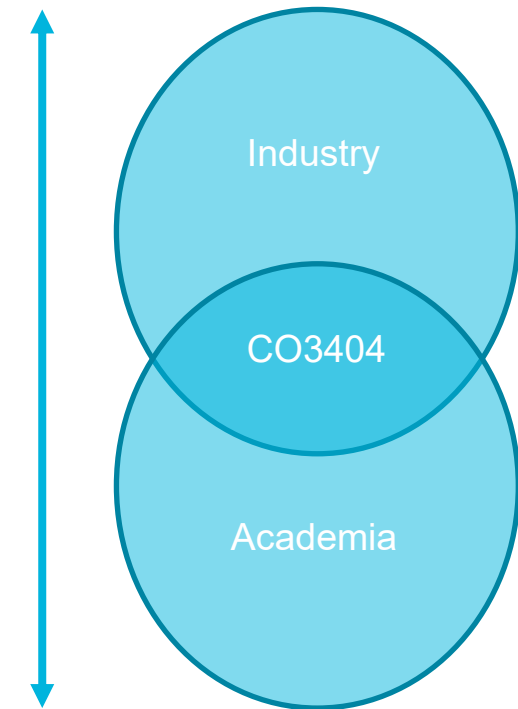
Dr Tony Nicol BSc (hons), MSc, PhD, MIET, CEng, FHEA, FBCS, CITP

DWP: Head of Engineering: tony.nicol@dpw.gov.uk

UCLAN: Senior Lecturer in Computer Science: ajnicol@lancashire.ac.uk

My background

- ▶ Working in Industry and academia for 50 years. A few examples:
- ▶ Royal Marines Commando
- ▶ Marine Research: sonar system for nuclear Subs – Electronics / Software Engineer
- ▶ Naval Systems: sonar system for Trident class sub – Electronics / Software Engineer
- ▶ Crypto: Crypto systems. Hardware, Software, Firmware
- ▶ UCLAN: Electronics Engineering, Computing, Software Engineering Course Leader
- ▶ University of Liverpool: taught MSc in Computer Security & Cyber Forensics
- ▶ DWP: Solution Architect, Technical Architect, Data Architect, Deputy Director
- ▶ Back at UCLAN



The value of this module

- ▶ Enable development of current industry best practice skills and knowledge
- ▶ Full blown distributed systems are mainly used in large enterprises – i.e. big and complex
- ▶ Large organisations (enterprise) include Gov departments, large retailers such as Amazon, Asda, Banks
- ▶ Most large organisations are going through a technology and data transformation – you need to be part of that, and this module will ensure you are, as that's where the well-paid jobs are
- ▶ This module aims to give you a taste of the real world of IT so you stand out at interview and can be part of the next generation to develop enterprise systems with new technological approaches
- ▶ Distributed systems is a huge topic, so I'll concentrate on the current & future and not dwell on the past

Job prospects

Microservices Jobs in England

[Insights](#)[About](#)[Find Jobs](#)

1 to 25 of 755 Microservices Jobs in England

Java Software Engineer

England, United Kingdom

VASS UK&I

a team. Excellent communication and collaboration skills. Preferred Qualifications: Experience with cloud platforms (e.g., AWS, Azure) and containerization (e.g., Docker, Kubernetes). Knowledge of **microservices** architecture and RESTful API development. Familiarity with CI/CD pipelines and DevOps practices. Experience with test-driven development (TDD) and automated testing frameworks. VASS [more »](#)

Posted: 4 days ago

Senior Software Development Engineer

Greater London, England, United Kingdom

PCCW GLOBAL Limited

object-oriented design, SOLID principles, distributed system design and software design patterns Experience in developing a multi-tier architecture system Previous exposure in SOA, **Microservices** or using API Management tools is an advantage Experience in using container technologies (e.g. Kubernetes, Docker, etc.) is an advantage Experience in using Cloud platform [more »](#)

Posted: Yesterday

Salary Guide

Microservices
England

10th Percentile
£43,750

25th Percentile
£52,500

Median
£65,000

75th Percentile
£87,500

90th Percentile
£110,000

[More Microservices
insights »](#)

Related

RESTful [314](#)

Spring Boot [127](#)

Node.js [388](#)

NoSQL [231](#)

Java [1,069](#)

React [899](#)

REST [335](#)

Spring [140](#)

OAuth [92](#)

AWS [2,159](#)

JavaScript [1,397](#)

AngularJS [528](#)

PostgreSQL [445](#)

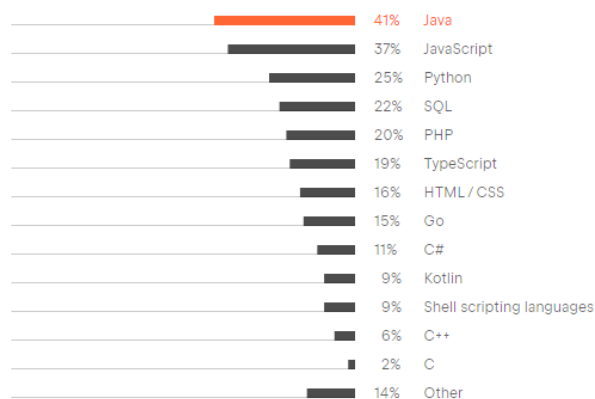
Serverless [394](#)

Kafka [269](#)

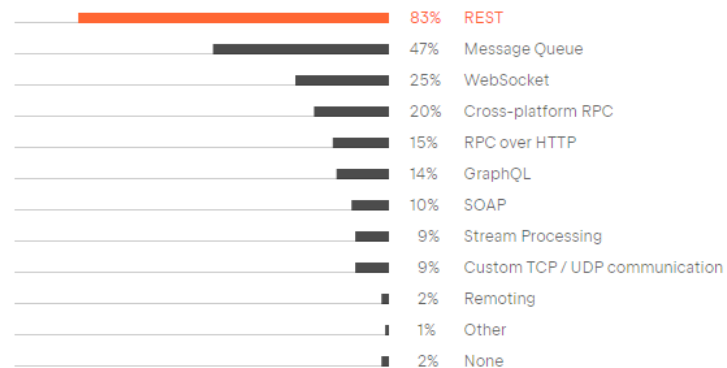
Relevant snippets

- ▶ There are other surveys of course ...

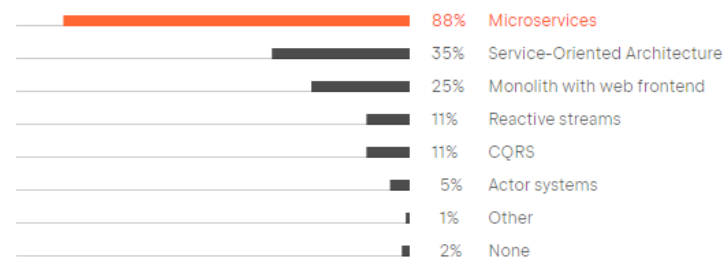
Primary languages among microservices developers



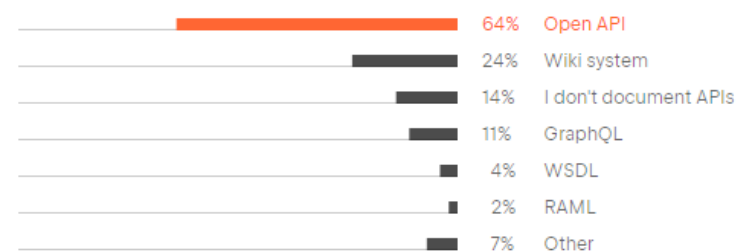
How do the distributed parts of your application communicate?



What approaches do you use in your system design?



How do you declare and document your APIs?



Module aims summary

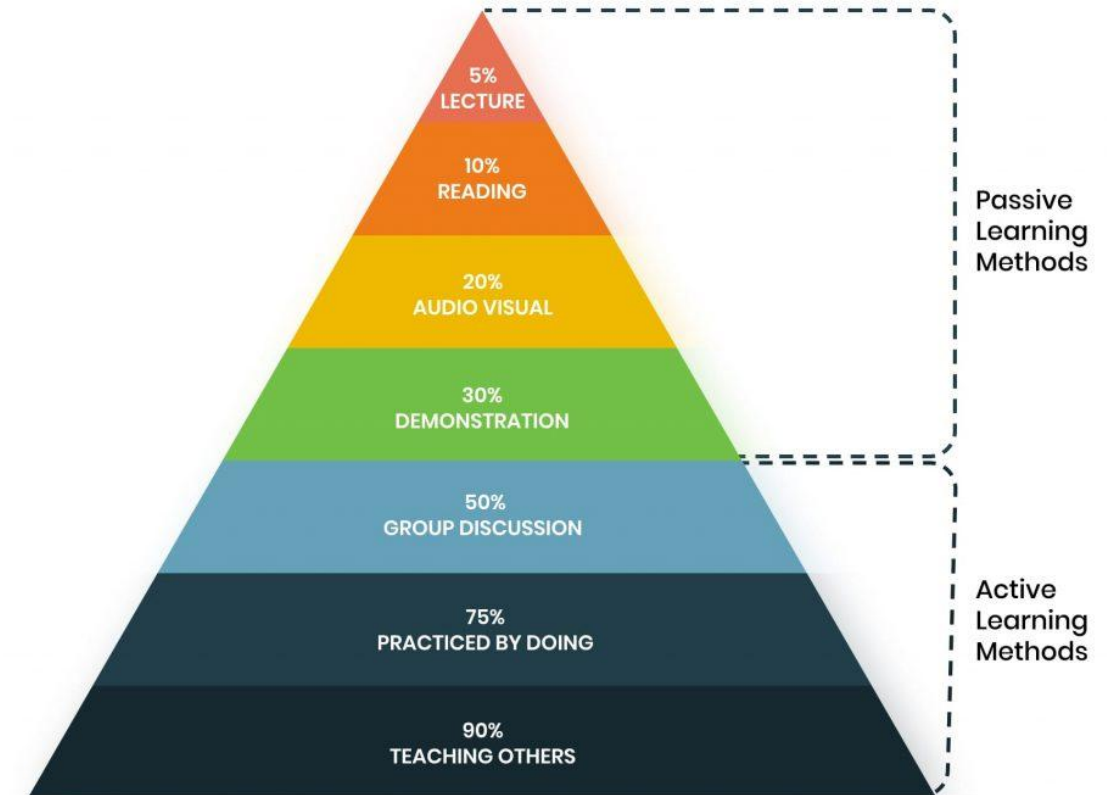
- ▶ 1. Create a distributed system application
 - ▶ Week by week I'll help you build on previous weeks leading to two assignments to create a distributed system
 - ▶ This will be built up practically with practical exercises where each week builds on the previous
- ▶ 2. Key design concepts such as scalability, security, heterogeneity, concurrency and containerisation.
- ▶ 3. Analyse technologies and patterns
- ▶ 4. How modern distributed systems support business
- ▶ 5. The need to deliver business value rather than technical perfection
- ▶ These 5 areas are a rough outline and don't carry equal weight. We will spend a lot more time on the technology, but you do need to understand how it relates to business needs and constraints
- ▶ Assessment is in the form of a major assignment and an exam weighted at 50% and 50%
- ▶ My approach is: lecture 1h, demo/tutorial/Q&A 1h (2h alternate weeks) - drop-in labs
- ▶ May be lecture and demos in both depending on the relevance and timing

It is essential to practice!

- ▶ **Lectures** - are valuable but can't rely on them alone
- ▶ **Reading** - you need to re-read the slides as they are comprehensive. Research areas needing detail
- ▶ **Audio visual** - make sure you watch the videos I create. Don't think a 1 hour video takes one hour to watch - you will need to pause and research
- ▶ **Demonstrations** - will be provided most weeks illustrating practical implementation of concepts. You should also do the demos to move into active learning
- ▶ On the active learning side, this is learning by doing
- ▶ **Group discussion** - collaborative quizzes
- ▶ **Practice by doing** - you will have lab exercises - problem solving and implementing solutions will improve retention and understanding. This is an **ESSENTIAL** component of learning
- ▶ You will be given demonstration code to work with to provide a learning framework building the skills needed mainly for the assessed major assignment. Run study, understand and experiment with them
- ▶ **Teaching others** - interactive sessions. If answering questions in class is unbearable, tell me privately



The Learning Pyramid



Feedback from last year before we get started

▶ Positives

- ▶ Some students did extremely well with grades in the 80s and two at 100% for coursework
 - ▶ "I got a job at the BBC directly because of this module"
 - ▶ "I didn't like the look of this module but now understand its importance in industry. It's now my favorite module"
 - ▶ "After discussing my microservices assignment I was offered a job at first interview"
 - ▶ "Having demos to work with to get an understanding of key concepts helped enormously with my assignment"
 - ▶ "This module directly helped me with my project. I changed the architecture to microservices"
 - ▶ You get the idea ...
-
- ▶ The knowledge I gained from you during the Distributed Systems module was absolutely invaluable for my work in this team. My understanding of APIs, how they work, and how they run in the cloud was extremely helpful and it meant that I was able to start working on tickets and pushing code changes to production in my first week. Thank you again for making your module so insightful, engaging, and up to date with the industry.

Challenges

- ▶ A substantial number of students failed. My observations and feedback from students:
 - ▶ Missing classes. Attendance is monitored. If you get behind, tell me and we can work out how to catch-up
 - ▶ Underestimated the module. "I thought it was web development so expected to breeze through. That was a big mistake"
 - ▶ Poor planning. Spent too much time on other modules or project - need to distribute time effectively
 - ▶ Achievement coaches are available to talk to if you are struggling with planning, motivation, confidence building etc. ecanuti2@uclan.ac.uk
 - ▶ Listening but not hearing. "I'll watch the videos when I need to": procrastination is a very flawed strategy
 - ▶ Note the learning pyramid - you need continuous learning and practice to consolidate - binge learning doesn't work on challenging technical courses
 - ▶ Missing demos - these build the framework for the assignments. I build up week by week, you must practice then consolidate over the whole year
 - ▶ Not undertaking personal study - this is essential each week if you are to do well in the assignments - 152 hours is minimum
 - ▶ Not submitting assignment - yes - it's true. Several didn't. If you are struggling, you must ask me for help - don't be an ostrich
 - ▶ Student mentoring. Go to a session to discuss with student mentors. It looks good on your cv if you become a student mentor
 - ▶ Late submission. MCs are there to help but they can put you on the back foot so use sparingly
 - ▶ I don't need to know this stuff - I'm a C++ desktop developer. Good luck with that then. A degree won't get you a job - it may get you an interview
- ▶ Recent feedback
 - ▶ Don't focus on the web development and front-end JavaScript as core as it creates a false simplified impression of the module
 - ▶ Show an assignment up-front to make it clear how the different topics build up to it
 - ▶ Got too far behind for various reasons and didn't speak up to ask for help soon enough. (I will help if asked)
 - ▶ As lectures and demos recorded, I convinced myself I'd watch them when needed so focused on modules with less resources. That was a mistake

Assumptions

- ▶ You understand basic networking from Y1
- ▶ You understand basic database concepts such as normalisation, relationships - from Y1
- ▶ You can program - from Y1 and Y2
- ▶ You understand how to test code - from Y1 and Y2
- ▶ If this is not the case, e.g. you haven't studied the standard course from Y1 or have completely forgotten this stuff, then you need to tell me, and I'll cover in the tutorial or individually if necessary
- ▶ The HTLM and CSS we do is very basic so you should be able to pick it up in your own time - but I'm happy to go through these things and any other topic in the tutorial or drop-in labs
- ▶ You must tell me if you need help before it's too late. I'm more than happy to help you get over mental blocks or clarify any of the concepts, but you must ask as I won't know
- ▶ I'll provide the lectures and tutorials / demos / Q&A online but this is only so you can join in the class to better see the code. If you have a valid reason not to be physically in the class, I need to be told ahead of time and I may allow you access from outside the class for exceptional circumstances

Responsibilities: Students

Timeliness

- Arrive in time to be prepared for the lecture/workshop
- Manage your time for study effectively

Communication

- Talk to us for support, to raise issues and to seek advice/guidance
- To monitor and use university communication mechanisms

Participation

- Attend all scheduled sessions and activities
- To display professional conduct at all times
- Contribute to group discussions and activities

Personal study

- Personal investigation of concepts and ideas
- Prepare for assessments and exams
- 152 hours minimum outside the module

Behaviour

- To treat all with respect: treat others as you would expect to be treated
- To recognise and address/report all inappropriate behaviours and attitudes

Responsibilities: Ours

Timeliness

- Arrive in time to be prepared for delivery before the start of the lecture/workshop
- To ensure all scheduled sessions and activities run effectively and as planned

Communication

- To provide clear communication of any changes to scheduled events and activities
- To clearly communicate issues and adjustments that may affect study and assessment
- To respond, in a timely manner, to issues raised

Participation

- To support and challenge learning and the development of understanding
- To model and display professional conduct at all times
- To always challenge inappropriate professional behaviours and attitudes whenever they occur

Study

- To professionally prepare and deliver high quality teaching and assessment
- To fairly assess and report individual application of knowledge, skills and understanding

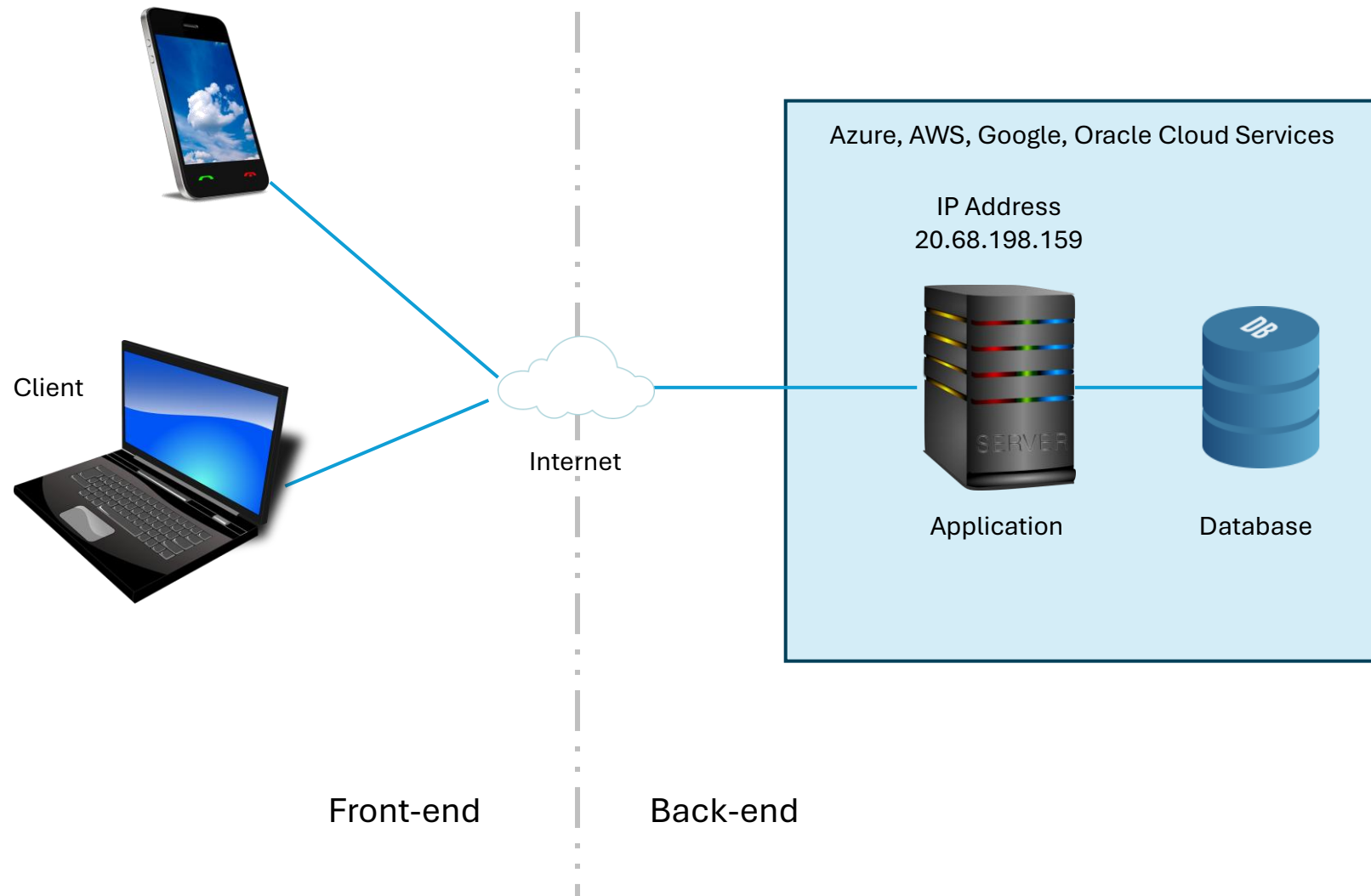
Behaviour

- To treat all with respect: treat others as you would expect to be treated
- To recognise and address/report all inappropriate behaviours and attitudes

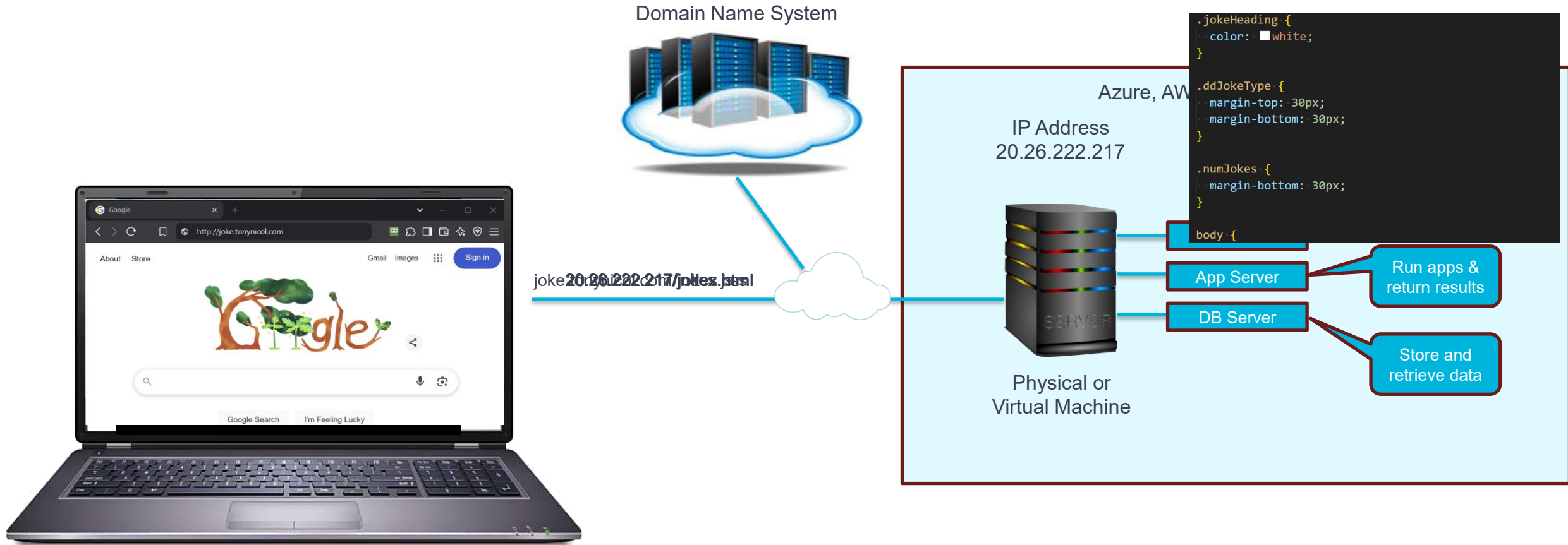


Basic 3-tier
Client Server Architecture
Monolithic

3-tier architecture



3-tier Architecture Demo



Demo joke site in debugger

Monolithic architecture

- ▶ Many if not most organisations implement a **monolithic** architecture which is basically one big program
- ▶ The application, even if it's designed to be modular, can still end up being effectively a single program with hundreds of thousands or millions of lines of code
- ▶ The application typically uses a single database which does make data management simpler, data consistency and performance is good
- ▶ The application may run on a cluster of servers for resilience, but as the cluster or server is increased to get more performance, the whole application needs to be replicated and scaled yet it may only be a small area of the application needing a performance improvement
- ▶ Code changes and testing can be extremely long winded and complicated because the applications are so large, complex, and business processes are tightly coupled - changing one area could create unintended side effects in another area so the whole application must be retested functionally and for performance. This is referred to as **Regression Testing**
- ▶ This level of testing can take months so organisations tend to save up problems and improvements leaving users to find work arounds until the next release - which could be months or years
- ▶ However, the monolith can be very performant through in-process function execution

- 
- ▶ In many situations, a monolith probably the most suitable solution
 - ▶ If the application is not particularly complex or large, e.g., a basic website for a small business, the monolithic approach like the one just shown, is perfectly reasonable
 - ▶ The application is small enough for a person to understand and test so code changes won't take months
 - ▶ However, large organisations such as Tesco, Amazon shopping, Netflix, Government departments etc., need to be able to adapt quickly to changes in business need or changes in government policy and they have huge user numbers of complex business processes
 - ▶ Taking six months or more to make modifications to a business platform is no longer acceptable; a new architectural approach is needed

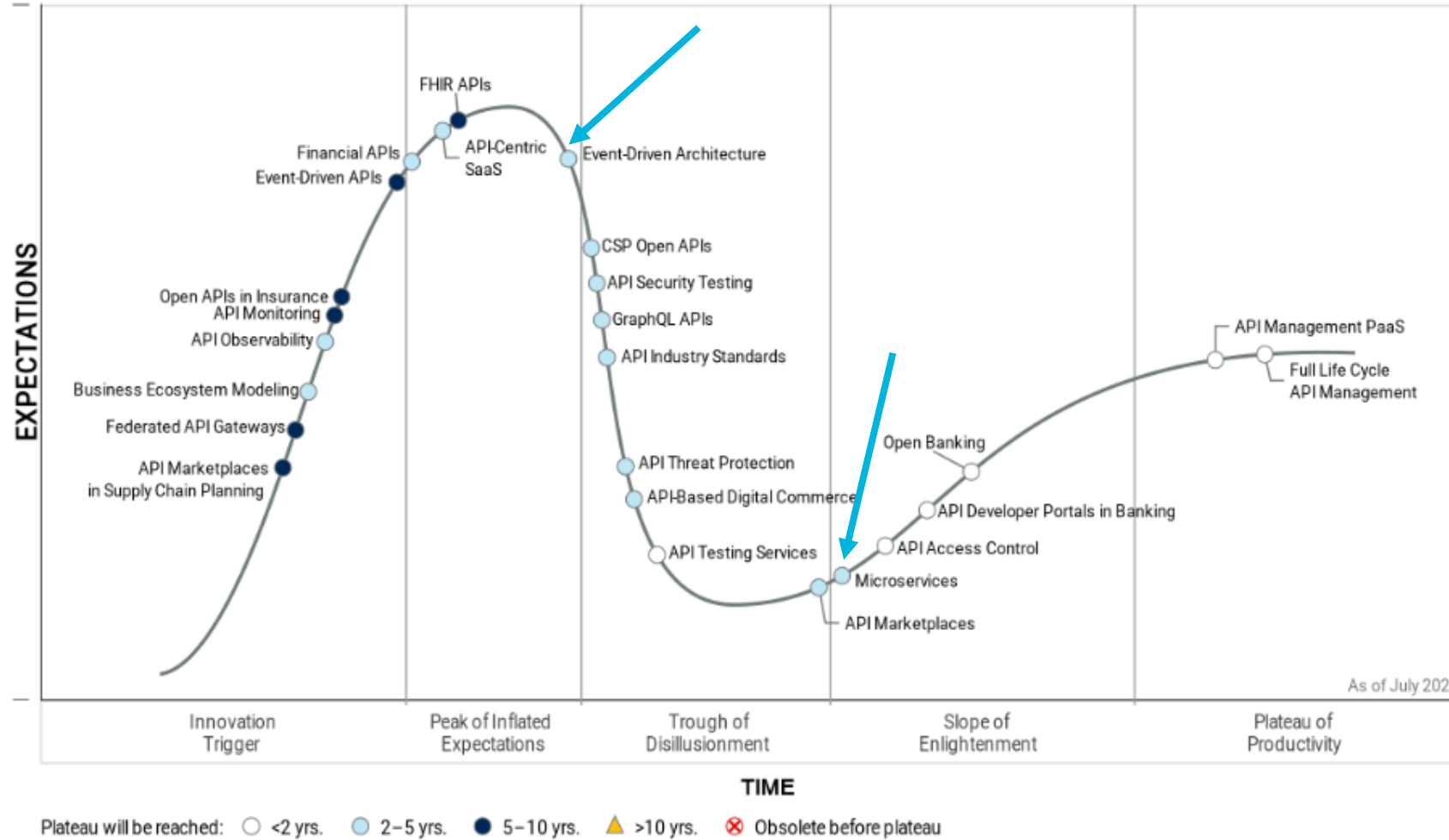


Distributed Systems

- 
- ▶ A **distributed system** appears to be a single system but made up of multiple smaller systems
 - ▶ When the smaller and simpler systems are used in combination, they appear to be a single complex system, but the underlying sub-systems are quicker to create and easier to maintain and test
 - ▶ There are various architectural patterns such a service-oriented architecture (SOA), Event Driven Architecture, **Microservice** Architecture and others
 - ▶ Currently the most popular approach today for distributed systems is a combination of event driven architecture and microservices to create an event-driven microservice architecture
 - ▶ This is not a solution to all problems; each problem needs to consider the best architectural pattern which may well be a monolith or some other approach. This is something for solution and enterprise architects to ponder - engineers will design and implement the operational solution based on the architecture designed by the architects

Microservices

Hype Cycle for APIs, 2023



Microservices

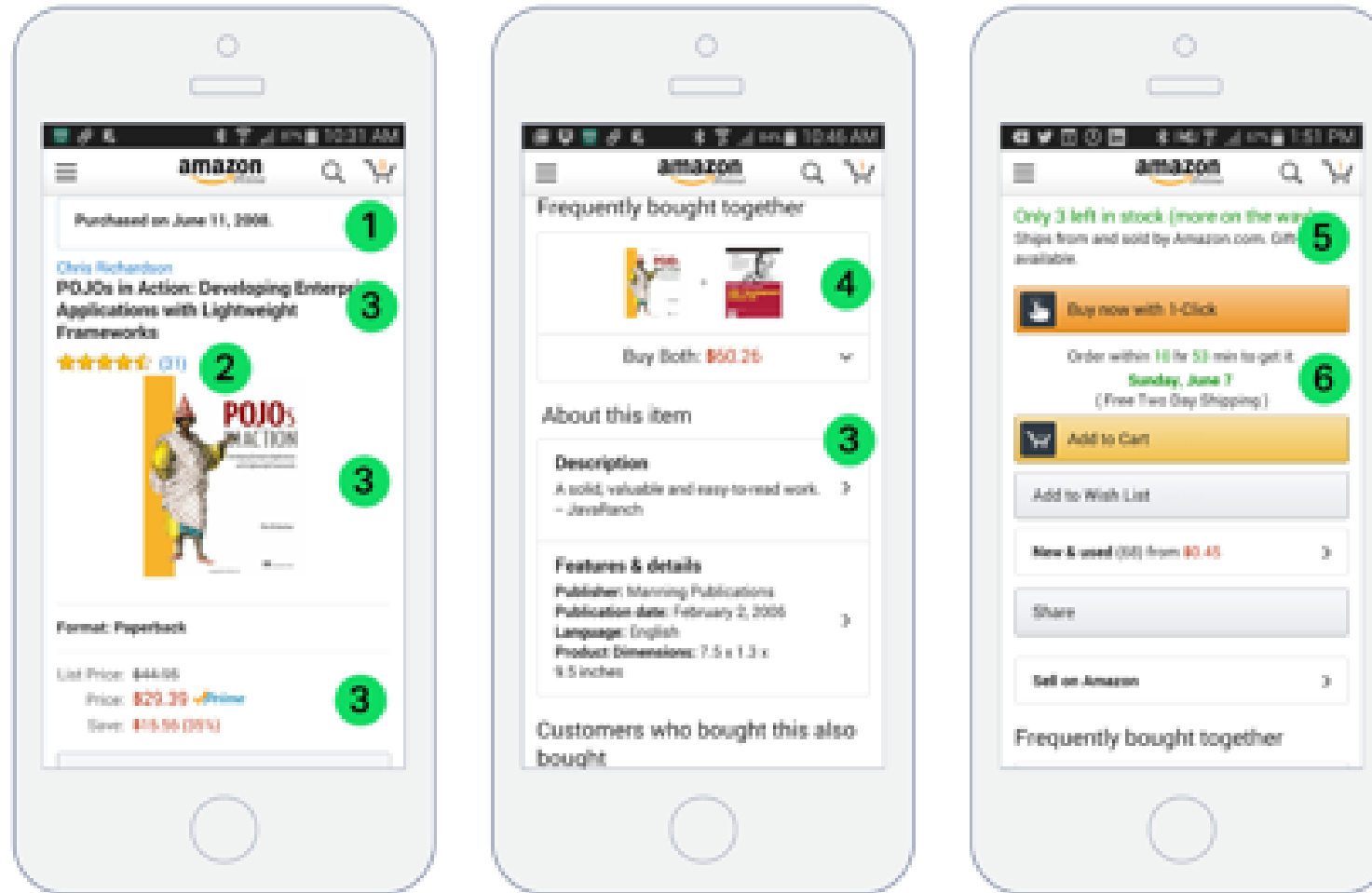
- ▶ A microservice architecture aims to provide an agile response to continuous business change
- ▶ Rather than attempt to create one big application to solve a business problem, the problem is broken into business domain problems, potentially using domain driven design (DDD). i.e., individual areas of the business that can be implemented independently are identified
- ▶ This is similar to general software decomposition where functions or classes are identified, developed and tested in isolation then integrated into the final application. But, in a microservice architecture, the granularity of decomposition is far coarser, each component is a business service and is implemented as a complete application
- ▶ Common examples include:
 - ▶ Identity - login, registration etc
 - ▶ User profile - manage user details such as name, email, preferences etc
 - ▶ Order - manage order creation, status updates and history
 - ▶ Inventory - tracks stock levels, reservations and availability
 - ▶ Payment, notification, shipping and many more

▶ A microservice:

- ▶ is an independent application running on its own hardware with its own database if it needs one - databases aren't shared which prevents changes to data models from impacting other services
- ▶ is loosely coupled to other microservices and a web front-end with well-defined and documented APIs
- ▶ is typically developed, tested, deployed and maintained by a single multi-disciplinary team of 5-9 people including a product owner ("two pizza team")
- ▶ is large enough to provide a business process, but small enough to enable daily or even hourly application updates
- ▶ is agile in that the product pipeline generally used is fully automated CI/CD. e.g. at DWP, developers use DDD for the architecture and test-driven development (TDD) for the code. A lot of effort goes into the tests such that any code changes can be tested automatically and if successful, the code is deployed automatically to the target system
- ▶ can be scaled independently. If one area of the business requires a performance improvement, that microservice can be scaled independently of the others so is more cost effective than the whole application scaling of a monolith
- ▶ is resilient. If one microservice fails, that part of the business is unavailable, but the rest of the business can carry on
- ▶ is a component of a heterogeneous architecture - as it is independent, the programming language, operating system, database technology, hardware etc can be whatever is the most appropriate for that business process

- ▶ Consider Amazon shopping. This is a large complex business - too large and complex for a monolith
- ▶ Independent business areas are identified and microservices implemented. At a high level, let's split that into six services:
 - ▶ Order history
 - ▶ Reviews
 - ▶ Product information
 - ▶ Recommendations
 - ▶ Inventory
 - ▶ Shipping
 - ▶ etc
- ▶ These six (there will be more than this) can be developed in isolation but once deployed, are available as a component making up the whole business service that is Amazon shopping
- ▶ To the user, it looks like one business as the front-end requests resources from various microservices to populate the web page

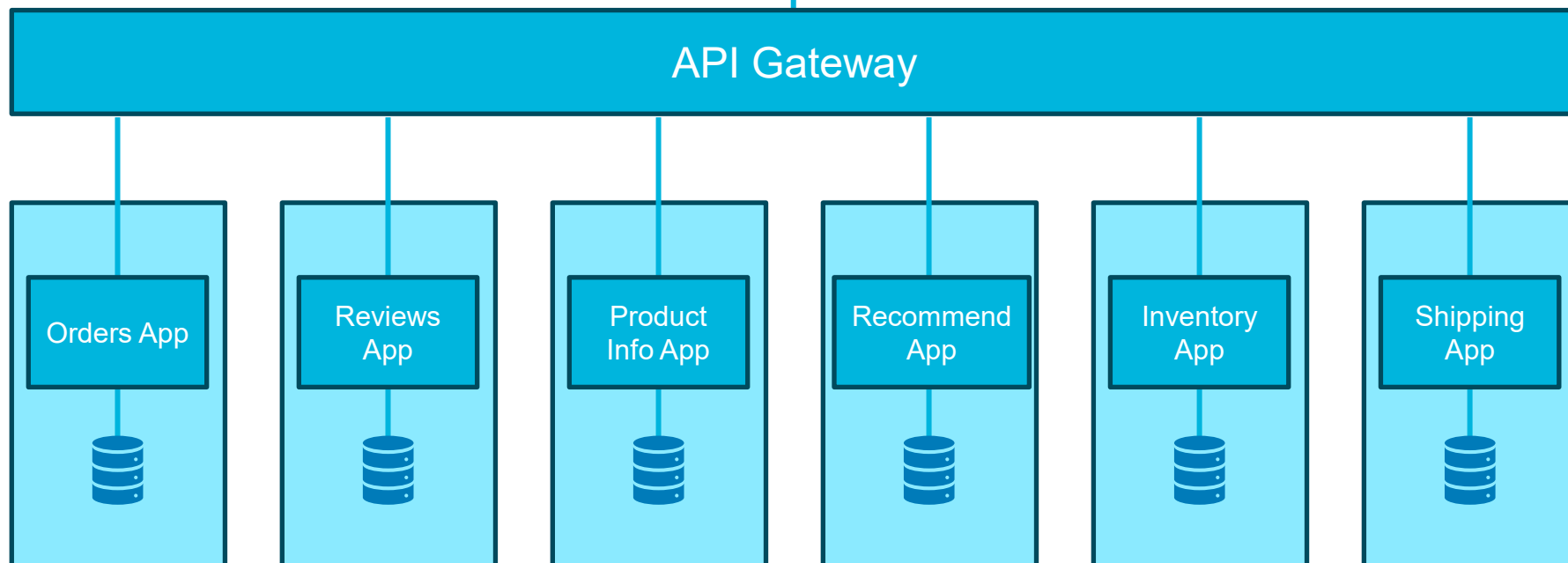
Example shopping site



1. ORDER HISTORY
2. REVIEWS
3. BASIC PRODUCT INFO
4. RECOMMENDATION
5. INVENTORY
6. SHIPPING



Web
Server



- ▶ Like all good ideas, there are challenges that need to be considered:
 - ▶ As components of the service interact across a network, latency is introduced - unlike the monolith
 - ▶ Data consistency is a challenge when data comes from various databases as opposed to one. e.g. for speed, you may want a local copy of some data, so managing consistency with the master can be a challenge
 - ▶ Fault finding can be more difficult as each team focusses on one area it can be difficult to track the problem through the whole system
 - ▶ Communication between microservices can become complex
 - ▶ It is a myth that microservices promote reusability or they are cheaper to build and operate
 - ▶ If your service does not require frequent or at least regular business change, then microservices are probably not appropriate as the key benefit is agility to business change
- ▶ So microservices are not a silver bullet that will kill the monolith, each have their advantages and disadvantages. So, like all design challenges, a decision as to which approach to use is based on the current and anticipated future business needs

- 
- ▶ A Gartner definition of a microservice:
 - ▶ Microservices are application components that are tightly scoped, highly cohesive, strongly encapsulated, loosely coupled, independently deployable and independently scalable. They are deployed as services behind open, standard network-accessible interfaces with well-defined API contracts. These components own the data they present, and only present data that they own

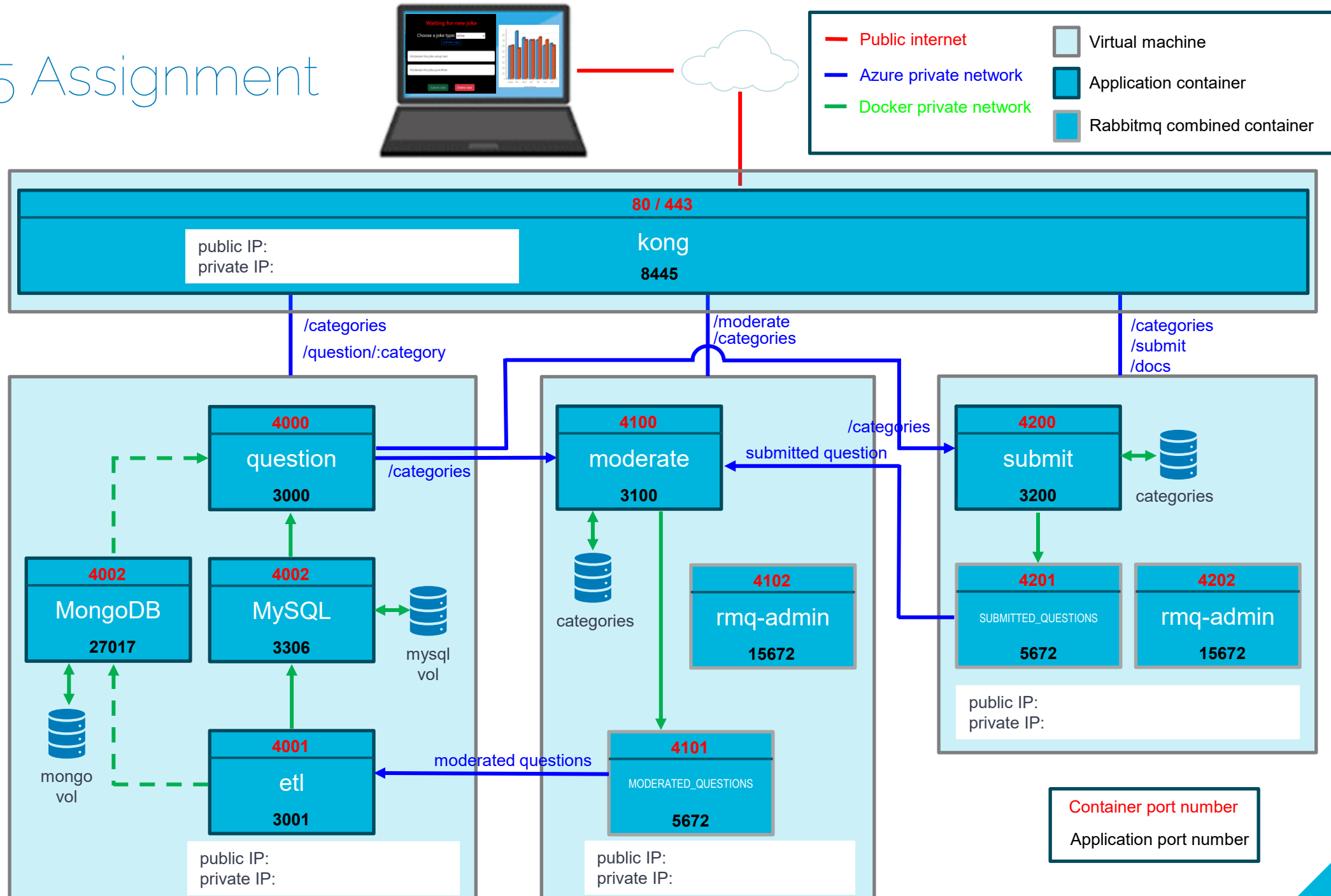
[Gartner, Inc. | G00745207, page 33]

- ▶ A microservice architecture is an application architecture deployed as a distributed system

Business needs

- ▶ A system is there to satisfy a need which is typically a business need. i.e. we don't build a solution then ask what can we use it for, we take a business need and create a solution to satisfy that need
- ▶ The business need may be a new business line or an improvement of current business lines to **respond more quickly to changing customer needs and market trends**
- ▶ The business needs will be gathered by a business analyst and maybe a product owner, then passed to an architect. The architects (solution, network, data, application, enterprise etc) will provide the blueprint for the solution which is then implemented by software engineers, network engineers, security engineers, cloud engineers etc. None of these people do everything but need an awareness of it all
- ▶ The key point is that the business drives the IT not the other way round and it is a team task and not that of an individual so being able to work well in a multidisciplinary team is important as is having a working knowledge of what the other team members are specialists in; this module attempts to give you that
- ▶ There needs to be a business case for funding, technical options are presented and considered against costs and benefits. Considerations such as data protection, system resilience, performance, security, accessibility, operational cost, software licenses, maintenance, telemetry, operating model are all business considerations for any system so requirements tend to fall into two areas: functional and non-functional
 - ▶ Functional requirement: what a system **must do** e.g. "When I press the green button, the red light should light"
 - ▶ Non-functional requirement: **how well it does it**. e.g., "When the green button is pressed, the red light should light within 0.5s"
- ▶ Both will be provided and assessed in the assignments

2024/25 Assignment



Development Approach

- ▶ I'll build up the knowledge of the components in the architecture week-by-week
- ▶ This means that successive weeks build on previous weeks, so missing classes or labs is not advisable
- ▶ I'll discuss and provide demo code for most of the components
 - ▶ You need to work through them to familiarise yourself with the operation
 - ▶ You need to do the lab exercises to practice
 - ▶ If you don't, your assignments will be very difficult to get working



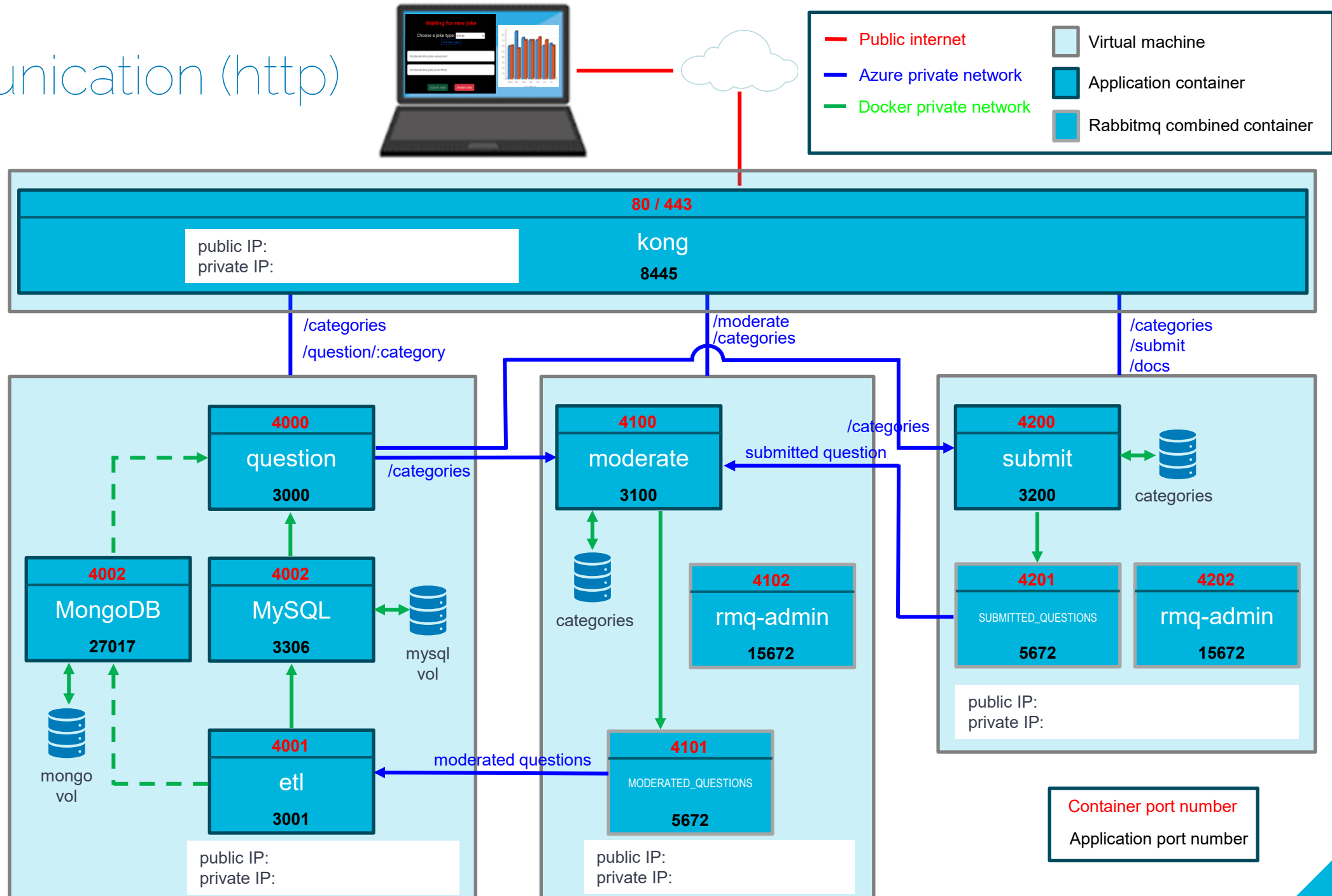
REST and intro to NodeJs

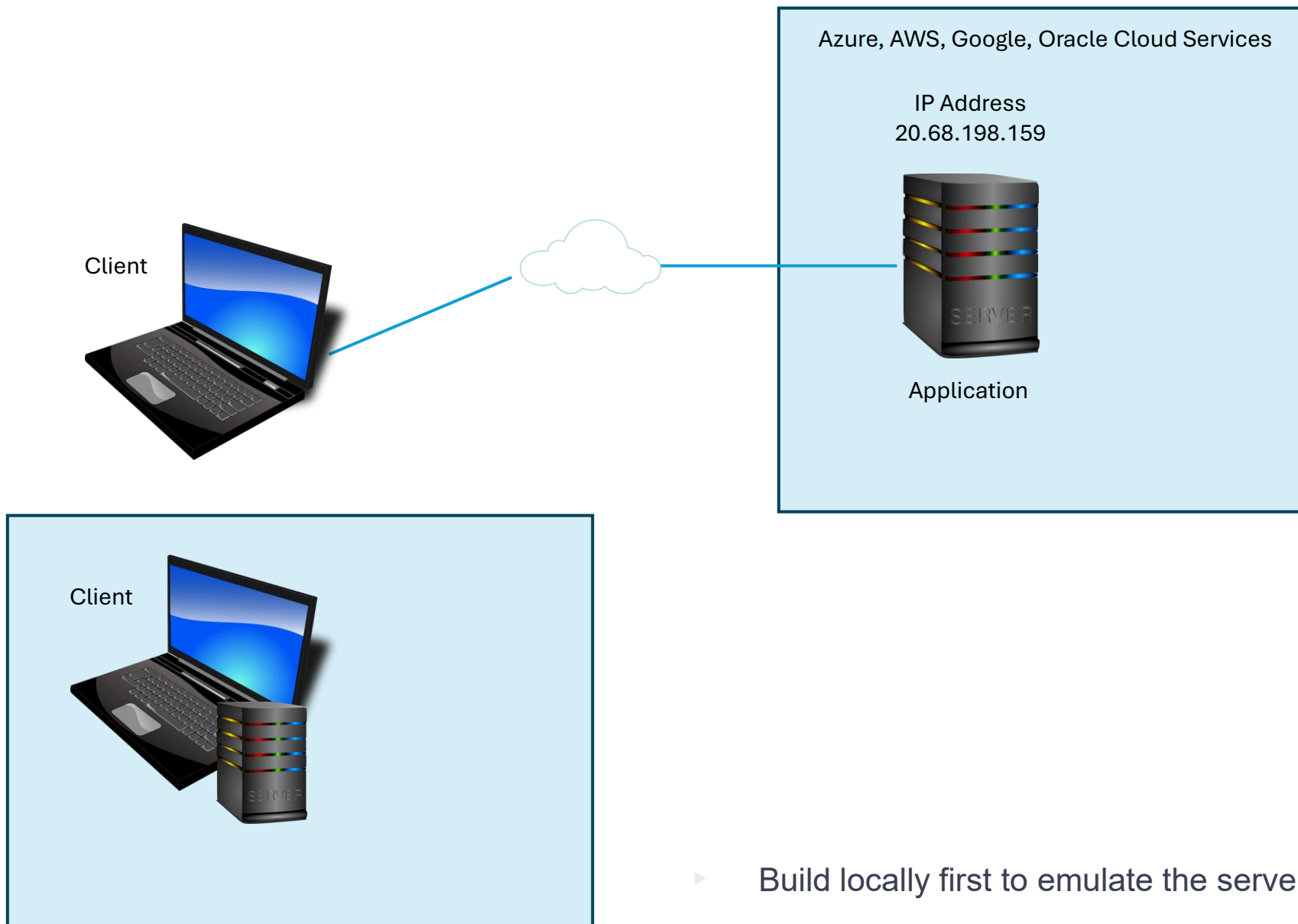


Learning objectives

- ▶ Upon successfully completing this section, you will be able to:
 1. Explain how to structure data using JSON
 2. Explain the structure of a URL
 3. Explain the concept and operation of a web API
 4. Explain the concept of RESTful APIs
 5. Design an API URL with path and query parameters
 6. Create and test a NodeJs / Express API

Communication (http)





- Build locally first to emulate the server for simplicity

Communication using http

- ▶ Monolithic (single app) and distributed (multiple sub-apps), web apps need network communication
- ▶ http is the most popular protocol used to get data from 'a' to 'b', machine to machine
The data structure passed from 'a' to 'b' is typically **H**yper**T**ext **M**arkup **L**anguage (HTML), text, **e**Xtensible **M**arkup **L**anguage (XML) or **J**ava**S**cript **O**bject **N**otation (JSON)
- ▶ If you use JavaScript for the backend too, then json is ideal for data transfer in both directions
- ▶ The key thing is that these data formats are all text – i.e. characters. This avoids incompatibility between different programming languages, operating systems, processor types or whatever as they all support a basic character set, so data transferred as characters is ubiquitous (there are exceptions)

JSON – JavaScript Object Notation

ID	First Name	Last Name	Age
1	Jim	Smith	35
2	Joe	Soap	52
3	Jane	Doe	55

- ▶ Syntax
Data is comma delimited **name:value** (sometimes called key:value) pairs. E.g. "firstName":"John"
- ▶ We need the double quotes for anything other than numeric values
- ▶ JSON equivalent of the table discussed previously:

```
[  
  { "ID" : 1, "firstName":"Jim", "lastName":"Smith", "age":35},  
  { "ID" : 2, "firstName":"Joe", "lastName":"Soap", "age":52},  
  { "ID" : 3, "firstName":"Jane", "lastName":"Doe", "age":55}  
]
```
- ▶ Syntax is native JavaScript, almost no conversion is necessary to send or receive and interpret the data. Structure at both ends still needs to be known. The example will map directly to an array of objects
- ▶ JSON syntax diagrams are clearly explained here: json.org

Web API

- ▶ Typically use http or https to get a web page
 - ▶ <http://20.26.222.217/index.html>, or <https://great-site.com/start.html>
 - ▶ An API is an application programming interface. i.e. a well-defined interface contract (e.g. function name, parameters, types) enabling one program to use functionality of another - e.g. calling Windows OS API functions
- ```
#include <windows.h>
int main() {
 PlaySound(TEXT("C:\\Windows\\Media\\my_sound.wav"), NULL, SND_ASYNC);
 return 0;
}
```
- ▶ Windows is written in C & C++, but can be called from other languages such as Python by mapping a python function call and types to the Windows C call based on contract
  - ▶ A web API is similar but the contract is with a remote server. The contract this time is passed in a url but the principle is the same

# URL

- ▶ `scheme://host[:port][/path][?query]` where [ ] indicates optional. Default port = 80 for http, 443 for https
  - ▶ `http(s)://something.com/something.html`
  - ▶ `http(s)://someAPI.com:3000/weather?city=preston&country=uk`
  - ▶ `http(s)://someAPI.com:3000/weather/{country}/{city}?hours=24&type=json`
  - ▶ scheme and domain are case insensitive, but the rest should be case sensitive
  - ▶ query parameter is search or filter term
  - ▶ you can use 0-9, Aa-Zz, -, ., \_ and ~ Other chars are for some form of control and should be escaped with %
  - ▶ to escape, use % followed by the hexadecimal character code. For example, you are not allowed a space character in a url. If you want one, you would use %20 which is the hex char code for space
  - ▶ e.g. quaint café would need to be written `quaint%20caf%C3%A9`
    - ▶ utf8 charset is here: <https://www.fileformat.info/info/charset/UTF-8/list.htm>
- ▶ The browser will make a request to the server, the server returns a response; this is **synchronous** - browser has to wait
- ▶ Could return html for rendering in a browser or data, typically in json format
- ▶ Web api example (not very good RESTful url):
  - ▶ <http://universities.hipolabs.com/search?country=united%20kingdom>
  - ▶ <http://universities.hipolabs.com/search?country=united%20kingdom&name=university%20of%20central%20lancashire>

# So how does a web API work?

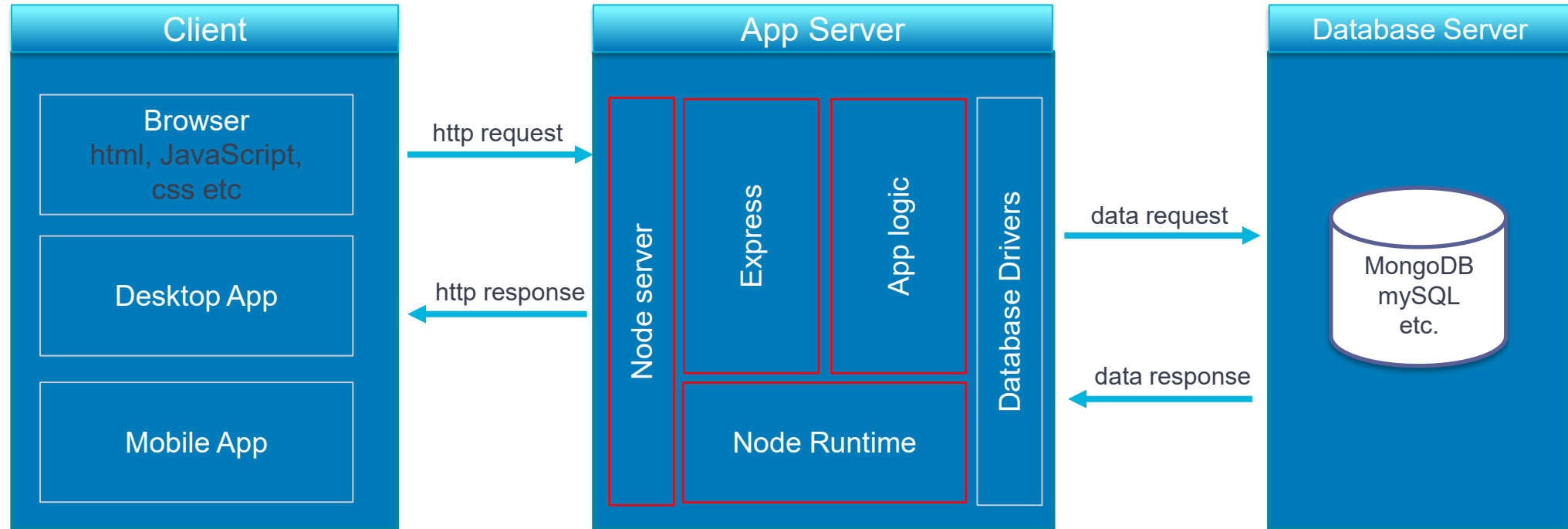
- ▶ A server at IP address, e.g., 20.123.80.240 listens for a request on a specific port number
- ▶ A url is used to form the contract with the api
- ▶ Server parses the request ... e.g. <http://x.com:3000/books/{type}/{author}?count=20>
- ▶ Node.JS is a JavaScript runtime environment
- ▶ To deal with restful APIs, we need a framework to avoid us having to write all this parsing and routing code, so we can install express.js. This has been the de facto framework for Node for many years; other frameworks are available ...
- ▶ Frameworks and libraries avoid you reinventing the wheel. You may have your favourite, or your organisation may stick to one for consistency. It really doesn't matter

- ▶ REST is currently the most popular approach to web API calls so we will focus on that (legacy may use SOAP)
- ▶ REST (REpresentational State Transfer) was introduced in 2000 by Roy Fielding in his PhD.  
[https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding\\_dissertation\\_2up.pdf](https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation_2up.pdf)
- ▶ HTTP has 9 methods. These are not methods in the function() sense, but just tokens the client can send to indicate a required action so are sometimes called http verbs
- ▶ Of the 9 http methods, we typically use five: GET to get data, POST to submit data – e.g. create a new row in a relational database, PUT to update **all** columns in a row, PATCH to update some of the columns, and DELETE to delete a whole row - this enables us to implement all CRUD actions on a database
- ▶ These methods don't do anything other than inform the server of what action the client is requesting
- ▶ An http request contains headers and sometimes a body. The body carries the “payload” i.e. data being sent or returned. The header is more about control or additional information other than the data we are sending or receiving
- ▶ We don't need to worry about many of the headers but just to ensure you know what I'm talking about when I say “http header” a few examples are:
  - ▶ cookies: small text files that are used to remember state and other things
  - ▶ content-type: e.g. application/json if that's the format of the payload to inform the client
  - ▶ content-length: number of bytes in the body
  - ▶ user-agent: which web browser you are using
  - ▶ last-modified: date the file was modified. If server has same, it returns 304 and no data. i.e. no change so use cached version

# Terminology

- ▶ **URL:** full address used to access a resource. e.g., `http://api.example.com:3000/users/123?count=10`
- ▶ **Origin:** the scheme, domain and port. e.g., `http://api.example.com:3000`
- ▶ **Path:** the part of the URL that comes after the domain. e.g., `/users/123` 123 is a path parameter
- ▶ **Query String:** acts as a filter or limiter of returned data. e.g., `?count=10`
- ▶ **Endpoint:** a specific URL (or path) that represents a resource or action in an API. Usually with method
  - ▶ e.g. `GET /users/123` or `DELETE /users/123` are different endpoints
- ▶ **Route:** In backend frameworks (like Express.js, Spring boot, Flask), a route is a **server-side mapping** between a URL pattern and a function or controller

# Architecture



- ▶ So we need Node.js to run the JavaScript
- ▶ We need Express.js to simplify creating the server and to process requests and return responses
- ▶ We need to write some logic – i.e. our app and data access and processing
- ▶ We need an appropriate database driver / SDK to enable our code to access the database
- ▶ We will focus on the areas in red first - the app server to create web APIs

# Node.JS and NPM intro

- ▶ Node is just a runtime environment so doesn't have a graphical UI as it is more of a machine to machine type environment. We can see results in the console window of what is returned to the browser
- ▶ `console.log()` is the method to output to the terminal
- ▶ Rather than bloating the app with loads of libraries you may not use, you can import what you need
- ▶ For example, if we want functionality to parse file paths, we can use the node built-in module 'path'
  - ▶ `const path = require('path');`
- ▶ If you require functionality that isn't included with node, you can install packages created by others who share their code via the node package manager (npm)
- ▶ npm documentation is at <https://docs.npmjs.com/>
- ▶ Search packages at <https://www.npmjs.com/>

# Create App Server and API

- ▶ When creating a new project that depends on external packages, those dependencies are recorded in a package.json file. You can create that manually but it's easier to get npm to set it up before installing any
  - ▶ Enter **npm init** then answer the questions. You can stick to the defaults – just press enter. You can edit later - or use **npm init -y** to answer yes to everything
  - ▶ This way, when storing your code in say git hub, or passing it to someone else, you don't need to save all the packages as they are recorded in the package.json file for npm to install later
- ▶ You will notice the packages.json and node\_modules appear in the file explorer. There can be quite a lot of packages as the packages may themselves include packages
- ▶ If you have just the source files, say clone from git, type **npm install** on its own and npm will read package.json and install all the dependencies listed in that file

[demos/api/minimalAPI](#)

[demos/API/firstAPI/app.js](#)

[demos/API/simpleAPI/app.js](#)



# Server response status codes

- ▶ When returning a response, we need to send an appropriate server response code. By default it is 200
- ▶ e.g. `res.sendStatus(404)`, `res.status(201).json(object)`
- ▶ Status code. This is really important as it indicates the level of success and redirection. There are 5 types:
- ▶ full detail is here: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>
  - ▶ 1xx: **informational**. e.g. 100 indicates a request has been received and is being processed. For example, if you want to check the server will accept your request and headers, don't send the body yet but look for a 100 to mean – good to carry on – i.e. send me your body
  - ▶ 2xx: **success**. The most popular. 200 - Request succeeded. 201 – request success and new resource created ...
  - ▶ 3xx: **redirection**. 300 - there is more than one possible response – you need to choose one. 301 – url changed permanently. Or 302, URL temporarily moved ... New url is returned in the location
  - ▶ 4xx: **error**. The next most popular. 400 – server cannot or will not process the request. 401 – unauthenticated. 404 – resource not found on the server. 429 – too many requests. There are quite a few of these
  - ▶ 5xx: **server error**: 500 – internal server error. 501 – request not implemented. 503 – service unavailable ...



# JavaScript functions

# Learning objectives

- ▶ Upon successfully completing this section, you should be able to:
  1. Write named functions, anonymous functions and arrow functions
  2. Write a function that takes another function as a callback parameter
  3. Write an anonymous function as a function argument
  4. Call an asynchronous API functions such as `setTimeout` and `fetch`
  5. Explain the basic operation of code that uses promises
  6. Create and test a simple node express API application

- ▶ There are a confusingly large number of ways to define / declare a function in JavaScript. You may come across any or all of them, so I'll cover them briefly here
- ▶ A named function is effectively “hoisted” to the top of the file by the parser if not in a block, so can be called before its declaration:

```
getData()
function getData() { ... }
```

- ▶ A function expression like the following creates an **anonymous** function and effectively assigns its address to the variable `getData`. `getData()` can call the function. As `getData` is available from the point of declaration.

```
getData()
const getData = function() { ... }
```

- ▶ The assigned function has the scope in which it's defined so you have more control over it clashing with other similarly named functions (function names have to be unique in the same scope). e.g. the example will throw an error as the function is being called outside the scope in which it was declared because `let` and `const` have block scope
- ▶ Typically these days, function expressions are used to provide better scope control unless a named function is needed or more appropriate, but for what we need, there's no important difference so use what you want

# Arrow functions

- ▶ These are essentially the same as assigned anonymous functions but the syntax is supposed to be more succinct, and some argue easier to read; I'm not one of them. However, they are very popular

```
Syntax is: fn = (param list) => {
 ...
 return ...
}
```

- ▶ If there is only a single statement to execute, return is implied
- ▶ If there is only one parameter, parentheses are optional
- ▶ "this" is not part of the function but inherited from the calling scope

[demos/more functions/arrowFunctions.js](#)

# Callback function

- ▶ Callback functions are used a lot in JavaScript when dealing with asynchronous execution (later)
- ▶ A callback function is a function passed as an argument to another function to be called sometime later
- ▶ The first example shows different functions passed as callbacks, declared in different ways

[demos/more functions/callback1.js](#)

- ▶ The second example shows a callback passed to an asynchronous API function declared in different ways
- ▶ Asynchronicity will be covered in detail later

[demos/more functions/callback2.js](#)

- ▶ We typically use arrow functions in API routes so compare the two demos:

[demos/API/firstAPI/app.js](#)

[demos/API/firstAPI2/app.js](#)



# Asynchronicity in JavaScript

# Learning objectives

- ▶ Upon successfully completing this section, you will be able to:
  1. Explain the difference between synchronous and asynchronous operations
  2. Explain how JavaScript handles asynchronous operations
  3. Explain the purpose and operation of JavaScript callbacks and promises
  4. Use promises in your code for asynchronous operations
  5. Explain the concept of middleware and use it in your solutions
  6. Explain and use same origin requests
  7. Create a front-end that displays API data retrieved from your application and web server



# Sync and Async

- ▶ Synchronous operations are those which take place sequentially. For example, you get to the head of the queue at a sandwich bar at a train station. You place your order for a toastie with the assistant. They go off and do their thing and make the sandwich – maybe chatting to someone or staring into space whilst the sandwich is toasting. Eventually they come back to you and ask “Is there anything else?” You order a coffee. They go off and do their thing – maybe stare into space whilst the milk boils then returns with your coffees. “Anything else?” By now you are getting disapproving looks from others in the queue so say no. You are asked for payment. You pay cash so they go off to the till and cash the money but they don’t have change so have to go into the office to get some change. They eventually complete the payment and you can be on your way whilst half of the queue have given up and gone for their train. Synchronous operations are simple but block execution whilst waiting
- ▶ Ok, you could employ 3 people with 3 tills and 3 toasters etc. This is clearly more efficient but also more expensive. This is parallel processing and now has 3 people (CPUs) staring into space at various times.
- ▶ A different approach could be to employ an asynchronous process. An attendant takes your order but whilst the sandwich is in the toaster, they take your next order – maybe that’s for coffee, but if not they can take an order from the next person. So whilst the toaster is still toasting and the milk is still boiling, they can take another order. At some point, the milk is boiling – maybe the toaster is still toasting. The assistant can stop taking orders and finish the coffee – which started after the toaster so is out of order. During finishing the coffee, the toaster finishes so once the coffee order is complete, they can go to the toaster and finish the toastie order. The key point here is that a long process can be kicked-off but this doesn’t block the assistant. They can do other things whilst waiting. This is more efficient but more complex as the orders may be completed out of order and the person needs to remember which order is for whom. Java uses multithreading for parallelism, JavaScript is single threaded so relies on asynchronous function calls to avoid blocking the single application thread

# Calling an API from the frontend

- ▶ Response from the app server may be slow so waiting for a response will block the single thread
  - ▶ The UI becomes unresponsive whilst waiting - this is synchronous
- ▶ A JavaScript function, `fetch`, will make the API call but asynchronously
  - ▶ The call is made but we don't wait for the response
- ▶ The UI remains responsive whilst `fetch` is waiting for a response in the background
  - ▶ effectively running in a separate thread (discussed later)
- ▶ We also need to consider not blocking the server thread if it needs to do something lengthy
  - ▶ e.g. search for data in a database. Make this server call asynchronous too
- ▶ When the response is received from the app server, it is somehow dealt with in the main thread
- ▶ A couple of quick examples to illustrate this so you can do your practical exercise
- ▶ Detail of how it all works follows

[demos/API/fetchBasic1](#)  
[demos/API/fetchDemo1](#)

```
function level1() {
 level2();
}

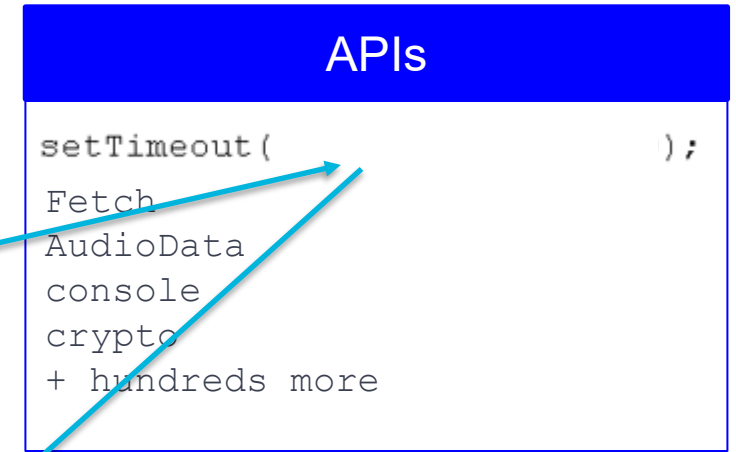
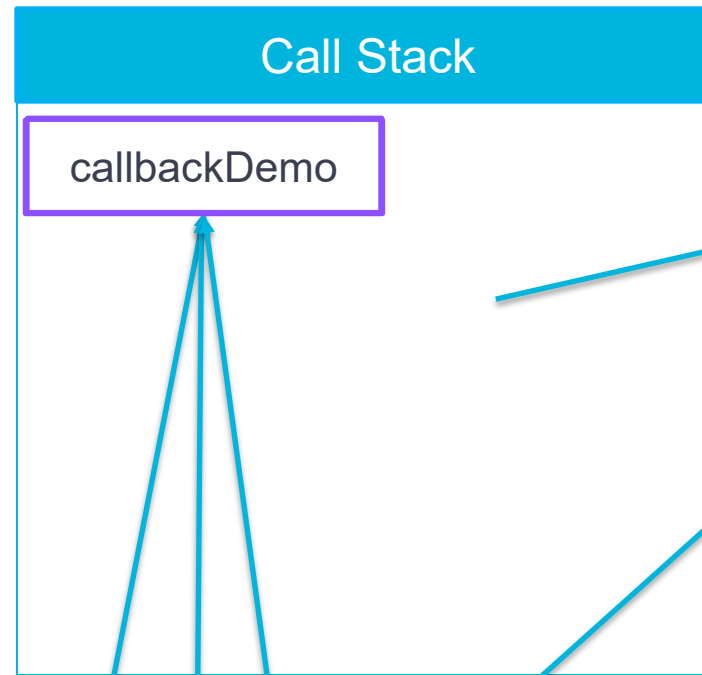
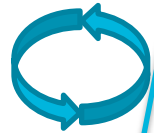
function level2() {
 level3();
}

function level3() {
 setTimeout(callbackDemo, 2000);
}

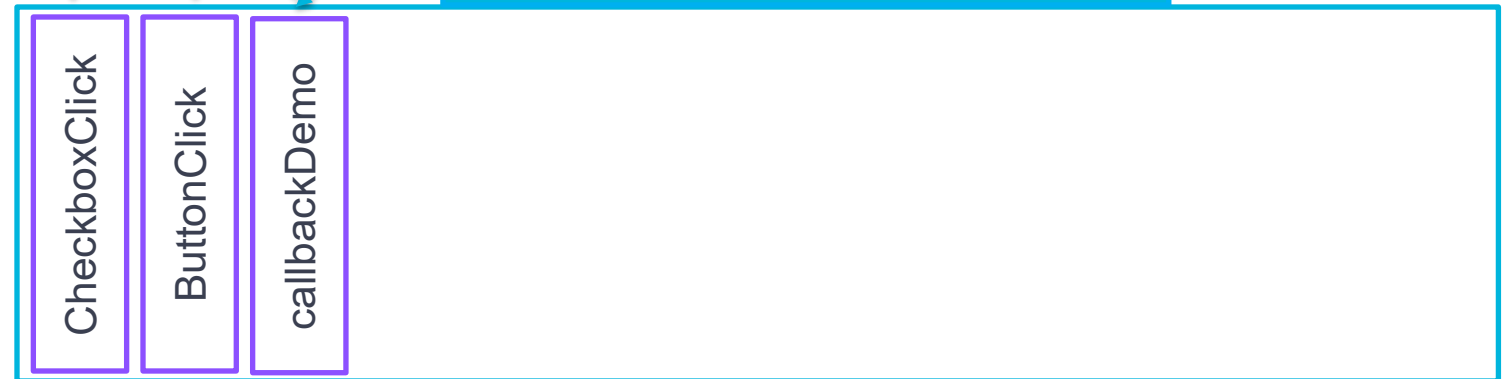
const callbackDemo = function () {
 console.log(`In callbackDemo`);
}

level1();
```

Message  
Handler



Event queue (FIFO)

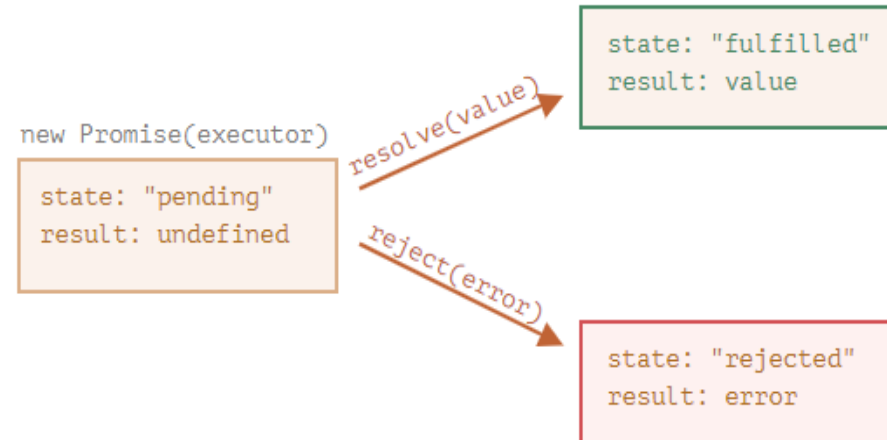


# Asynchronous approaches

- ▶ We've seen how callback functions like `setTimeout` work in JavaScript to make an async call
- ▶ But, multiple callbacks within callbacks can get quite complex, especially in error conditions, so the concept of a **promise**, was added to the language. Callbacks are still used
- ▶ A further addition to the language is **async** and **await**. These further simplify the coding of asynchronous operations using promises
- ▶ Callback, promise and `async / await` are a little tricky to get your head around at first and are areas that interviewers often ask about to differentiate basic JavaScript programmers from those with an understanding of asynchronous operations handled by these JavaScript language features
- ▶ To reiterate: a callback is a function which is passed to another function as an argument. The function "registers" the callback. i.e. it notes the function's execution address. If the function is asynchronous, when the main function, e.g. `fetch` or `setTimeout`, is ready, e.g., reading data from a file or database or whatever it is that's likely to take time, it will pass the registered callback function to the event queue to be executed when the thread is available. Using a promise is just a syntactically better way of achieving the same result
- ▶ If the callback is not asynchronous, when the parent function is ready, it will execute the callback synchronously

# Promises

- ▶ A promise is called that because it is a promise, agreed now, but to do something later. A promise is an **object** with two key properties: state and result. It has three states: pending, fulfilled or rejected. Whilst it waits for an asynchronous event to complete, it has a status of “pending”. This is just a newer way of implementing asynchronous operations instead of callbacks
- ▶ We can create promise-based code and we use promise-based code from others



```
const smallNum = new Promise((resolve, reject) => {
 setTimeout(cbTimeout, 2000, resolve, reject)
}).then(cbThen)
 .catch(cbCatch)
 .finally(cbFinally)
```

```
function cbTimeout(resolve, reject) {
 let num = 6
 if (num < 5) resolve(`${num} is good`)
 else
 reject(`${num} is too big`)
}
```

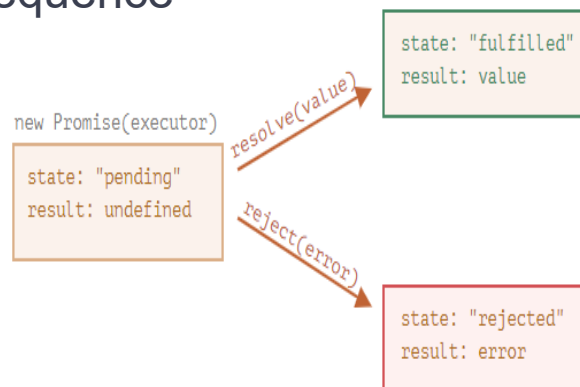
```
function cbThen(result) {
 console.log(result)
}
```

```
function cbCatch(result) {
 console.log(result)
}
```

```
function cbFinally() {
 console.log("Promise finally")
}
```

```
console.log("Done")
```

- ▶ Written using full functions to illustrate sequence
- ▶ Typically use arrow functions
- ▶ Typically create promise in a function



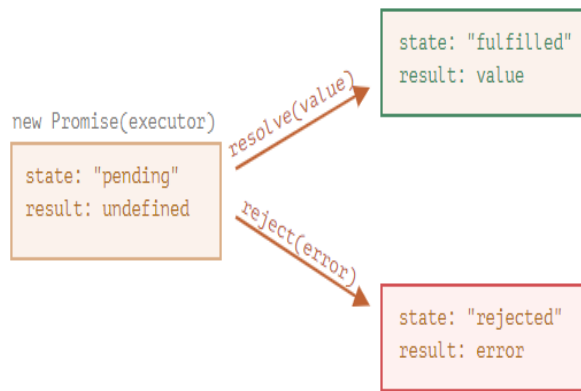
[demos/async/promise1.js](#)

[demos/async/promise1b.js](#)

# Async / Await

- ▶ Promises were introduced to simplify callbacks but they are still not particularly intuitive as code is executed out of sequence so you need to think differently when dealing with asynchronous processing. Async / await has been introduced to simplify things even more by enabling you to make the code look more like synchronous code
- ▶ A promise needs to be set into a state from pending to have its **.then()** or **.catch()** function callback called to handle the response to the asynchronous event. Async / await enables some, arguably better readability by enabling the result of the async function to be dealt with in what looks like a more synchronous manner. Async / await uses promises but is abstracted further from the detail
- ▶ Async/await is an option, not a replacement – there are situations where you may prefer to use one over the other, or both - it's up to you
- ▶ Many library functions now support promises - fetch is one of them
- ▶ Note: to be able to **await** a response, await must be within an **async** function

```
checkSmallNum(5)
console.log("Done")
```



- ▶ Looking at the same example
- ▶ Still uses a promise but async await syntax

```

function getSmallNum(num) {
 const prom = new Promise((res, rej) => {
 // Immediate execution of the callback
 // Reg timer callback - not executed
 setTimeout(() => {
 if (num < 5) res(`${num} is good`)
 else rej(`${num} is too big`)
 }, 2000)
 })
 // Call function returning promise
 return prom
}

// Needs to be async to use await
async function checkSmallNum(num) {
 try {
 // Pending
 const result = await getSmallNum(num)
 // Resolved
 console.log(`${result}`)
 } catch (err) {
 // Rejected
 console.log(`${err}`)
 } finally {
 // Do this regardless
 console.log('Promise Finally')
 }
}

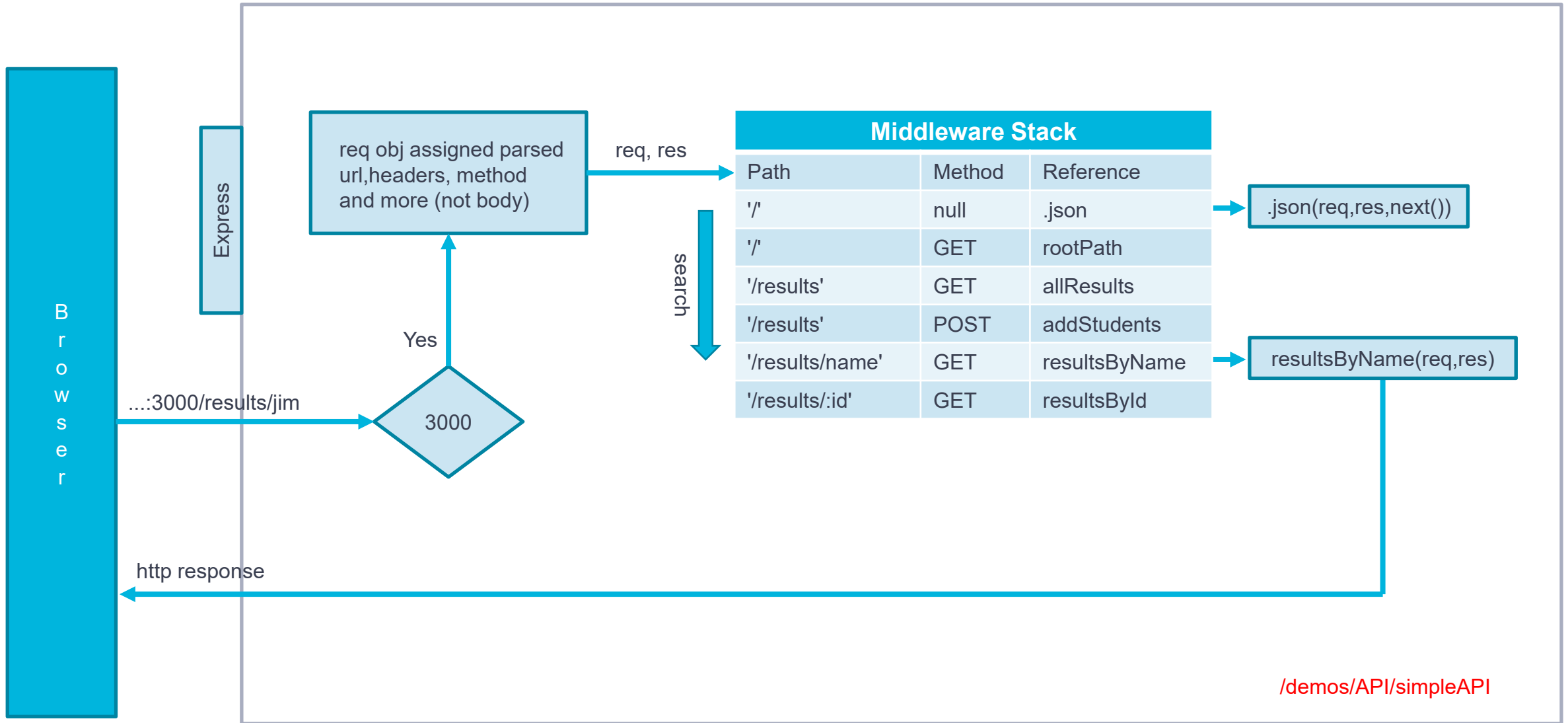
```



- ▶ Note: if you're a detail person, there is also a microtask queue:
- ▶ JavaScript uses an **event loop** to manage asynchronous operations. It has **two main queues**:
  - ▶ **Macrotask queue** (task queue or event queue on the diagram)
    - ▶ Includes: setTimeout(), setInterval(), fetch(), I/O, UI events
    - ▶ Scheduled to run **after** the current call stack is empty and **after all microtasks** have been processed
  - ▶ **Microtask queue**:
    - ▶ Includes: Promise callbacks (.then, catch, finally) and a few others.
    - ▶ These are processed **immediately after the call stack clears, before** any macrotasks

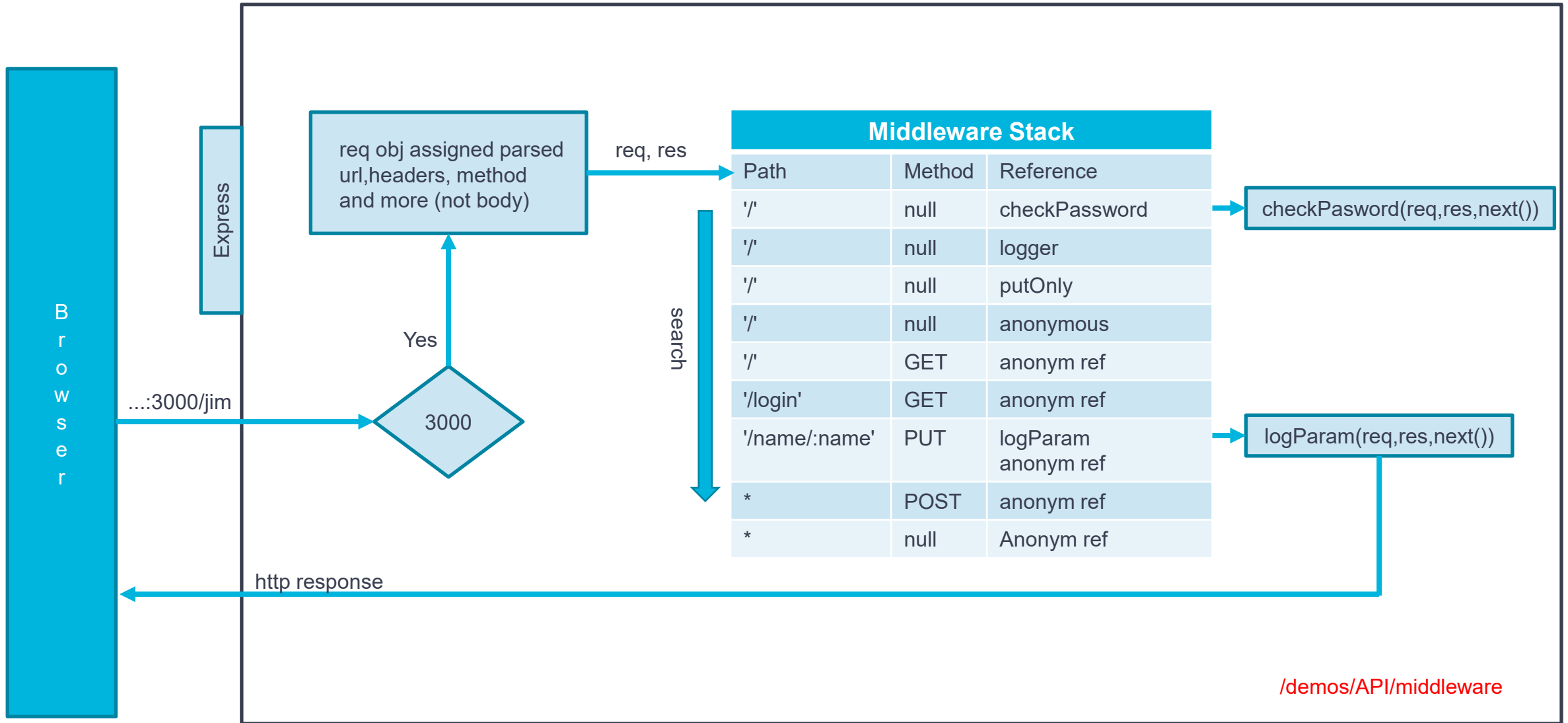


# Express and Middleware



# Express Middleware

- ▶ Middleware in this context is basically functionality that you have written or library packages that can be added to a call list (middleware stack) and have access to req and res
- ▶ Middleware is added to the middleware stack, typically using `app.use()` to add a function to the middleware stack
- ▶ Middleware is called in the order it appears on the stack. If a middleware completes, e.g., `res.send()` none of the others are called. However, middleware functions, if they can't complete the job, or they do something en-route to an endpoint, maybe add some stuff to the req or response object they need to call `next()` to pass control to the next middleware in the list etc.
- ▶ We won't get involved in anything complicated but will add some middleware that gets called before our route for pre-processing then generally pass control to the route's callback function to complete with a `res.send()` or similar
- ▶ The middleware stack has middleware added to it in the order it appears in the source file so be wary of that



- ▶ The middleware functions appear on the stack in the order they are defined
- ▶ Each is checked in order and if they don't terminate, call next to keep the search going
- ▶ The putOnly middleware isn't special, it is called for all methods but the called function only responds if the method is PUT
- ▶ `app.use('/login', checkPassword)` example, adds checkPassword to the /login route for ALL methods
- ▶ The `/name/:name` is different. As the middleware only applies to it explicitly, i.e. `PUT /name/:name`, it is added into the stack array element object and called before the path and on completion calls the route function by calling `next()`. There could be any number of these middleware functions passed as parameters in the path and they will be stacked in order from left to right
- ▶ The middleware can modify the content of the request and response objects that it passes on



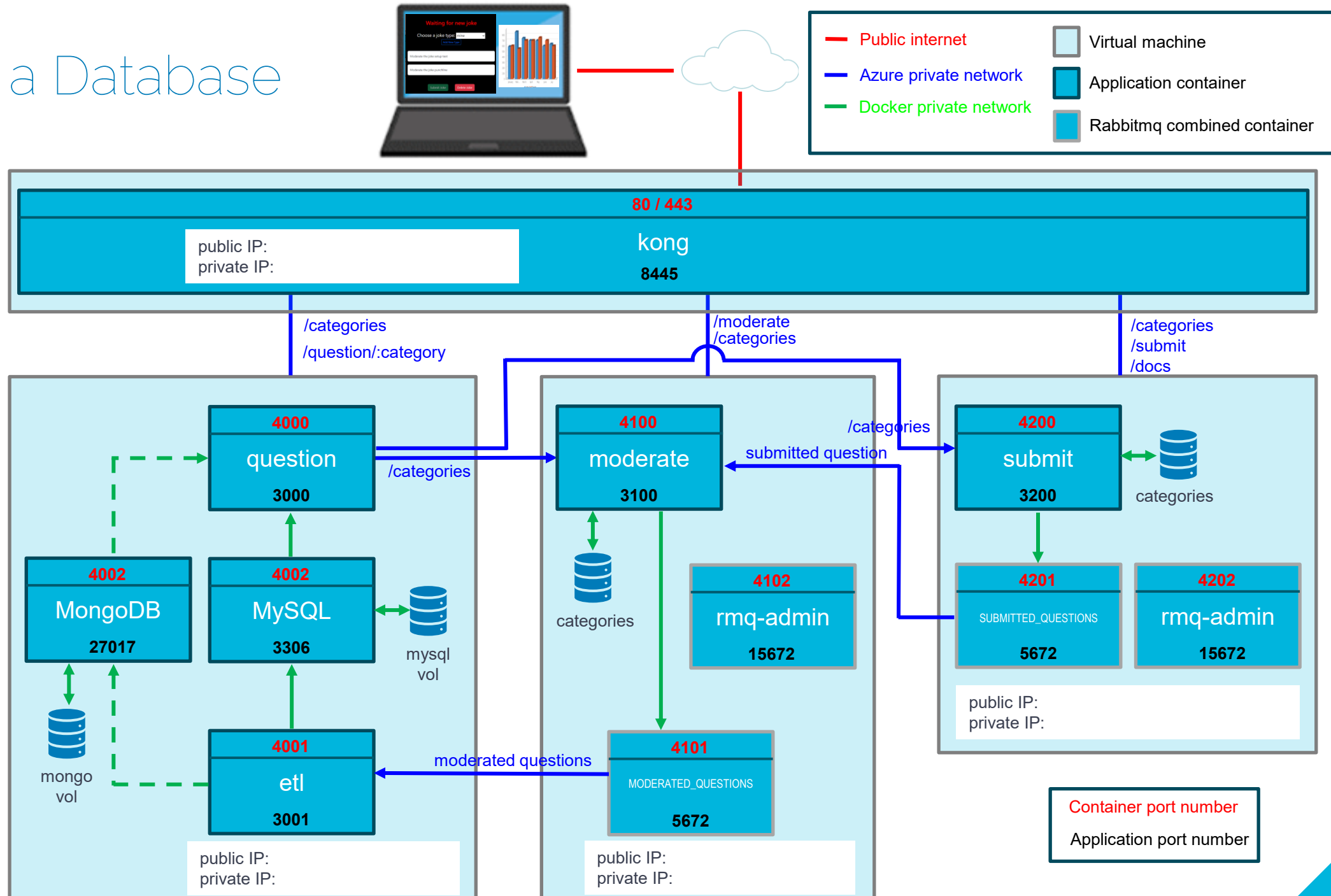
Adding a database to the stack

# Learning objectives

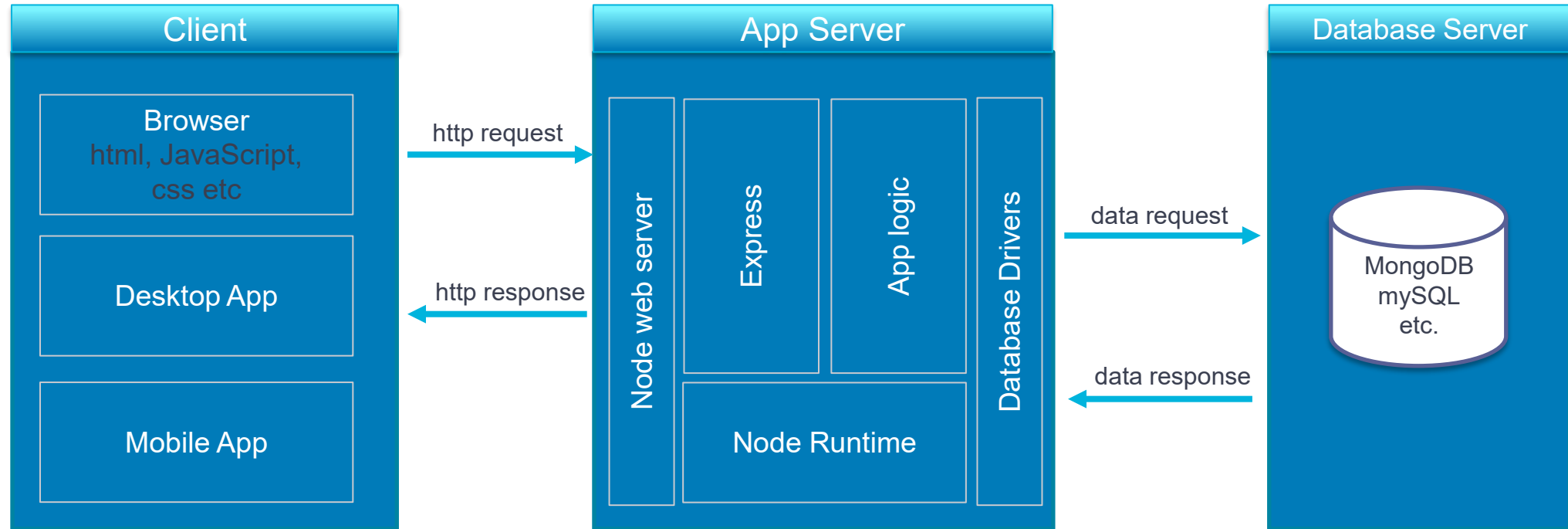
- ▶ Upon successfully completing this section, you will be able to:
  1. Explain the ETL process
  2. Explain cardinality in an entity relationship diagram
  3. Explain and create table relationships using primary and foreign keys
  4. Explain and create referential integrity rules
  5. Interpret and write basic SQL
  6. Explain SQL injection and how you can prevent it
  7. Explain and use the different scopes of environment variables
  8. Explain the benefits of using environment variables in maintainability and application configuration
  9. Explain the benefits of database connection pooling in enterprise systems
  10. Explain and implement database connection pooling which manages multiple concurrent connections
  11. Create an application which adheres to the principle of separation of concerns



# Adding a Database



# Architecture



- ▶ So we need Node.js to run any back-end code in JavaScript and provide the static web server
- ▶ We need Express.js to simplify creating the server and to process requests and return responses
- ▶ We need to write some logic – i.e. our app, data access and processing
- ▶ We need an appropriate database driver / SDK to enable our code to access the database
- ▶ We need a database management system / database server

- ▶ It's time to add a database to complete the 3-tier architectural stack
- ▶ There are various database types, mainly relational and various no SQL
- ▶ We will look at relational first using MySQL server, now owned by Oracle
  - ▶ You should install mySQL server and Workbench from <https://dev.mysql.com/downloads/mysql/8.0.html>
- ▶ This is not a database course, you did enough in the first year to cover this module, so I'll stick to basics
- ▶ MySQL is a very popular relational database system, RDBMS, but since Oracle took it over, it has been forked and some prefer to use the fork, MariaDB. They are basically the same, but we will use MySQL
- ▶ A relational database uses multiple tables which are related in some way. A process of data normalisation is used to remove repetition from the data and determine relationships
  - ▶ If you can't remember how to normalise from the first year, just look it up as revision
  - ▶ We will not create complex data models or require in-depth normalisation
- ▶ We'll do this in two parts
  - ▶ Quick and dirty to get the database working and integrated using a single simple table
  - ▶ A more engineered approach

[demos/data/basic/app.js](#) (create db and table manually)

[demos/data/mysql2\\_basic\\_promise/app.js](#)

# Data Engineering

- ▶ Software Engineers work closely with Data Engineers so need a good grasp of data operations
- ▶ To summarise a software developer should be able to analyse, standardise, **E**xtract & clean, **T**ransform, and **L**oad (ETL) data into data structures they or a data engineer have modelled to enable others to use and analyse - i.e. create related normalised tables and write the data to the tables if using RDBMS
- ▶ Various tools are used in this process, or you may write code for continuous ETL into a database
- ▶ e.g., one system creates a log of all the changes made to its database each day. In the evening, a batch process job may be run to transfer that data to an analytical system to be added to data from other systems for analysis. That data may be transferred as comma separated (or other) values
- ▶ A data engineer / software developer will need to clean the data
  - ▶ scan for missing column values, blank records, ensure compliance to data standards etc
- ▶ They will **extract** the data from the file in whatever format it is in
- ▶ They will **transform** the data - e.g. set all names to upper or lower case, make sure date formats are correct etc, align to organisational data standards, maybe separate data based on table destination
- ▶ Then **load** the data into a table or a set of related tables
- ▶ The ETL process is a very popular process in data processing systems

| first_name | last_name | marital_status | grade              | name          | fee     | qualification | title | name   |
|------------|-----------|----------------|--------------------|---------------|---------|---------------|-------|--------|
| Kevin      | Bacon     | Married        | Distinction        | Physics       | 3200.00 | MSc           | Dr    | Spock  |
| Kevin      | Bacon     | Married        | Distinction        | Music         | 1200.00 | Diploma       | Prof  | Finkle |
| Billy      | Wibble    | Single         | Merit              | Chemistry     | 1200.00 | Diploma       | Mrs   | Jones  |
| Jane       | Doe       | Single         | Upper Second Class | Mathematics   | 1500.00 | BSc           | Dr    | Bob    |
| Jane       | Doe       | Single         | Upper Second Class | Mathematics   | 1500.00 | BSc           | Dr    | Love   |
| Jane       | Doe       | Single         | Pass               | Music         | 1200.00 | Diploma       | Prof  | Finkle |
| Joan       | Jet       | Single         | Pass               | Electronics   | 800.00  | City & Guilds | Prof  | Plumb  |
| Joan       | Jet       | Single         | Pass               | Electronics   | 800.00  | City & Guilds | Mr    | Mann   |
| Joan       | Jet       | Single         | Merit              | Chemistry     | 1200.00 | Diploma       | Mrs   | Jones  |
| Paul       | Smith     | Married        | Merit              | Physics       | 3200.00 | MSc           | Dr    | Spock  |
| Paul       | Smith     | Married        | First Class        | Computer S... | 2000.00 | BSc           | Mr    | Smith  |

- ▶ There are duplicates
- ▶ If you wanted to find out Joan Jet's grade in Electronics, how would you do it?
- ▶ Searching will return two results - there could be two Joan Jets - there is no unique ID
- ▶ If it's the same person and she had her grade changed on appeal, if not all records updated then data is inconsistent
- ▶ There are two columns called **name**
- ▶ We need some uniqueness and to avoid checking many columns, so break the table up into tables

- ▶ A student can be on more than one course, but for simplicity, a tutor can only teach one course, but a course can have more than one tutor. I just made this up, but this information would be elicited from the business by a business analyst working in conjunction with an engineer
- ▶ Split into entities of related information using data normalisation, e.g. student, course and tutor. Each is given a unique ID to search on, this is the primary key. There are no duplicates
- ▶ The tutor table enables us to find the course they teach using the course primary key, e.g. staff number
- ▶ Grade is a many-to-many relationship as many students can be on a course and a course can have many students but a grade is unique to one student on one course. A single primary key on grade would give us nothing useful but a primary key made of two other primary keys to form a composite primary key is useful as it relates the grade to the student **and** the course. We deal with this scenario with a linking or joining table with both key values. e.g. What grade did Kevin Bacon get for Physics?
- ▶ There's repetition in grade, student and course - so marital-status, qualification and grade could have their own tables and be replaced by foreign keys. That would require more joins which may impact performance so it's a trade-off
- ▶ The 4 tables, functionally dependent on a key that will work are:

student

| id | first_name | last_name | marital-status |
|----|------------|-----------|----------------|
| 1  | Kevin      | Bacon     | Married        |
| 2  | Billy      | Wibble    | Single         |
| 3  | Jane       | Doe       | Single         |
| 4  | Joan       | Jet       | Single         |
| 5  | Paul       | Smith     | Married        |

course

| id | name             | fee     | qualification |
|----|------------------|---------|---------------|
| 1  | Computer Science | 2000.00 | BSc           |
| 2  | Mathematics      | 1500.00 | BSc           |
| 3  | Physics          | 3200.00 | MSc           |
| 4  | Chemistry        | 1200.00 | Diploma       |
| 5  | Music            | 1200.00 | Diploma       |
| 6  | Electronics      | 800.00  | City & Guilds |

tutor

| id | title | name   | course_id |
|----|-------|--------|-----------|
| 1  | Prof  | Plumb  | 6         |
| 2  | Dr    | Spock  | 3         |
| 3  | Dr    | Bob    | 2         |
| 4  | Mr    | Smith  | 1         |
| 5  | Mrs   | Jones  | 4         |
| 6  | Prof  | Finkle | 5         |
| 7  | Mr    | Mann   | 6         |
| 8  | Dr    | Love   | 2         |

grade

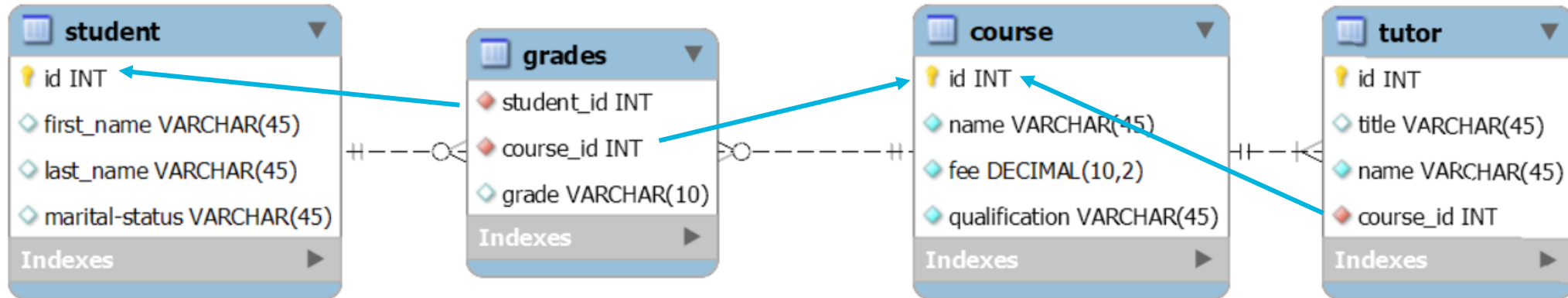
| student_id | course_id | grade              |
|------------|-----------|--------------------|
| 5          | 3         | Merit              |
| 5          | 1         | First Class        |
| 4          | 6         | Pass               |
| 4          | 4         | Merit              |
| 1          | 3         | Distinction        |
| 1          | 5         | Distinction        |
| 2          | 4         | Merit              |
| 3          | 2         | Upper Second Class |



# Data modelling

- ▶ You would likely be designing a new product and don't yet have any data. You work with the business analysts to determine what data you need.
- ▶ The design would be used by data engineers and software engineers to create the solution
- ▶ Modelling it is typically done using an entity relationship diagram ERD in a modelling tool such as Erwin
- ▶ MySql workbench has a basic modelling tool built-in so I used that to create the following entity relationship diagram (ERD) diagram but the tables we will use are easily normalised and modelled without this but it's handy for documentation
- ▶ Workbench can take an existing database and create an ERD from it
- ▶ Workbench enables you to draw the diagram and create all the SQL to create the tables from the diagram
- ▶ There are more professional tools for enterprise designs with many more features

- ▶ The notation in this example is referred to as "crow's foot" notation

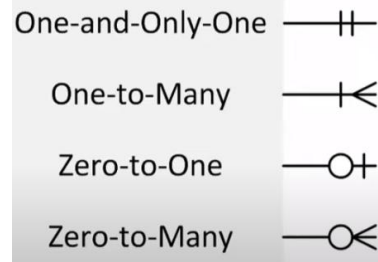


Left to right: each student can have zero or more grades

Right to left: a grade can be associated to 1 student.

Right to left: a course can have 0 or more students.

Right to left: a tutor can teach on only one course (in this example)



Left to right: A grade can be associated to only one course

Left to right: A course can have only 1 tutor

Decent tools show the relationship between primary and foreign key so I've added the blue arrows  
We may want a direct relationship between student and tutor. Is there a relationship between student and tutor beyond the course? Personal tutor? This is all part of requirements gathering



- ▶ Relational databases are accessed and operated using the Structured Query Language (SQL)
- ▶ With SQL we can create a database, tables, add data, modify data, request data etc
- ▶ You can use the workbench GUI but it still shows you the SQL you could have used
- ▶ We will only be using a subset of SQL so I'm not going to go through it all - there are many references
  - ▶ `SELECT * FROM student_db.course;` To avoid having to specify the database every time double click **schema** or specify it: **use student\_db;**
  - ▶ `SELECT name, qualification FROM course;`
  - ▶ `INSERT INTO student (first_name, last_name, marital_status)  
VALUES ('Tony', 'Nicol', 'Married');`
  - ▶ `DELETE FROM student WHERE id=7;`
  - ▶ If the data you need is in more than one table, you need to join them to create a resultant set. e.g., we want to list all tutors and the courses they teach. It is customary to put SQL keywords in capitals but not required:  
`SELECT tutor.title, tutor.name, course.name  
FROM tutor  
JOIN course on course.id = tutor.course_id;`

Demo some queries

- ▶ Join all tables to find what courses and grades Joan Jet is on and the tutor names
- ▶ This will create multiple rows if there are multiple tutors. If you don't want that you can use concat

```
select first_name, last_name, course.name, grades.grade, tutor.title, tutor.name
from student
 inner join grades on student.id = student_id
 inner join course on course.id = grades.course_id
 inner join tutor on course.id = tutor.course_id
where student.last_name = "Jet" and course.name = "Electronics";
```



| first_name | last_name | name        | grade | title | name  |
|------------|-----------|-------------|-------|-------|-------|
| Joan       | Jet       | Electronics | Pass  | Prof  | Plumb |
| Joan       | Jet       | Electronics | Pass  | Mr    | Mann  |

```
SELECT
 student.first_name,
 student.last_name,
 course.name,
 grades.grade,
 GROUP_CONCAT(CONCAT(tutor.title, ' ', tutor.name)SEPARATOR ', ') AS tutors
FROM student
INNER JOIN grades ON student.id = grades.student_id
INNER JOIN course ON course.id = grades.course_id
INNER JOIN tutor ON course.id = tutor.course_id
WHERE student.last_name = 'Jet' AND course.name = 'Electronics'
GROUP BY
 student.first_name,
 student.last_name,
 course.name,
 grades.grade;
```



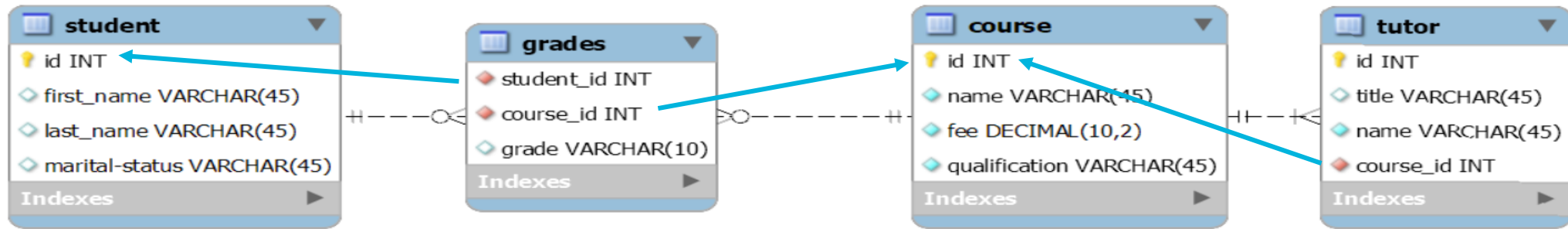
| first_name | last_name | name        | grade | tutors              |
|------------|-----------|-------------|-------|---------------------|
| Joan       | Jet       | Electronics | Pass  | Prof Plumb, Mr Mann |

# Referential Integrity

| id | title | name   | course_id |
|----|-------|--------|-----------|
| 1  | Prof  | Plumb  | 6         |
| 2  | Dr    | Spock  | 3         |
| 3  | Dr    | Bob    | 2         |
| 4  | Mr    | Smith  | 1         |
| 5  | Mrs   | Jones  | 4         |
| 6  | Prof  | Finkle | 5         |
| 7  | Mr    | Mann   | 6         |
| 8  | Dr    | Love   | 2         |

| id | name             | qualification |
|----|------------------|---------------|
| 1  | Computer Science | BSc           |
| 2  | Mathematics      | BSc           |
| 3  | Physics          | MSc           |
| 4  | Chemistry        | Diploma       |
| 5  | Music            | Diploma       |
| 6  | Electronics      | City & Guilds |

- ▶ Deleting Electronics would leave rows 1 and 7 orphaned
- ▶ Deleting electronics then re-adding would require all course id of value 6 to be changed to 7
- ▶ Imagine a database with millions of people, with many relationships - I've seen systems with over 100 tables joined. Finding the orphans and fixing them can be very tricky.
- ▶ Alternatively, if you really do want to delete Electronics as you will no longer deliver it as a course, you need to remove all the child records too or maybe set them to null
- ▶ We can deal with these scenarios by setting referential integrity rules



- ▶ To provide referential integrity between course and tutor, you can add a constraint onto the keys. On delete, you can **cascade**, **restrict**, **set to null** or **do nothing**.
- ▶ Cascade means it will automatically delete all the children of a deleted parent. E.g., deleting a row in the course table would delete all tutor entries for that course so leaves no orphans
- ▶ Restrict means, if there are records in the child table with reference to the parent you are trying to delete, don't allow the delete and throw an error
- ▶ There are the same options associated with update. If you update a parent record, e.g., change the course primary key from 6 to 7 for some reason, the cascade constraint will automatically update all child records with key value 6 to 7. If it is set to restricted, then trying to update a parent record will raise an error
- ▶ Alter the tutor (child) table | foreign keys | select key. Referenced column is id. By default, these are restricted

| Foreign Key Name | Referenced Table      | Column                                        | Referenced Column | Foreign Key Options        |
|------------------|-----------------------|-----------------------------------------------|-------------------|----------------------------|
| course_id        | `student_db`.`course` | <input type="checkbox"/> id                   |                   | On Update: <b>RESTRICT</b> |
|                  |                       | <input type="checkbox"/> title                |                   | On Delete: <b>RESTRICT</b> |
|                  |                       | <input type="checkbox"/> name                 |                   | - Foreign Key Comment      |
|                  |                       | <input checked="" type="checkbox"/> course_id | id                |                            |



# Separation of Concerns

# SQL Injection

- ▶ So the database has various protections but what about our code?
- ▶ SQL injection is where typically an application requires data from a user to say, look something up
- ▶ For example, an API where we want to get details on someone with a last name of martin
  - ▶ `localhost:3000/name/martin` The name would typically be read in the web UI  
[/demos/data/mysql2\\_basic\\_promise\\_sql\\_injection](/demos/data/mysql2_basic_promise_sql_injection)
  - ▶ But what if we passed `localhost:3000/name/`' OR 1=1; -- ``
- ▶ If we can determine column names we can modify content - e.g. give ourselves a 1<sup>st</sup> class degree
  - ▶ `localhost:3000/name/`; INSERT INTO names (first_name,last_name,age) VALUES ('tony','nicol',25);--``
- ▶ How do we protect against this?
  - ▶ Database configuration - `multipleStatements: false` (default in MySQL2)
  - ▶ Parameterised queries
  - ▶ Thorough code checking of what's passes in

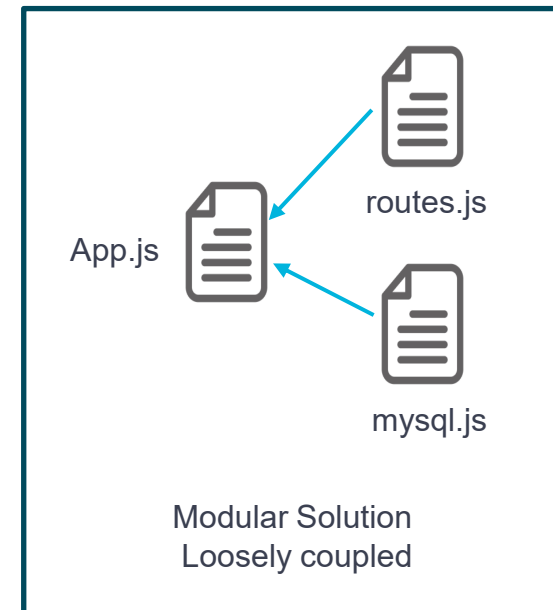
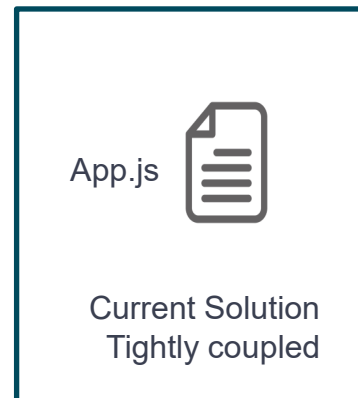
# Parameterised queries

- ▶ The previous example combines code and data; we need to separate these
- ▶ We need to make sure user input is treated only as data and is not executable SQL
- ▶ Data passed in for query is replaced with a placeholder, '?' and the data is added to an array
- ▶ The query with ? placeholders is passed as arg1 and data array as arg2 - the function safely combines
- ▶ Say we pass in 3 params in the path or in a POST body
  - ▶ `localhost:3000/student/billy/wibble/25`
  - ▶ `const params = [req.params.fName, req.params.lName, req.params.age]`
  - ▶ `const sql = "SELECT * FROM names WHERE first_name=? AND last_name=? AND age =?"`
  - ▶ `const [result] = await connection.execute(sql, params)`
- ▶ Note: **.query** works ok but **.execute** is preferred for security and query performance with ? params

[/demos/data/mysql2\\_basic\\_promise\\_sql\\_injection](/demos/data/mysql2_basic_promise_sql_injection)

# Separation of concerns

- ▶ Means designing a solution such that different parts of the code or architecture handle **distinct responsibilities**, without overlapping or interfering with each other
  - ▶ Each microservice/module/class/function has a single, well-defined purpose
  - ▶ Changes in one concern don't impact the others
  - ▶ Code is more maintainable - easier to test and extend

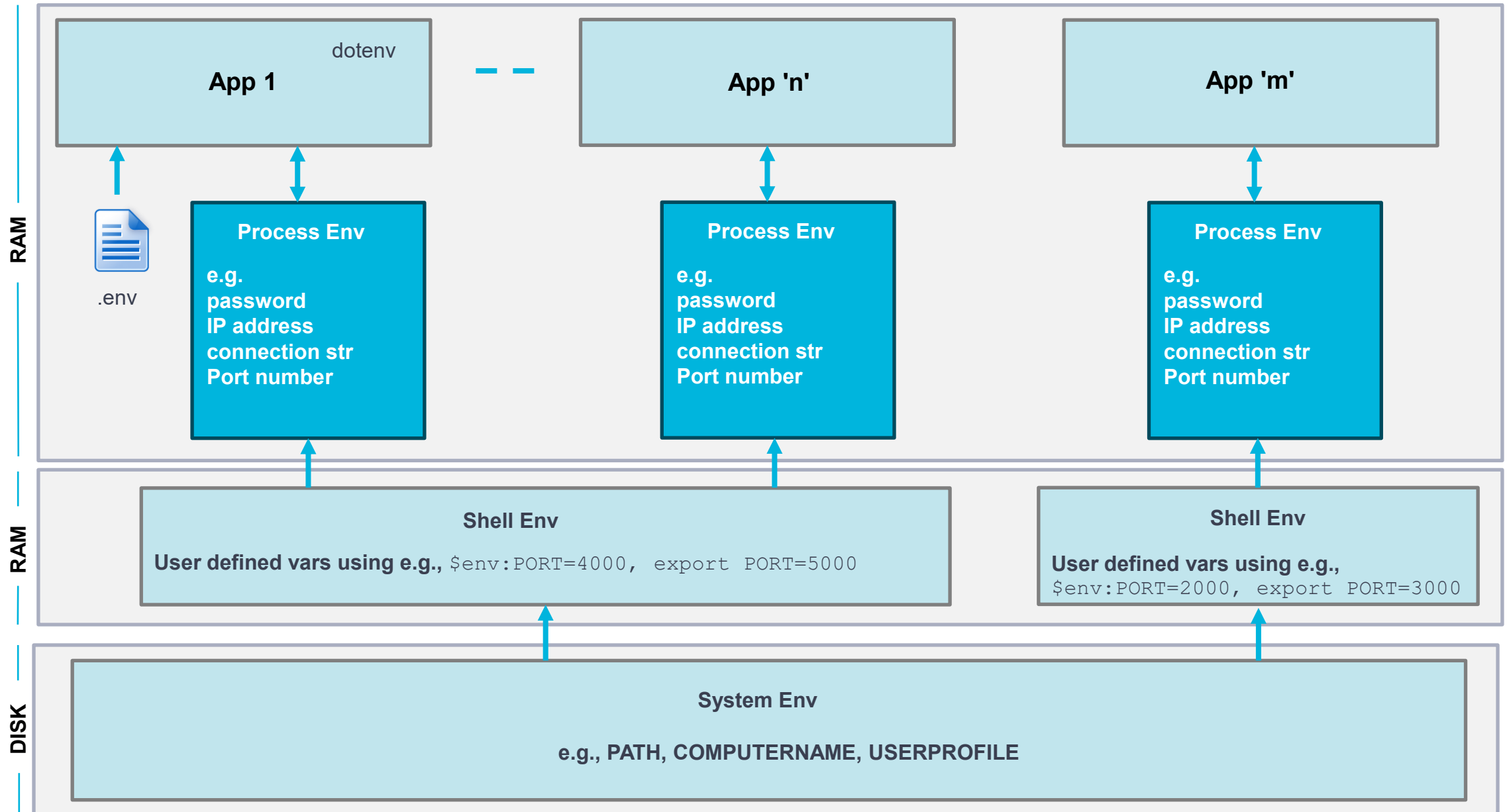




# Environment variables

- ▶ Before we get into modular database-application design, a quick sidetrack on environment variables
- ▶ Environment variables are variables held by the operating system so can be made accessible to applications. For example, the system scope env vars such as PATH in Windows, holds the paths to applications so you can access applications from any directory
- ▶ We can set env vars that are scoped to a process - i.e. a running app. These are not visible to other processes
- ▶ Our process (e.g. node app) can access our process scoped env vars **and** the inherited shell and system env vars
- ▶ We use environment variables to keep some of the sensitive or common settings out of our source files
- ▶ We can also use them to configure our application for different circumstances and environments at runtime
- ▶ They act like parameters to an app for flexibility - not sent to the app but available for the app to read
  - ▶ e.g. start server on a different port without changing code
- ▶ We can set env vars manually (don't leave space either side of =)
  - ▶ Windows power shell: `$env:PORT=4000` View with `$env:PORT`
  - ▶ Windows command shell: `set PORT=4000` view with `set PORT`
  - ▶ Linux: `export PORT=4000` View with `echo $PORT`
  - ▶ `const port = process.env.PORT` will read PORT from the environment; use it to start server on 4000

# Basic Env Var Architecture



- ▶ Rather than set the environment variables manually, we can use an npm package **dotenv**
- ▶ Write your environment variables in a file - default is `.env`. Include the npm package 'dotenv'. When the dotenv config function is called, it will read the variables from the file and write them into the process environment

```
require('dotenv').config()
```
- ▶ Create a file called `.env` in the root dir of your project.
- ▶ If you want to put the file somewhere else or you are having trouble opening it, use the following to make it explicit:

```
require('dotenv').config({ path: path.resolve(__dirname, '.env') })
```
- ▶ Using this approach you can call the file something else if you want to.
  - ▶ Note: path needs to be "required" but doesn't need npm as it is already included in modern versions of node
- ▶ To see all your env vars:
  - ▶ Windows powershell: `Get-ChildItem Env: or Get-ChildItem Env: | Sort-Object Name` to sort them on name
  - ▶ Windows command shell: `set`
  - ▶ Linux: `env` or `env | sort`
- ▶ Using `.env` to hold passwords is a little more secure than adding them in the code. If you are going to push to github, don't commit your `.env` file. To prevent that, add `.env` to a `.gitignore` file (add `./node_modules` too)
- ▶ If you already have an env var declared in your env, dotenv won't overwrite it - i.e. a shell var trumps a `.env` var

# Database connections

- ▶ When a query from the app is made on the database, typically in the simplest non-busy scenario
  - ▶ a connection object is created
  - ▶ the app authenticates using the connection string
  - ▶ the query is completed
  - ▶ the app closes the connection
  - ▶ improves security by minimising session hijacking or unauthorised reuse in very secure environments - limited time for hacker
  - ▶ may use per-user credentials for isolation and auditing
- ▶ Alternatively, a connection object can be created and kept open until the application closes
  - ▶ only take the object creation and authentication hit once so improves performance
  - ▶ slightly less secure but fine for all but the most sensitive data scenarios
  - ▶ cannot run concurrent queries needed in a multi-user environment - e.g. concurrent http requests
  - ▶ or even multiple queries from the same user to say restore a user profile when they login to the web app

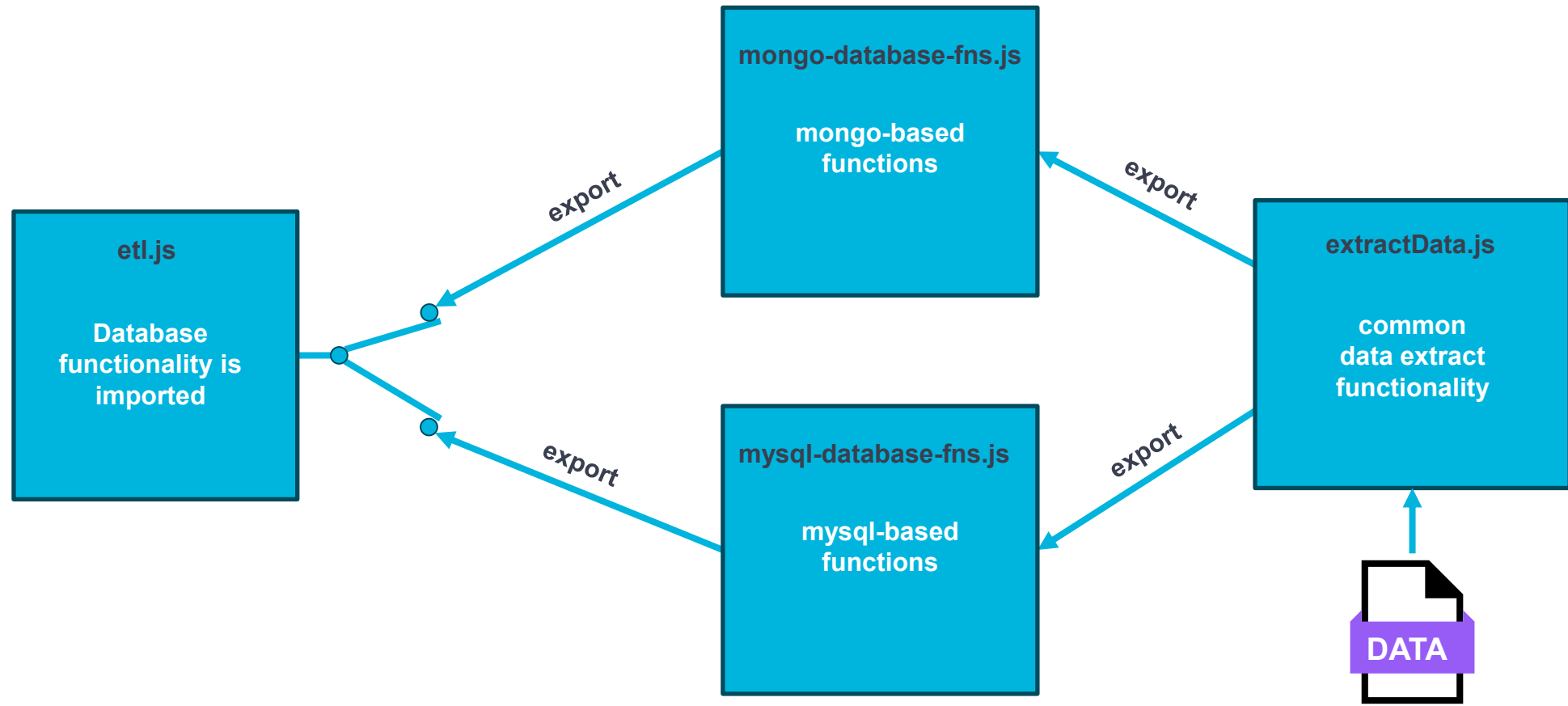
# Database Connection Pooling

- ▶ Connection pooling is a technique used to manage and reuse database connections efficiently
  - ▶ Especially in high-performance or multi-user applications
  - ▶ Avoids the time overhead of opening and closing a new connection for every database request
  - ▶ Creates a pool of reusable connections when the application starts and authenticates them all
  - ▶ Hands out an available connection from the pool when a query is made & returns it on query completion
  - ▶ Keeps connections alive and ready for reuse, reducing overhead of creating and authenticating
  - ▶ Supports concurrent users and concurrent queries from the same user
  - ▶ Can prevent connection leaks by using auto lifecycle management - `pool.query()` gets, uses then releases connection
- ▶ Can set limits. e.g., for `mysql2`
  - ▶ `connectionLimit`: limit concurrent connections to prevent resource exhaustion such as CPU, RAM, I/O. Default=10
  - ▶ `waitForConnections`: should pool exhaustion cause connection requests to be queued. Default=true
  - ▶ `queueLimit`: if queuing, how big do we allow the queue to grow? Default=0 which means unlimited

## Data engineering use-case

- ▶ I have a pdf document with trivia questions and answers. It's formatted with headers, footers, page numbers etc. I want to extract the data, transform it into a format I need to be able to load it into a database which I have modelled
- ▶ Step1: I exported into a word document so I could edit it
- ▶ Step2: I needed to clean the data - replace smart quotes, en & em dashes, other non standard characters into standard characters. Remove headers, footers, section breaks etc. This is automated using word's search and replace feature
- ▶ Step3: Data were exported into delimited text: tab to separates a question from an answer; newline to separate each question-and-answer pair. Database special characters need to be dealt with too - e.g., " and ' are used in databases to separate data but the data may contain " or '. So, all " need to be converted to either "" or \" and ' to " or \'. There are others. You could use backtick to avoid the change
- ▶ Step 4: **Extract**. Data is loaded from storage, e.g. a text file which is typically coma separated values (csv) or delineated in other ways (tab and eol in my case), maybe the format is json. Whatever, it needs to be extracted to some form of staging area so the original is not modified
- ▶ Step 5: **Transform**. Any more formatting and cleaning to get the data into a form that maps to our data table design
- ▶ Step 6: **Load**. Data is loaded into one or more tables in a database
- ▶ In this simple example, the database has only one table with three columns: id, question and answer
- ▶ The ETL is written in modular JavaScript. It loads data from a file, separates question and answer, creates a table then loads all the data from the file into the table.
- ▶ A use case could be: datafile lands. app.js triggered to do etl. Data file archived. Data available for API request

- ▶ The modular structure enables the database layer to be changed without changing the app





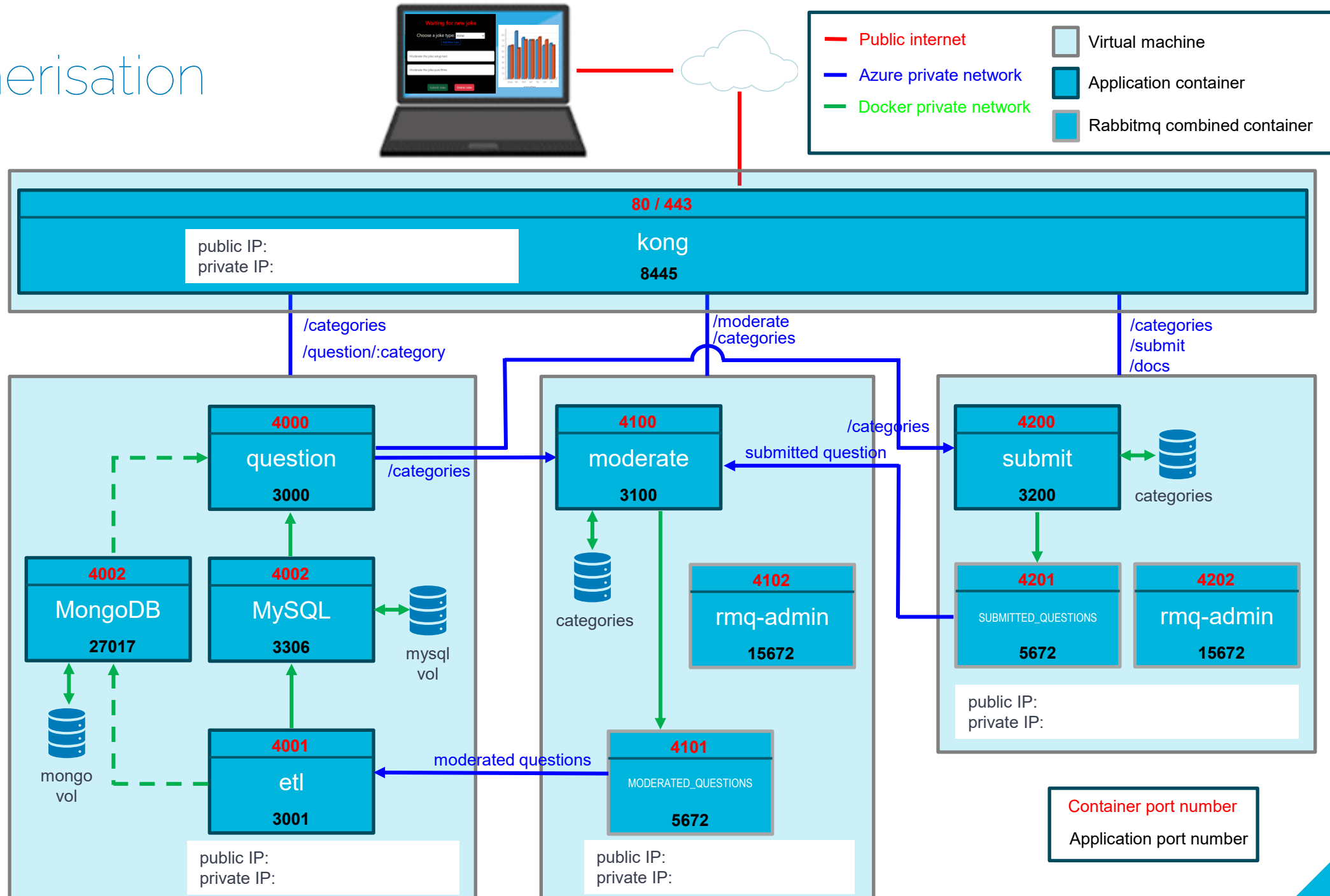
# Containerisation



# Learning objectives

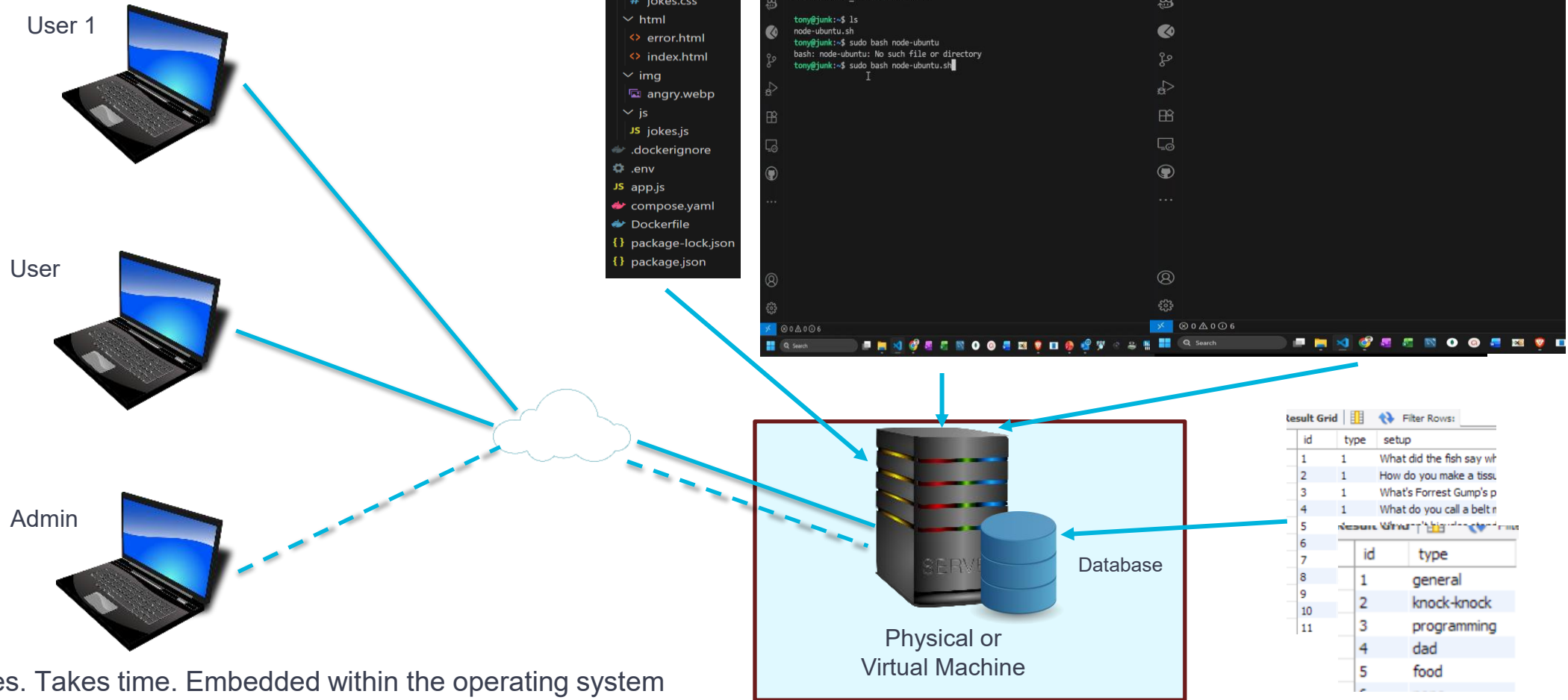
- ▶ On completion of this section you will be able to:
- ▶ explain the high level architecture of a container
- ▶ explain the concept, benefits and limitations of containers
- ▶ explain the container image creation process using a Docker file
- ▶ create a docker image of your application
- ▶ create, run and test a local container
- ▶ create and run a local container that interacts with a local instance of MySQL server
- ▶ use common docker commands
- ▶ use the Docker desktop application
- ▶ use Docker Compose to deploy multiple containers

# Containerisation



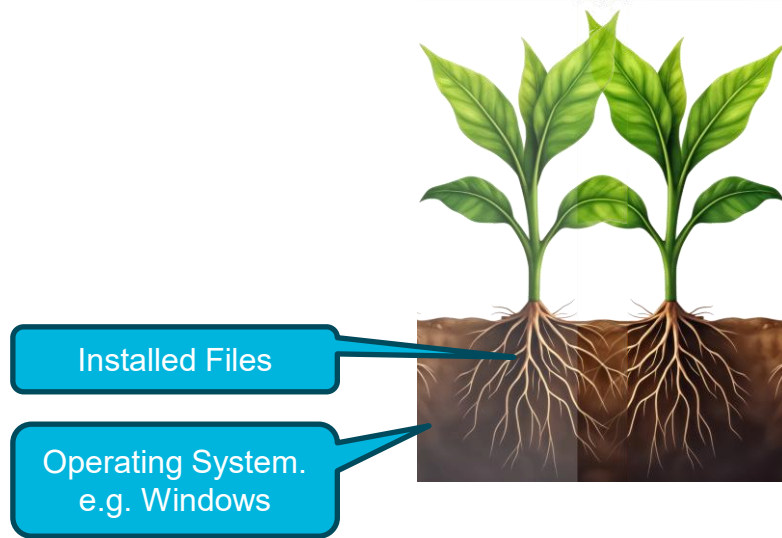
- 
- ▶ It is perfectly reasonable to install applications on your PC or server
    - ▶ e.g. install nodeJs, put your app on there, install MySql server - as you have on your client
  - ▶ In larger organisations, a more modern approach is to use containers so we will focus on that
  - ▶ We will deploy containers on the local machine, then onto a remote machine
  - ▶ The main tools we will use for containerisation are Docker, Docker Compose, and Kubernetes.
  - ▶ Let's compare a container to an application system installation

# Installing files on the remote

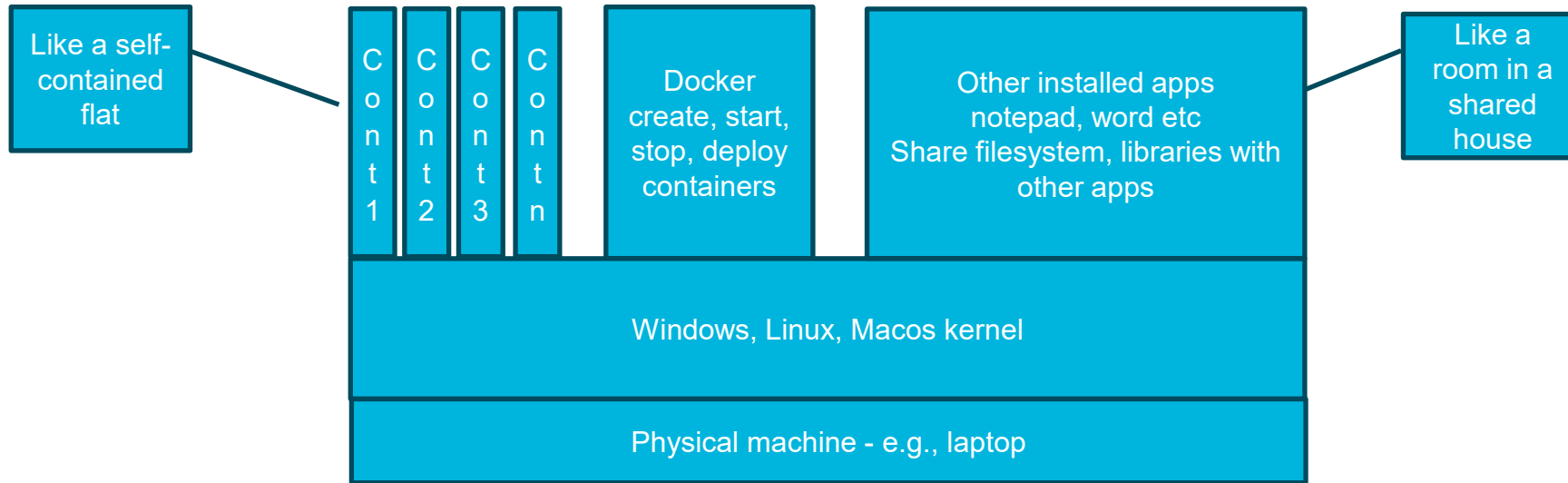


- ▶ Many files. Takes time. Embedded within the operating system
- ▶ Each VM build requires all this to be done again
- ▶ Database needs to be configured and populated
- ▶ Modern enterprise systems tend to use containers

# What are containers?



# Containers on a physical machine



- ▶ We need Docker installed to create and run and manage containers
- ▶ They have their own filesystem within the container
- ▶ Containers are self-contained and isolated from each other and other apps, and contain all app dependencies
- ▶ The container app uses the OS kernel for resource management similar to installed apps so kernels need to match
  - ▶ Linux containers work on most Linux distros, Ubuntu, Red Hat, Centos etc as the kernel is the same and not version sensitive
  - ▶ This is not the case for Windows. Versions need to match. They are not popular. Modern .net available on Linux
- ▶ Note: the hardware needs to be the same. e.g. both need to be X86 or maybe RISC if using mac
- ▶ So, we can test locally then deploy to a remote machine with a good level of confidence
  - ▶ Less of the "Well it worked on my machine when I tested it but ..."

# Docker image

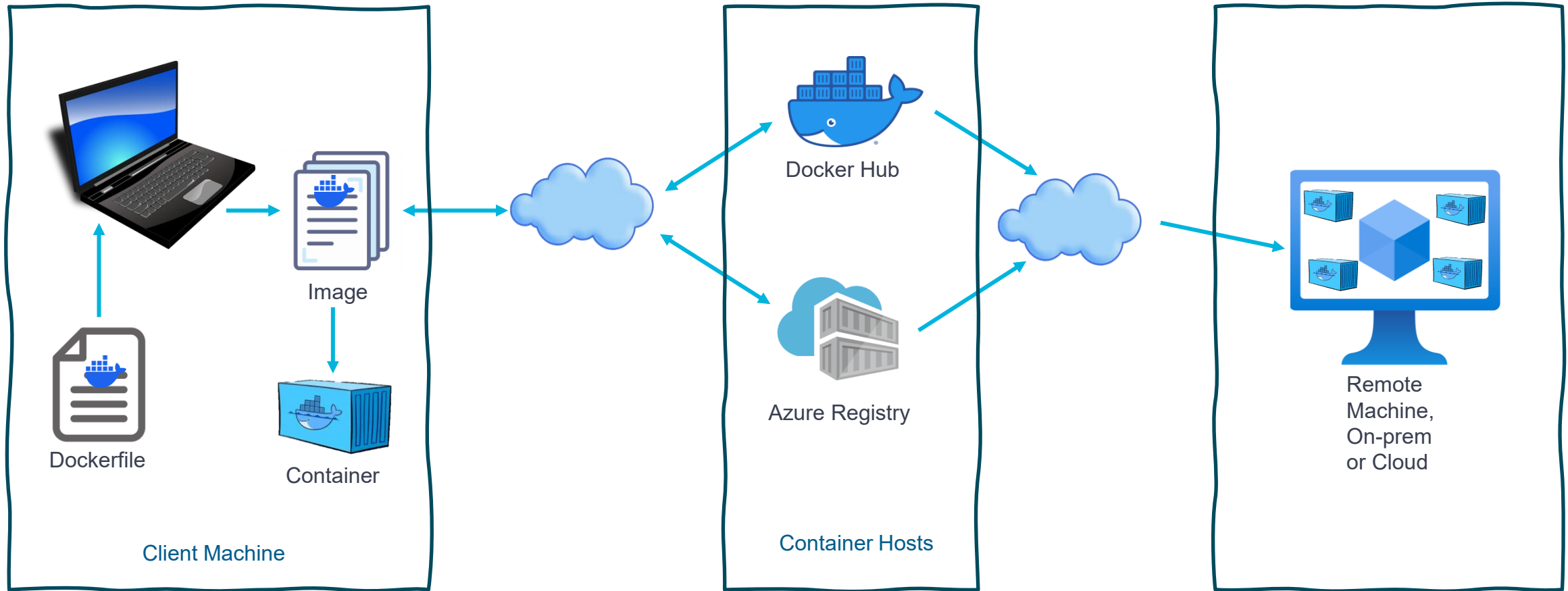
- ▶ A Docker image is a file containing multiple elements needed for an application to run, similar to a system snapshot
- ▶ Let's say we have only NodeJS installed on a machine and we take a snapshot of it; this would become a base layer for your image. That's not how it works but can be visualized. Node is built into an image using Docker
- ▶ There are other base images such as MySQL server, MongoDB, Nginx and various other core application people have built into a Docker image and made available for download
- ▶ Some images are free to use, some need to be paid for, some are maintained and some are not. Thousands are available on Docker hub
- ▶ You can take a base image such as NodeJs and add layers on top of it - i.e. your app and dependencies. A file called Dockerfile holds instructions on what to add into each new layer. Most instructions in a Dockerfile will create an additional layer on top of the image. Each layer is given a unique id
- ▶ Once we have added layers to the image such as any extra node packages, source code, config files or whatever we have that makes up our app and its dependencies, a new image which is the combination of those files is created
- ▶ From the image, Docker can build a container which is an isolated execution environment for our application

# Prerequisites

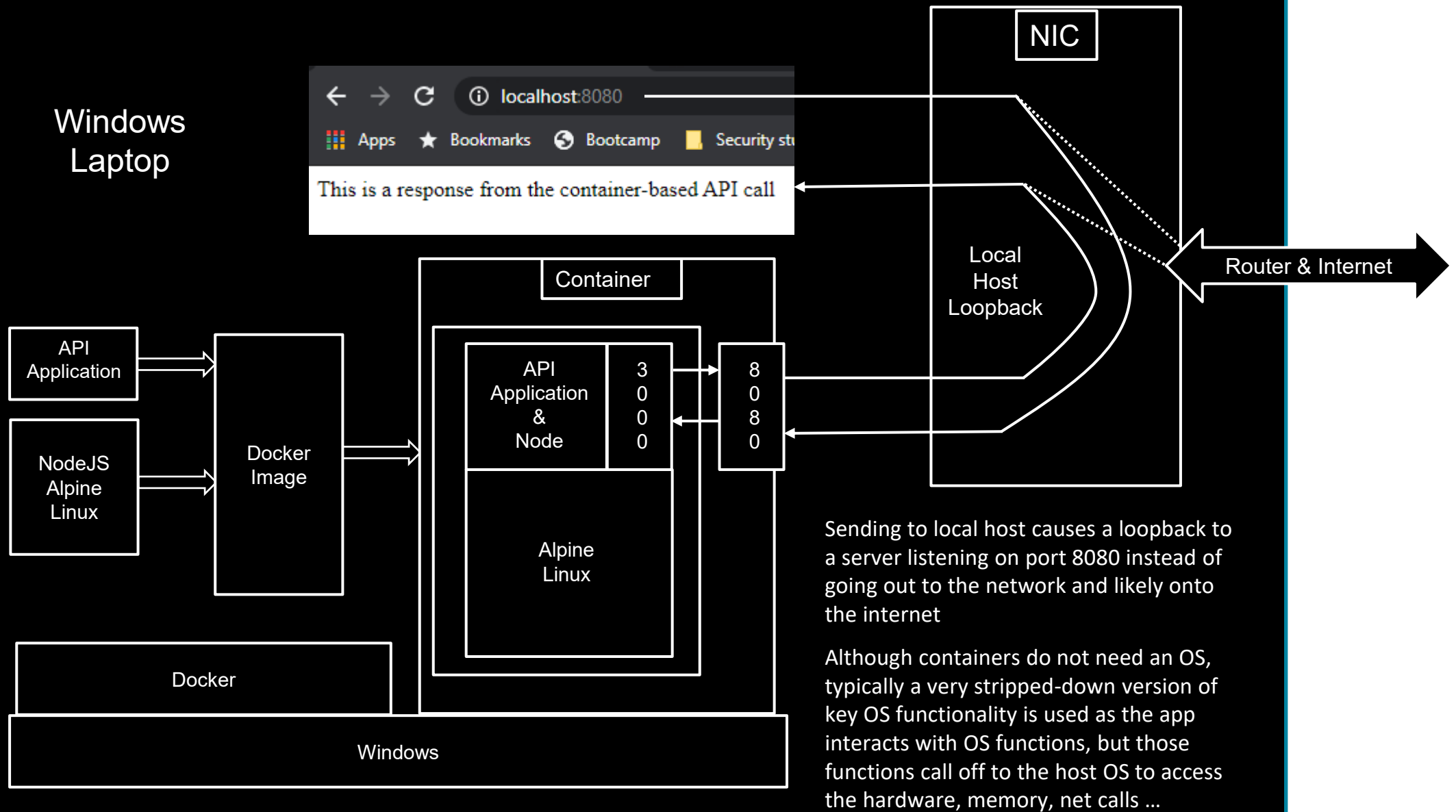
- ▶ Docker was developed for Linux but installs on most systems. WSL is faster than Hyper-V so enable it
- ▶ Make sure you don't have the old version installed
  - ▶ Open Turn Windows features on or off
  - ▶ Look for wsl. If it is checked, uncheck it
- ▶ To enable new WSL (Windows Subsystem for Linux) open powershell in admin mode and enter
  - ▶ **wsl --install**, reboot when done
  - ▶ Check it installed: **wsl --version**
  - ▶ If you need to update it at a later date use: **wsl --update**
  - ▶ Restart Vscode and you should have a Ubuntu wsl terminal option if you need Linux tools or prefer Linux
- ▶ Download Docker desktop from <https://docs.docker.com/get-docker/>
- ▶ Docker desktop is a GUI to interact with Docker instead of the CLI
  - ▶ CLI is still available and the docker engine is installed but containers can be started, stopped, inspected etc from the desktop
  - ▶ We won't have Docker desktop on the remote machine; we will use the CLI



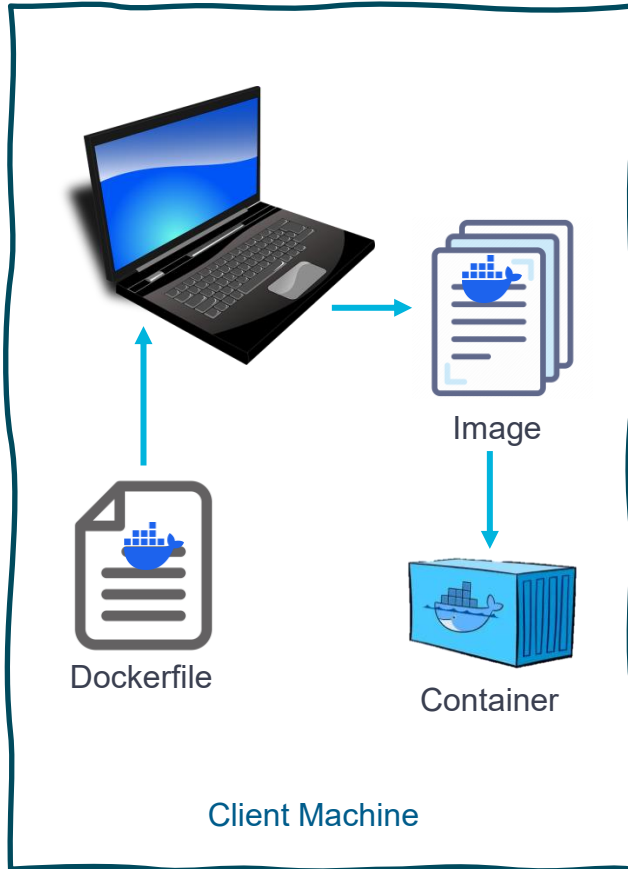
# Docker build and deploy process



## Windows Laptop



# Initial dev and test



- ▶ Write the Dockerfile to provide build instructions
- ▶ Docker reads the file and creates the image
- ▶ Docker loads this image into a container to execute the code

# Process

- ▶ Create a **Dockerfile** no extension - Note: # is a comment

```
FROM node:alpine # Base image
```

```
WORKDIR /usr/app # The directory in the container OS filesystem. The RUN command is executed in the WORKDIR
```

```
COPY ./package*.json . # COPY all package files (note the ".")
```

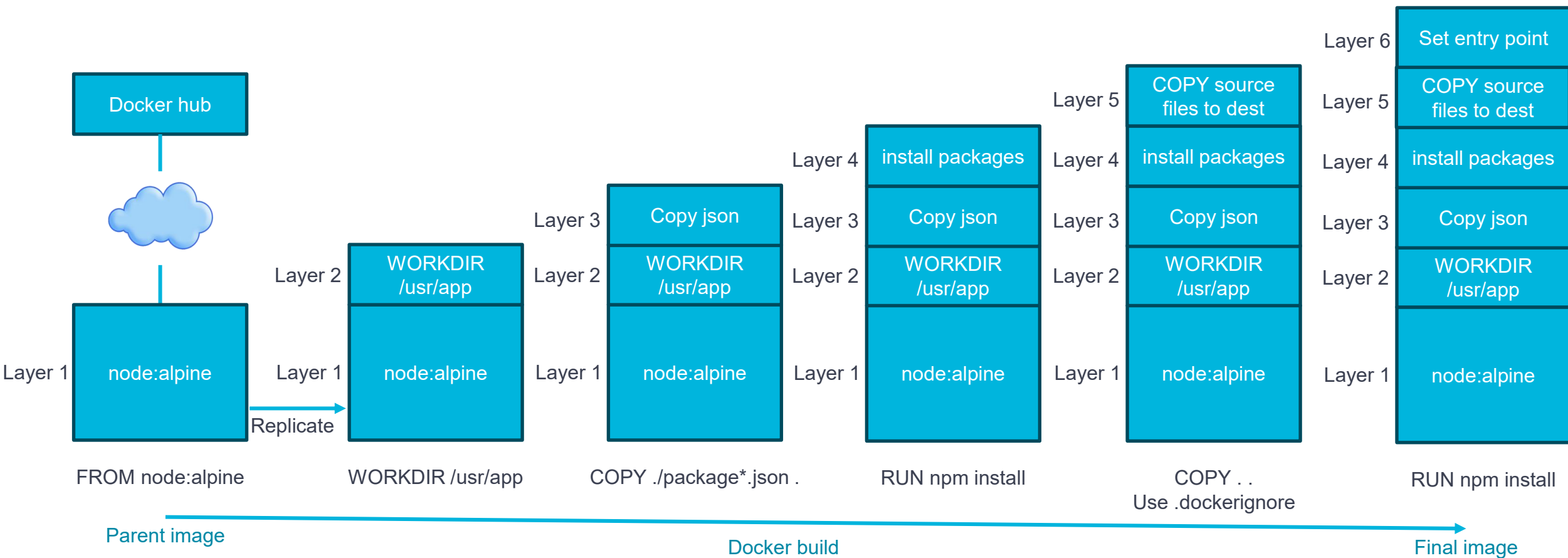
```
RUN npm install # npm will be run from WORKDIR to install dependencies in the package.json into the image
```

```
COPY . . # COPY "." all files in the current directory into WORKDIR except for any in .dockerignore
```

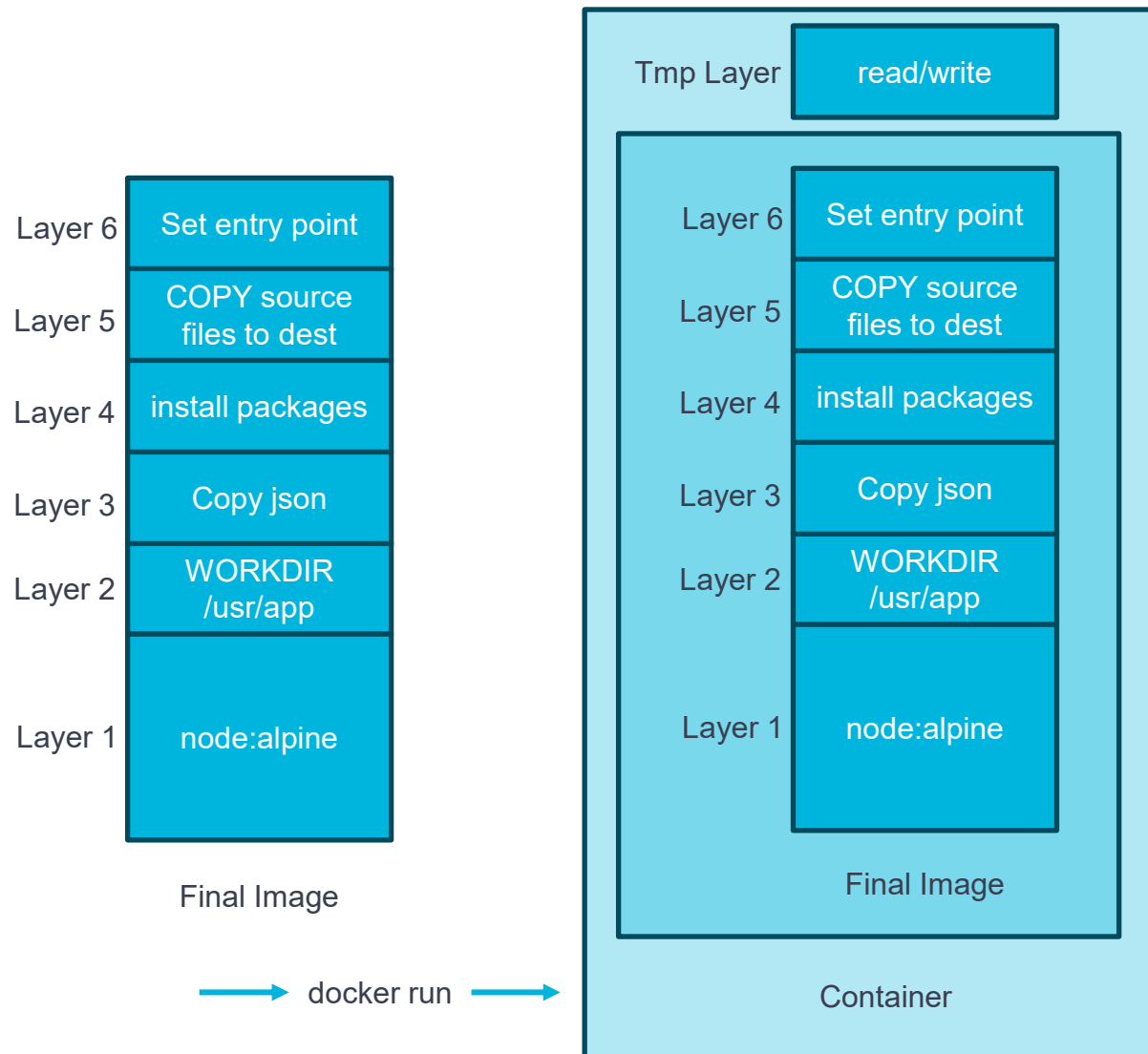
```
CMD ["node", "app.js"] # start app.js when container starts
```

# Build the image

- With reference to the six instructions in the Dockerfile, as each one is executed during the build process, a new layer is created. `.dockerignore` can exclude files from the copy process such as the `node_modules`



# Create and run the container



- ▶ Container has everything it needs to run on OS
- ▶ All layers are immutable
- ▶ Separate read / write layer added in the container to change data, e.g. delete a file, add a file, save some data
- ▶ When the container is recreated, data persists
- ▶ When the container is deleted, the tmp layer is deleted as it was created within the container and is not added to the image the container was built from
- ▶ This can be a problem if the container has a database
- ▶ To persist data outside the container, Docker provided volumes (later)

# Build the image

- ▶ **docker build .** (Make sure Docker desktop is running) Will build the image and include all files in the current directory (note the '.'). The . means find the Docker file in the current directory
- ▶ List the images with **docker images**
- ▶ This is fine and the image is given an image ID which is the first 12 bytes of the sha 256 hash of the image – i.e. not very user friendly so use the **-t** option to add a tag, i.e. a name to the image repository
- ▶ **docker build -t name:tag .** Will build a docker image with node, alpine and our app from which a container can be created
- ▶ A tag will use the first part, name, as the repository name. The second part is used as an information tag – typically a version number
- ▶ Image name is repo-name:tag-value. E.g. **docker build -t demo:v1 .** would create and name the image **demo:v1**

- ▶ `.dockerignore` lists files we **don't** want the `COPY . .` command to add to the image
- ▶ There is no need to add `.dockerignore` to the `.dockerignore` file, as docker knows not to copy it
- ▶ Ignoring the Dockerfile is optional. Add it if you want others to be able to look at it in the image to see how it was built or don't add it if you don't want them to see it or there are sensitive details in it
- ▶ We should add `node_modules` to `dockerignore` as we don't want to copy our local copy as they will only be overwritten when executing `RUN npm install` so is pointless and inefficient
- ▶ If you are using git or have other files you don't want to bloat the image unnecessarily, then add those to `.dockerignore`



# Create and run app in local container

- ▶ Note: Docker have changed the syntax for some cli commands. The container commands are affected. Any container commands are now docker container sub-command. e.g. **docker run** becomes **docker container run**. Not yet enforced so I'll use the simpler
- ▶ To create a container, run and make accessible on a network, you specify an image name and port mapping :
  - ▶ `docker container run -d -p 4000:3000 --name test --rm demo:v1` OR
  - ▶ `docker run -d -p 4000:3000 --name test --rm demo` is shorter
- ▶ -d means run in detached mode. i.e. in the background – otherwise the command line becomes unusable - but provides output
- ▶ -p means map ports. What port do you want the container to listen on and which port is your app listening on. For example, -p 8080:3000 would set the container listening on port 8080 and would forward to 3000 for your app
- ▶ If you only have one container and one app, you may just use -p 3000:3000 or whatever, as long as the port is free
- ▶ A container is a running instance of an image. You could run the same image in multiple containers:
  - ▶ `docker run -d -p 4000:3000 --name cont1 --rm demo`
  - ▶ `docker run -d -p 4001:3000 --rm demo`
  - ▶ Use `localhost:4000` to access the first, **localhost:4001** for the second. Both are listening on 3000 inside their container
- ▶ **--name** allows you to uniquely name the container rather than use the obscure Docker generated name such as frosty\_northcut
- ▶ **--rm** means remove the container (delete it) once it's stopped otherwise you need to do it as a separate step. This avoids building up loads of containers or having to manually delete them

# Useful Docker Commands

- ▶ **docker image build -t name:tag .** to create an image based on the Dockerfile script (was docker build)
- ▶ **docker images** will list your images
- ▶ **docker image history --no-trunc image\_name > image\_history.txt** shows all the layers (available in desktop now)
- ▶ **docker image rm** name or image id to delete an image (was docker rmi) - or use desktop
- ▶ **docker container run -d -p ContPort:AppPort --name ContName --rm ImgName:tag** (was docker run)
  - ▶ `docker container run -d -p 8080:3000 --rm --name test-cont test:1.0.0`
- ▶ **docker container run -it image\_name sh** if image has OS shell, enable access the contents (available in desktop)
- ▶ **docker container ps** will list your running containers
- ▶ **docker container ps -a** will list all containers including those stopped but not deleted
- ▶ **docker container stop name** will stop a container using the name if you gave it one using -- name
  - ▶ It will auto delete the container and delete the image if you used the **--rm** option when you ran it
- ▶ **docker container stop container\_id** use this if it's easier
  - ▶ If you didn't name the container, you can use the first few chars of its image ID e.g. **docker stop 9db** if say the image id was 9dbcd7be14d and there wasn't another starting 9db. Saves typing the container name
- ▶ **docker inspect <container>** shows the environment injected into the container

- ▶ **docker image tag OldName:tag NewName:tag** will change the name of the image
- ▶ **docker container rm [options] container** to delete a container. If it won't stop, use -f as an option to force it
- ▶ **docker push** repo-name/image:tag e.g. `docker push reg-name.azurecr.io/image:v1`
- ▶ **docker pull** repo-name/image:tag e.g. `docker pull dockerhub_name/image:tag`
- ▶ **docker image prune** deletes all dangling images - i.e. they have no container association
- ▶ **docker image prune -a** delete all images including those associated to a container as long as container is stopped
- ▶ **docker system prune** delete stopped containers, unused networks, dangling images & unused build cache
- ▶ **docker system prune -a** as above but also images linked to a stopped container
- ▶ **docker system prune -a --volumes** this will delete as above but also volumes and anything created by docker
- ▶ **docker volume ls** lists all volumes and their drivers
- ▶ **docker volume ls -q** list all volumes but just their names. -q = quiet
- ▶ **docker volume rm vol\_name** deletes the specified volume
- ▶ **docker volume rm \$(docker volume ls -q)** deletes the list of volumes - i.e. all of them. \$( ) expands to list for rm
- ▶ **docker network ls** to list all the networks
- ▶ **docker network connect network container** e.g. **docker network connect yin\_default yang** connect yin default net to container called yang in different docker network created by compose on same machine - e.g. local machine



# Deploy multiple containers

- ▶ Containers can be built, pulled and networked using Docker, but Docker provides a tool called Docker Compose
- ▶ We create a definition of the containers we need in a compose file (written in YAML), which points to the Dockerfile of each image to be built. Compose builds containers and creates a network between them so they can communicate
- ▶ As we will generally be using multiple containers for each microservice, we will focus on Docker Compose
  - ▶ we still need to understand Docker commands as the two tools work together
- ▶ I'll demo a simple example of running two of our demo containers
- ▶ Even if you only have one container you may still prefer this approach as there is less command line typing
  - ▶ The stuff we write on the command line for Docker can go into a YAML config file for Compose

# YAML

- ▶ Originally defined as “Yet Another Markup Language”, but as the point of it is to avoid markup such as XML start and end tags and JSON {}, it was renamed as a recursive name: “YAML Aint Markup Language”
- ▶ YAML is similar to JSON but is claimed to be more readable as it used indentation rather than delineation through {}
  - ▶ i.e. a bit like python is supposed to be more readable than "real" languages

## Comments

# starts a comment

## Name:Value pairs

name: tony-nicol  
description: YAML-example

## Object

connection:  
 port: 9000  
 app: my-app

## List / Array

list-name:

- url: http://1235.com
- url: http://1236.com
- url: http://1236.com

## List / Array of objects. An array element can contain an object

services:

- app: my-app
  - port: 9000
  - url: http://1234.com
- app: my-app2
  - port: 3000
  - urls:
    - url: http://1235.com # - can be aligned to parent or indented
    - url: http://1236.com # Whichever you choose they all need to be the same
    - url: http://1236.com # in that list

Lists with simple data members (not objects) can be written in two ways  
ports: [3000, 3001, 3002, 3003]

or

ports:

- 3000
- 3001
- 3002
- 3003

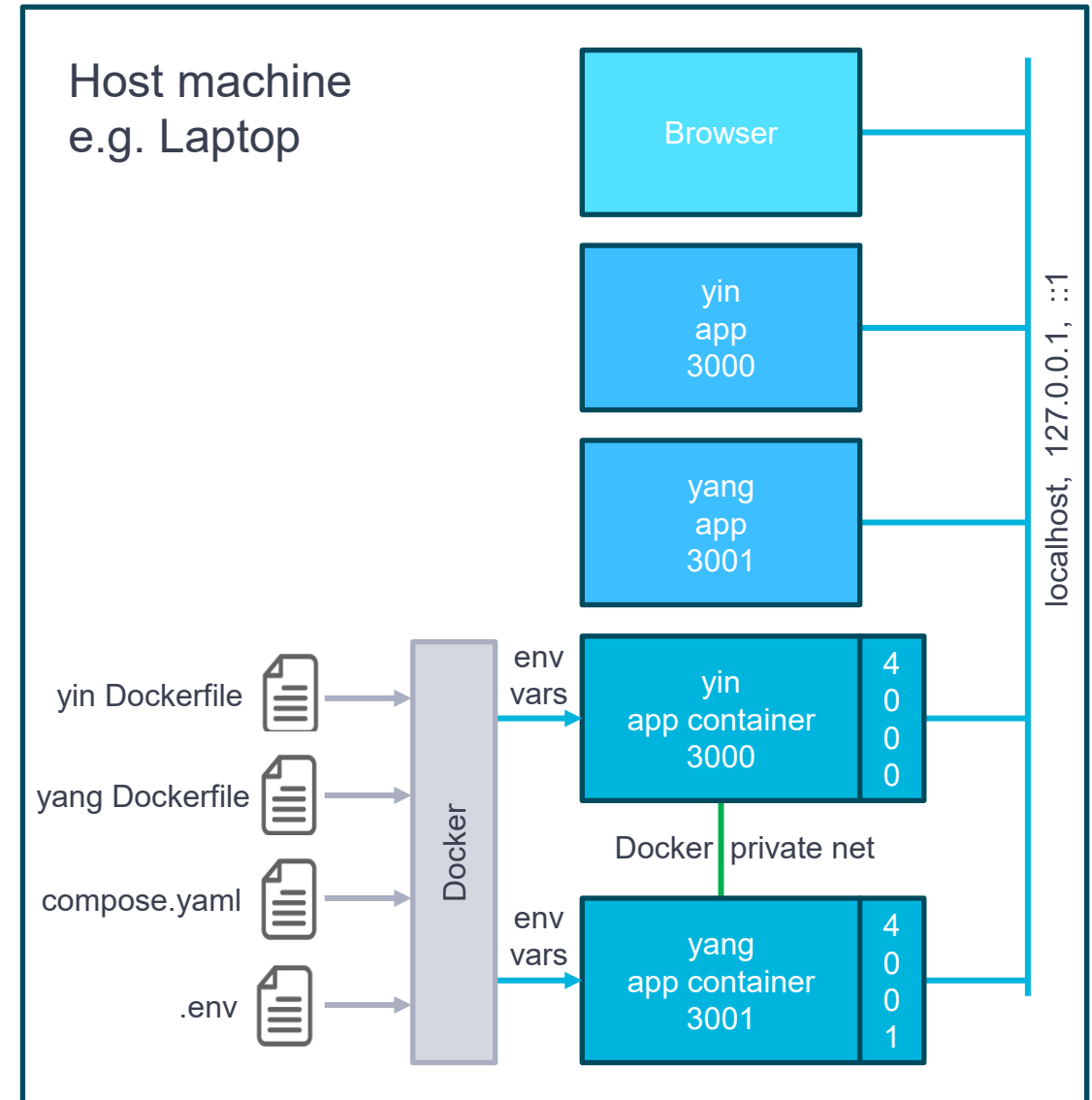
--- This effectively starts a new document within the same file



# Local app and container logical architecture

- ▶ yin and yang are two simple apps that synchronously communicate with each other just for illustration
- ▶ If containers are built in the same compose file, a default private network is created to connect them
- ▶ Everything communicates on localhost except inter-container comms which use private IP range
  - ▶ 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16
  - ▶ private IPs <-> host names are service names
  - ▶ port numbers are app port numbers
- ▶ Compose file references Dockerfiles for each image build instructions
- ▶ .env is referenced by compose and env vars sent to container process when started

[/Demos/docker/yinYang](#)





# Cloud Hosting Intro

# Objectives

- ▶ Explain the business and technical pros and cons of cloud computing
- ▶ Differentiate between the service models
- ▶ Explain the shared responsibility model
- ▶ Explain the difference between CapEx and OpEx
- ▶ Explain the concept of a virtual machine
- ▶ Compare and contrast a physical machine to a virtual machine
- ▶ Explain the differences between cloud deployment models
- ▶ Create a virtual machine on the cloud and deploy an application and runtime to it
- ▶ Use basic Linux commands
- ▶ Create and use an asymmetric key pair for an SSH session
- ▶ Create a container registry
- ▶ install Docker on a remote machine
- ▶ Push your image to a registry, pull the image and run a container on a remote machine

# Introduction into Cloud Hosting

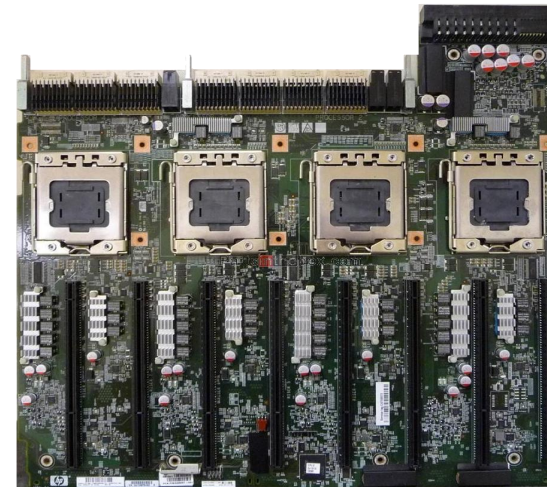
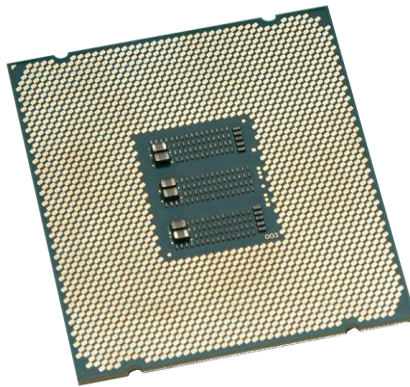
- ▶ You are now familiar with full stack development on your local machine. We now need to deploy remotely so potentially anyone in the world can access your application
- ▶ The backend can be implemented on a server in your home, in an office, in a data centre or in the cloud - as long as it is accessible from the internet
- ▶ Cloud hosting is just a load of data centres hosting lots of servers that you rent rather than buy
- ▶ An Azure and AWS data centre is generally referred to as an availability zone (AZ), there are exceptions
  - ▶ Google refers to them just as "zones"

# Cloud Business Model

- ▶ For an enterprise, you need a lot of compute and storage. You need premises with strong physical security, redundant power, high speed networking, racks of servers with redundant power and network cards, lots of storage for files and databases, infrastructure security and the list goes on. All this comes as a sunk cost referred to as Capital Expenditure. You pay for it then the assets depreciate over time, about 5 years. You need specialised staff to install, protect and maintain all this equipment
- ▶ If you want to scale your application, i.e. give it more CPU or Storage, you need to buy and install more stuff which can take a long time. Once you have scaled up, if the traffic for your service reduces, you are stuck with a load of kit that's hardly being used
- ▶ Amazon found themselves in this position and at times had a lot of spare capacity so why not rent it out? At least until you need it back. That idea grew into what we have today where the data centres are specifically used to rent these services
- ▶ This means businesses can replace CapEx with OpEx – i.e. operating costs which are service costs rather than depreciable asset costs. The rental model is basically pay for what you use. If you want more then pay more. If you want to scale down then do so and pay less
- ▶ The cost and hiring difficulties and permanent salary costs of all the specialist staff required to maintain and operate the infrastructure moves to cloud service provider so this is seen as a good business model
- ▶ Businesses can capitalise on economies of scale



# What does "the cloud" & "on-prem" look like?

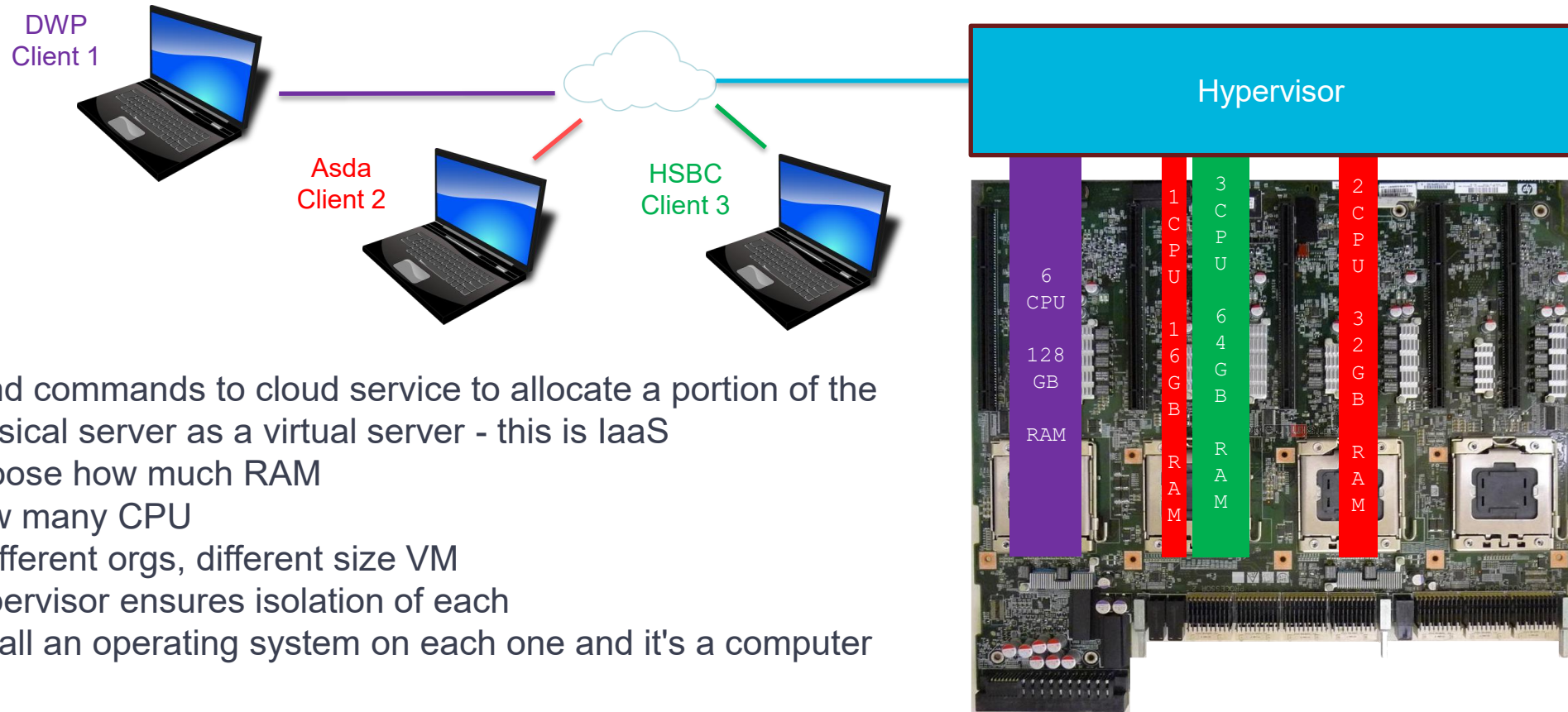


Intel Xeon (Xeon Sierra Forest CPU) 288 cores per chip - 1,152 in this server

# Virtual Machines

- ▶ It would be very inconvenient for the cloud provider if we wanted to rent 10 physical servers one day then reduce to 5 the next day. Having to re-route networking and security – pulling cables and plugging them in elsewhere is infeasible – but possible (and is one model)
- ▶ You may find that the extremely powerful physical machines are too big for your application, and you can't get the most out of them and a whole machine is expensive.
- ▶ Businesses want to be able to scale their infrastructure regularly so a software approach to infrastructure architecture is required so the cloud providers can allocate machines that have been released, for immediate use for someone else
- ▶ A virtual machine is a software emulation of a physical machine hence the name virtual machine
- ▶ A physical machine can host multiple virtual machines which to the user, look just like a physical machine and although there is some performance overhead, virtual machines are now very mature and performant

# What are: Virtual Machine, Public Cloud, IaaS?



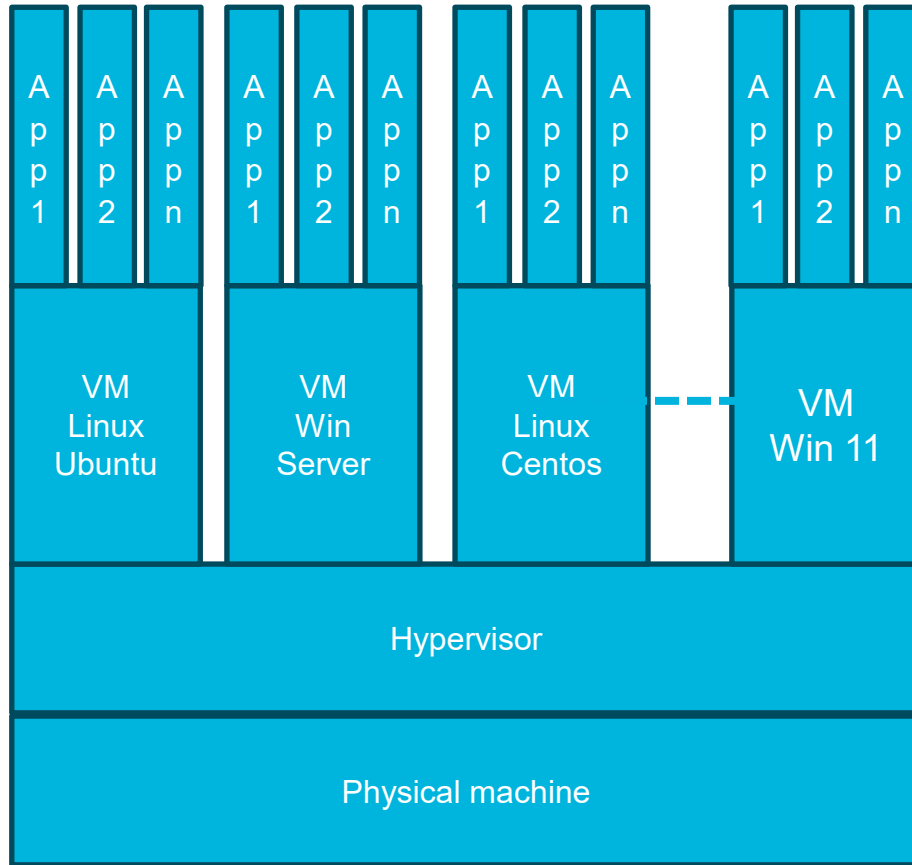
- Send commands to cloud service to allocate a portion of the physical server as a virtual server - this is IaaS
- Choose how much RAM
- How many CPU
- 3 different orgs, different size VM
- Hypervisor ensures isolation of each
- Install an operating system on each one and it's a computer

<https://azure.microsoft.com/en-us/Free>

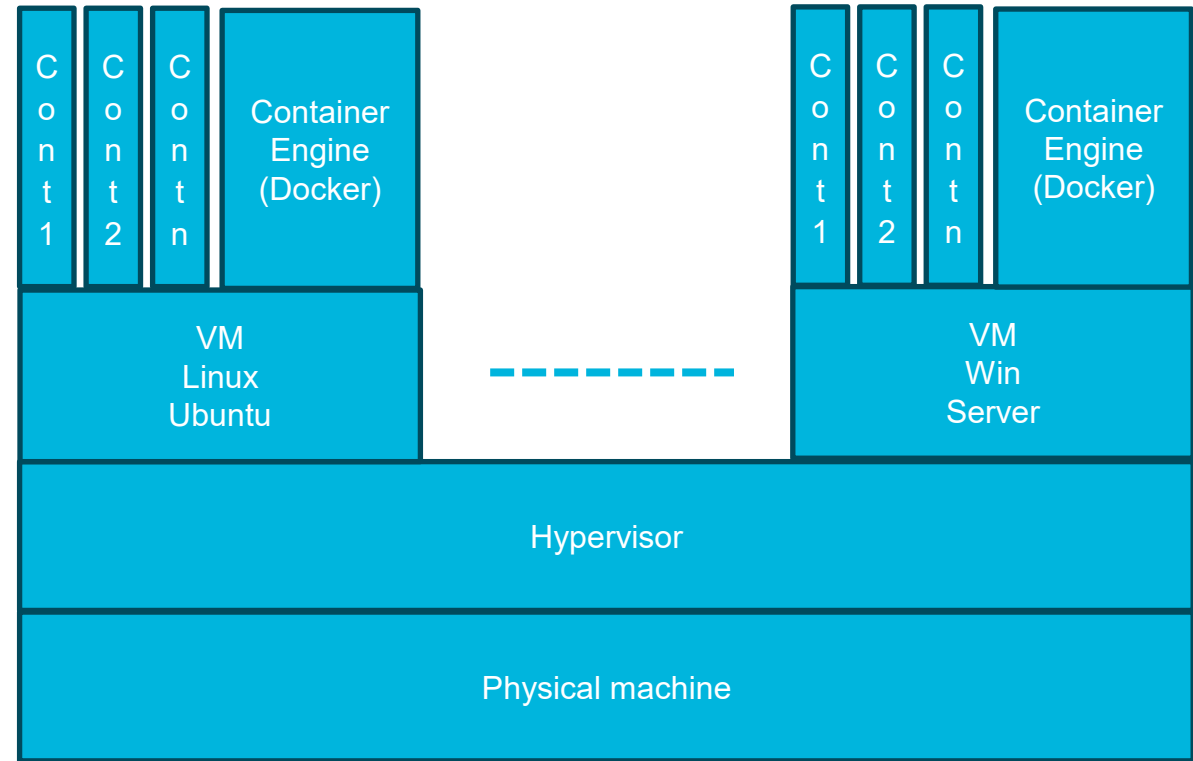
<https://aws.amazon.com/free>



# VM vs Container structure



Virtual Machines



Containers

- ▶ A VM is an isolated whole machine with its whole operating system and applications – like individual servers or PCs. They are isolated at the hypervisor level when using a type1 hypervisor as would be used in a data centre
- ▶ Because of the size of some of the operating systems – Win server 2025 requires 32GB of disk space (Ubuntu server is about 1.9GB), it can take minutes to load and start, and you often don't need most of the stuff in it
- ▶ If you only want to run a single app you have written, you don't need all the OS tools, hundreds of drivers and bloat
- ▶ You can write your program on a laptop say, then, using a tool such as Docker, create an image and run as a container
- ▶ A container engine, e.g. Docker, runs on a single shared OS and creates a runtime environment for the containers
- ▶ The container is isolated from other containers at a process level – they are only accessible via a network IP

# Cloud Deployment Models

## Private cloud

- ▶ Can be on-prem to provide complete control of infrastructure
- ▶ Can be in the cloud but completely private from others. No one else shares your physical servers
- ▶ Better security and privacy. Hosted in own data centre or cloud private managed servers to control sensitive data and data sovereignty
- ▶ Downside: all admin is provided by the customer

## Public cloud

- ▶ Share infrastructure with unknown others – relies on good security configuration for segregation
- ▶ Costs less as when you switch something off, someone else can use that resource
- ▶ Access from anywhere in the world
- ▶ Security and privacy is more complex and important to get right

## Hybrid

- ▶ Some systems on prem and some in the cloud
- ▶ Maybe for security - or as part of a phased migration – or to keep sensitive data on prem
- ▶ Applications and data can span both
- ▶ Much more complex

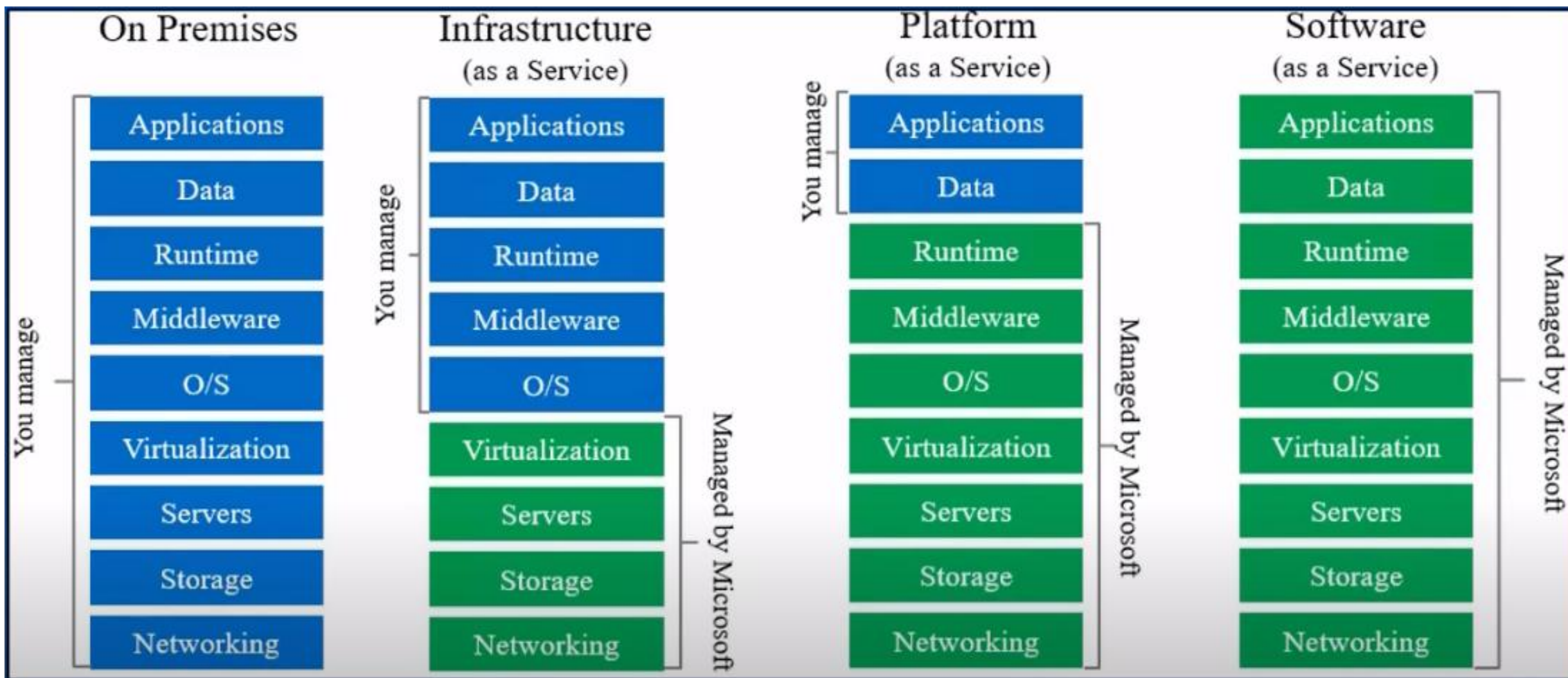
## Community Cloud

- ▶ A cloud environment shared by several organisations with common goals or requirements, such as security, compliance, or policy. e.g. Government, Education, healthcare

# Cloud Service Models

- ▶ **IaaS:** Infrastructure as a service – you deploy and manage architecture, operating system, apps, databases etc
  - ▶ Basically rent servers / memory / disk and related power, networking etc
  - ▶ Fast server scaling
  - ▶ No ownership of hardware - no more CapEx, move to OpEx
  - ▶ Most control but also need more skilled staff such as architects, network specialists, dev-ops, maintenance team
- ▶ **PaaS:** Platform as a Service
  - ▶ IaaS plus OS, tools, database etc. You just need to create and deploy the apps onto it and it will scale for you
  - ▶ No need to worry about OS licenses, patching, upgrades etc.
  - ▶ Less skilled resources needed – once architected, only need mainly developers. All patching etc is part of the service
- ▶ **SaaS:** Software as a Service
  - ▶ Infrastructure, resilience, scaling, DR, elasticity etc is a managed service and so is the software and licenses
  - ▶ Examples include: O365, SAP applications, Seibel CRM etc
  - ▶ Unlike desktop apps, you can access SaaS applications from anywhere with any device that can access the internet
- ▶ **DBaaS:** Database as a service - full service between PaaS and SaaS in the cloud - e.g. MongoDB Atlas

# Cloud Shared Responsibility Model

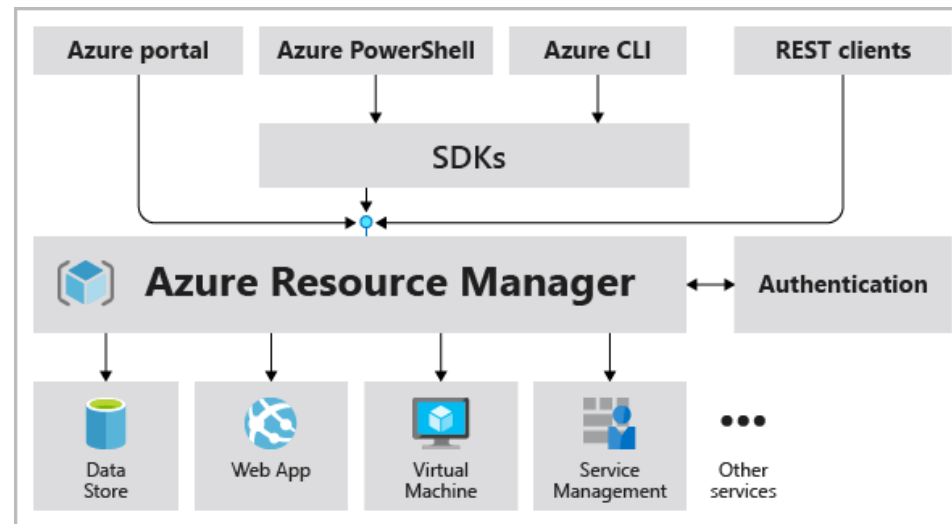




Create a VM

# Infrastructure as a service

- ▶ There are many cloud data centres around the world. They are generally referred to as availability zones
- ▶ We will start off just using one to deploy a simple application
- ▶ This is not a cloud computing course so I'll stick to a need-to-know approach, you can research the rest if interested
- ▶ Azure (and others) is accessed using a web API where you can create, start, stop and remove resources
- ▶ You create a local area network, deploy virtual machines, install operating systems, deploy your application and data all via the web API
- ▶ The Azure resource manager creates resources you ask for. In Azure, all resources must belong to a resource group



- ▶ The portal is effectively an Azure web site that creates the API requests in a point and click approach. The portal is very convenient but not very practical other than for learning or trying stuff out
- ▶ Resources are normally built using code which interacts with the ARM through its API
- ▶ Code can be written in any language as long as it generates appropriate ARM API calls
- ▶ Or use the command line interface (CLI). The CLI needs to be installed. With this you can access the ARM API from your local machine with:
  - ▶ Microsoft Powershell with pre-written commandlets (basically pre-written functions)
  - ▶ Or shell script to issue CLI commands
- ▶ BASH and Powershell CLI is also available using Azure Cloudshell from a browser in the portal – this is an online CLI to write infrastructure as code rather than using your local machine
- ▶ So, quite a few options
- ▶ The free account only includes access to one type of VM for free - the B1S Burstable B series that's usable for up to 750 hours per month
- ▶ If you are using your student account, check the limitations. e.g. 4 cpu per region, 3 public IP addresses



# Deploy to the cloud

- ▶ We are nearly ready to deploy apps to the cloud
- ▶ If you don't have a Microsoft Azure account, then:
- ▶ You will need a Microsoft account first <https://account.microsoft.com/account/Account>
- ▶ You should have a student account. This gives to \$100 and this isn't used if you use free tier services
- ▶ Student account is quite limited - e.g., 4 cpu and 3 public IP limit but there are ways around its limitations that I'll discuss. Access at: <https://azure.microsoft.com/en-us/free/students>
- ▶ You could create a free account as well: <https://azure.microsoft.com/en-gb/free> Note: you will need a credit or debit card for this. You won't be charged if you remain within the free credit. There is no 4 cpu and 3 public IP limit using this approach
- ▶ Once you have your account, you will have root access. It is not advisable to use this unless you absolutely must, and it's worth setting 2FA on it
- ▶ It's worth then setting an admin account for yourself and use that generally. If your admin account is compromised, e.g. someone hacks you and changes the password, you can still access your account using root and create a new admin account. There is a step-by-step guide here: <https://learn.microsoft.com/en-us/azure/role-based-access-control/role-assignments-portal-subscription-admin>
- ▶ I can't show any of this stuff on a corporate account as I don't have access to these things

# Billing alert

- ▶ Set up a billing alert so you don't leave something running by mistake and run up a bill
- ▶ From the portal:
  - ▶ Search for budgets
  - ▶ Add
    - ▶ Unique name
    - ▶ Amount
    - ▶ Next
    - ▶ Type & percentage to trigger first alert
    - ▶ Add email address
    - ▶ That's it. You will get an email if your spend or projected spend hits x% of your limit
- ▶ Note: this is an alert. It will not stop you being billed so make sure your email isn't in spam

# Create a VM on Azure

- ▶ Various sizes and operating systems available
- ▶ Various methods available to create them - we will start with the Azure portal
- ▶ I've provided a couple of videos on how to do this from the portal
  - ▶ "Create an ssh key pair in Azure in 4 minutes.mp4" (4 mins)
  - ▶ "Create linux vm on Azure with ssh keys in 3 minutes.mp4" (3 mins)
- ▶ To connect remotely to a VM you need to create it with either a username and password or using an asymmetric key pair where the public part can be held by the cloud provider and the secret part by you. The key is used to encrypt the secure shell session SSH from your client machine to the VM
- ▶ Once the VM is setup, we have options. We could install nodejs then deploy our files and run the code in the same way we do on the local machine. You can install a database in the same way we have on our local machine so there is nothing new there, but the more modern and scalable approach is containerisation

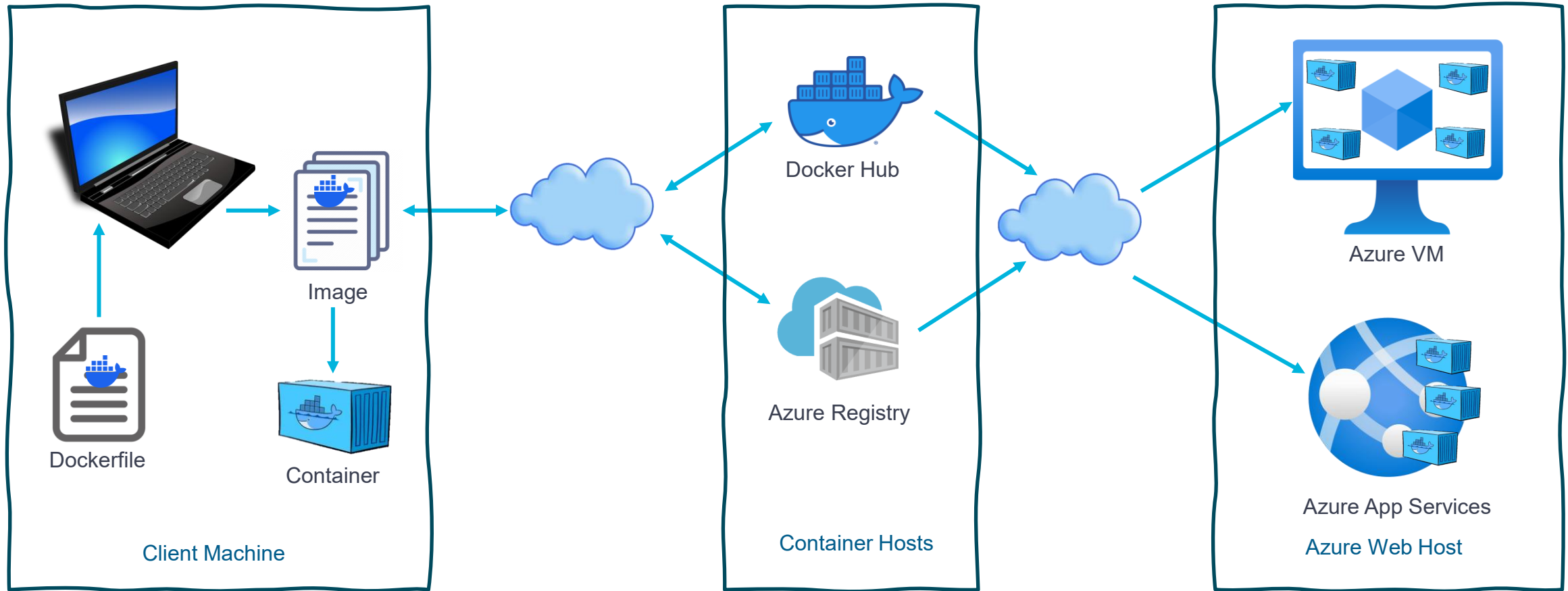
- ▶ Create a key pair
- ▶ Create a linux vm from the portal
- ▶ Connect to the vm from the client (or cloudshell - but you need to create a disk for this which is not free)
  - ▶ **ssh -i filepath/name.pem username@ipaddress**
  - ▶ or **ssh username@ipaddress** then enter the password when asked
- ▶ ssh uses port 22 so this needs to be open in the vm firewall (network security group)
- ▶ Issue a command to see the server is alive
  - ▶ `lsb_release -a`
  - ▶ move around between dirs

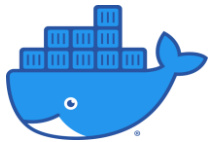
create basic vm



# Container deployment

# Container deployment process





Docker Hub



Azure Registry

Container Hosts

- ▶ A registry is a place you store your images - typically online
- ▶ This enables access to the image to others to create a container from it
- ▶ There are various container registries: Amazon Elastic Container Registry (ECR), Azure Container Registry (ACR), Docker Hub Container Registry, GitHub Container Registry and others
- ▶ You can make images available for others to download or you use theirs. I will use Docker Hub as it is the most popular by far, and free

# Create a container registry

There are various options but a popular one is Docker hub

Create a Docker hub registry

- ▶ Go to <https://hub.docker.com/> and Sign up
- ▶ Enter docker ID - needs to be unique to you. Fill in other stuff - choose the free one unless you want to pay
- ▶ Note: you can now only have **one** private repo. You can have many public repos
- ▶ `docker login docker.io -u oldgitdocker` or, `docker login -u oldgitdocker` assumes `docker.io`
- ▶ To push an image called `oldgitdocker/tst`, use **`docker push oldgitdocker/tst`**

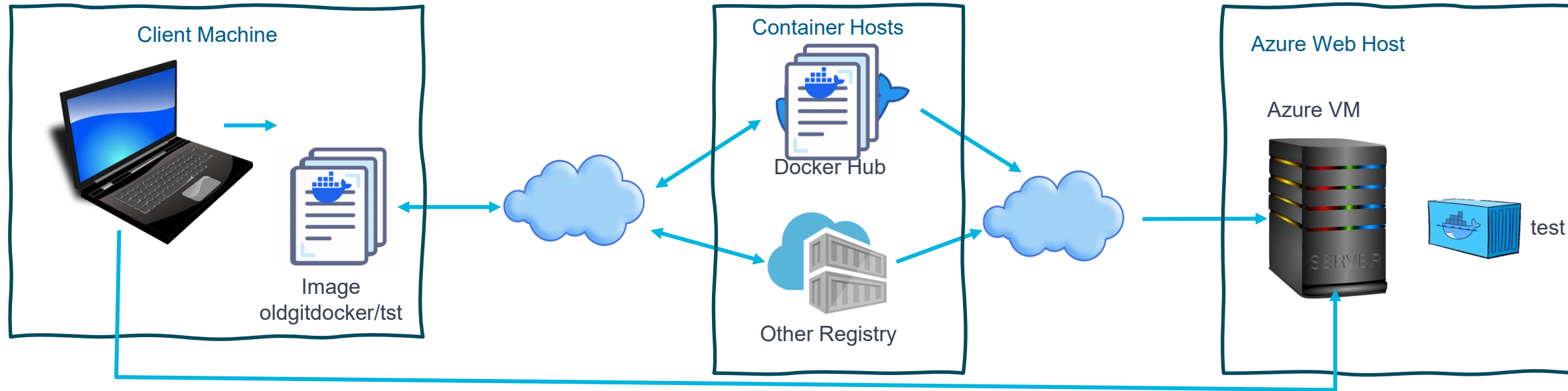


# Deploy to a VM basic instructions

- ▶ Initial prerequisites:
- ▶ Create a vm - make sure you open appropriate ports - use password or key (aws doesn't support password)
- ▶ See: Create linux vm on Azure with ssh keys in 3 minutes.mp4
- ▶ Copy docker install shell script to install docker. Use winscp or other GUI app or just scp at the command line
- ▶ `scp -i c:\your_key docker-az.sh you@your_ip:/home/you/docker-az.sh`
- ▶ `scp -i c:\your_key compose.yaml you@your_ip:/home/you/compose.yaml`
- ▶ If using WinSCP, see **using winscp.mp4**
- ▶ Execute the docker install script on the remote machine by logging in to VM and entering
  - ▶ `bash docker-az.sh`
  - ▶ It takes a couple of minutes to install
- ▶ We can now build and deploy docker images and start them as containers on the VM

# Deploying image

- ▶ The following illustrates the build and deploy process



```
docker compose build
docker login -u oldgitdocker (push)
docker push oldgitdocker/tst
ssh -i c:\key\az.pem tony@40.81.137.204
scp -i c:\key\az.pem path\compose.yml tony@40.81.137.204:/home/tony
```

```
docker compose up
http://40.81.137.204:4000/tst/tony?count=30&format=json
docker compose down
```

# Container vs VM

- ▶ Both are used to create an isolated virtual environment - but they are different
- ▶ VM pros:
  - ▶ Less expensive than physical computer as you can run more than one on the same hardware
  - ▶ Necessary for cloud as you have no access to the physical machines
  - ▶ Single hypervisor can manage all VM resource allocation and management
  - ▶ Each process is isolated at the hypervisor level. Each can have a different OS
- ▶ VM cons:
  - ▶ Take up a lot of resources - they are large - especially Windows VMs, many GBs so use a lot of disk space and RAM
  - ▶ A VM rarely uses all allocated resources, so is less cost efficient, but, still more cost efficient than a physical machine
- ▶ Container pros:
  - ▶ Can be small - a few MBs so potentially 1000 times smaller than a VM
  - ▶ Due to their size and not having to deploy a whole OS, they can be deployed and quickly scale through replication
- ▶ Container cons:
  - ▶ Unlike a VM, processes in a container are only isolated at the host kernel level so a little less secure
  - ▶ Because the kernel is shared, each must work with the same OS that hosts Docker
  - ▶ If one container misbehaves and causes instability in the kernel, it could affect the others
  - ▶ If data needs to persist on container deletion, or restarted, a volume needs to be created on the host or network

# When to use one over the other

- ▶ A VM may be a better choice if you need to:
  - ▶ Manage a variety of operating systems
  - ▶ Manage multiple apps on a single server - i.e. make use of task switching
  - ▶ Run an app that requires all the resources and functionalities of an OS
  - ▶ Ensure full isolation and security
- ▶ A container may be a better choice if you need to:
  - ▶ Maximize the number of apps running on a server - as they are small
  - ▶ Deploy multiple instances of a single application
  - ▶ Have a lightweight system that quickly starts
  - ▶ Develop an application that runs on any underlying infrastructure
  - ▶ If it runs on the test machine, it's highly likely to run on the target machine



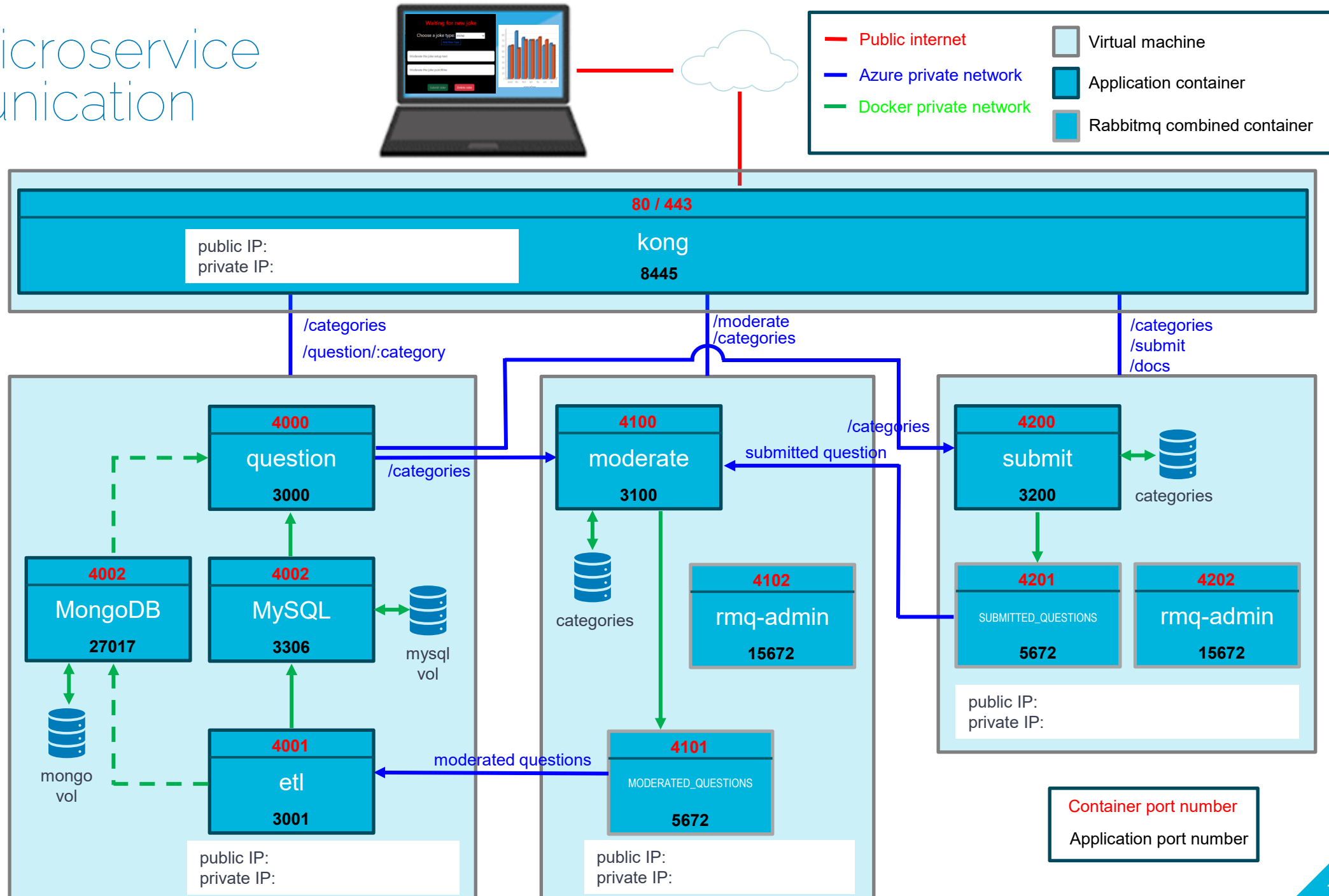
# Database container & Volumes

# Objectives

On completion of this section you will be able to:

- ▶ Explain the need for Docker volumes
- ▶ Differentiate between a named volume and a bind mount
- ▶ Create a named volume and bind mount
- ▶ Deploy your application and database to the cloud the run in containers
- ▶ Use a .env file with compose to configure your system
- ▶ Use a Docker private network and explain how Docker creates private IPs and DNS names

# Inter-microservice communication



# Docker volumes

- ▶ You could store data in the image and container. For example, the data files could be stored in, and accessed from, the container by copying them at build time or posting data into them using an API. However, this just makes the image and container larger, and the data is lost when the container is deleted as they are ephemeral
- ▶ There are essentially five volume types, but we will focus on the main two: **named volumes** and **bind mounts**. These are used for persisting or accessing data outside the container. These are created on the host but managed by docker. Multiple containers can share a named volume
- ▶ As a volume exist on the host, it's stored in a protected docker directory, so on container deletion, the data persists
- ▶ A database for example needs somewhere to store its data outside the ephemeral container. A volume satisfies this
- ▶ Volumes can be created in a Dockerfile but as we are creating multiple containers, we will use compose so will configure the volumes in there

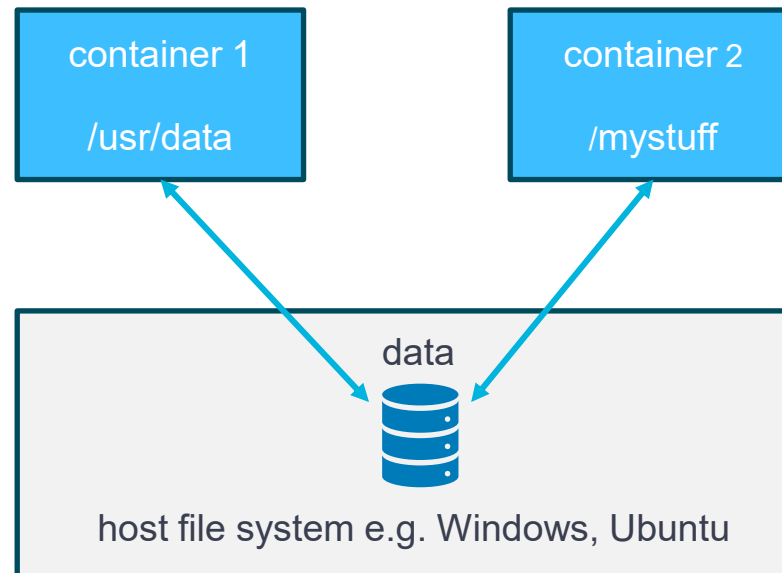
## Named Volume

- ▶ This will associate a name with the folder value in the name value pair. It's as if you had created a disk with the specified name. Writing to **/usr/src/data** in the container, will write to a protected docker folder on the file system called **data** (or whatever you call the volume) as if it was a logical disk if declared as follows:

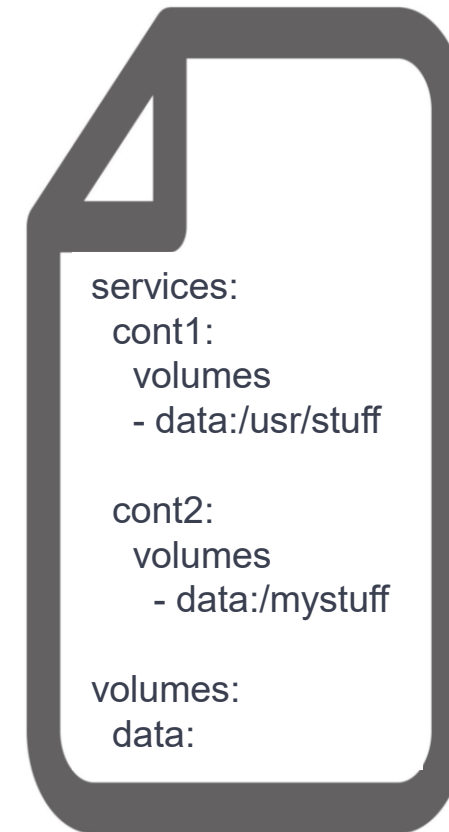
**volumes:**

**- data:/usr/src/data**





compose.yaml



- ▶ Compose file maps specific service folder to a volume
- ▶ Compose declares the creation of a volume called **data** on the host OS - can be elsewhere also e.g. S3

# Docker bind mounts

- ▶ A bind mount maps a directory in the container to a directory on the host machine
  - ▶ Unlike a volume, no space is created, write to one and data is seen in the other
- ▶ Say a file or dir outside the container is needed in the container at `/usr/app/whatever`
- ▶ But the data is actually on the host in say `/home/username/stuff`
  - ▶ A bind mount which maps these will enable the container to access `/usr/app/whatever` but the actual data accessed is `/home/username/stuff` on the host
- ▶ e.g., declare a bind mount as a volume but not to create a volume, just map two folders onto each other
- ▶

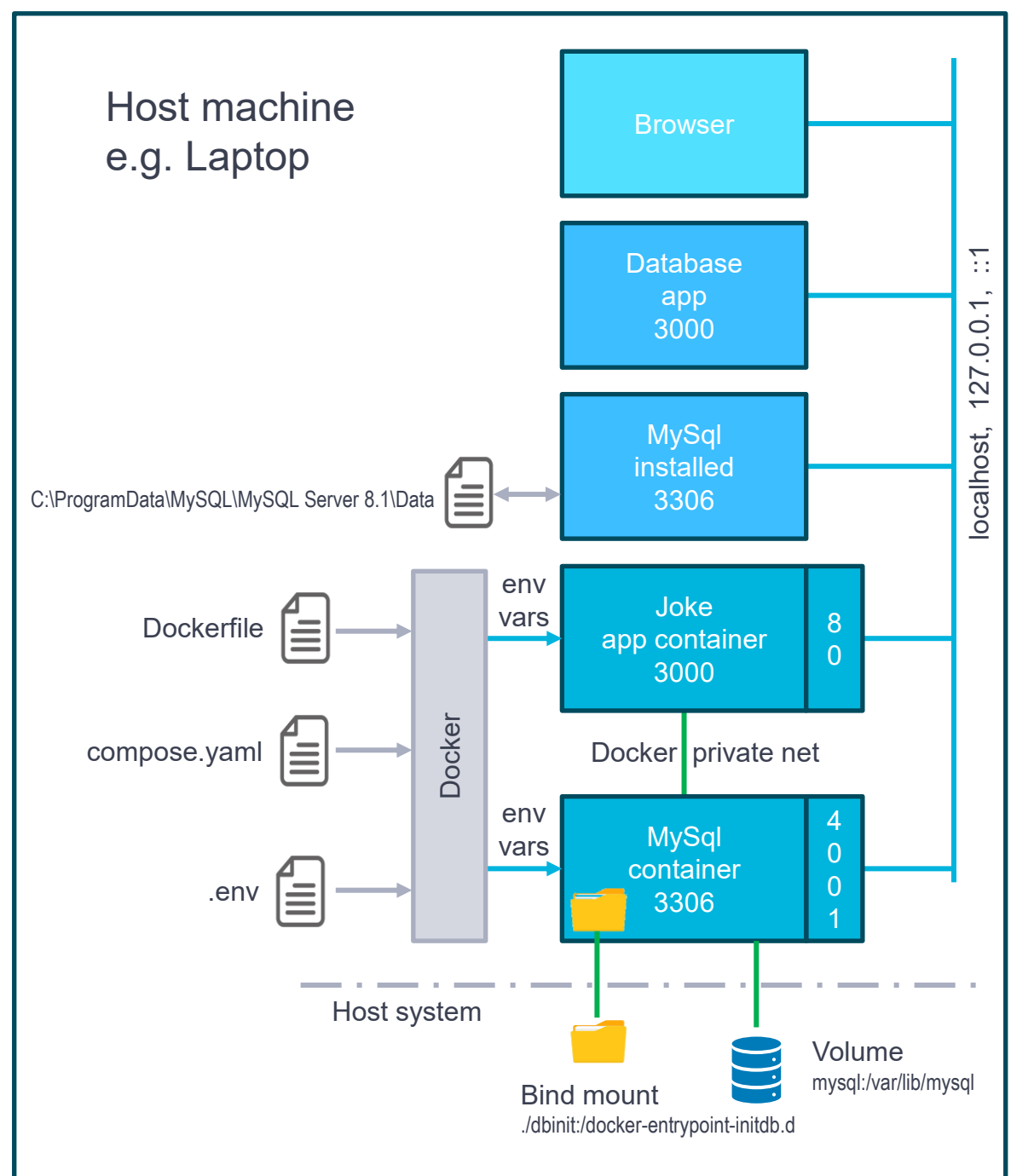
```
volumes:
 - /hostPath:/containerPath
 - /home/username/stuff:/usr/app/whatever
```

# App & Database containers

- ▶ MySQL data stored in files on host file system (Win)
- ▶ MySQL data Container files stored on file system (Linux)
  - ▶ But that is the Alpine Linux in the container
  - ▶ Volume maps data outside the container
  - ▶ Alpine dir mapped to Ubuntu Docker dir
  - ▶ Docker dir Ubuntu: /var/lib/docker/volumes/
  - ▶ Docker references as mysql or whatever you named it
- ▶ In Windows, volumes are stored in wsl at:
  - ▶ \\wsl\$\docker-desktop-data\data\docker\volumes

`\Demos\data\poor joke sql to improve in lab\app.js`

`jokePoorMySQLContainers.mp4`



# Some useful compose commands

- ▶ Docker compose has a cli like Docker. However, most of the commands can be used in the desktop if you don't like typing. Details here: <https://docs.docker.com/compose/reference/>
- ▶ **docker compose build** will build all the images defined in the compose.yaml file
- ▶ **docker compose up** will start all the containers
- ▶ or with one command, **docker compose up --build** will do both
- ▶ **docker compose up -d** will run the containers in detached mode
- ▶ **docker compose exec -it etl sh** This will start the shell if sh is available. Some images have bash in them too
- ▶ **docker compose exec -it mysql-svr bash** The mysql image has bash and sh
- ▶ **docker compose exec -it caller sh** This only has sh as it is built from alpine. -it gives us access to a terminal
- ▶ **docker exec** does the same thing but uses the container name not the service name
  - ▶ **docker exec -it etl-cont sh**
  - ▶ **docker exec -it mysql-cont bash**
  - ▶ **docker exec -it caller-cont sh**
- ▶ **docker compose down** will stop and delete all containers, networks, volumes created with up
  - ▶ named volumes won't be deleted unless you add **--volumes or -v**
  - ▶ images are not deleted unless you add **--rmi local | all**
- ▶ **docker system prune --all --volumes** deletes everything
- ▶ **docker compose logs -f** useful to see all console.log statements when on cloud
- ▶ **docker compose restart service-name** to restart a stopped container
- ▶ **docker compose config** you can check, even before building, what the compose file looks like after parsing all env substitutions for checking

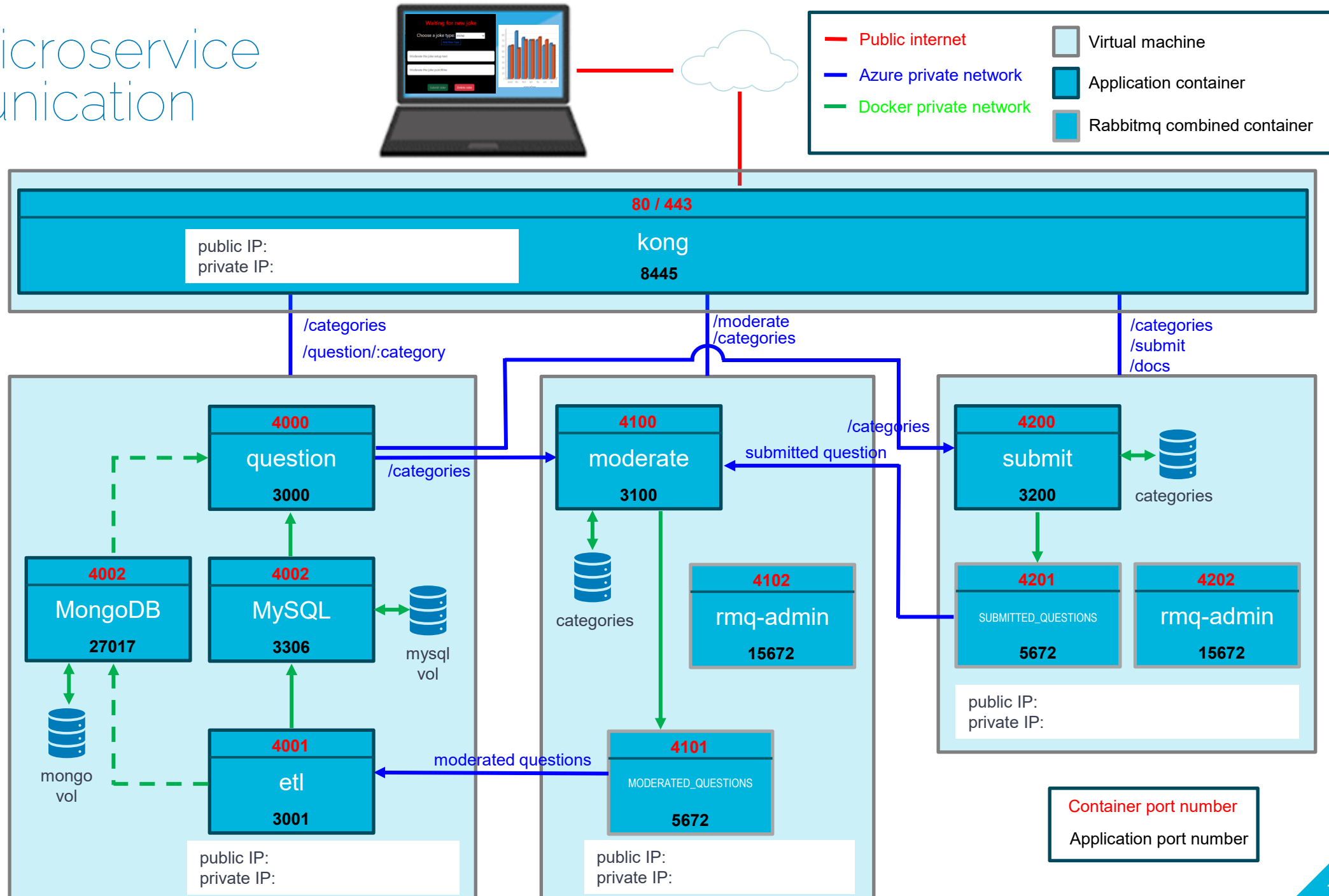


# Microservice Inter-communication

# Learning objectives

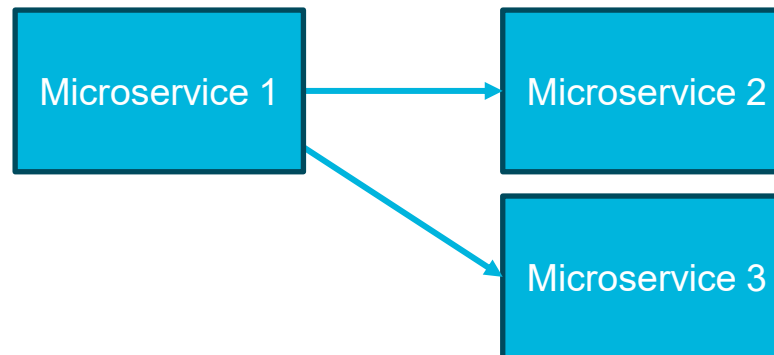
- ▶ Upon successfully completing this work, you should be able to:
  - ▶ Explain the differences between synchronous and asynchronous message patterns
  - ▶ Discuss, a use-case explaining the benefits and challenges of using synchronous messaging
  - ▶ Discuss, a use-case explaining the benefits and challenges of using asynchronous messaging
  - ▶ Explain the difference, with a use-case, of provider / consumer, and publisher / subscriber patterns
  - ▶ Implement synchronous and asynchronous message patterns for inter-microservice communication

# Inter-microservice communication



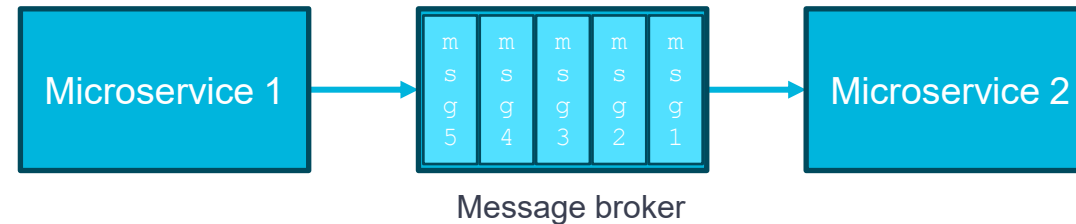
# Inter Microservice Communication

- ▶ HTTP is synchronous
  - ▶ JavaScript overcomes thread blocking with callbacks or promises but an http request still waits for a response, so http is synchronous
  - ▶ Synchronous HTTP API is fine in some circumstances but does more tightly couple the services than an asynchronous approach - e.g. if the endpoint changes, it affects all the callers as the contract is broken for all of them or at best, inconsistent . Could build-in versioning into the path to minimise this
  - ▶ Response is generally fast as it's a direct point to point call
  - ▶ What about destination service failure? Http doesn't account for this unlike SOAP. e.g. you post some data and get a 500 response - what has happened to the data? How do you recover? More code needed
  - ▶ What about adding another microservice destination? e.g. You are already calling a "Change of circumstances" microservice with say an address change. A new payment service, also requiring the change of address, would result in a new interface from the caller





- ▶ Asynchronous messaging may be a better option in some cases
  - ▶ A microservice posts to a queue and that's it. The response is immediate, but the message must wait
  - ▶ Ms 1 doesn't need to know if ms 2 can cope with a burst of messages that it can't cope with
  - ▶ The microservices are decoupled - the contract is with the queue. Ms1 need know nothing about Ms2
  - ▶ If Ms2 goes down, Ms1 doesn't know or care
  - ▶ Shared message queue pattern, as illustrated below, typically uses a dynamic FIFO data structure - a queue

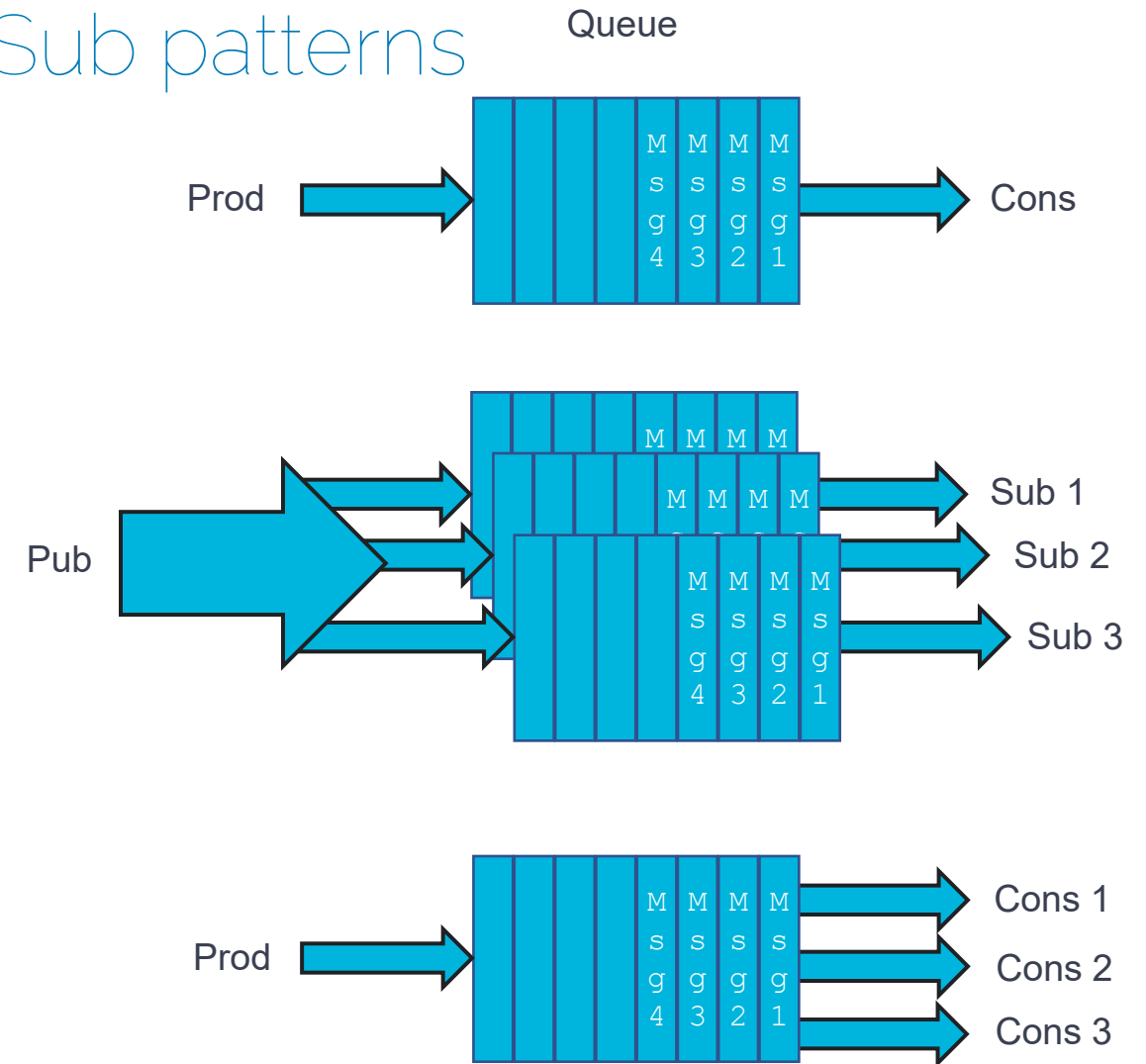
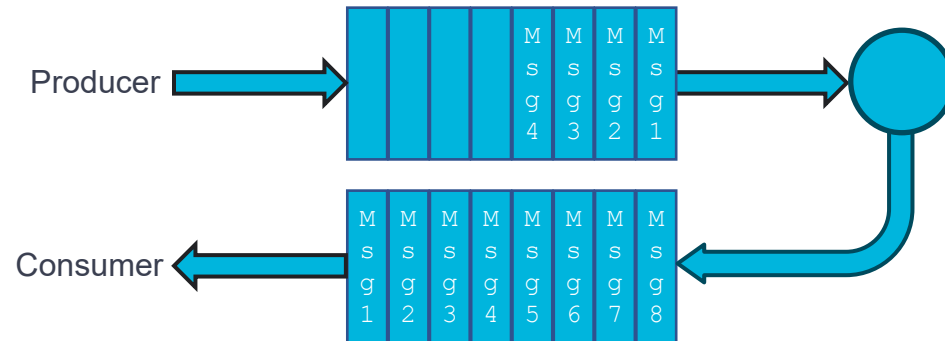


# Message brokers

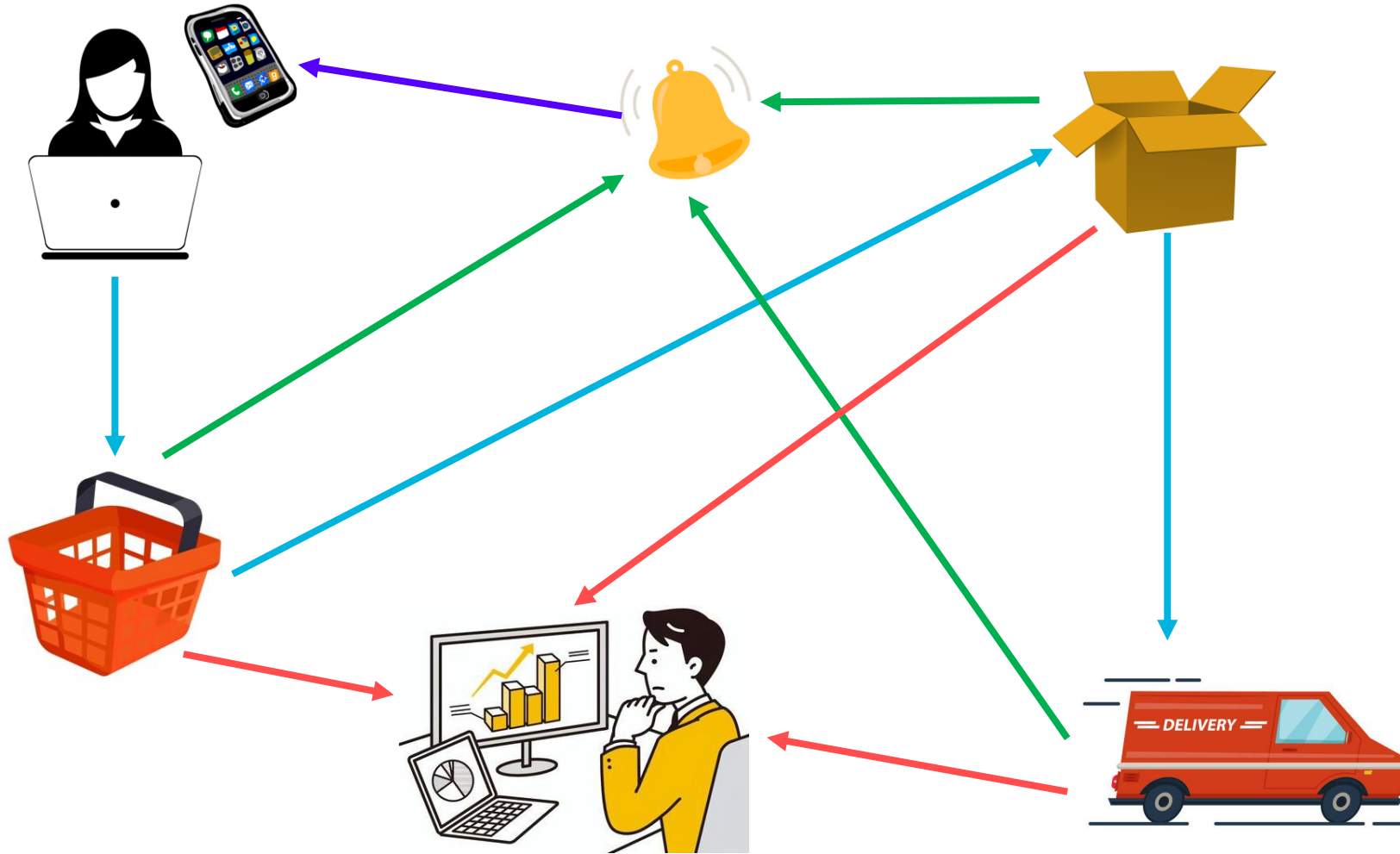
- ▶ Message brokers implement the asynchronous message pattern. There's quite a selection:
  - ▶ RabbitMQ - open source
  - ▶ Apache Kafka - open source
  - ▶ Amazon Simple Queue Service (SQS) - AWS
  - ▶ Azure message broker and Azure storage queue (ASQ)
  - ▶ and more ...
- 
- ▶ We will have an introductory look at RabbitMQ and Apache Kafka as they are cloud agnostic
  - ▶ However, if all-in with a particular cloud provider, their services may be preferential

# Producer/Consumer and Pub/Sub patterns

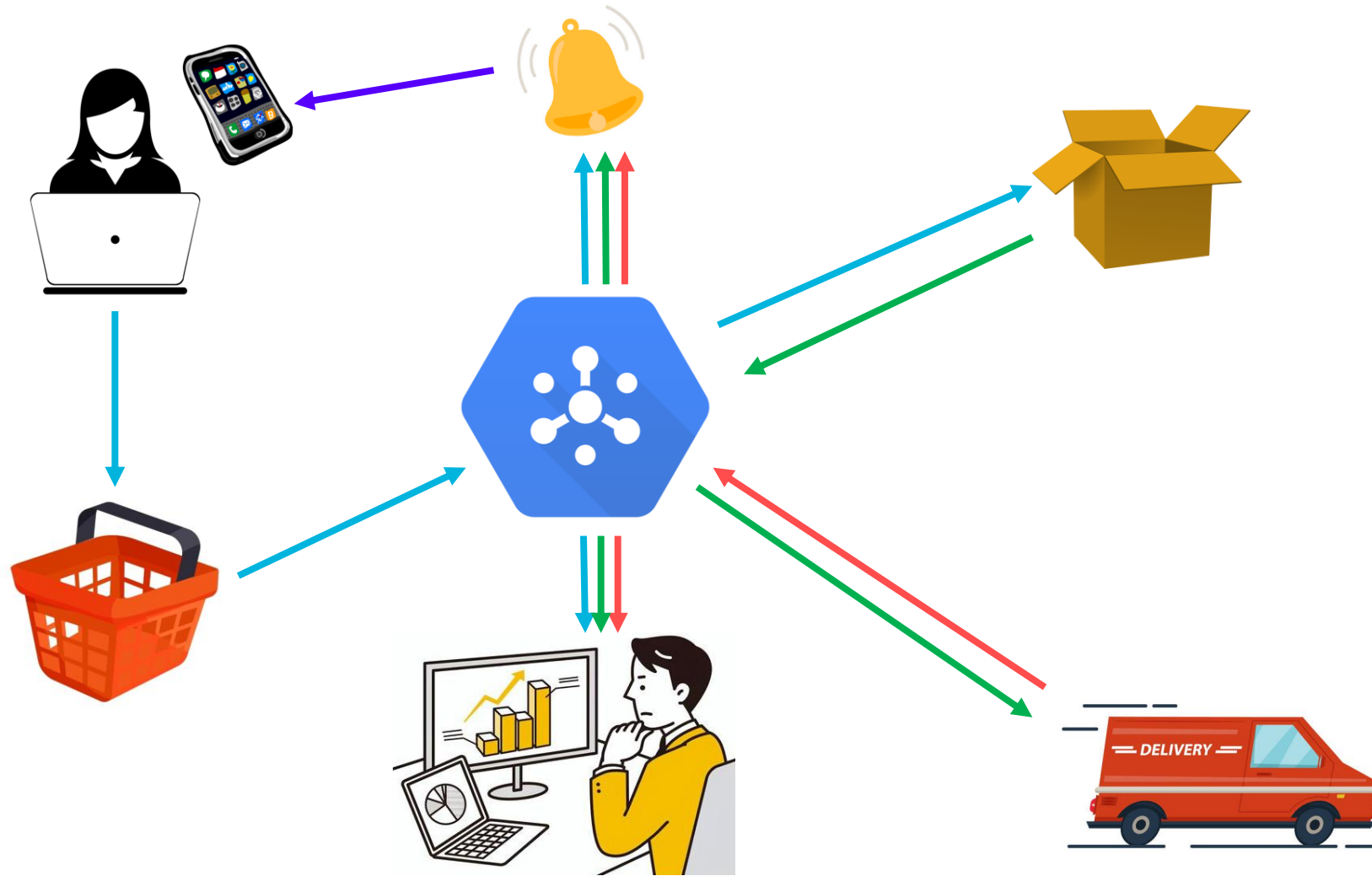
- ▶ Asynchronous service to service messaging
- ▶ One producer and one consumer
- ▶ Use fan-out for pub/sub - adding a new microservice
- ▶ Generally, uses push to consumer
- ▶ Resilience is built-in to avoid SPOF
- ▶ Implemented using a message broker
- ▶ Several patterns



## Simple synchronous scenario



# Publisher / Subscriber asynchronous alternative



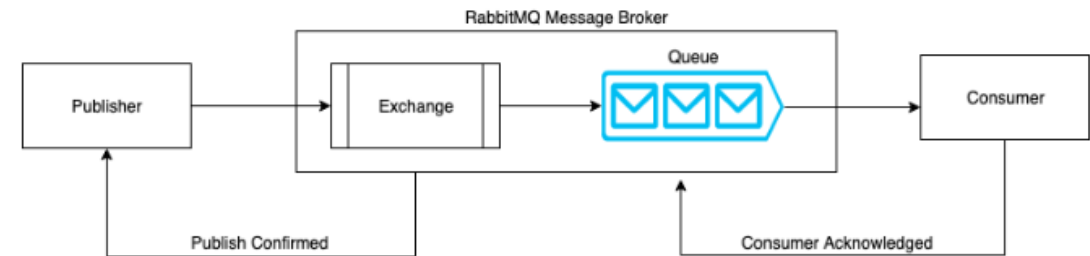
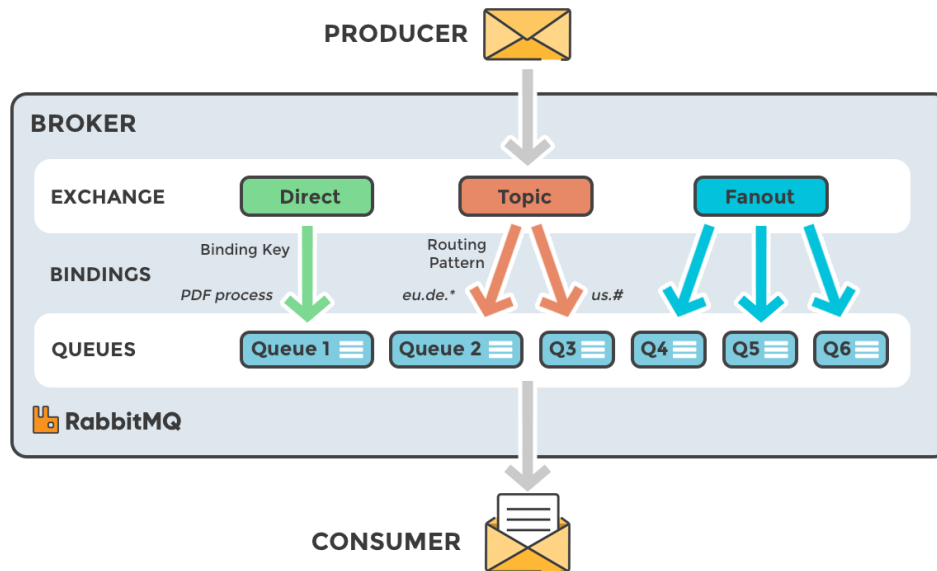
# RabbitMQ terminology

- ▶ RabbitMQ implements the Advanced Message Queueing Protocol (AMQP) standard which is a platform agnostic protocol to transfer information between applications. The source and dest are: producer and consumer
- ▶ Connection
  - ▶ This is a network connection between an app and the queue. Similar to a database connection
- ▶ Channel
  - ▶ virtual connection inside a connection. Multiple channels can be created within a single connection, allowing for concurrent operations and communication between different parts of a client application
  - ▶ For example, you might have one channel for produced messages and another for consuming messages within the same connection to RabbitMQ
- ▶ Queue
  - ▶ A FIFO-based message storage and delivery component
  - ▶ It holds messages sent by producers until they are consumed by consumers
- ▶ RabbitMQ is available as a docker container
  - ▶ It can implement queues and message routing through a message exchange component which can route messages to various queues based on a routing key
  - ▶ It has a management interface to setup and monitor the queues

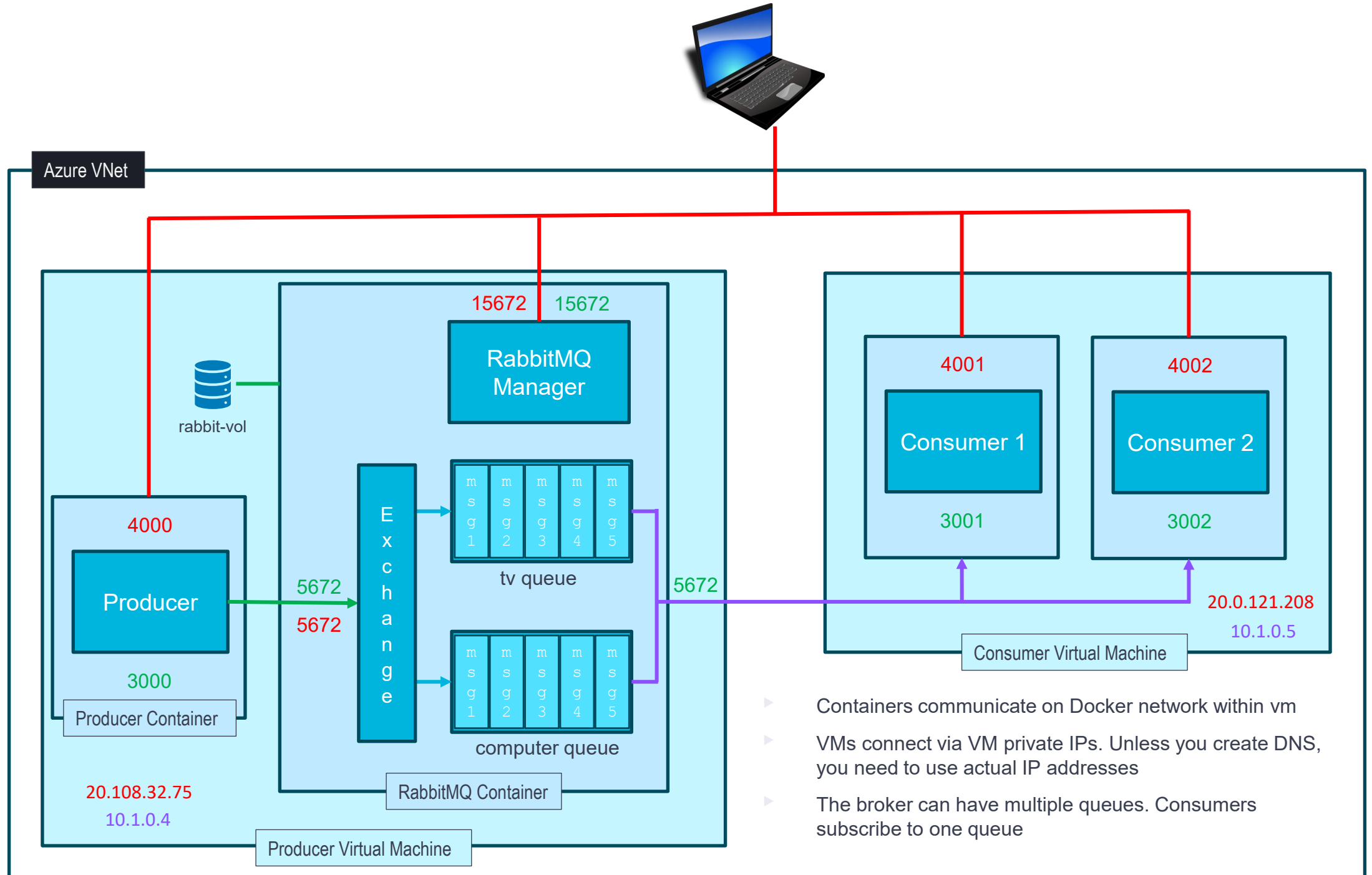
- ▶ **Producer / Publisher**
  - ▶ An application that sends a message to the broker. The broker confirms the message is queued
- ▶ **Consumer / Subscriber**
  - ▶ Application that consumes the message. Consumer informs rabbitmq that it received the message (acknowledges it) so it can be removed from the queue. Without ack, the message stays where it is
- ▶ **Exchange**
  - ▶ Receives messages from producers and routes them to the correct queue based on rules associated with a routing key
- ▶ **Routing key**
  - ▶ This can be a word, series of words, wildcard. e.g. all messages that contain pr\* as a post code are sent to a queue for consumers of Preston based messages - e.g. for a targeted mail shot
- ▶ **Virtual host**
  - ▶ Segregates applications using the same rabbitmq instance. Think of it as separate broker instances each with different permissions and credentials but built into a single instance, e.g. a single container

# Rabbitmq exchange types

- ▶ Direct
  - ▶ a producer and consumer are using one queue based on the binding key (we use this & default exchange)
- ▶ Topic
  - ▶ a wildcard match between the routing key and the routing pattern determines the queue. Like the Preston example
- ▶ Fanout
  - ▶ routes a message to all the queues - e.g. message sent to multiple microservices - pub / sub pattern
- ▶ All this is summarised in the Rabbitmq documentation as:





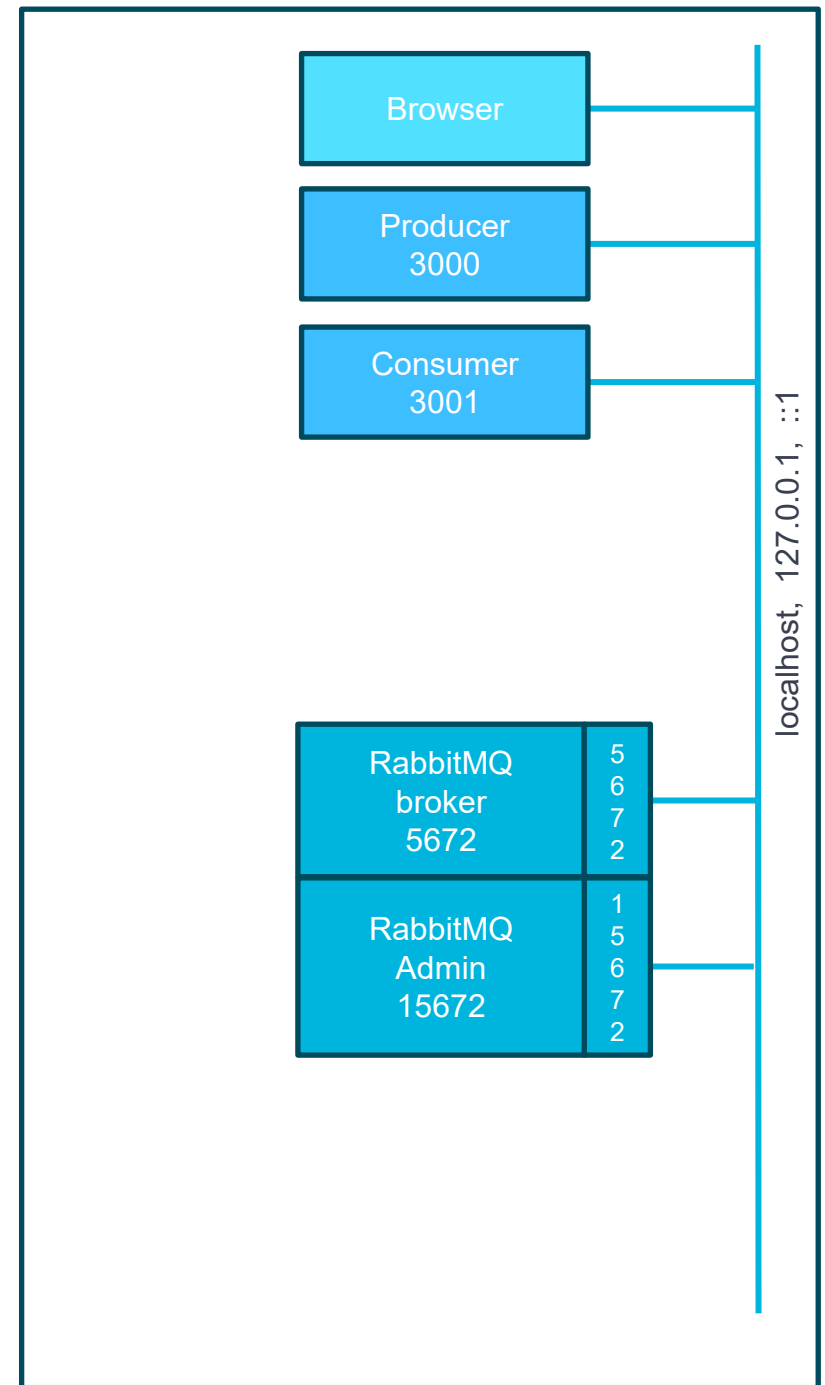


- ▶ Containers communicate on Docker network within vm
- ▶ VMs connect via VM private IPs. Unless you create DNS, you need to use actual IP addresses
- ▶ The broker can have multiple queues. Consumers subscribe to one queue

# RabbitMQ demo

- ▶ One VM will host the producer and message broker
- ▶ Another will host the consumer
- ▶ Test in vscode
  - ▶ If consumer is down, producer should still work
  - ▶ Start consumer and it should clear the queue

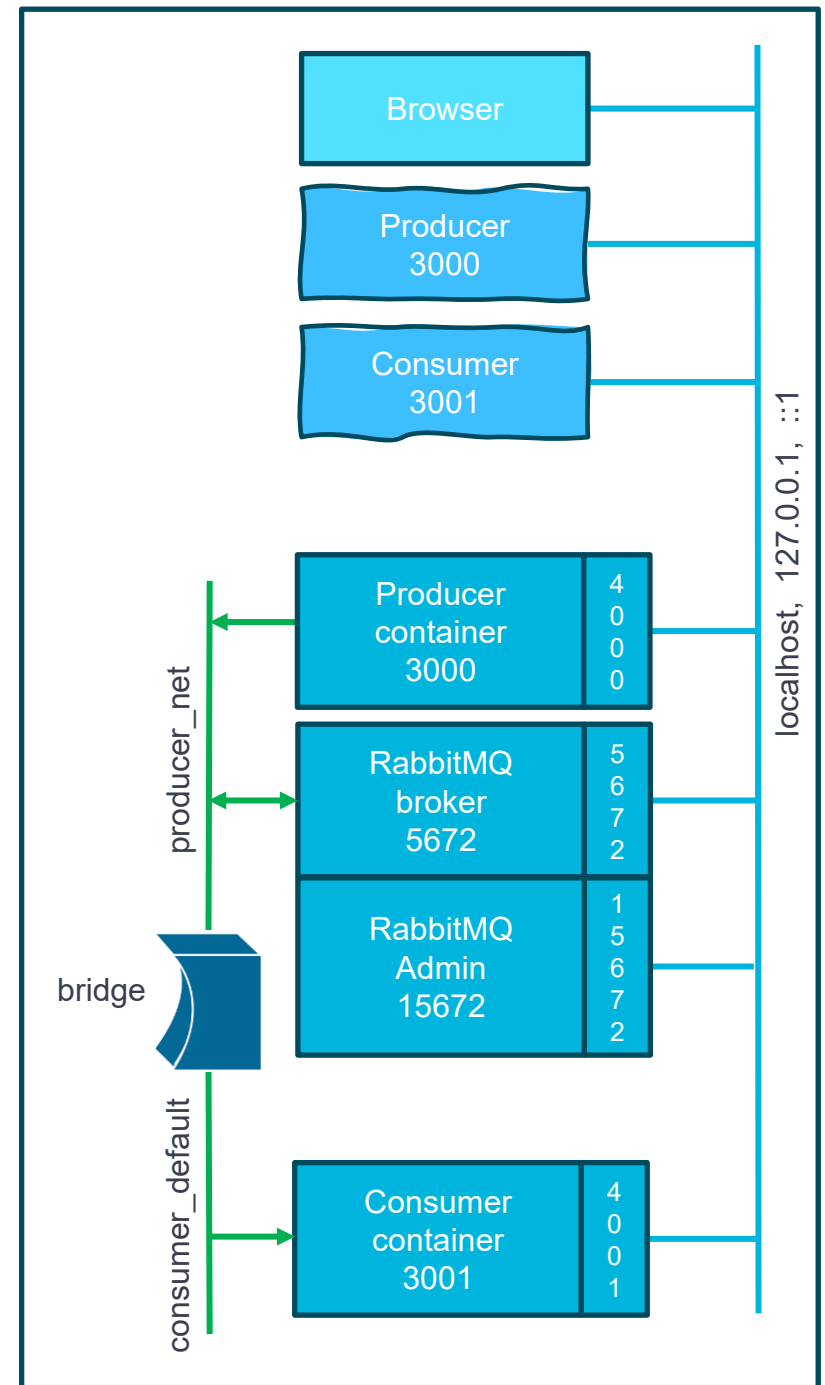
Demo in vscode  
`\demos\architecture\rabbitOneQueue\`



# RabbitMQ demo

- ▶ Create two sets of containers as we will deploy on different VMs
- ▶ As they are on different networks, we need to join them to test
  - ▶ If container recreated, you need to rejoin to network
  - ▶ **docker network ls** to list them
  - ▶ **docker network connect producer\_net consumer**
  - ▶ **docker network connect consumer\_default producer**
  - ▶ **docker network inspect *network\_name*** - look in container list
- ▶ Deploy images to registry
- ▶ Create two VMs and install Docker
  - ▶ Remember to change consumer IP to producer private address
- ▶ Pull images and start containers
- ▶ Test from public IPs

[\demos\architecture\rabbitOneQueue\](#)







# API Gateway

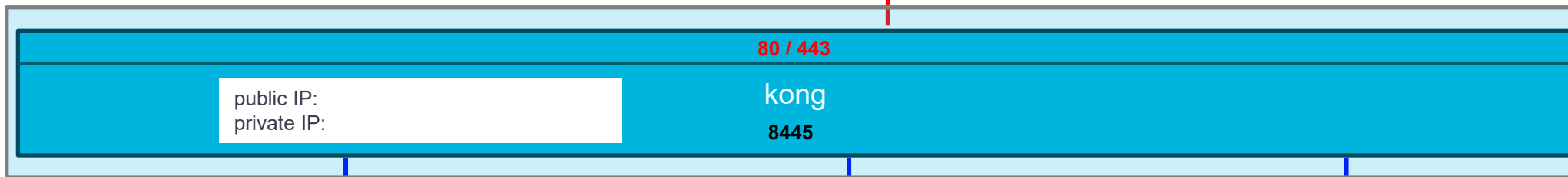
# Learning objectives

- ▶ Upon successfully completing this work, you should be able to:
  - ▶ Explain the benefits of using an API gateway including
    - ▶ Architectural
    - ▶ Security
    - ▶ Orchestration
    - ▶ Rate limiting
    - ▶ Monitoring
    - ▶ Authentication
    - ▶ Caching
  - ▶ Explain any disadvantages of using an API gateway
  - ▶ Implement an API gateway in local containers and in the cloud

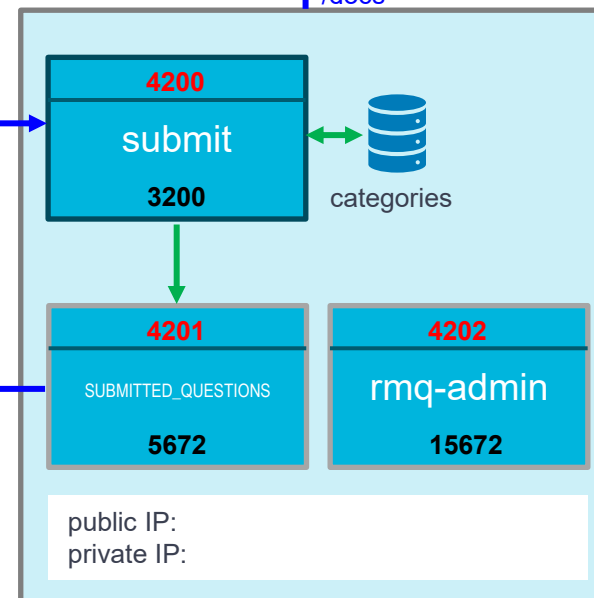
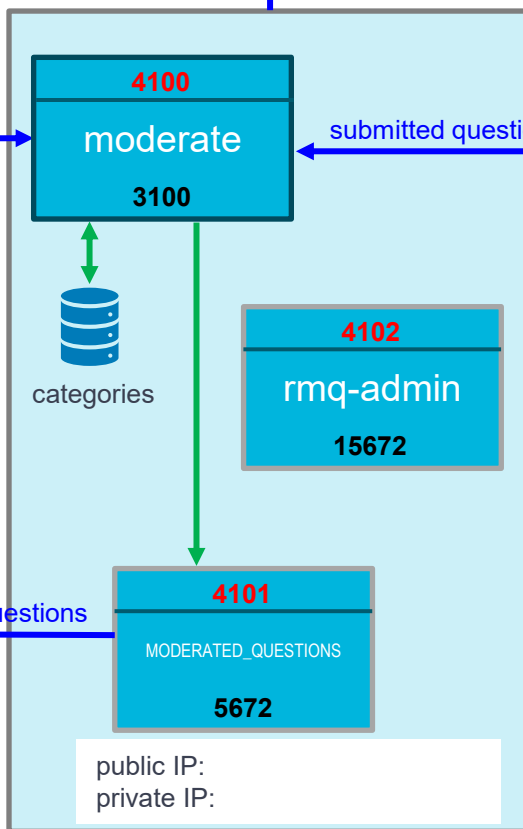
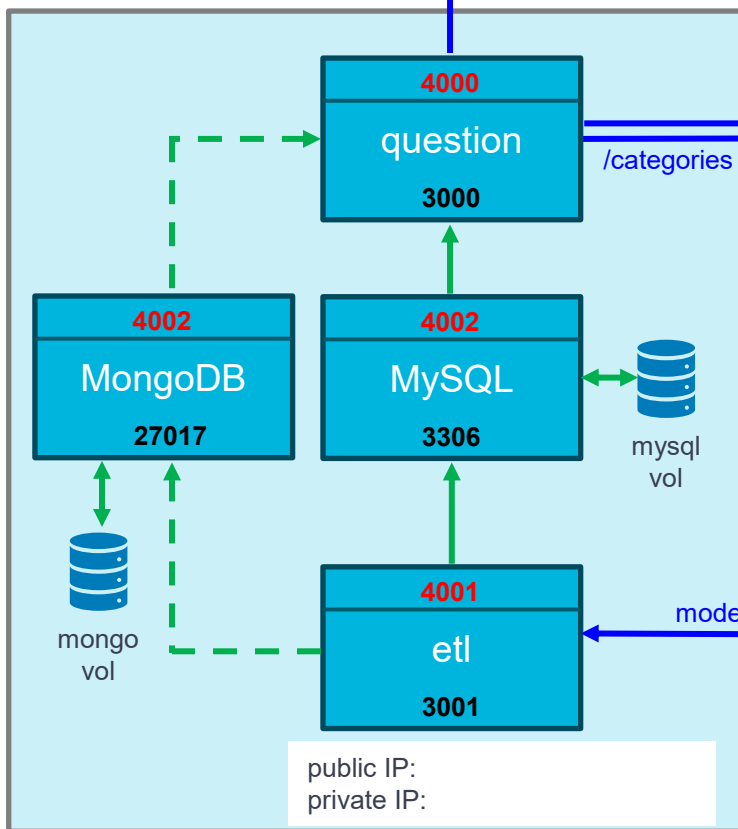


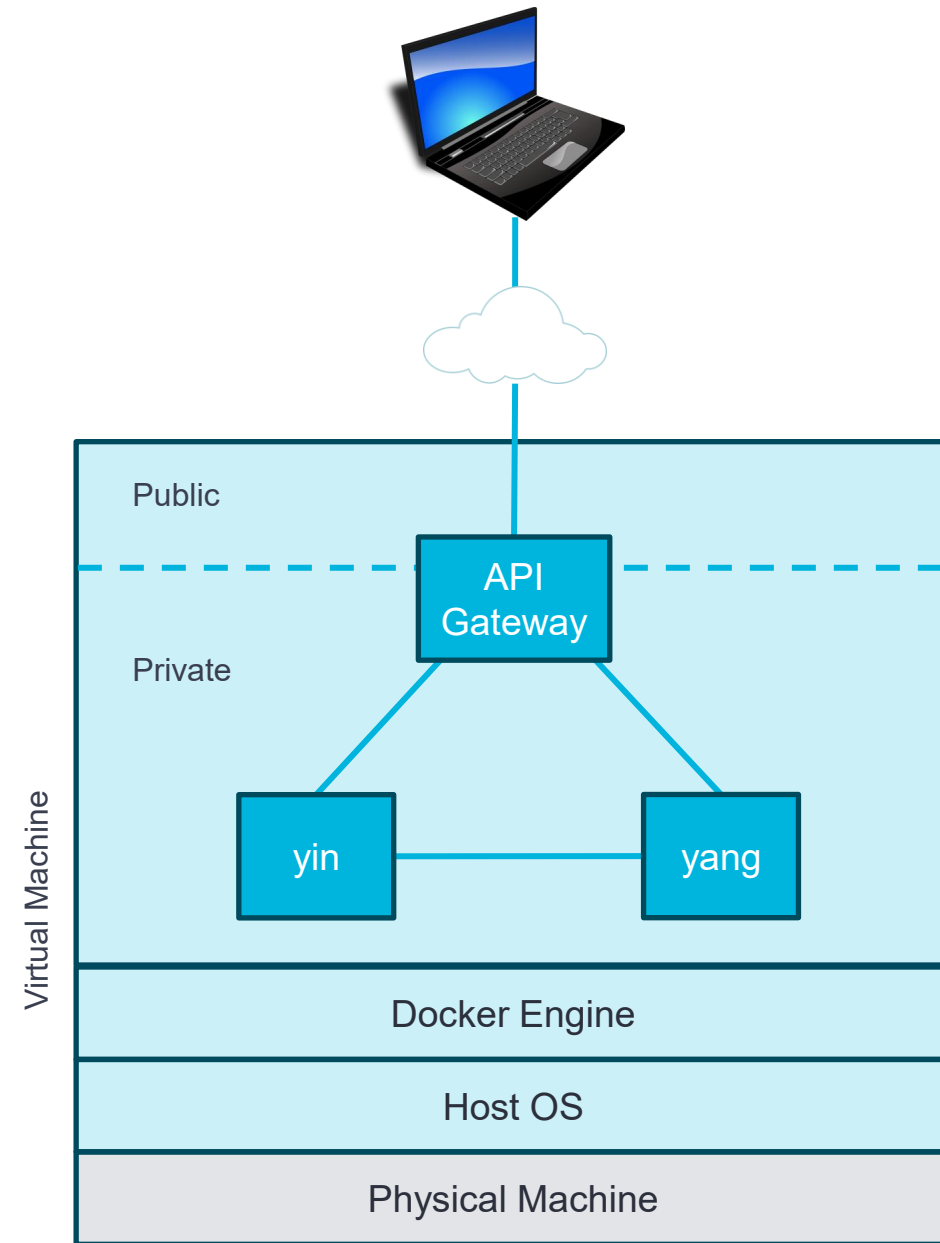
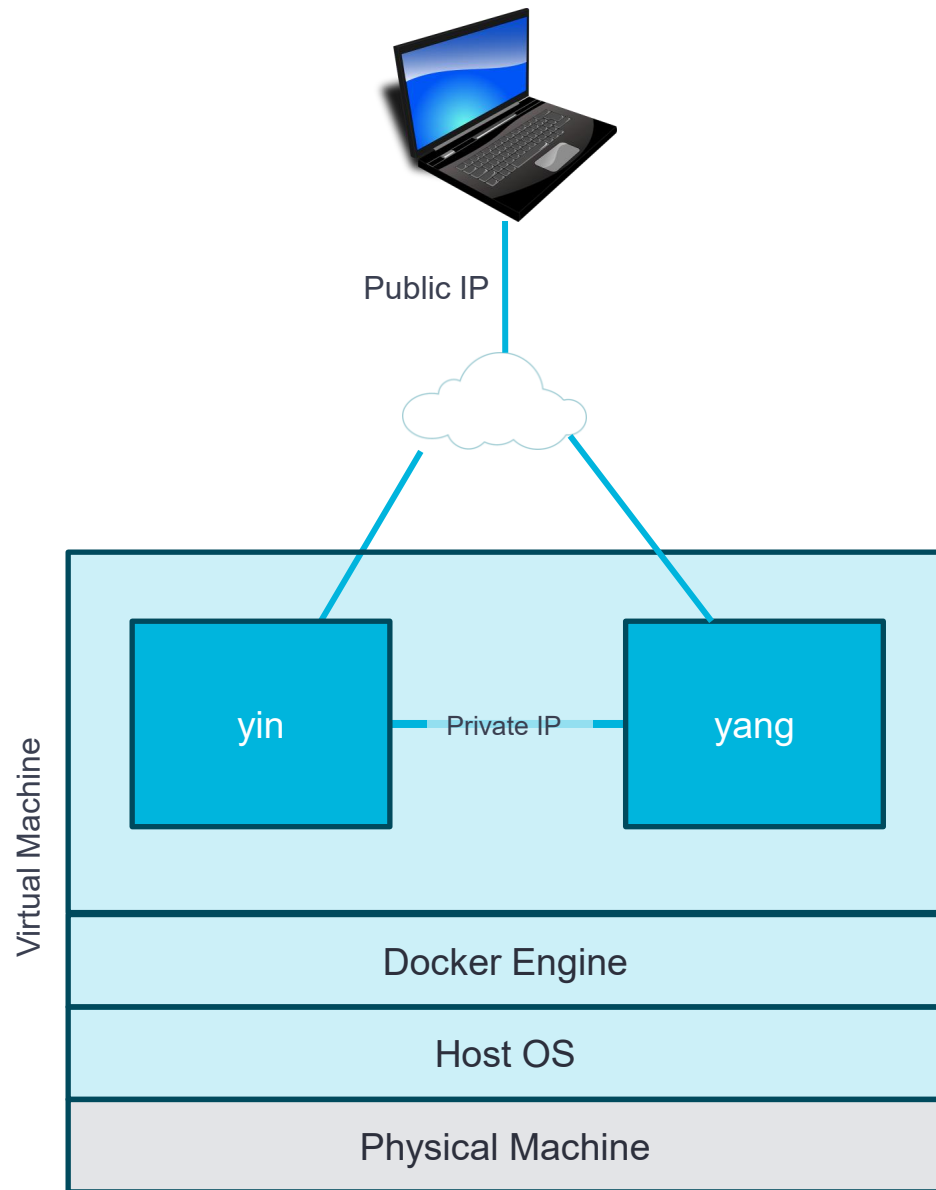
- Public internet
- Azure private network
- Docker private network

- Virtual machine
- Application container
- Rabbitmq combined container



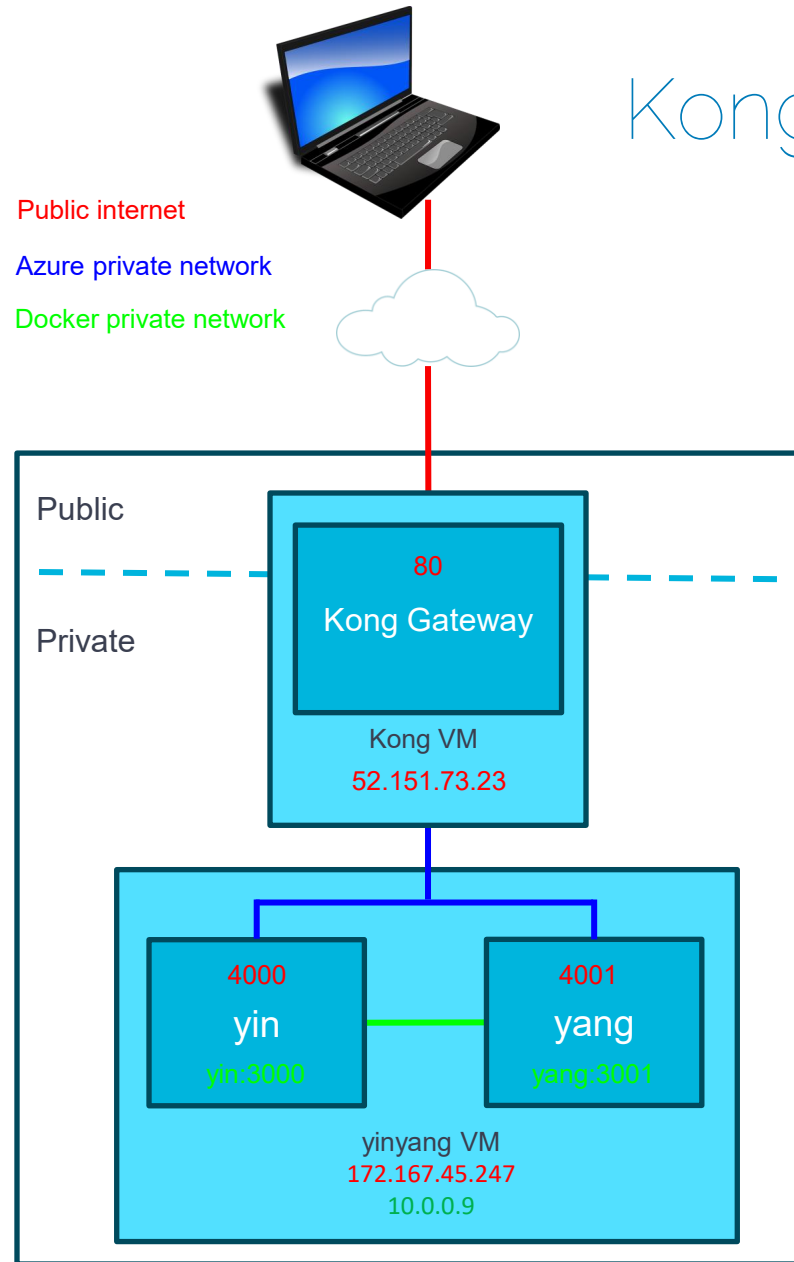
Container port number  
Application port number





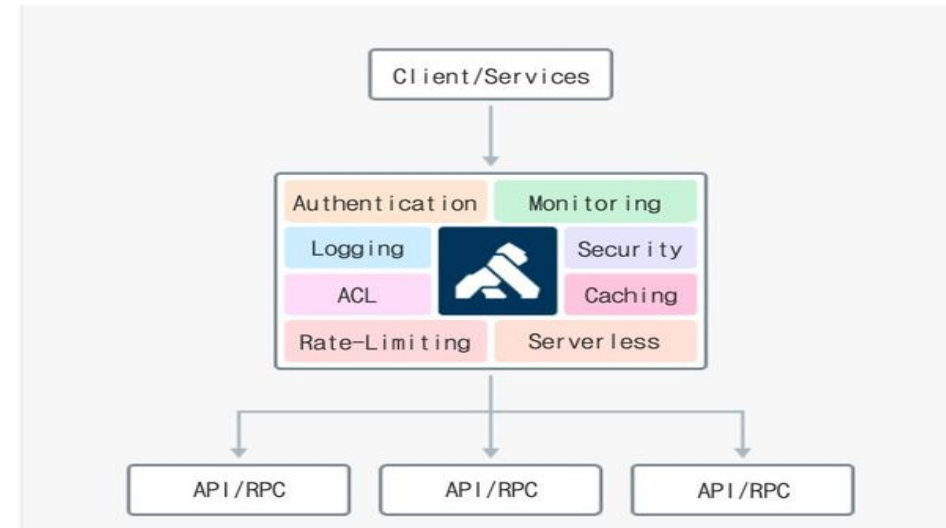
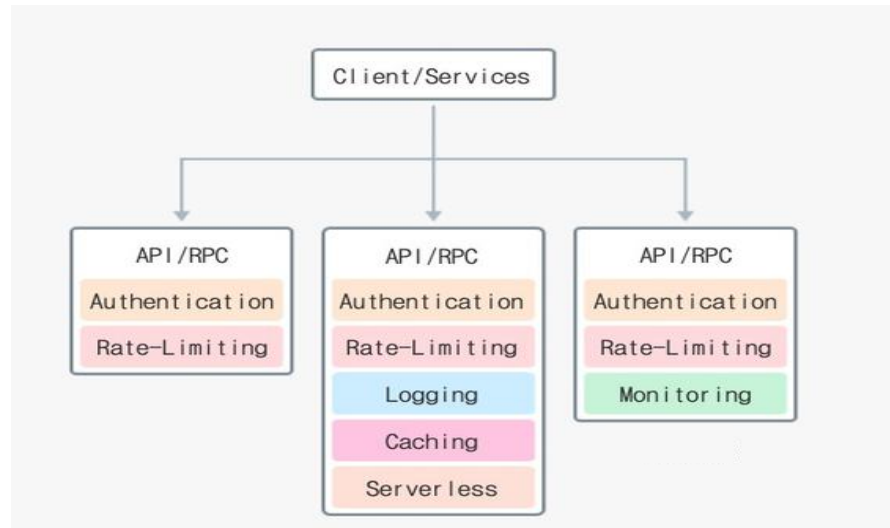


# Kong API Gateway



- ▶ Single point of entry to all applications. No need for multiple ports and IPs
- ▶ Gateway acts as a reverse proxy, forwards requests to applications so manages what is available to the public
- ▶ All interaction between containers and gateway are private so don't need encryption unless you really need the extra security
- ▶ User connects to kong public ip with required path. e.g. **52.151.73.23:80/yin/yin.html** to get the main page and resources
- ▶ Kong will forward to the private ip of the yinyang vm. e.g. **http://10.0.0.9:4000** on the Azure private network to the yin container
- ▶ The container forwards the request onto the node application at 3000
- ▶ The containers communicate between each other on a private docker network if on the same VM. e.g. yin makes a request to yang at its DNS name: yang:3001
- ▶ The gateway and applications are independently scalable

# Kong high level features



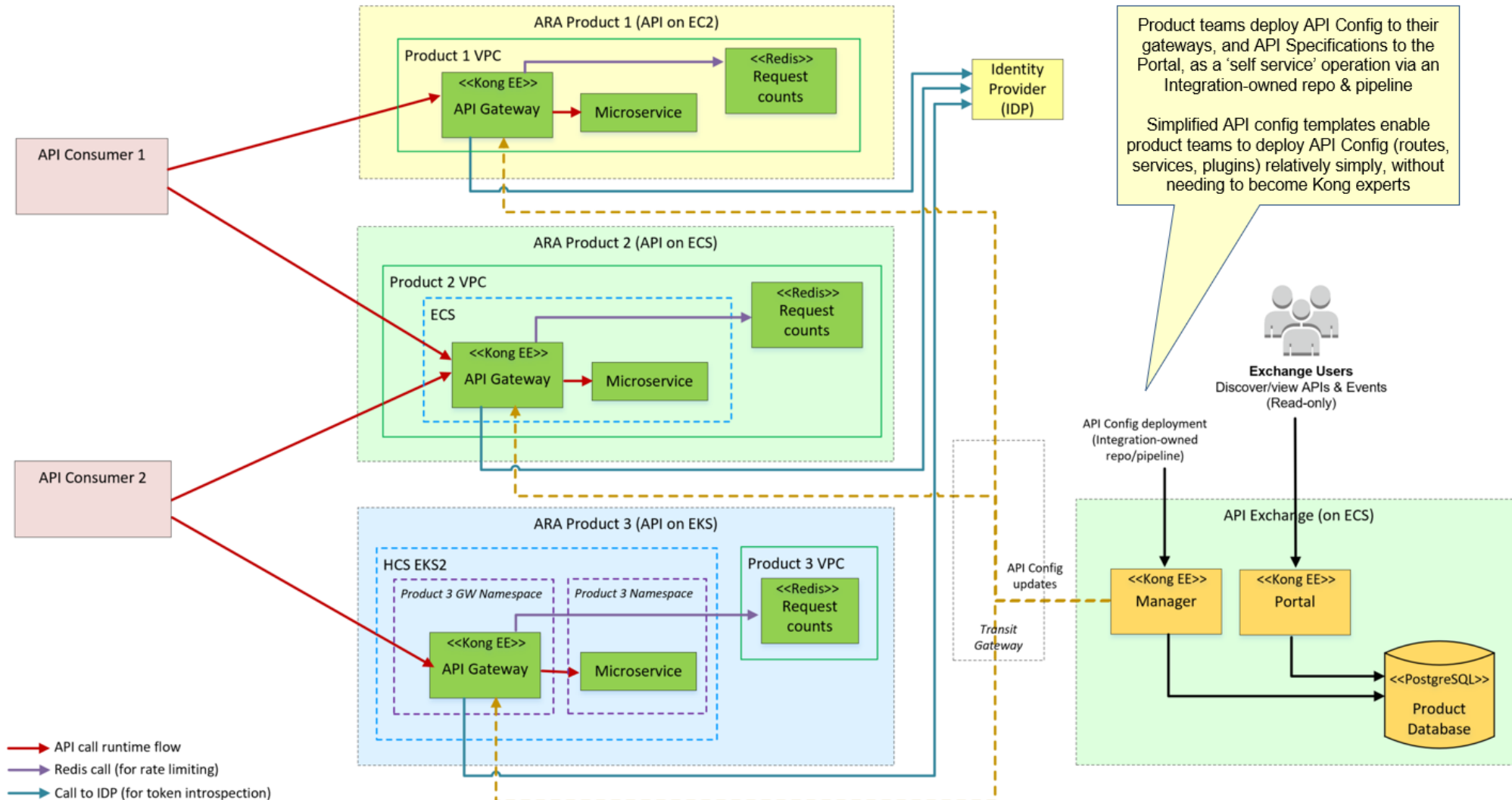
- ▶ Avoid replicating functionality such as rate limiting and authentication - add using Kong plugins
- ▶ Single entry point so https can be implemented once
- ▶ Aggregation of API calls - e.g. one request to multiple api calls, Kong can orchestrate that - reduces bandwidth
- ▶ Acts as a reverse proxy to protect services from the internet - users never see the endpoint IP
- ▶ Operates in two different modes: with database or without (db-less mode)
  - ▶ Database - typically Postgres or Cassandra - centrally stores all config info
  - ▶ In db-less, config is stored on each node - simpler deployment as no database dependency
  - ▶ db-less uses declarative code, db approach uses imperative via API

# API Gateway properties

- ▶ Request Routing: route client requests to the appropriate microservices based on the URL or request parameters
- ▶ Load Balancing: load balancing evenly among multiple instances of the same microservice
- ▶ Security: authentication, authorization, and security in one place
- ▶ Rate Limiting: restricts the number of requests per min. Prevents abuse, protects backend resources
- ▶ Throttling: sets rate of processed requests, often by slowing down responses (queue) to smooth traffic spikes
- ▶ Caching: cache microservice responses. Reduce backend load and improve response times
- ▶ Logging and Analytics: provide logging to analyse traffic patterns and API usage
- ▶ Aggregation: aggregate multiple microservice responses into a single client response
  - ▶ reduces latency for client as they can make one rather than several individual calls

- 
- ▶ Disadvantages - not many but they need to be considered
    - ▶ Single point of failure. Must be designed to be resilient and scalable - multi-node with load balancer
    - ▶ Extra hop may introduce a bit of extra latency
    - ▶ Adds complexity to the architecture - needs configuring, monitoring and maintenance (e.g. patching)
    - ▶ Extra cost: cost to run and knowledgeable staff to understand the technology so adds to cost
    - ▶ Security, this is an advantage but, as all services are accessible from here, it increases the attack surface for hackers so security of the gateway must be carefully designed

DWP



- ▶ The gateway is relatively simple to use:

routes are defined in kong.yaml and associated with a forwarding address

```
- name: yin
 url: http://10.0.2.4:4000
 routes:
 - name: yin
 paths:
 - /yin
 strip_path: false
```

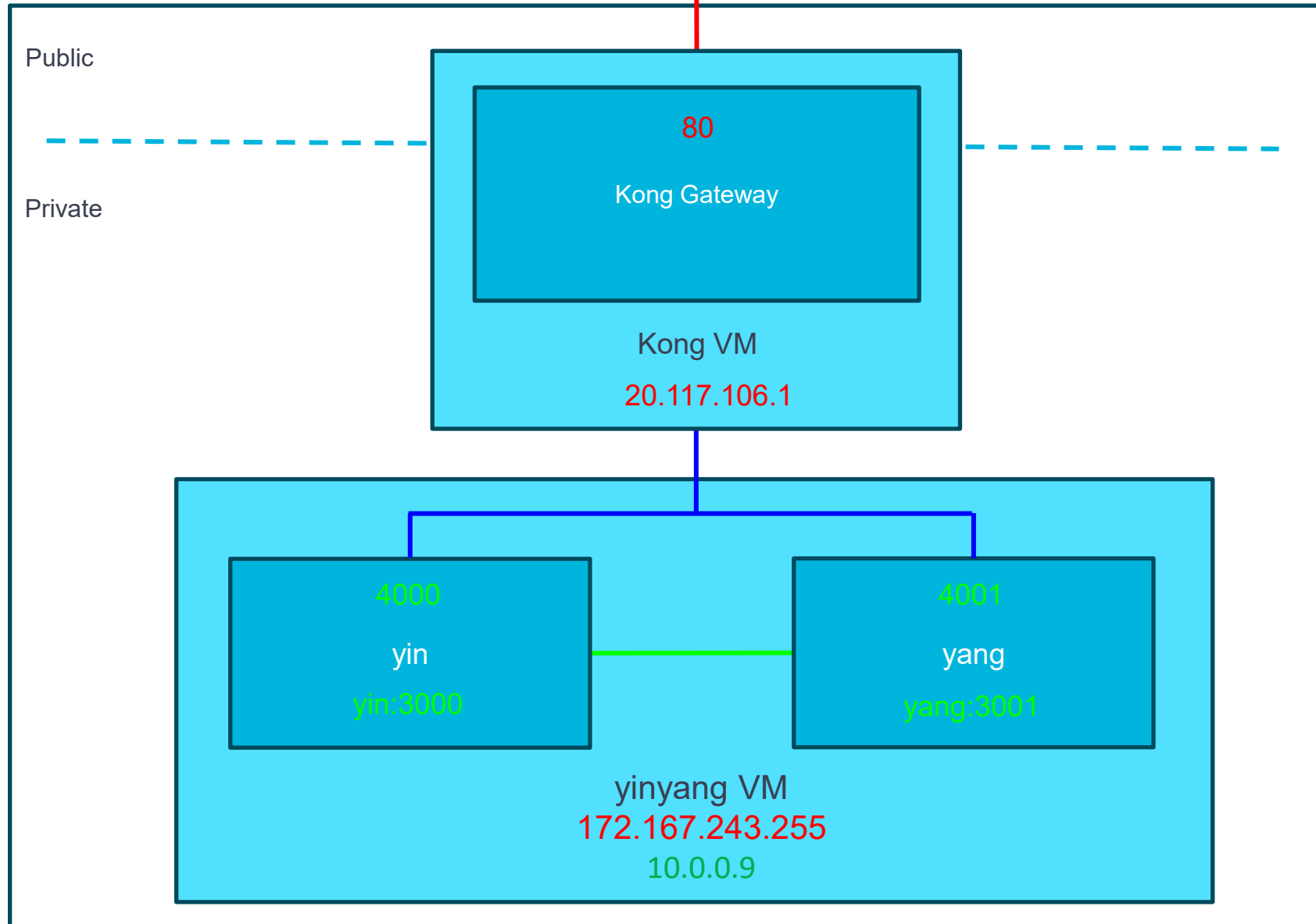
```
- name: yang
 url: http://10.0.2.4:4001
 routes:
 - name: yang
 paths:
 - /yang
 strip_path: true
```

- ▶ http:// 52.151.73.23/yin is matched on path then origin changed to http://10.0.2.4:4000/yin because strip\_path is false
- ▶ http://52.151.73.23/yang/home is forwarded to http://10.0.2.4:4001/home because strip\_path is true so yang is removed
  - ▶ strip\_path is **true by default** and will remove from the source path, the path element specified in the path property
  - ▶ Can be useful for example, for versioning:  
kongIP/api/v1-1-2/books/:media could be routed to 10.0.2.3/books/:media by specifying /api/v1-1-2 in the paths and strip\_path: true
  - ▶ kongIP/api/v1-2-0/books/:media could be routed to 10.0.3.5/books/:media as kong will match the source path to the paths definition to find a match, then remove the source path and replace the IP address
- ▶ So, it's up to you what you use - strip\_path enables flexibility in how traffic is forwarded
- ▶ For more complex path filtering, regex can be used

Public internet

Azure private network

Docker private network





# Event driven architecture

## RabbitMQ and Kafka



# Learning objectives

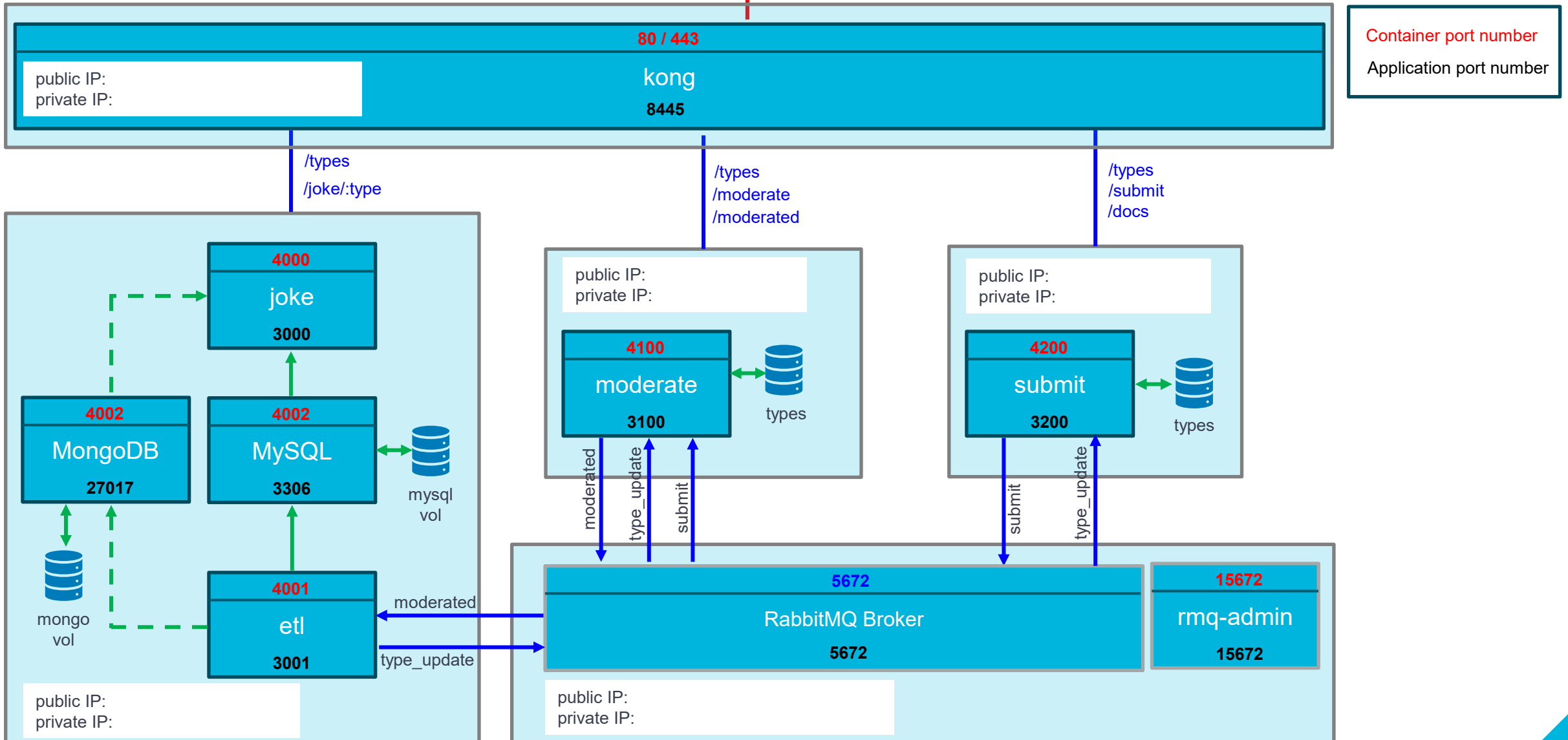
- ▶ Upon successfully completing this section, you should be able to:
  - ▶ Explain the benefits and limitations of various messaging and architectural patterns including:
    - ▶ request-response
    - ▶ point-to-point asynchronous
    - ▶ publisher-subscriber
    - ▶ event notification
    - ▶ event streaming
    - ▶ event carried state transfer (ECST)
  - ▶ Design and draw and explain a basic event-driven architecture
  - ▶ Differentiate between "Thick" and "Thin" events and explain the advantages and disadvantages
  - ▶ Explain the properties and basic operation of Kafka
  - ▶ Compare an event driven architecture to a message driven architecture
  - ▶ Compare the key properties of Kafka to RabbitMQ
  - ▶ Explain how Kafka partitions enable parallel processing

# Event-driven



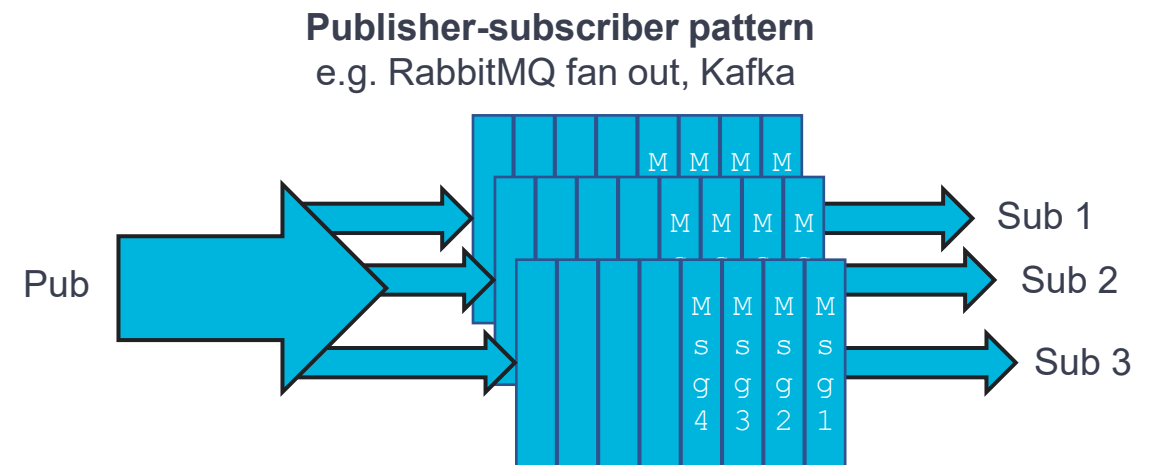
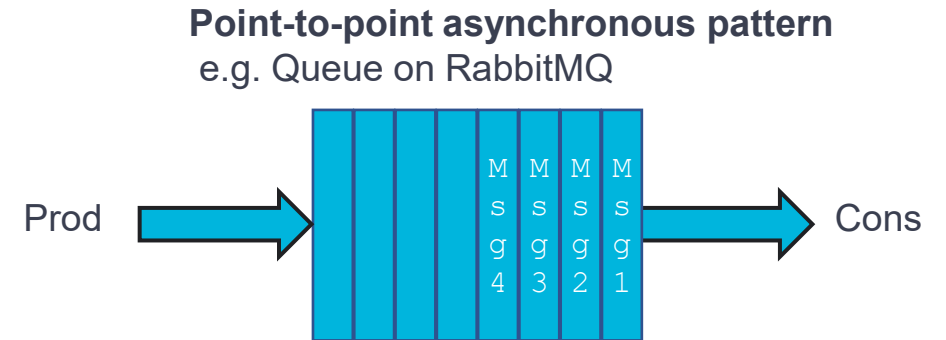
Public internet  
Azure private network  
Docker private network

Virtual machine  
Application container  
Rabbitmq combined container



# Inter-microservice messaging

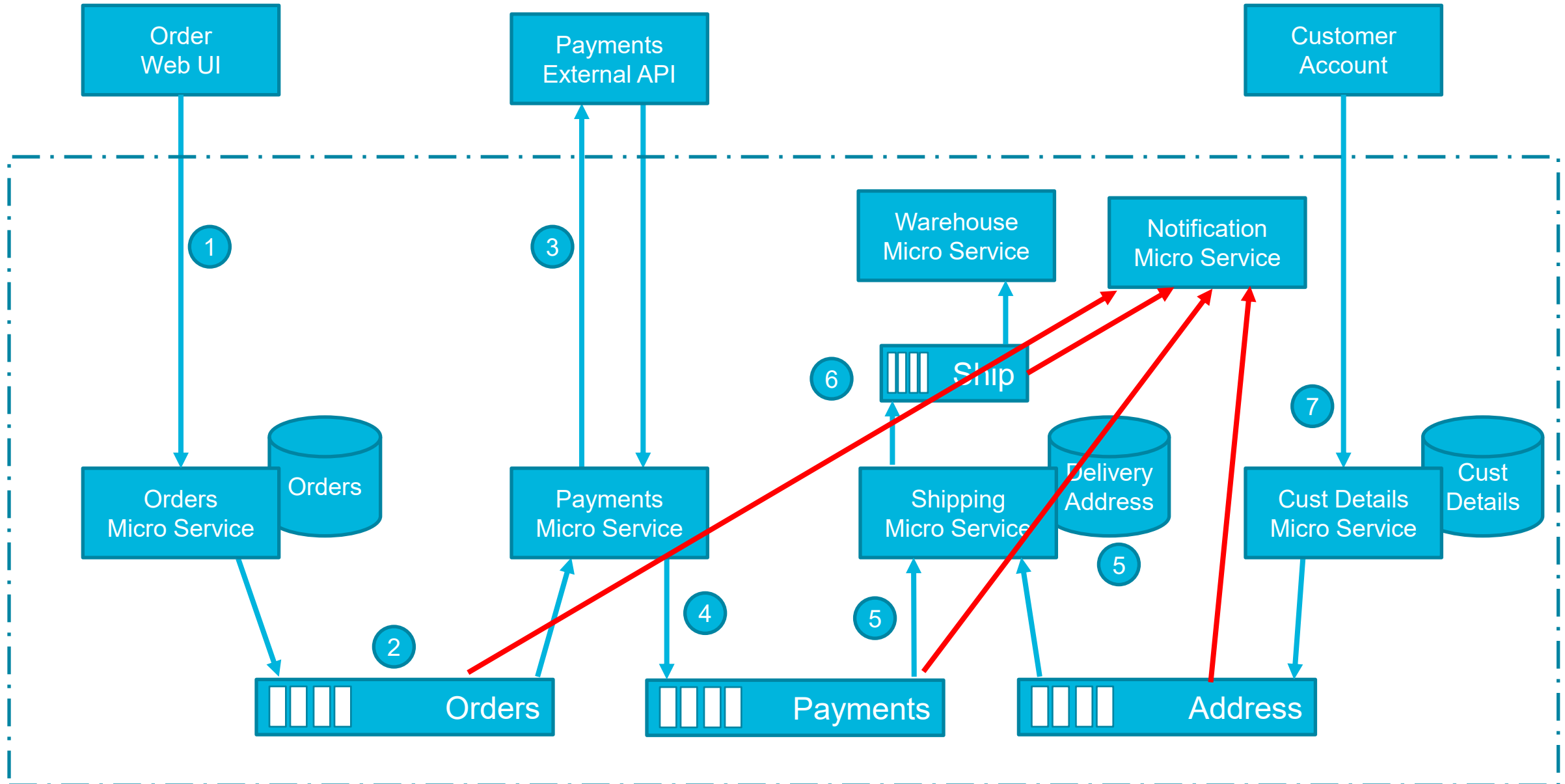
- ▶ As a reminder of what we have already done
- ▶ **Request-response pattern.** http is the simplest messaging pattern, and given there is no intermediary isn't usually even classified as a messaging pattern. More of an interaction pattern. Tight coupling.
- ▶ Point-to-point (p2p) asynchronous message pattern. Typically implemented with a queue. Single producer and single consumer. Loose coupling.
- ▶ Publisher-subscriber pattern. Typically shown as one publisher and many subscribers but also supports many publishers if required. Often implemented using RabbitMQ in a topic configuration if we want subscribers to subscribe to specific topics. The exchange will filter on topic, e.g. PR\* in a postcode all go to queue1 for example.
- ▶ Alternatively, RabbitMQ can be configured with a fanout exchange, where every message published to the exchange is delivered to every bound queue, and therefore to every subscriber. Unlike topic exchanges, no routing key or filtering is applied.
- ▶ This reminder is because this is a popular exam topic, and I will use RabbitMQ fanout in pub-sub mode for event processing

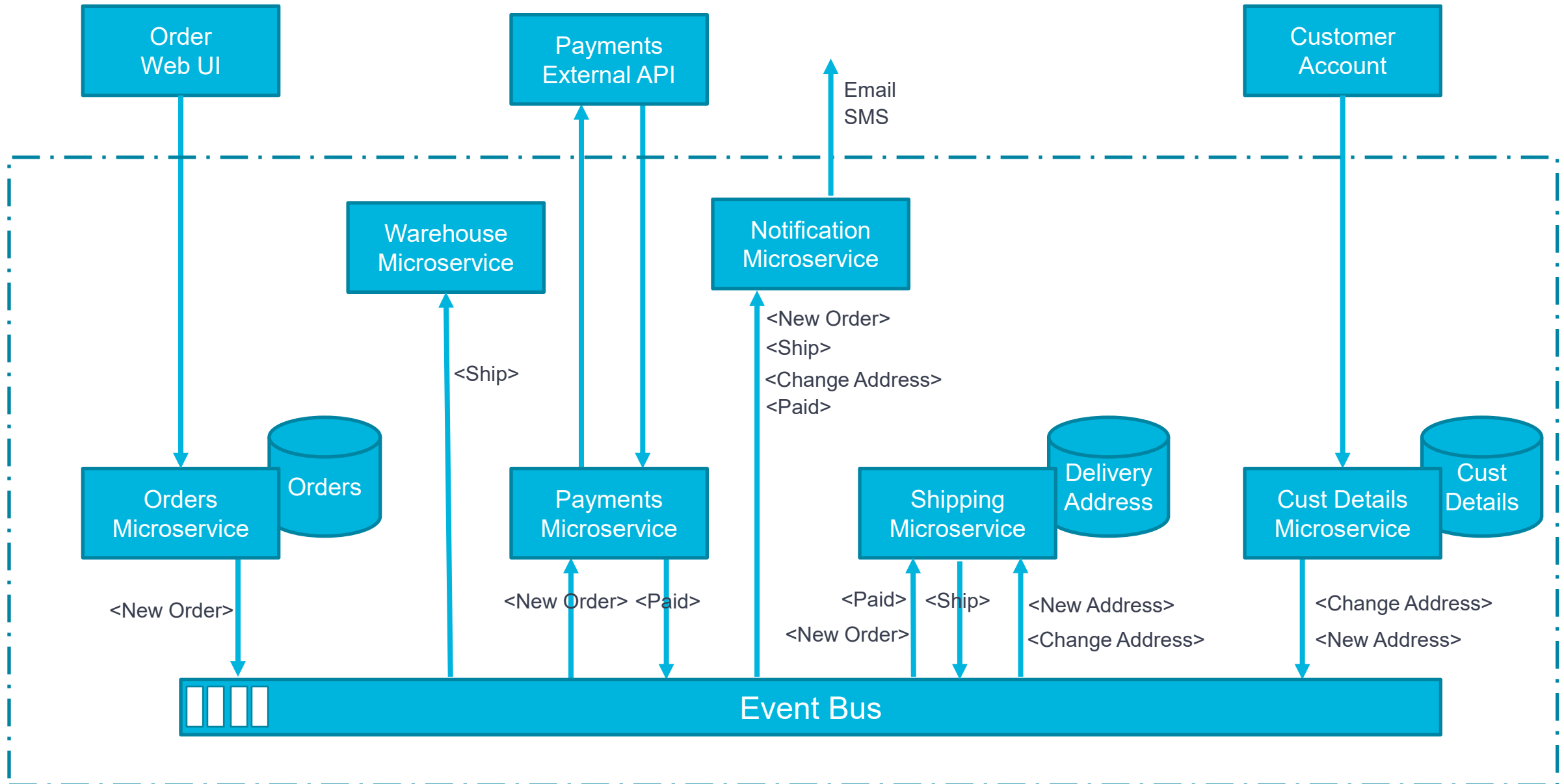


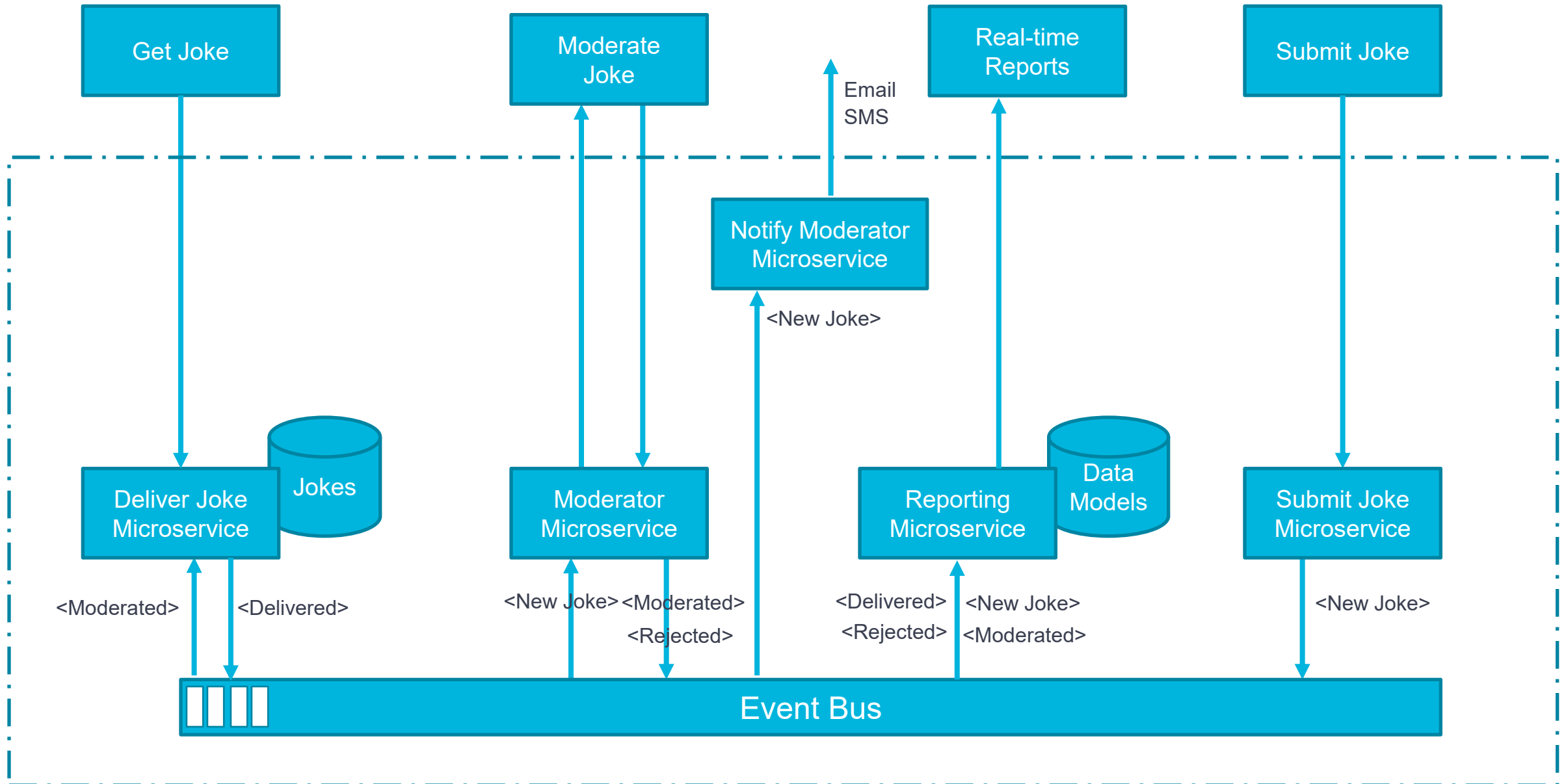
# What is an event

- ▶ A significant change in state or occurrence within a system that is captured and communicated as a discrete message to notify other components
  - ▶ Represents state change: e.g., "OrderPlaced", "PaymentProcessed", or "UserLoggedIn"
  - ▶ Immutable: An event is a statement of fact - we don't change it
  - ▶ Context: Data describes the change. e.g., orderID, userDetails, timestamp or whatever
  - ▶ Asynchronous: generally published to an event bus or broker (e.g. Kafka, RabbitMQ)
- ▶ Use case:
  - ▶ In an event-based microservice architecture, messaging tends to act like a state machine. A change in one service may need to be known by others
  - ▶ For example, services tend to have a master data store - often called a System of Record (SoR) - single source of truth
  - ▶ In a distributed system such as microservices, other microservices may need access to that SoR data
  - ▶ We'll look now at a basic event-driven architecture and discuss event-messaging

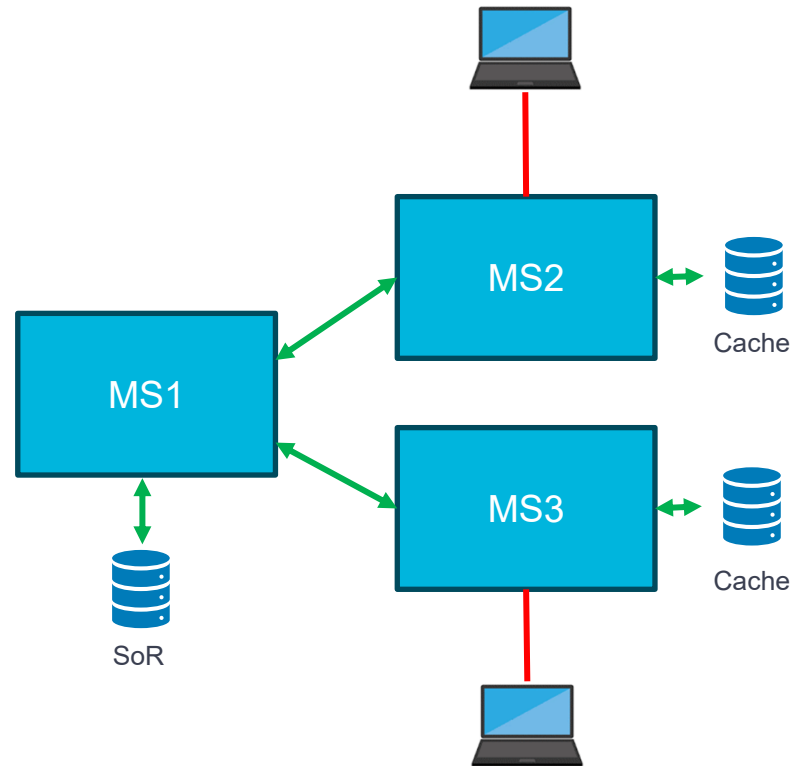
# Event driven architecture







# Point to point synchronous messaging



- ▶ User submits data to MS2
- ▶ MS2 writes data to the SoR. If it needs access to the data, it reads from the SoR
- ▶ MS3 writes data to the SoR. If it needs access to the data, it reads from the SoR
- ▶ If a user needs data, MS1 or MS2 need to access the SoR. This is inefficient and tightly coupled - if the SoR fails, the whole system fails
- ▶ MS2 and MS3 could keep a copy of the SoR for resilience and performance but it would need to regularly poll the SoR for changes. e.g. MS3 writes to the SoR, MS2 doesn't know about it until it polls. This is a bit more resilient to SoR failure
- ▶ This is the pattern used in a previous assignment

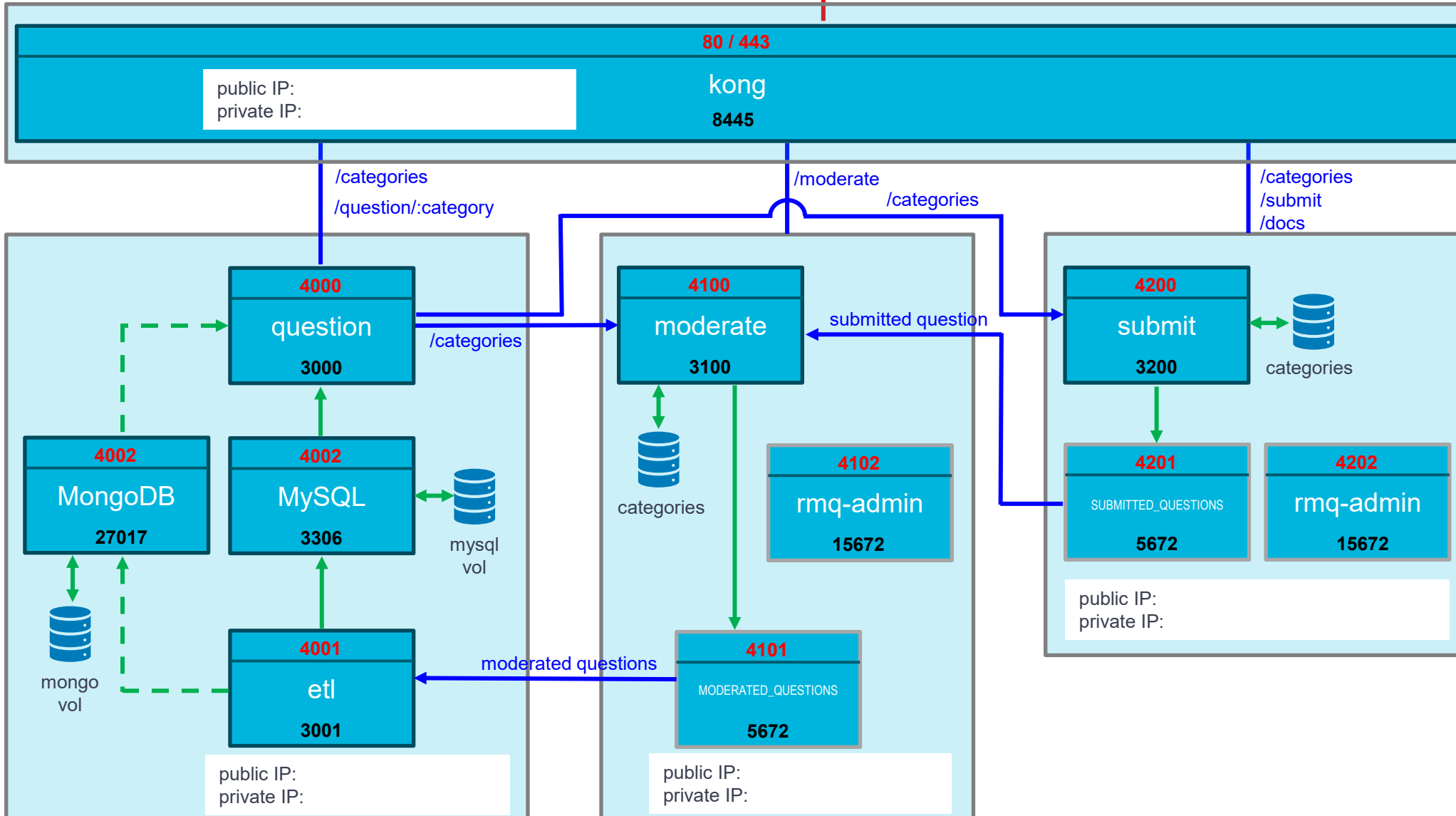




- Public internet
- Azure private network
- Docker private network

- Virtual machine
- Application container
- Rabbitmq combined container

Container port number  
Application port number

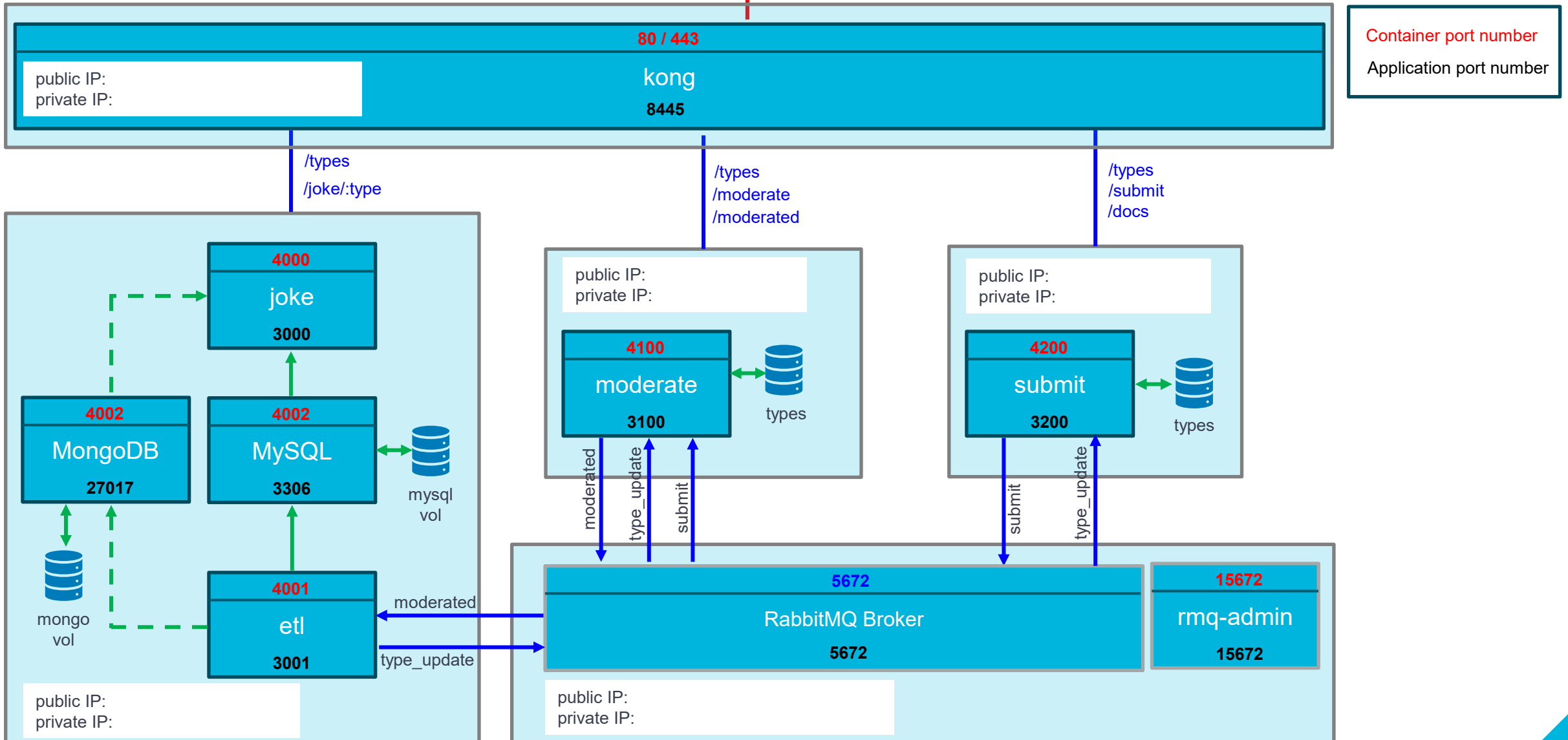


# Event-driven



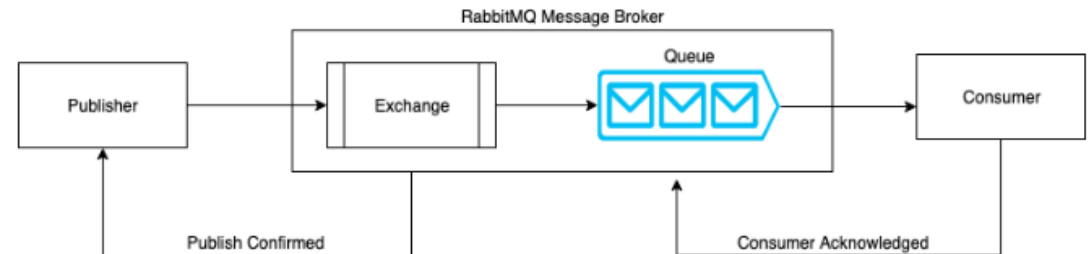
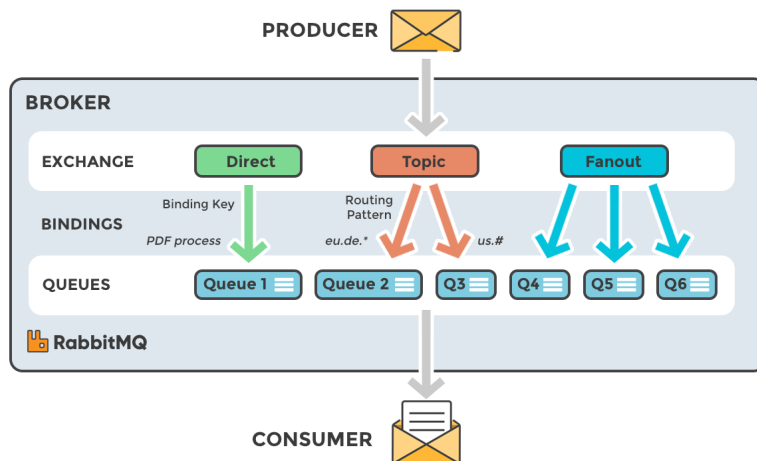
Public internet  
Azure private network  
Docker private network

Virtual machine  
Application container  
Rabbitmq combined container



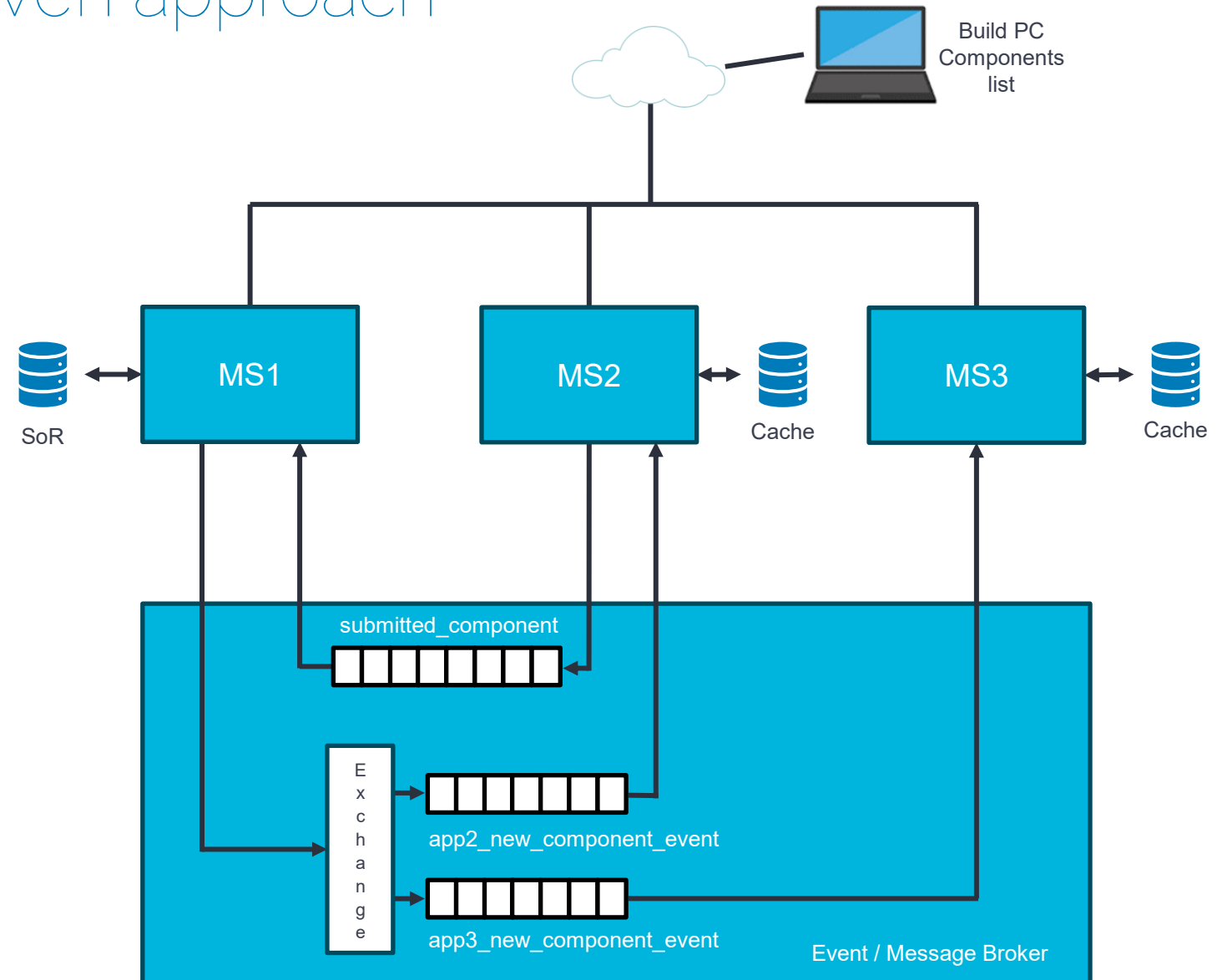
# RabbitMQ message broker recap

- ▶ Let's create an event driven architecture using RabbitMQ
- ▶ A broker is an intermediary between the message producer and message consumer - like post-office
  - ▶ Asynchronous pattern - producer doesn't wait for the consumer to acknowledge. Contract is with the broker
  - ▶ Loose coupling between producer and consumer
  - ▶ Consumer can consume from the queue in its own time - queuing smooths spikes to prevent consumer overload
  - ▶ Supports single queue one producer to one consumer
  - ▶ Supports publisher / subscriber message distribution with multiple subscribers to the same messages
  - ▶ Generally pushes data from the broker to the consumer
  - ▶ Generally message deleted from the queue when the consumer acknowledges receipt of the message
  - ▶ RabbitMQ is resilient as can run a cluster of brokers so caters for failure of one or more nodes

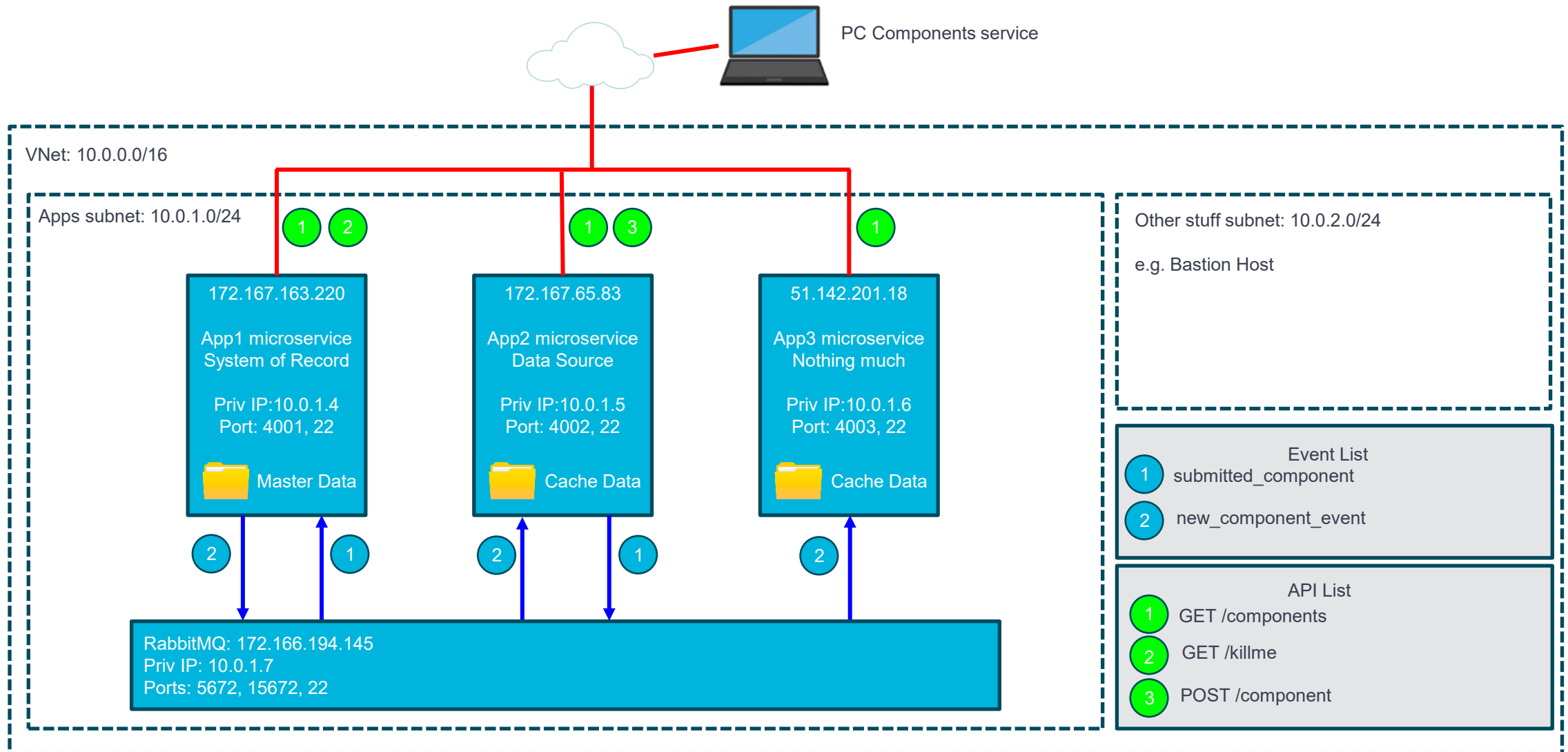


# Asynchronous Event-driven approach

- ▶ Simple use-case: add components to a list with no duplicates. All services informed of new entries
- ▶ User sends data to MS2
- ▶ MS2 updates the SoR (MS1) via a queue
- ▶ Message pattern is Asynchronous point-to-point
- ▶ MS1 decides if it should update the SoR and if so, creates an update event with the data
- ▶ Event is published (to an exchange in RabbitMQ)
- ▶ MS2 and 3 subscribe to their queue
- ▶ Queues push the event to each of MS2 and MS3
- ▶ They update their local cache with event data
- ▶ The data reaches eventual consistency
- ▶ This arch pattern is called "**Event-Carried State Transfer**" (ECST) using pub-sub messaging
- ▶ Users access data from cache so quicker
- ▶ System is resilient if SoR or any microservice fails
- ▶ Data transfer loosely coupled in both directions
- ▶ An alternative to RabbitMQ is Kafka
- ▶ Clearly more complex than point to point but more resilient to failure of one or more microservices



# Detailed architectural diagram



- ▶ "Thick" or "fat" events, carry all the data (state). Used in "**Event-Carried State Transfer**" architectural pattern to update subscribers to the event and its state
- ▶ Thick event benefits include
  - ▶ Potentially lower latency as no additional API call to get data as it's all in the event
  - ▶ Asynchronous so provides loose coupling and hence more resilience to failure
  - ▶ Producer does not need to scale as more consumers are added
- ▶ Challenges
  - Likely not a practical solution when integrating to legacy without lots of nugatory legacy improvement
  - All data is always sent even if not all subscribers want all the data so network and broker load can be increased
  - Load on the broker has increased so that needs to be considered

# Thick and Thin events

- ▶ A "thin" event only carries a small data package such as time of the event and some sort of ID to identify the event. The data payload is not sent with the event. This approach is used in "**Event Notification**" pattern
  - ▶ producer publishes an event indicating that something happened, maybe also an ID and timestamp
  - ▶ state, i.e., the data, requires consumers to query the producer to obtain the data, typically via an API call
- ▶ Thin event benefits include
  - ▶ less data transferred via the broker
  - ▶ consumer can ask for whatever data it needs from one or more APIs rather than take all of it
  - ▶ popular for legacy monolithic systems that don't support thick events but do have API functionality
- ▶ Challenges of thin events
  - ▶ Synchronous API call adds tight coupling so reduces resilience
  - ▶ Slower as an event is processed then the API is called
  - ▶ APIs can suffer due to concurrent burst-type requests from subscribers requesting data on receipt of the event
- ▶ Both patterns are valid depending on the use case. If consumers can tolerate synchronous dependence on a producer, thin messages or synchronous calls may be preferable; whereas fat events are better suited to asynchronous, decoupled integration where resilience and autonomy are prioritised e.g. distributed systems

# Kafka

- ▶ Popular open source event-based pub / sub streaming application
  - ▶ Developed by LinkedIn to track user activity on its platform
  - ▶ It's now been made open source as part of the Apache foundation and managed by Confluent
  - ▶ Very popular in event driven microservice architectures
  - ▶ Used by Netflix, Spotify, Uber, LinkedIn, Twitter and many other well-known enterprises
  - ▶ As in RabbitMQ, Kafka uses producer/consumer model and supports prod/cons and pub/sub message (event) pattern
  - ▶ In principle it is very similar to RabbitMQ but is functionally different in that it uses persistent logs instead of queues
- ▶ It captures and stores events
  - ▶ Events such as IoT sensor data, Web-site click stream, track and monitor delivery vehicles etc., are captured in real-time into a log - i.e. append-only read-only storage so maintains sequence and can't be changed
  - ▶ As in MQ, there can be multiple producers and multiple consumers
  - ▶ Anyone interested in the Payments topic would subscribe to that topic
  - ▶ Events are stored however you want, text, xml, whatever as data is serialised like in MQ



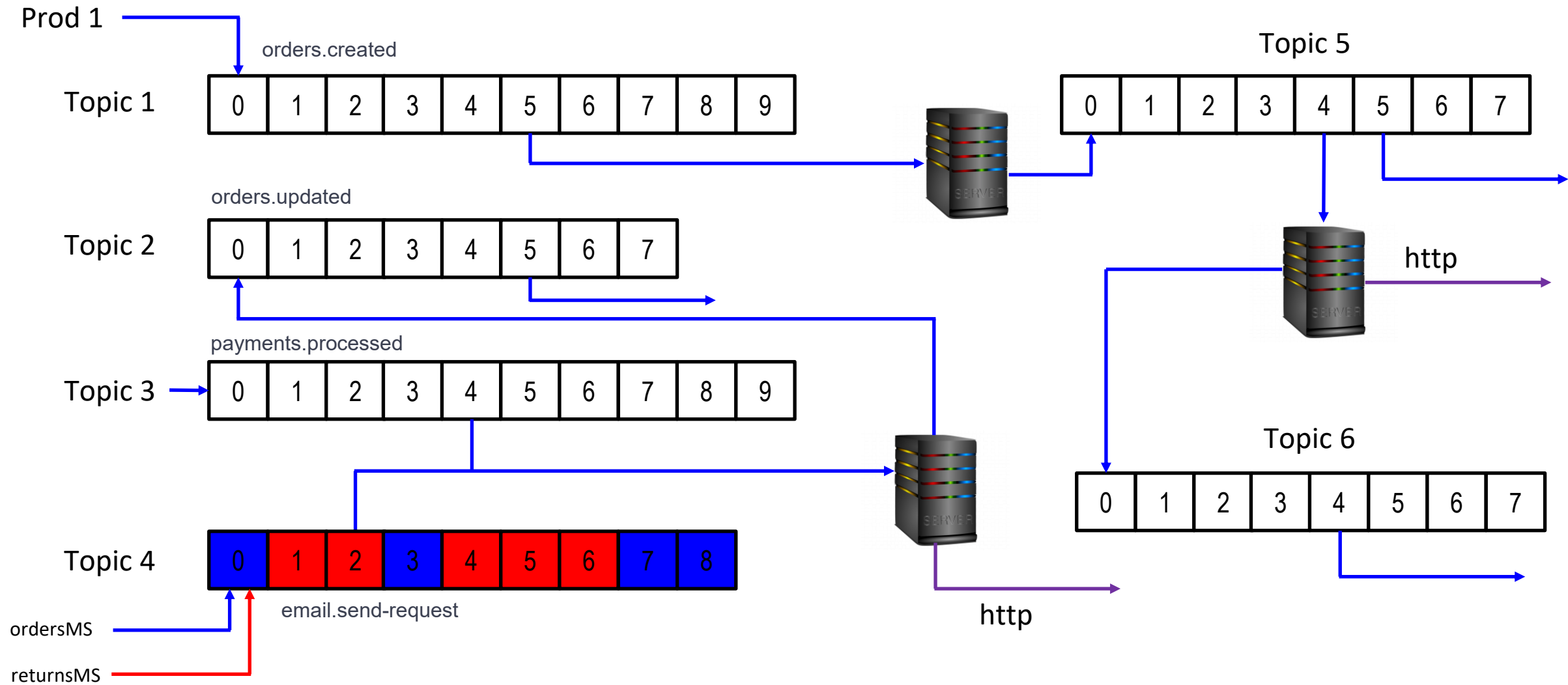


▶ **Kafka is a distributed application**

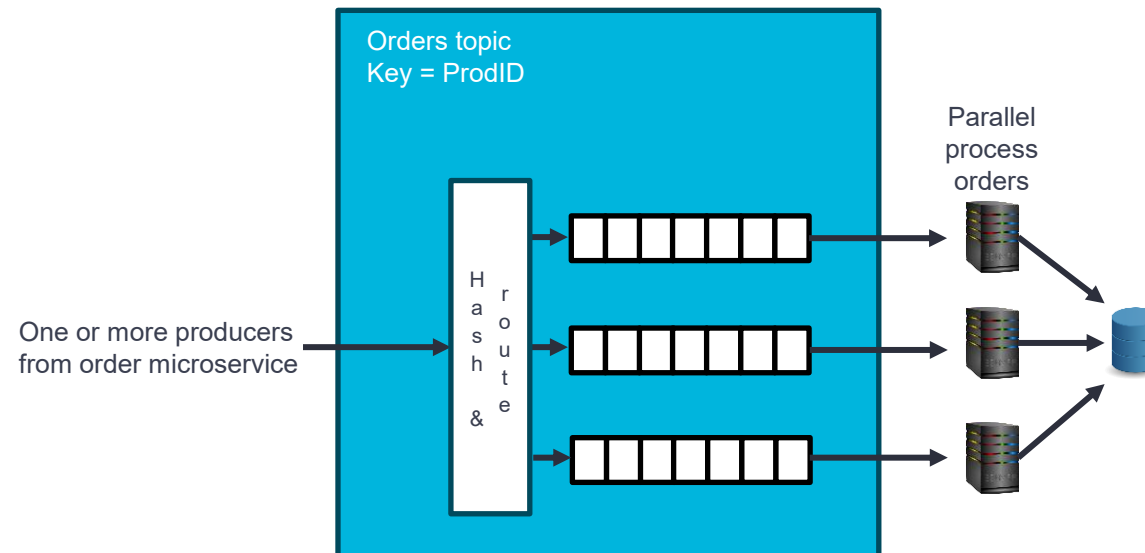
- ▶ Fault tolerant, horizontally scalable, enables parallel processing across multiple nodes

▶ **Kafka and Queue key differences**

- ▶ A basic queue has one producer. Kafka can have multiple producers as can MQ by using an exchange
- ▶ A basic queue has one consumer. The queued message is deleted once read. Kafka can have many subscribers (consumers in Kafka Speak) to the same topic because reading the data doesn't cause it to be deleted (removed from the queue) as data is not deleted once read; Kafka events are persisted. A bit like reading an array or file using a pointer for each consumer
- ▶ Kafka data retention can be set from minutes through to infinity, by default 1 week, or by storage size
- ▶ Unlike RabbitMQ which can be configured as a priority queue, Kafka stores events / messages as an immutable append log - i.e. in the order in which they come in is maintained
- ▶ Kafka consumers have an index into the log (basically a file pointer) which increments as they read messages. If they need to go back, they can change the read index on the log - basically like accessing and reading a log file
- ▶ Rather than replicating data in a one queue per subscriber, Kafka has one copy of the events and subscribers read the events using their own offsets into the event stream
- ▶ Kafka supports data streaming. e.g. data can be streamed in through a "Kafka connect" connector to a streaming data source such as a click stream, IoT, video etc. The stream can be processed by a consumer, then output to another topic which is streamed back out through "Kafka stream". e.g. IoT data streams in, the consumer processes it in real time, event by event, and streams the output through another topic to real time analytics service or whatever
- ▶ You can add whatever events you want to a topic from multiple producers, the consumers will filter on what they want to see based on a key of some sort, e.g., customerID



- ▶ The example showed a single topic partition - there are various options to improve performance and resilience
- ▶ When a cluster is created you can specify the number of partitions (previous discussion was one), say we want 3
- ▶ The producer will provide a key, say a user ID. Kafka will hash that to choose the partition:  $p = \text{hash}(\text{key}) \% 3$
- ▶ This ensures strict ordering of users or products or whatever based on a key this enables parallel processing
- ▶ A Kafka cluster can support any number of topics. LinkedIn has 10s of thousands of topics, millions of partitions



- ▶ Events are not pushed by the broker as in RabbitMQ, a consumers polls the partition - this sounds inefficient but ...
- ▶ A subscriber tries to fetch data. If there is none, it can deliberately block until there is data then it is returned
- ▶ It can also request a return only when a specified number of events have landed
- ▶ So we need to use JavaScript async calls or Java threads

# MQ and Kafka basic comparison

| Feature          | Kafka                                                      | RabbitMQ                                                                        |
|------------------|------------------------------------------------------------|---------------------------------------------------------------------------------|
| Primary Role     | Distributed event streaming platform                       | Message broker for reliable messaging                                           |
| Message Storage  | Persistent log (messages retained for a configurable time) | Transient - messages deleted                                                    |
| Ordering         | Guaranteed. No prioritisation                              | Guaranteed but can be config for priority queue - prod can set a priority level |
| Replay           | Supported (consumers can re-read old events)               | Not supported after ack                                                         |
| Throughput       | Extremely high (millions of messages/sec)                  | Lower than Kafka, optimized for reliability (uses ack for pub and sub)          |
| Consumer Scaling | Consumer groups to partitions for parallel processing      | Multiple queues or competing consumers on one queue - e.g. round robin          |
| Protocol         | Custom (binary)                                            | AMQP (Advanced Message Queuing Protocol)                                        |
| Serialization    | Opaque byte arrays                                         | Opaque byte arrays                                                              |

# Summary of patterns

## Messaging and architectural patterns

- ▶ **Request–Response:** Synchronous. Caller waits for a reply. Tight coupling. Not formally a message pattern
  - ▶ Typical tech: HTTP, REST, gRPC
- ▶ **Point-to-Point Asynchronous Messaging:** One message, one consumer. Asynchronous. Message lives in a queue. Loose coupling
  - ▶ Typical tech: RabbitMQ queue, Azure Service Bus Queue, AWS Simple Queueing Service (SQS)
- ▶ **Publish–Subscribe Messaging:** Typically, one message many subscribers. Asynchronous. Loose coupling. Subscribers receive independent message copies
  - ▶ Typical tech: Kafka topics, RabbitMQ fanout/topic exchanges
- ▶ **Event Notification:** Event signals that something happened. Carries minimal data (e.g. ID only - thin event). Consumers typically calls synchronous API for state so creates tight coupling
- ▶ **Event Streaming:** Continuous sequence of events. Replayable so ordering matters
  - ▶ Typical tech: Kafka, Amazon Event Hubs, AWS Kinesis
- ▶ **Event-Carried State Transfer:** How a distributed system propagates state between services without creating runtime coupling. Typically uses pub-sub pattern but not a requirement. Event carries the state (thick / fat events) - no callback needed
- ▶ **Event-driven architecture:** Components react to events, are loosely coupled, asynchronous, usually using pub/sub messaging
- ▶ Good Kafka intro video here: <https://youtu.be/QkdkLdMBuL0>

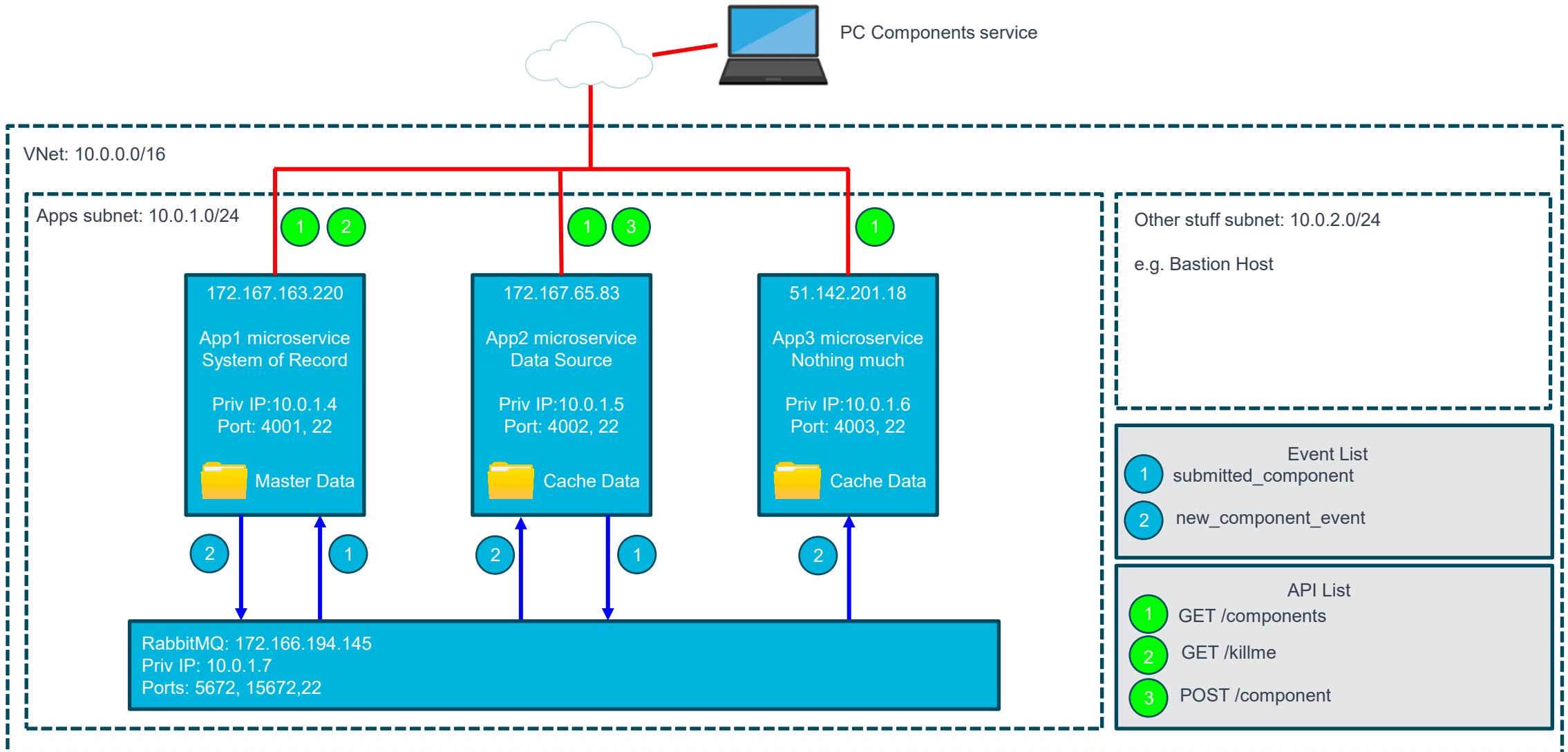


# Subnets & Bastion Host

# Learning objectives

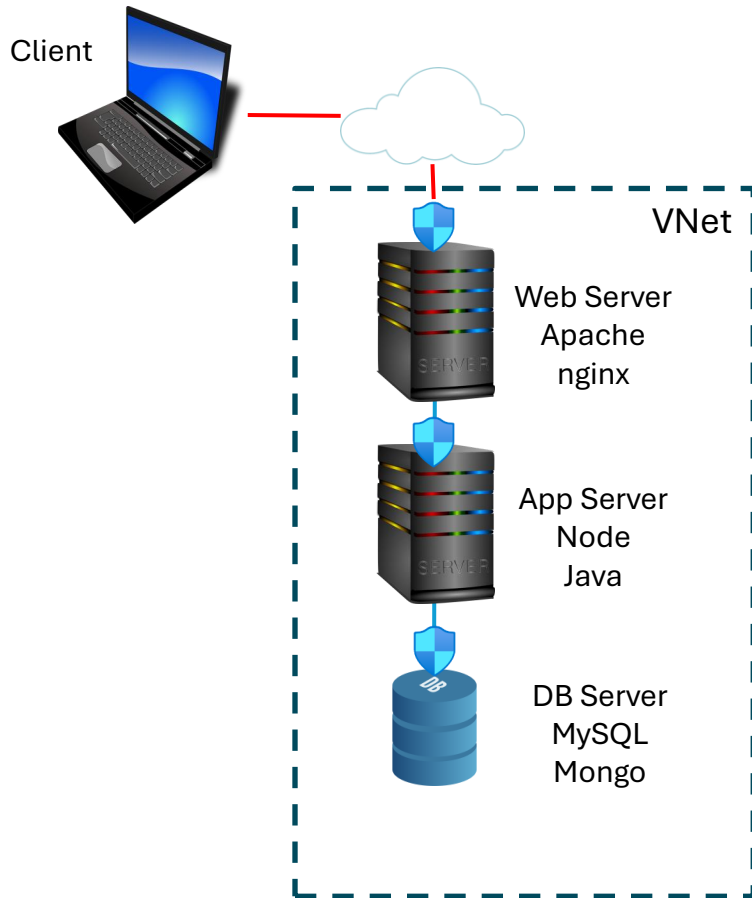
- ▶ Upon successfully completing this section, you will be able to:
  - ▶ Create subnetworks
  - ▶ Explain the benefit of subnetworks
  - ▶ Explain an architectural need for subnetworks
  - ▶ Create subnetwork security rules
  - ▶ Create a bastion host in the Azure portal to work with last week's demo or any VMs to enable admin access
  - ▶ Explain the purpose and properties of a Bastion host
  - ▶ SSH into any VM using redirects from the SSH config file
  - ▶ Explain virtual network peering
  - ▶ Peer two virtual networks across different regions

Could use Kafka here



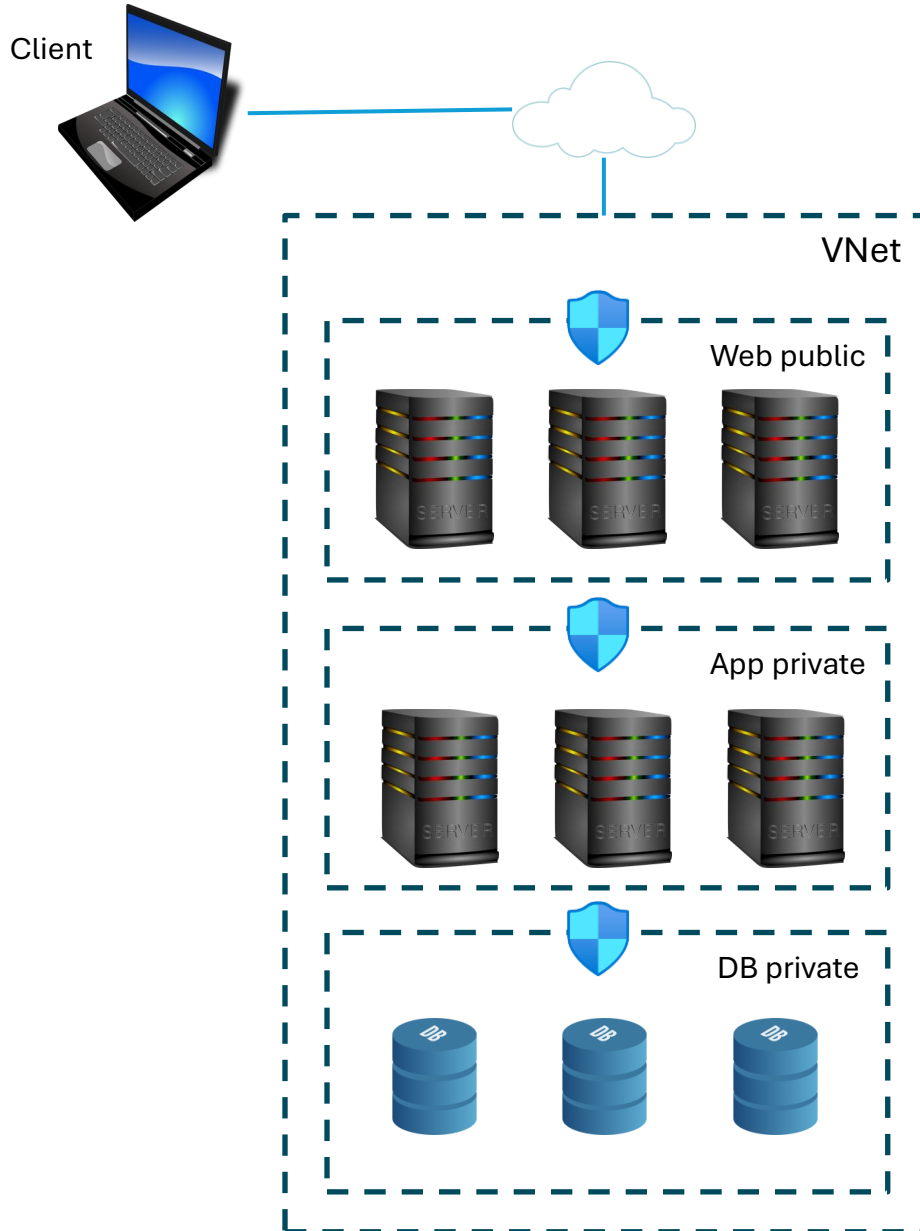


# 3-tier architecture



- ▶ If each tier runs on a VM, in Azure the default is:
  - ▶ Put all VMs in the same VNet - typically 10.0.0.0/16
  - ▶ Allocate public and private IP addresses to all VMs
  - ▶ Allocate NSG to each VM
- ▶ All are exposed to the internet via IP & network so easy targets
  - ▶ So remove the public IPs of the app server and database
  - ▶ But now we can't access them for OS updates etc
  - ▶ If a hacker gets through a vulnerability in the web server, they can access the app server and DB by guessing private IP addresses, but the "gold" is in the database tier
  - ▶ Each VM has an NSG so could limit access. e.g. App server only allows http traffic from private IP 10.0.0.5 and only open port 3000
  - ▶ That works, but if we scale either tier, we have to deal with IPS

# subnet isolation in 3-tier architecture

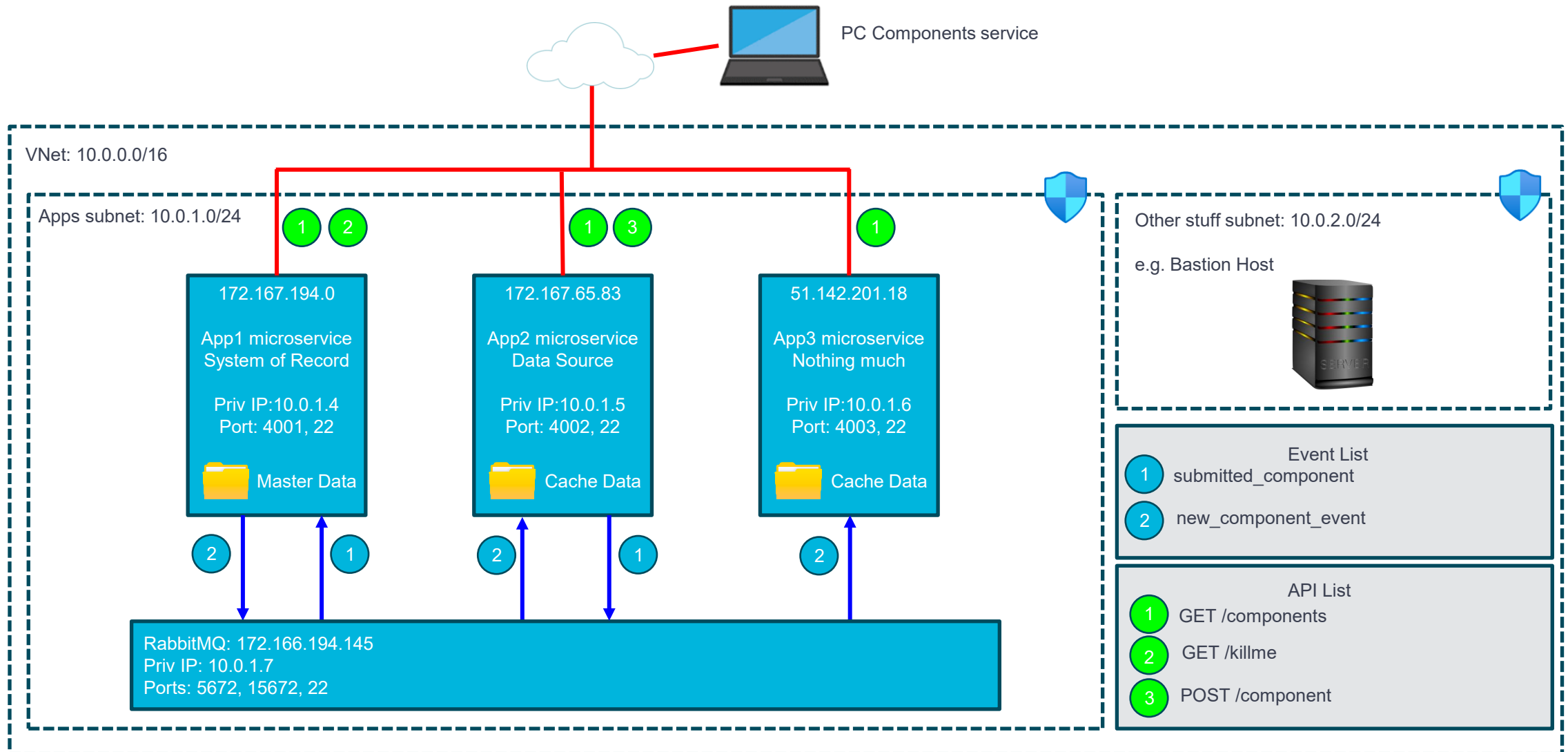


- ▶ We may have many VMs if we scale the solution so many NSG rules need to be set if left at the VM level
- ▶ Use sub-networks and attach an NSG to the sub-net
  - ▶ All rules apply to all resources in the subnet
  - ▶ Set rules to only allow traffic from specific subnet
- ▶ Great. But we still can't access the VMs for admins in the App or DB tier as there are no public IPs
- ▶ And, we don't really want admin access via the web tier even through port 22 as anyone in the world can reach it and have a go at trying to get through
- ▶ Although a microservice is not strictly a 3-tier architecture, it is composed of multiple tiers, API routing, business logic and data so can be protected in the same way using subnets and NSGs

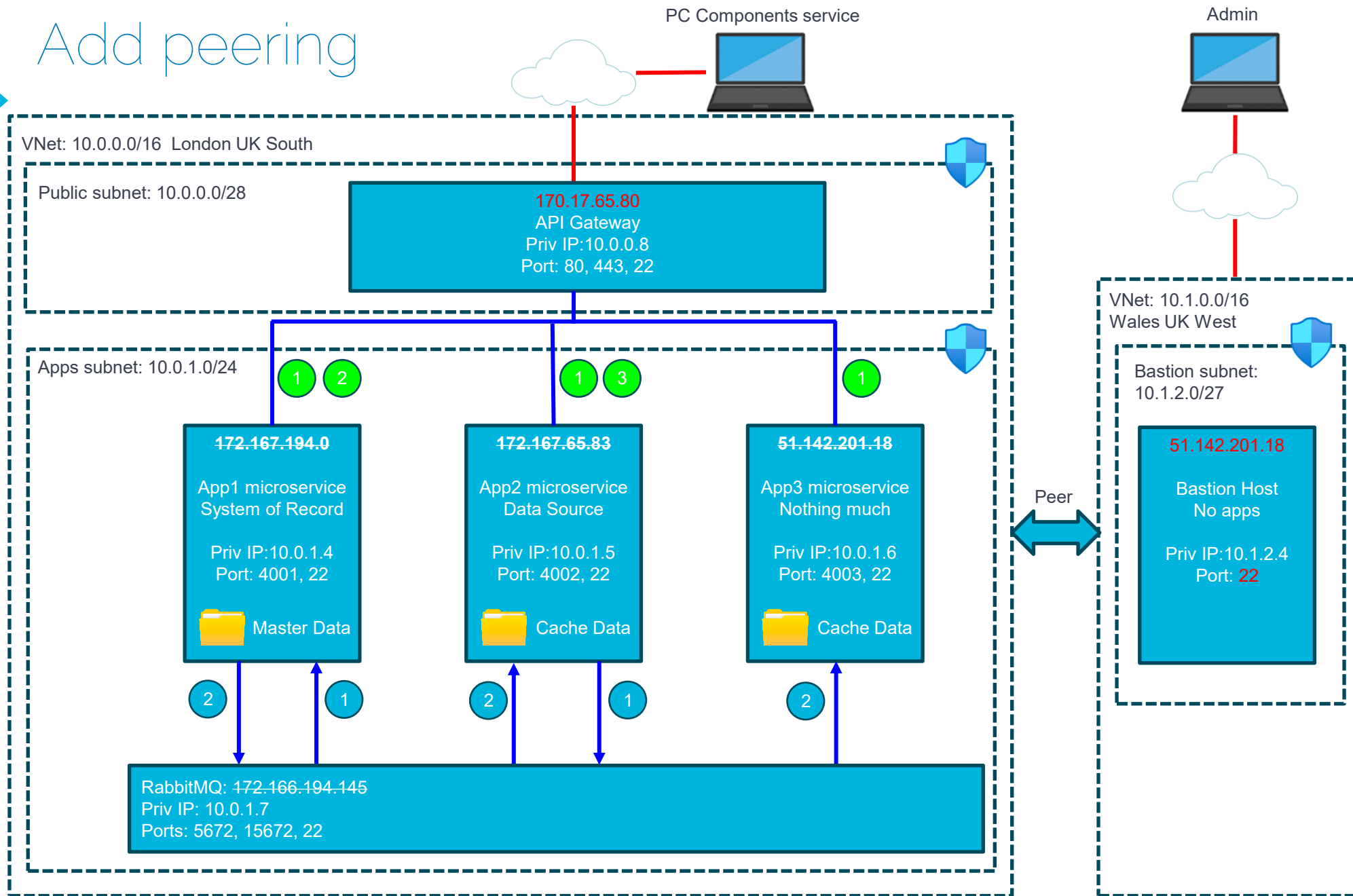
# Bastion Host (Jump host or Jump box)

- ▶ We need a solution to securely provide admin access to our VMs
- ▶ A bastion or jump box is used by system admins and maybe engineers to access VMs on a private network
- ▶ Modern fully cloud-native and zero-trust architectures are diverging from traditional bastion hosts, but bastion hosts remain widely used and are still popular in many cloud and hybrid environments and very much for on-prem
- ▶ The Bastion is exposed to the internet as a single entry-point for administered machines
- ▶ It is the only system component accessible to SSH (or RDP) from outside the private network
- ▶ The assumption is that this VM WILL be attacked so it is hardened, monitored and limited functionality
  - ▶ Hardened means, removing non-essential software, including in the OS, closing all ports other than SSH, all non-essential services, no root login via SSH, limited users, password authentication disabled (must use key), admin IP whitelist
  - ▶ Basically, hardening is the deliberate removal of trust, functionality, and exposure until only what is essential remains
- ▶ All VMs not needing internet access (e.g. web server, API Gateway do) do not have public IP addresses
- ▶ Firewall / NSG limit access only from the Bastion host for admin traffic e.g. ssh on port 22
- ▶ Bastion is placed in its own subnet
- ▶ Reduces the attack surface but does provide a "Once in you can access all" so needs protection
  - ▶ Intrusion detection
  - ▶ Suspicious behaviour alerts
  - ▶ Command and user access auditing
  - ▶ Does not store private keys - simply forwards SSH to the target system

# Add a Bastion



# Add peering



## NSG rules

### Bastion NSG

Dir: Inbound  
Source: public IP (x.x.x.x/32)  
Protocol: TCP  
Dest port: 22  
Dest addr: 10.0.0.0/28, 10.0.1.0/24  
Action: Allow  
Public admin -> bastion

### API Gateway NSG

Dir: inbound  
Source: Internet (0.0.0.0/0)  
Protocol: TCP  
Dest ports: 80, 443  
Action: Allow  
anywhere -> Kong

### Apps subnet NSG

Dir: inbound  
Source: 10.0.0.0/28 (API GW)  
Protocol: TCP  
Dest ports: 4001-4003, 22  
Action: Allow  
Kong -> Microservices

Dir: inbound  
Source: 10.1.2.0/27 (API GW)  
Protocol: TCP  
Dest ports: 4001-4003, 22  
Action: Allow  
Kong -> Microservices

All outbound use the defaults  
as we can send outbound  
traffic anywhere



# Infrastructure as Code

# Learning objectives

- ▶ Upon successfully completing this work, you should be able to:
  - ▶ Explain the term "Infrastructure as Code" and its benefits
  - ▶ Demonstrate an awareness of various IaC approaches
  - ▶ Explain imperative and declarative programming and when you may use one over the other
  - ▶ Explain the key benefits of Terraform over other approaches
  - ▶ Explain the key components in Terraform such as resources and provisioners
  - ▶ Explain a given piece of terraform code
  - ▶ Explain the concept and benefits of a CI/CD pipeline and general code deployment automation
  - ▶ Deploy infrastructure using Terraform

# Basic infrastructure

- ▶ Creating a VM / EC2 from the portal seems so easy. But a lot goes on in the background and various defaults are used; this is ok for experimenting but not in real design:
  - ▶ A VM type and size needs to be specified
  - ▶ A VM OS image needs to be specified
  - ▶ We need a system disk for the VM image
  - ▶ We need a network within which our VM can communicate
  - ▶ VM needs a network Interface Card for network connection
  - ▶ We need firewall to allow only ports we are using
  - ▶ We need a sub-networks for efficiency and security
  - ▶ VM needs a private and public IP address
- ▶ All this gets created using defaults and you can see them in the resource group, but for automated deployment and system design control, **we** need to specify these things

| <input type="checkbox"/> Name ↑↓                                                                                                                           | Type ↑↓                |
|------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------|
| <input type="checkbox"/>  junk                                          | Virtual machine        |
| <input type="checkbox"/>  junk                                          | Network Interface      |
| <input type="checkbox"/>  junk                                          | Network security group |
| <input type="checkbox"/>  junk                                          | Public IP address      |
| <input type="checkbox"/>  junk                                          | Virtual network        |
| <input type="checkbox"/>  junk_OsDisk_1_ec2f444defcc4fa281868c12c4be... | Disk                   |

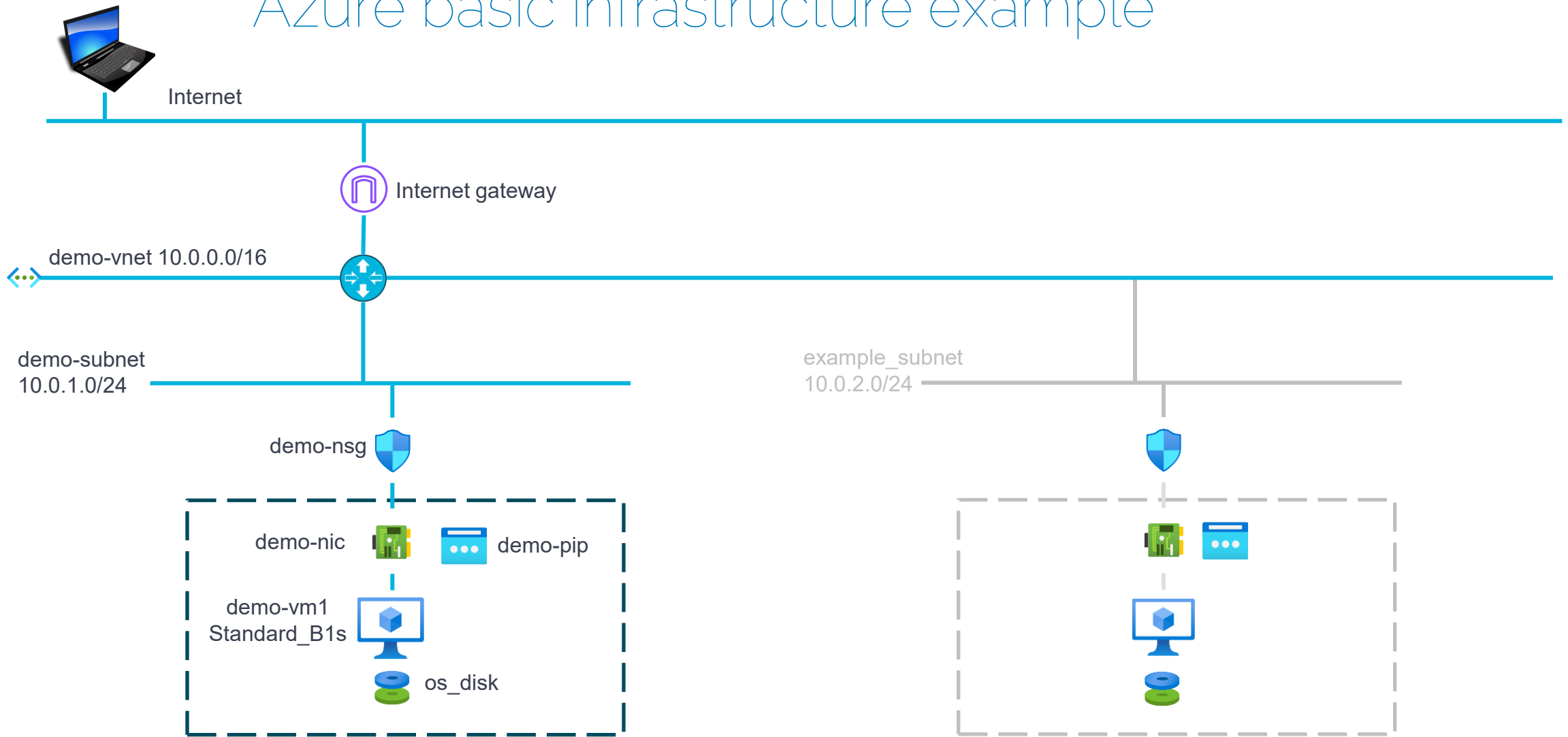


# Deleting resources

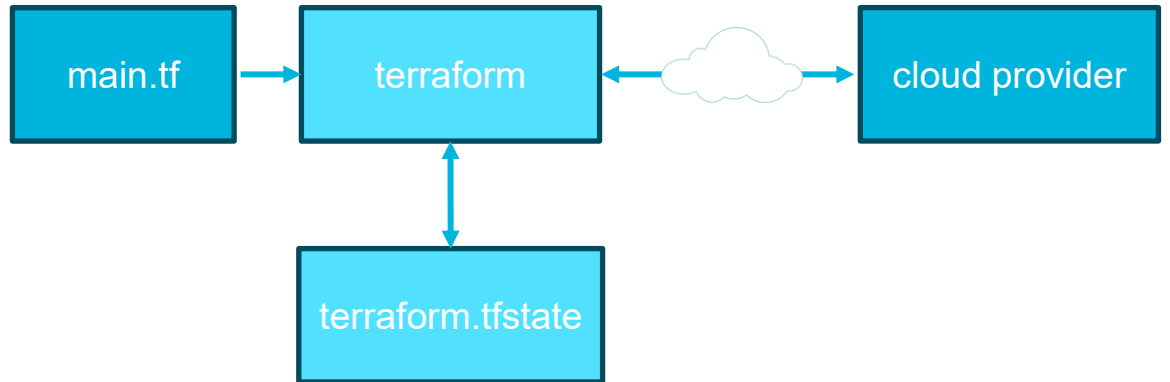
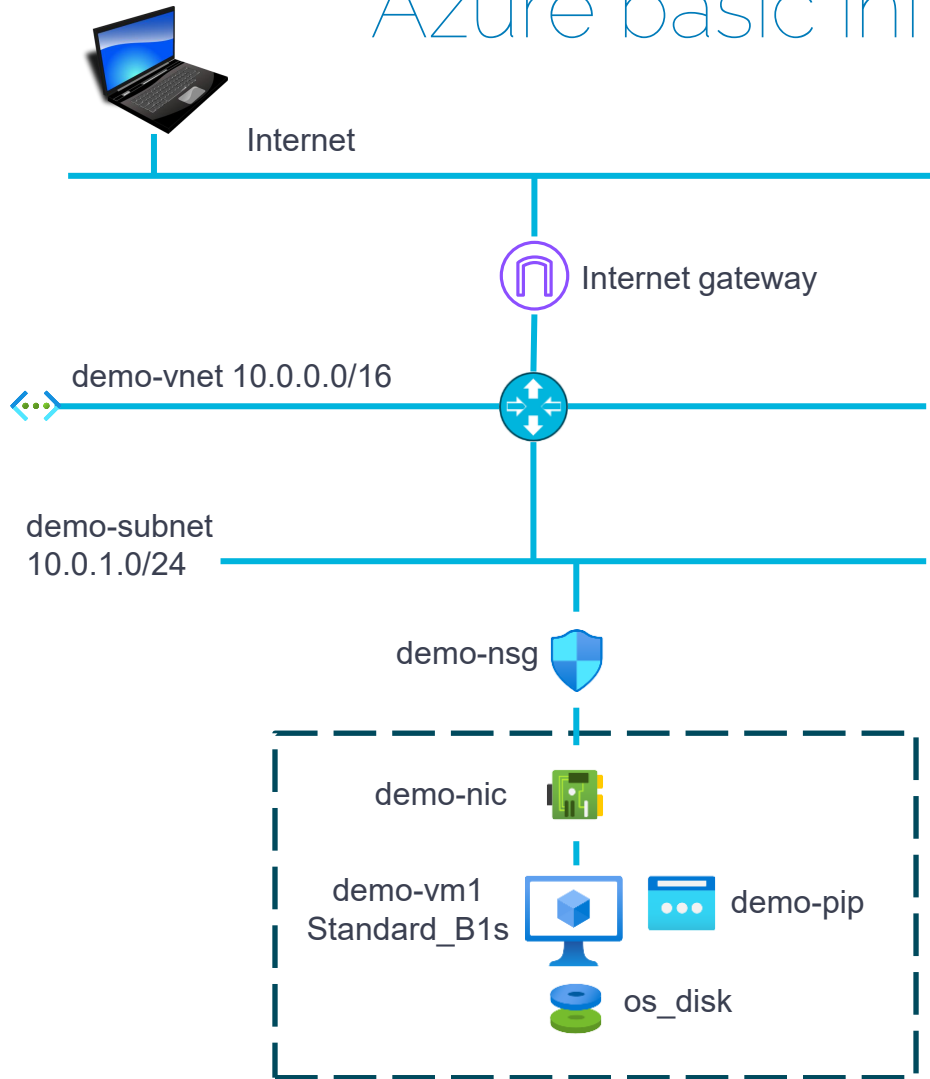
- ▶ If you choose to delete all the resources, say to save cost, then you want to recreate them again
  - ▶ Azure: delete the resource group - simply deletes all resources in the group. That may not be what you want.
  - ▶ AWS: you need to remember all the resources and manually delete each one
  - ▶ To recreate, you need to go through all the console steps again - imagine a more complex architecture
- ▶ Organisations need to automate the creation and destruction of their architecture and code deployment
- ▶ Cloud providers enable various ways of creating, managing and destroying resources programmatically
  - ▶ Software Development Kit: use your favourite programming language - e.g. call a function to create a VM
  - ▶ CLI: install the command line interface which provides various functions - e.g. to create a VM
  - ▶ Use a shell / batch script to execute command line commands imperatively
  - ▶ Use the cloud provider's built-in shell e.g. cloud shell to issue command line commands
  - ▶ Azure Resource Manager (ARM) templates for declarative IaC
    - e.g., In resource manager | Export template creates a reusable template of your infrastructure
  - ▶ AWS CloudFormation templates for declarative IaC
  - ▶ Use third party software such as Terraform which is cloud provider agnostic **declarative** approach
- ▶ I've posted a couple of videos showing CLI and template approach. We will stick to industry approach of Terraform

Videos: Azure CLI IaC and Azure templates

# Azure basic infrastructure example



# Azure basic infrastructure example



- ▶ **terraform init** - check which provider then download plugin
- ▶ **terraform apply**
  - ▶ Builds dependency graph
  - ▶ Checks for changes in state
  - ▶ Deletes and creates resources as required
- ▶ **terraform destroy** - all resources in state destroyed

[demos/terraform/Azure/minimal\\_vm/main.tf](#)

# Terraform

- ▶ Just to do something relatively simple such as create a vm is messy using command line and scripts
- ▶ In industry we need more control of the infrastructure than using defaults
- ▶ We need reliable repeatable and maintainable automation
- ▶ Use something like Terraform - which is probably the most popular approach (not perfect but what is)
  - ▶ Declarative so you don't need to worry about the creation order (usually)
  - ▶ Don't need to wait for resource creation to move on e.g. to get IP as it is declarative - it will do it
  - ▶ Only one language to learn for any cloud provider - HashiCorp Configuration Language (HCL)
- ▶ Terraform is a module in its own right, so just an intro here due to its importance in industry
- ▶ None of us can remember all the commands of all the tools we use so use documentation
  - ▶ AWS: <https://registry.terraform.io/providers/hashicorp/aws/latest/docs>
  - ▶ Azure: <https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs>
- ▶ Terraform app is available at:
  - ▶ <https://developer.hashicorp.com/terraform/install>
  - ▶ Choose AMD64 - latest version for Windows

# Initial key components

- ▶ **Configuration file**
  - ▶ One or more .tf files containing Terraform code. Primary file typically called main.tf
  - ▶ Used to define terraform "things" like vm, nsg or whatever
  - ▶ This is a declarative as opposed to imperative approach - we declare the outcome and let it work out how to do it
- ▶ **Provider**
  - ▶ Defines plugin to connect to provider's API to construct & manage resources based on the declarative Terraform code
- ▶ **Resource (thing)**
  - ▶ Represents resource components such as VM, Storage, Network etc
  - ▶ Consists of:
    - ▶ Resource type: e.g. "aws\_instance" for an AWS ec2, "azurerm\_virtual\_machine" etc. These are defined by the provider's API
    - ▶ Resource name: a unique name for you to refer to in code. i.e., like a constant
    - ▶ Args and Attributes: configuration detail for a specific resource. e.g. instance\_type = "t2.micro",
- ▶ **Variable**
  - ▶ Like parameters passed into the file or between blocks within the file similarish to function parameters
  - ▶ Input variables: pass in values from cli, env vars or a variables file to, say, config a resource - e.g. VM type
  - ▶ Local values: used within the code to reduce repetition - like constants
  - ▶ Output values: extract values from configurations for display or pass into other config files - a bit like function return values
  - ▶ Various types: string, number, bool, list (like array), map (like json object)

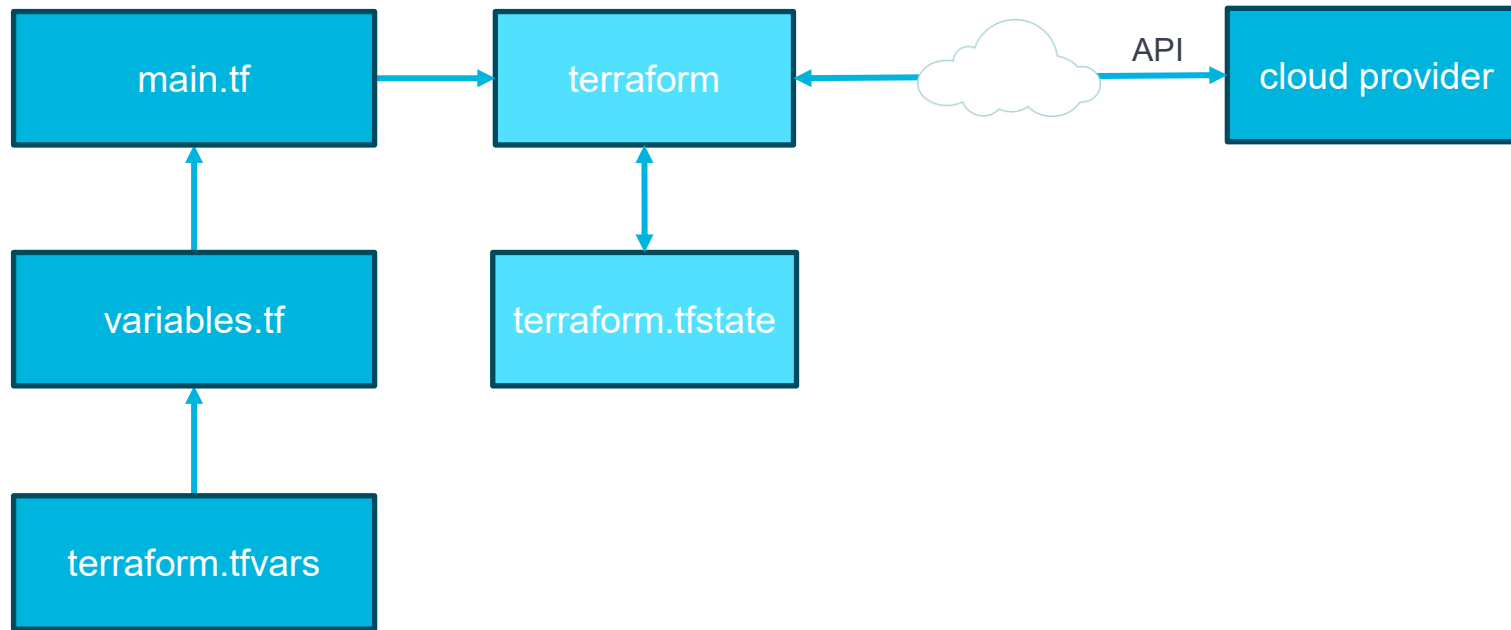
# Create a VM

- ▶ It seems complicated but think about it as being like building a computer, connecting components
- ▶ Script is reusable
- ▶ Can build or destroy all the components with a single command
- ▶ Declarative, like in docker compose, so create a target state
- ▶ Terraform creates a state file that holds the exact configuration created so it can be used to delete too
  - ▶ The state file will appear once you build something and is called **terraform.tfstate**
  - ▶ Terraform refers to this when you change something - may not have to rebuild everything
- ▶ Note: For first use, switching providers, switch directories and various other scenarios, you need:
  - ▶ **terraform init**
  - ▶ The environment will usually tell you if you need to do this. It will download a provider .exe file to use as the API integration point to the cloud provider. So you will see a new folder appear: .terraform
  - ▶ A new file appears: .terraform.lock.hcl this is used to ensure that the exact version of the provider used to create infrastructure is used in the future. e.g. if a team works on the same project, this file would be shared. Like the package-lock.json file
- ▶ **terraform apply** will execute code in the main.tf file - in fact all the .tf files in the current directory
- ▶ **terraform destroy** will delete everything defined in the state file

# Variables

- ▶ Don't hard-code properties such as image and instance type, use in and out variables. Generally, store in variables.tf
- ▶

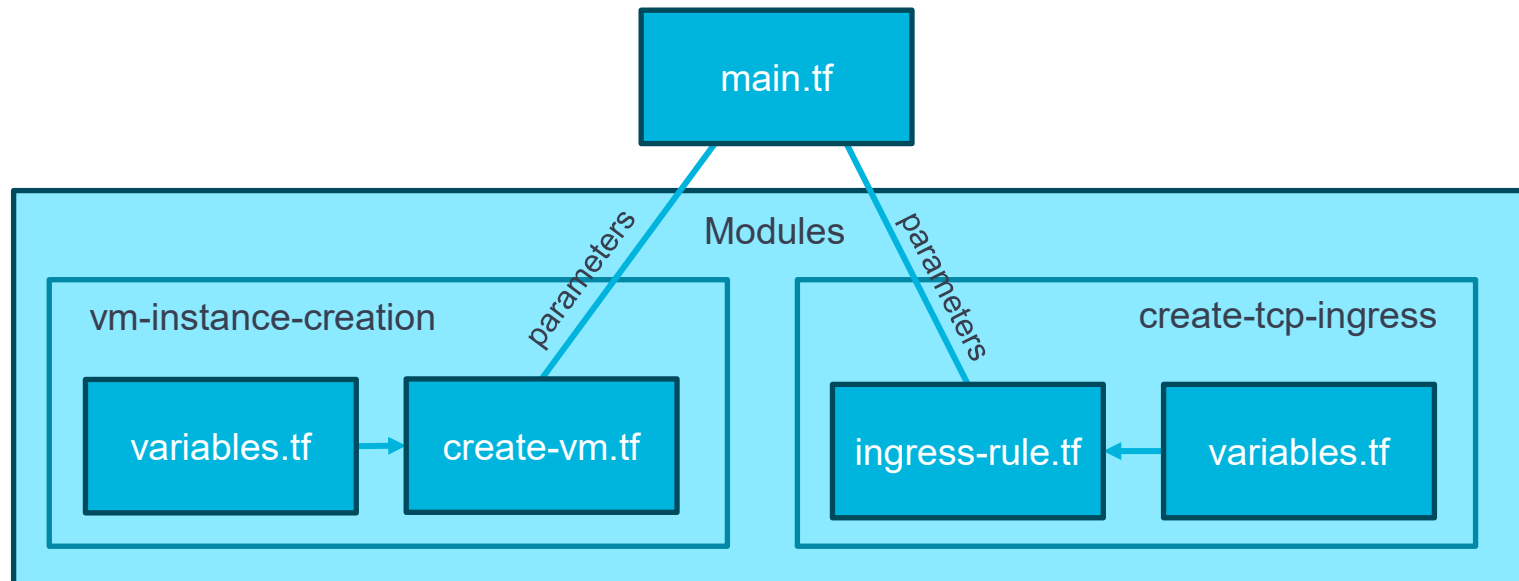
```
variable "machine_type" {
 description = "Azure Virtual machine"
 type = string
 default = "Standard_B1s"
}
```
- ▶ Variable name **machine\_type**. This is initialised to Standard\_B1s unless a non-default value is passed
- ▶ A variable value can be passed in from a file with terraform.tfvars extension or at the command line during apply
- ▶ This approach promotes code reuse. In terraform.tfvars:
  - ▶ `machine_type = "Standard_B2s"` This overrides the default B1s
- ▶ Here, the var instance\_type is not fixed but assigned the value of machine type. If not provided, it defaults to Standard\_B1s. Defaults are optional and often not desirable. Generally best to specify what is used
- ▶ You could have different .tfvars files say for dev and test, e.g. dev.tfvars, test.tfvars set to different values. Just provide the name of the file on the command line during apply
- ▶ Output variables can be added to a default file **output.tf** if required or left in the main.tf file - they all get combined during an apply



- ▶ Terraform apply: values in .tfvars are copied into variables in variables.tf
- ▶ main.tf uses the values in the variables to create the required resources
- ▶ Terraform creates a state file of resources and dependency map of creation order
- ▶ Terraform converts the state file into API calls to the cloud provider to create or destroy resources
  - ▶ Note: if a change is made and some resources are not affected, the unaffected resource definitions are ignored
- ▶ Terraform will read the output generated at the cloud end and display progress on the local machine



- ▶ For common actions, such as VM creation, code can be structured into modules - like functions
- ▶ A main.tf references the module by pointing to its location so is really just a file that references one or more modules to create a more complex infrastructure without code duplication - similar to main program referencing functions by their location
- ▶ A new main is written for the bigger project, and it will reference modules (modules folders)
- ▶ Values are passed from main to module (or module to module) in a similar way to calling functions
- ▶ Main would specify the provider and region which is inherited by modules



# Terraform Provisioners

- ▶ Terraform is an infrastructure deployment application but it is also capable of deploying code
- ▶ A provisioner enables OS command execution on the client or remote machine and are added to a resource block
- ▶ local-exec provisioner
  - ▶ Execute commands on the client machine - e.g. your laptop
  - ▶ For example, pipe terraform output to a file instead of cluttering the console
- ▶ remote-exec provisioner
  - ▶ Execute system commands on the remote machine - e.g., Azure VM, AWS ec2
  - ▶ e.g. can use to install node, or docker or whatever
- ▶ file provisioner
  - ▶ Securely transfer files from the host to the remote machine - e.g. rather than using scp or winscp manually
- ▶ Note: as resource blocks are only called when creating the resource, the provisioners are only executed during resource creation. e.g. Maybe during a VM creation we want to copy a Docker install script to the remote vm and run it to install Docker. That only happens once during VM creation
- ▶ This is a useful config feature for smaller or test projects, In industry you would more likely use:
  - ▶ Ansible, Chef or Puppet

# Terraform null-resource

- ▶ Provisioners are really handy in executing local or remote commands but they are only executed when the resource is being created - i.e. generally once. That's fine for installing Docker
- ▶ But say we want to automate a file copy if the file changes. E.g. a source code file or compose.yaml file, we can't unless we delete a resource and recreate it
  - ▶ Resource blocks are only executed when they are created and the provisioners are in there
  - ▶ Terraform provides a special "fudge" resource block called a **null-resource** that doesn't create a resource but acts like it does. i.e. it is called once when trying to create a resource, but as it is a null-resource, it creates nothing so doesn't affect the infrastructure, but its provisioners are executed

# Terraform null resource

- ▶ When terraform apply is executed, terraform compares the current state, held in the state file, to the desired state held in .tf files and creates a plan indicating any changes it needs to make
- ▶ Changing the content of a file you want to copy to the vm is not a resource change so will not trigger the file-deployment within the resource. A null-resource can deal with this using a trigger

```
resource "null_resource" "copy_file" {
 triggers = {
 file_md5 = filemd5("path/to/local/file.txt")
 }

 provisioner "file" {
 source = "path/to/local/file.txt"
 destination = "/home/ubuntu/file.txt"

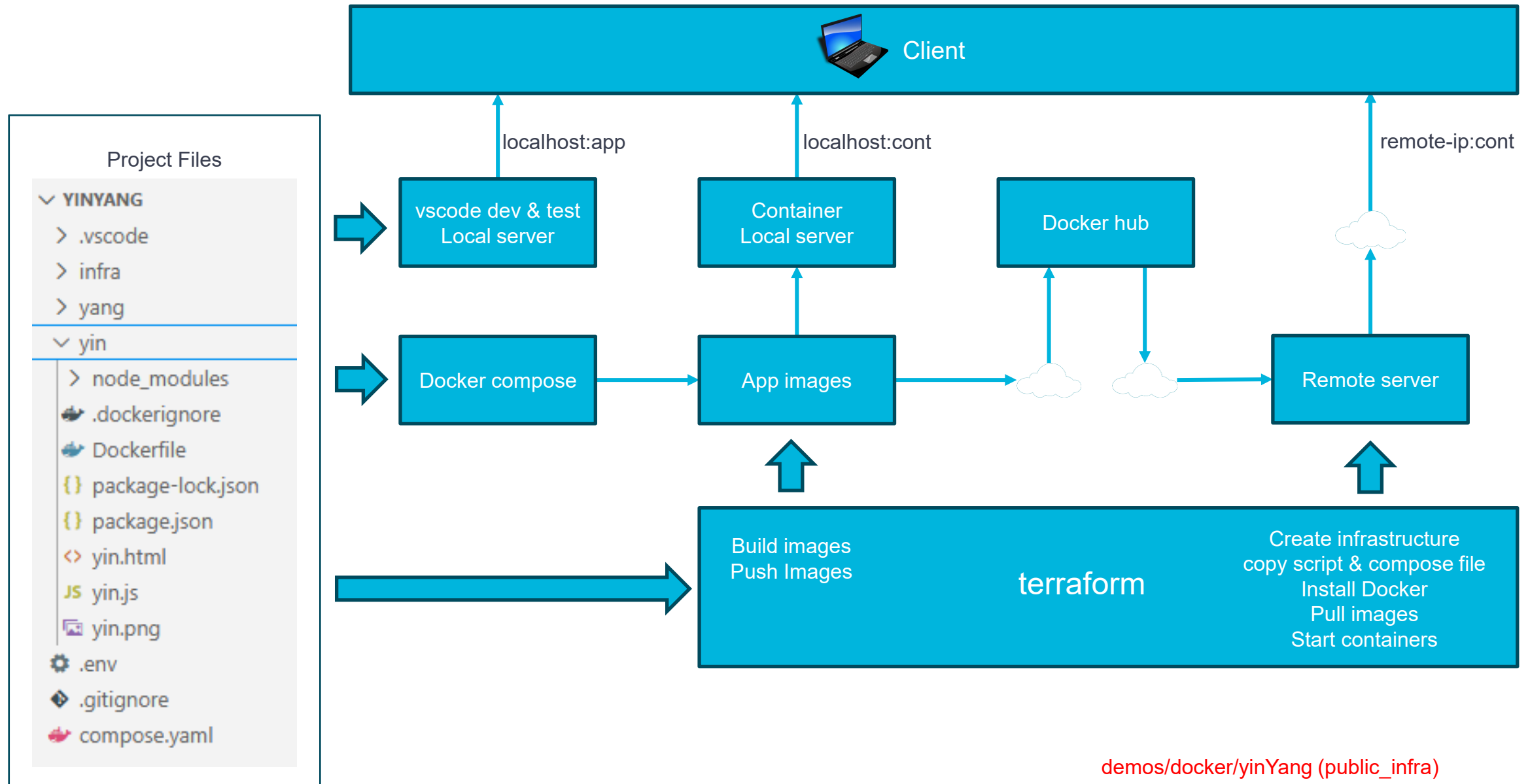
 connection {
 type = "ssh"
 user = "tony"
 private_key = file("path/to/private_key.pem")
 host = azurerm_linux_virtual_machine.demo-vm.public_ip_address
 }
 }
}
```

- ▶ The resource block is triggered by some condition and executed. In this case, the file hash is compared to that in the state and if different, triggers this resource block so the file provisioner is executed

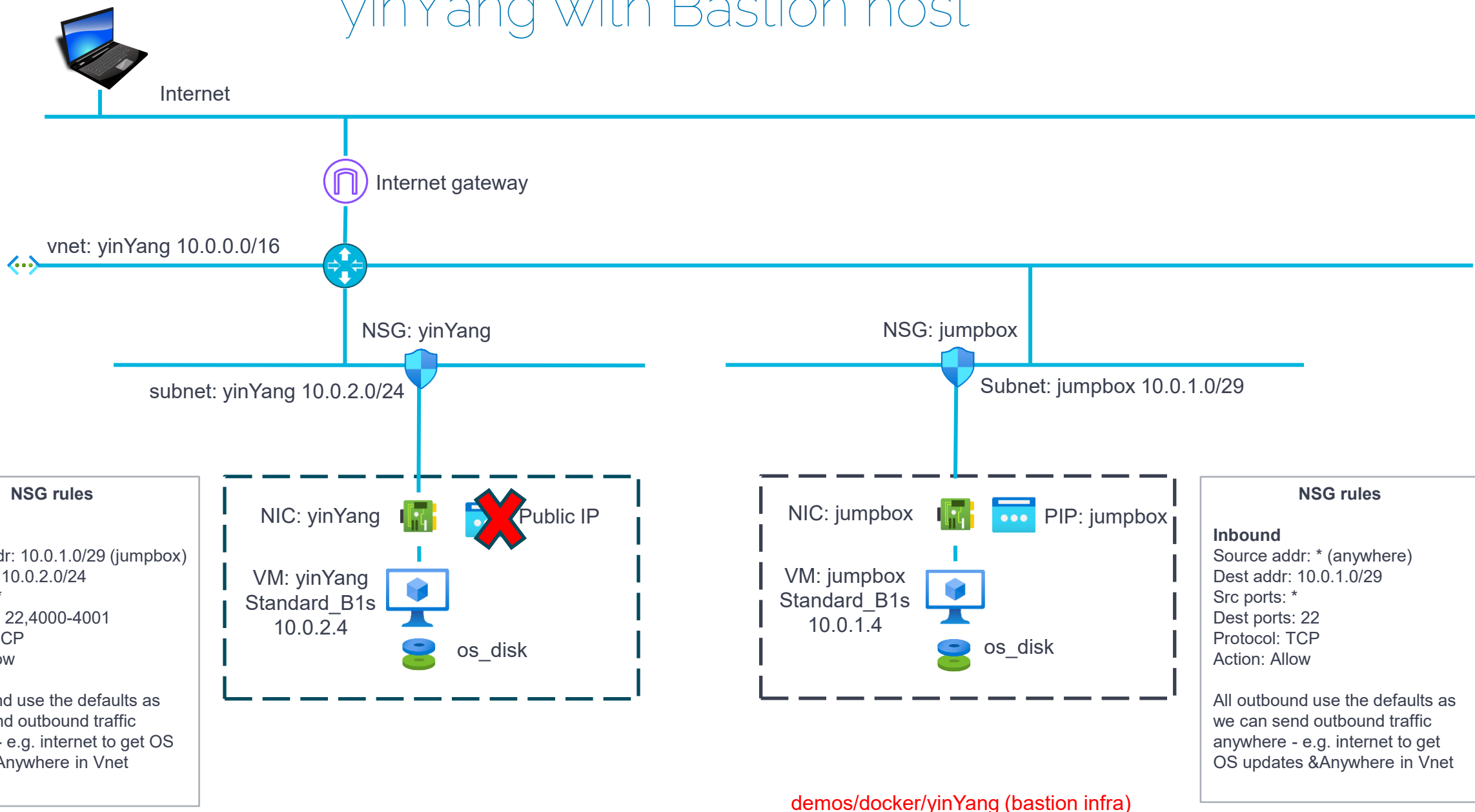
# CI/CD Pipeline

- ▶ Modern organisations tend to use a CI/CD approach for software development
- ▶ Continuous integration (CI)
  - ▶ teams use gitlab, or similar, to work independently on parts of a project
  - ▶ each would debug and test their code then when ready, merge into a single project ready to test as a whole
  - ▶ tests can be automated and a popular approach is test-driven design
- ▶ Continuous deployment (CD)
  - ▶ Automatically deploy tested code into the runtime environment e.g. dev, test, production or whatever
- ▶ So CI/CD automates testing of new or modified code, and if successful deploys it
  - ▶ this enables regular code releases so enables rapid response to changes in business needs
  - ▶ reduces the potential for human error in manually transferring and configuring files between environments
- ▶ There are various CI/CD tools available such as gitlab, Jenkins, GitHub actions, and cloud services
- ▶ The following demo will use Terraform to:
  - ▶ create the target infrastructure
  - ▶ create a basic CI/CD pipeline to automate deployment of a previous simple basic app

# Terraform basic CI/CD



# yinYang with Bastion host



# Authentication

- ▶ Terraform needs to login to Azure to do its thing
- ▶ There are various ways but probably the easiest is to use the Azure CLI
- ▶ Terraform doesn't use the CLI to create resources but it can use it to authenticate
- ▶ Download the cli. For Windows I chose the MSI for 64 bit
- ▶ You can now also use the CLI instead of Terraform if you want to for quick test builds
- ▶ **az version** # check it works
- ▶ **az login** # authenticate through browser popup
- ▶ **az account list --output table** # get a list of accounts - make sure yours is set to default
- ▶ **az account set -subscription "your subscription number"** # get from portal. Search for subscriptions
- ▶ You should now be able to use Terraform to build Azure infrastructure





# Security - TLS

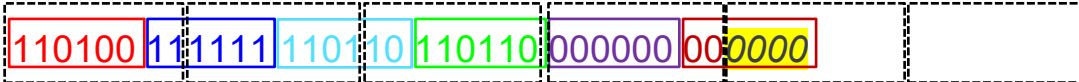
- ▶ On completion of this section, you should be able to:
  - ▶ Explain what base 64 is and the difference between it and base64url
  - ▶ Explain the properties, basic operation and use of the following security approaches:
    - ▶ Hashing
    - ▶ Message Authentication Code
    - ▶ Symmetric key encryption
    - ▶ Diffie-Hellman key exchange
    - ▶ Asymmetric key encryption
    - ▶ Digital Signature
  - ▶ Explain the properties, basic operation and use of a digital certificate
  - ▶ Explain how to request and deploy a digital certificate
  - ▶ Explain the chain of trust
  - ▶ Explain how the browser uses the certificate to establish trust with the server
  - ▶ Create and deploy a digital certificate for an application
  - ▶ Explain the TLS handshake process
  - ▶ Explain the generation and use of session keys

- ▶ This is a very broad subject
- ▶ We will focus only on the underlying principles of Transport Layer Security used to enable https
- ▶ https enables encryption in transit to provide secure communication
- ▶ TLS uses the following crypto processes, so we need to study them briefly:
  - ▶ Hashing
  - ▶ MAC / HMAC
  - ▶ Symmetric key encryption
  - ▶ Asymmetric key encryption
  - ▶ Digital signature
  - ▶ Key exchange
  - ▶ Digital Certificate

# Base 64

- ▶ Data is a sequence of binary bits
- ▶ The binary number block is split into 6 bit blocks then stored as bytes. i.e. 4 chars in 3 bytes
- ▶ The 6 bits represent the position in the base64 alphabet where 'A'=0, 'a'=26, '0'=52, '+' = 62, '/' = 63. Split into 6 bit blocks
- ▶ Say we have a file of only 4 bytes: D3 FD B6 00

110100111111101101101100000000



- ▶ 0 / 2 2 A A == In base 64 -
- ▶ As the smallest byte sequence to fit 6 bit numbers is 24 bits or 3 bytes, we need to pad a byte with zeros then indicate how many bytes are missing from a 3 byte block. Pad the last byte to create a char = A in this case. As we are 2 bytes short of a 3 byte block, we add two = to indicate that the last triple has 3 bytes missing converting 0/22AA back into binary
- ▶ Doesn't work with urls as +, / and = are used for other things in internet-based data. + is a space and / is a path separator. = means = so has been dropped. Because base64-url is used for structured data, e.g., JWT, OAuth tokens, URL query parameters etc, the receiver knows how to interpret this data as it can be inferred by the protocol or an expected number of bytes

| Index | Binary | Char | Index | Binary | Char | Index | Binary | Char | Index | Binary | Char |
|-------|--------|------|-------|--------|------|-------|--------|------|-------|--------|------|
| 0     | 000000 | A    | 16    | 010000 | Q    | 32    | 100000 | g    | 48    | 110000 | w    |
| 1     | 000001 | B    | 17    | 010001 | R    | 33    | 100001 | h    | 49    | 110001 | x    |
| 2     | 000010 | C    | 18    | 010010 | S    | 34    | 100010 | i    | 50    | 110010 | y    |
| 3     | 000011 | D    | 19    | 010011 | T    | 35    | 100011 | j    | 51    | 110011 | z    |
| 4     | 000100 | E    | 20    | 010100 | U    | 36    | 100100 | k    | 52    | 110100 | 0    |
| 5     | 000101 | F    | 21    | 010101 | V    | 37    | 100101 | l    | 53    | 110101 | 1    |
| 6     | 000110 | G    | 22    | 010110 | W    | 38    | 100110 | m    | 54    | 110110 | 2    |
| 7     | 000111 | H    | 23    | 010111 | X    | 39    | 100111 | n    | 55    | 110111 | 3    |
| 8     | 001000 | I    | 24    | 011000 | Y    | 40    | 101000 | o    | 56    | 111000 | 4    |
| 9     | 001001 | J    | 25    | 011001 | Z    | 41    | 101001 | p    | 57    | 111001 | 5    |
| 10    | 001010 | K    | 26    | 011010 | a    | 42    | 101010 | q    | 58    | 111010 | 6    |
| 11    | 001011 | L    | 27    | 011011 | b    | 43    | 101011 | r    | 59    | 111011 | 7    |
| 12    | 001100 | M    | 28    | 011100 | c    | 44    | 101100 | s    | 60    | 111100 | 8    |
| 13    | 001101 | N    | 29    | 011101 | d    | 45    | 101101 | t    | 61    | 111101 | 9    |
| 14    | 001110 | O    | 30    | 011110 | e    | 46    | 101110 | u    | 62    | 111110 | +    |
| 15    | 001111 | P    | 31    | 011111 | f    | 47    | 101111 | v    | 63    | 111111 | /    |

demo using <https://cryptii.com/pipes/hex-to-base64>

# Security Aims of TLS

- ▶ **Confidentiality** - no one other than the intended person can read the data - i.e. secret
  - ▶ **Integrity** - we can be assured that the data has not been interfered with or corrupted
  - ▶ **Authenticity** - we know for sure who / where the data came from
  - ▶ **Non-repudiation** - the provider of data cannot claim at a later date that they didn't send it
- 
- ▶ TLS provides crypto approaches that deal with most of these assurances
  - ▶ TLS doesn't provide non-repudiation but a technique it uses can - we will discuss each
  - ▶ TLS enables us to use https which is security of data in transit through encryption and authentication

# Hashing (Integrity)

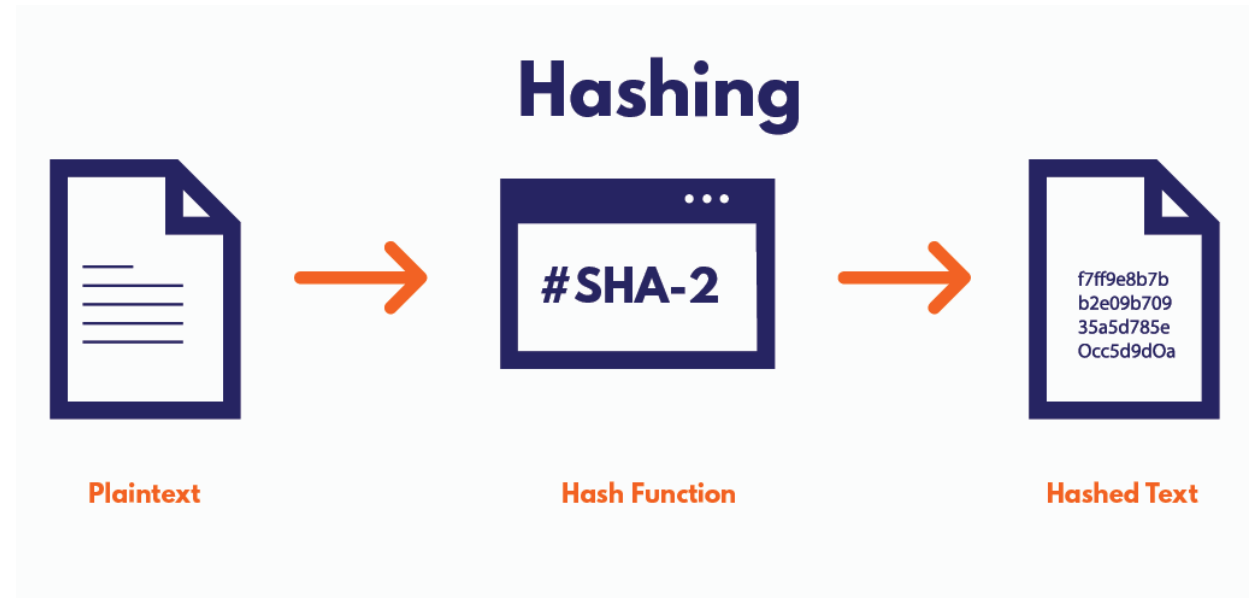
- ▶ Fast search
- ▶ **One-way crypto function**
- ▶ **Fixed length number (digest)**
- ▶ Use cases include

- ▶ password protection

- ▶ Pa55w0rd : SHA 256=590561E975AA1CBDAA6FF00DA013DFB85656990EEF11E504E9FEBE48A87D3642
- ▶ Can't be reversed but rainbow table may expose the password so still need strong password
- ▶ Brute force attack
- ▶ Use long complex passwords and a password manager
- ▶ Use Single sign-on
- ▶ Use Multi factor authentication

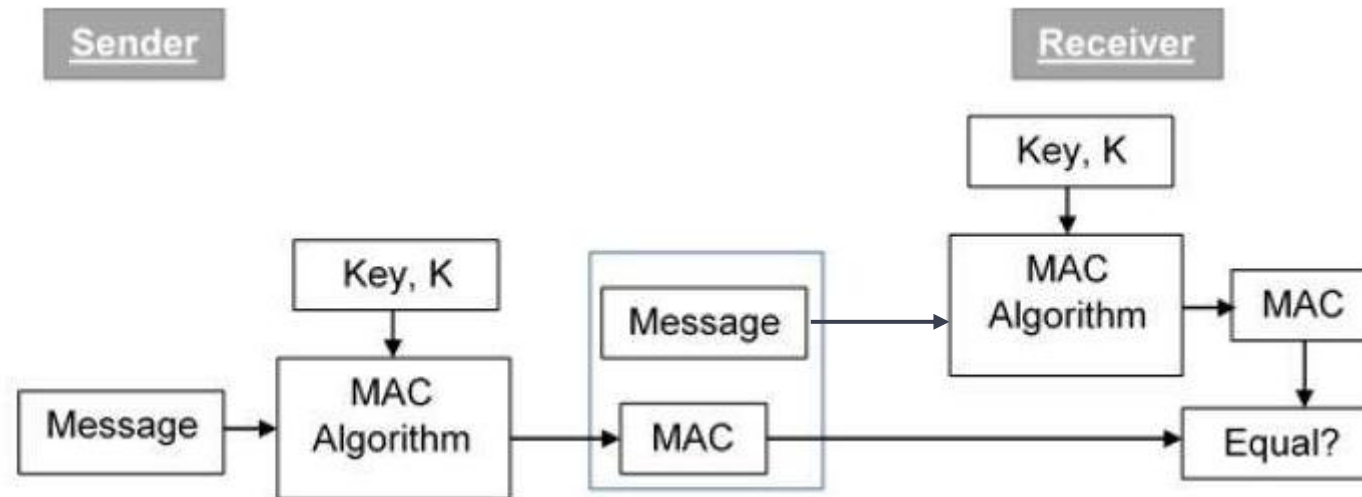
- ▶ file / message integrity check

- ▶ File data integrity, photo steganography



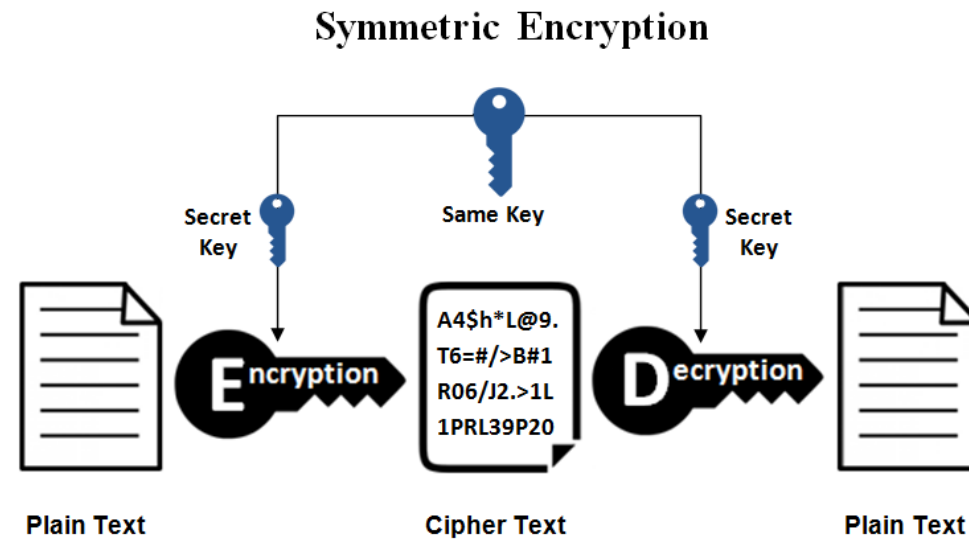
# Message Authentication Code (Integrity and authenticity)

- ▶ Hashing is good for integrity but not security
- ▶ MAC fixes this as it adds authenticity as long as the shared secret is kept secure
- ▶ HMAC is a MAC generated using a hashing function



- ▶ Cannot provide non-repudiation - I can be assured of who it came from and it's not been tampered with, but what if the sender claims they didn't send it, what proof do I have? I have a message and hmac which I could have created myself and claimed it was sent to me

# Symmetric key encryption (confidentiality)



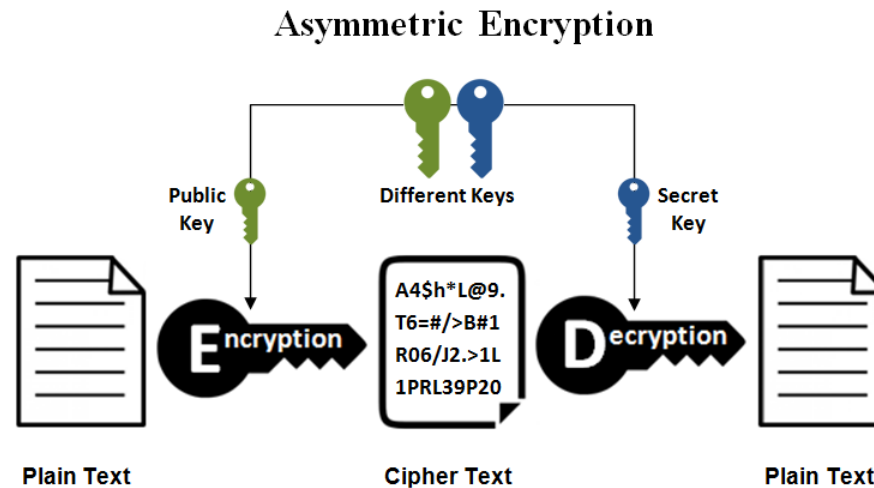
- ▶ The same key is used to encrypt as to decrypt
- ▶ Advanced Encryption Standard is most common
- ▶ AES has a 128 bit key (can accommodate 192 and 256 bit keys)
- ▶ AES128 for home routers & https, Bitlocker & Whatsapp use AES256 and NSA approves it for Secret level data
- ▶ Very fast so good for large data blocks
- ▶ Challenge: getting the key to the second party securely



# Secure key agreement (not exchange)

- ▶ Diffie-Hellman is a very popular method of key agreement
- ▶ Each party creates values that are exchanged and from those values, the secret keys are computed
- ▶ The beauty of this method is that the secret keys themselves are not exchanged so can't be intercepted
- ▶ The values which are exchanged, if intercepted, cannot be used to compute the shared secret
- ▶ If there is a man in the middle attack, they would need to intercept both exchanges and substitute both values
  - ▶ To protect against this, the server value also provides a digital signature (later) so the value can't be changed without detection
- ▶ From the exchanged values, both parties can compute the same value which is the shared secret
- ▶ Although the shared secret could be used now for encryption, generally create working keys based on these shared values as will be seen in TLS later

# Asymmetric Key Encryption (confidentiality)



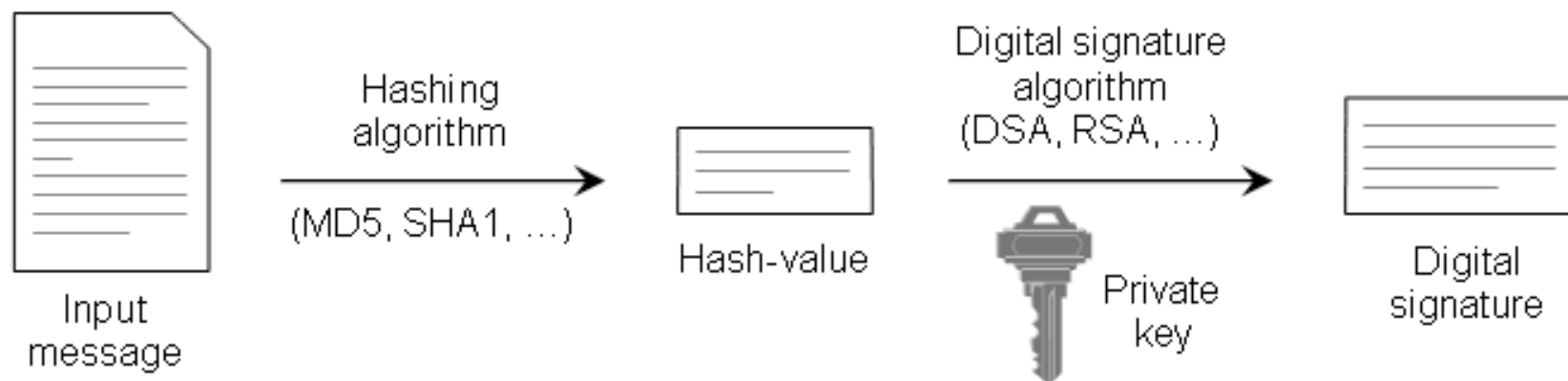
- ▶ Uses a pair of complementary but different keys: private key and public key
- ▶ Public key is made public
- ▶ A message encrypted with a public key is decrypted with the secret key - this is used for **confidentiality**
- ▶ A message encrypted with the secret key can be decrypted with a public key - not used for confidentiality, it's just a fact
- ▶ But who sent the encrypted message?
- ▶ How do we know your public key came from **you** when we downloaded it or received it in an email? - i.e. someone could claim to be Alice, spoof her email and send an encrypted message to Bob using his public key
- ▶ Complex maths so slow. Not good for large data blocks

</demos/crypto/asymmetricKeyGen.js>  
</demos/crypto/asymmetricEncrypt.js>  
</demos/crypto/asymmetricDecrypt.js>

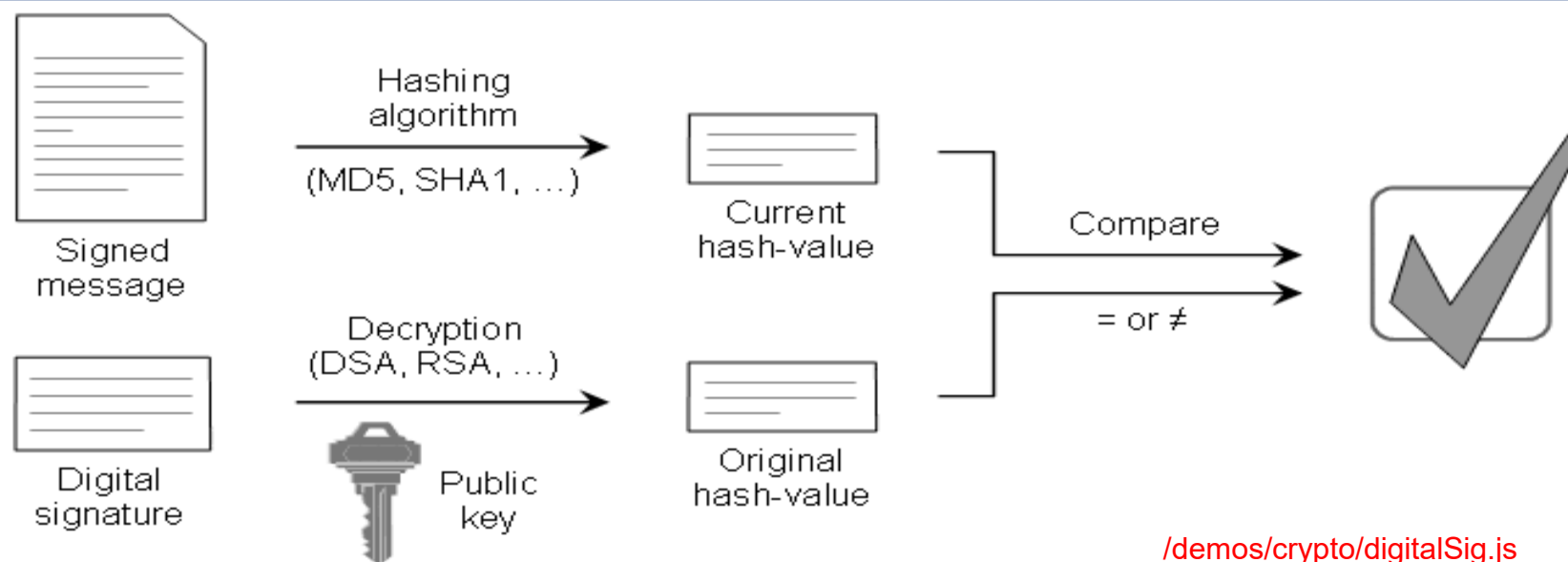
- ▶ Popular asymmetric algorithms are RSA and Elliptic curve
- ▶ RSA (N) where N is the key size
- ▶ The National Institute of Standards and Technology (NIST) define crypto strength in "bits of security"
  - ▶ 112 bits is good until about 2030
  - ▶ Approximate equivalencies:
    - ▶ 2048 bit RSA key
    - ▶ 224 bit ECC key
    - ▶ 128 bit AES

# Digital Signature (authenticity, integrity, non-repudiation)

- ▶ A digital signature has a wealth of features:
  - ▶ More secure alternative to a handwritten signature
  - ▶ Guarantees **authenticity** of the sender
  - ▶ Provides **non-repudiation** - as long as the secret key isn't shared so for legal stuff, use a HSM, smart card or similar to hold the secret key
  - ▶ Ensures **integrity**
- ▶ Bob wants to send me a binding contract of employment
  1. Bob sends me his public key
  2. Bob hashes the contract to create a unique number (digest)
  3. Bob encrypts the digest using his private key
  4. I receive the email with the plain text contract and the encrypted digest (digital signature)
  5. I decrypt the digital signature using Bob's public key to get the hash value (not confidentiality)
  6. I hash the contract file to get a digest (hash) to check it's the same as Bob's
- ▶ We need a diagram



Create a Digital Signature



Verify a Digital Signature

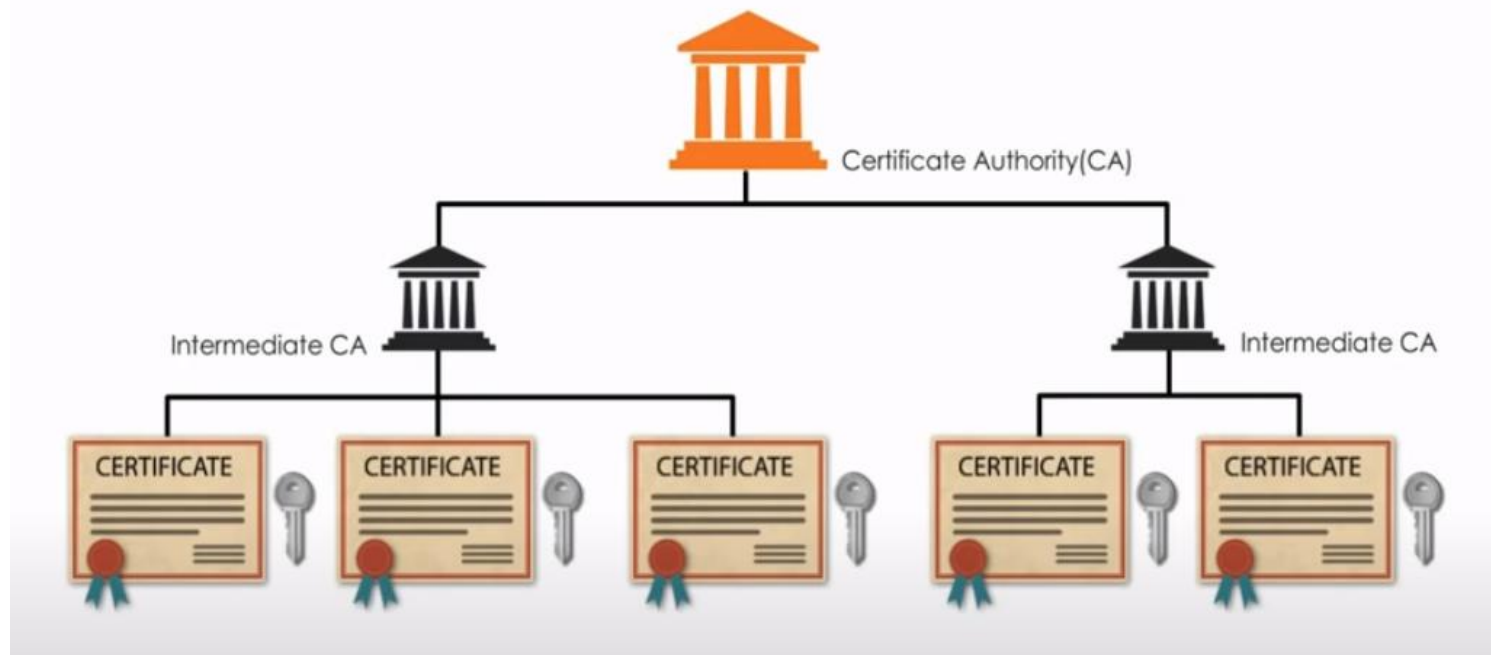
</demos/crypto/digitalSig.js>  
</demos/crypto/digitalSigCheck.js>

# Summary of basic properties

- ▶ **Symmetric keys** are the same for encryption and decryption - provides **confidentiality**
  - ▶ Encryption and decryption is very fast
  - ▶ Securely getting a symmetric key to the second party is the difficult bit. Diffie-Helman is a popular approach to solving that problem as the actual keys aren't exchanged but calculated based on exchanged data
  - ▶ Advanced Encryption Standard (AES) is popular for symmetric encryption (data in transit and at rest)
- ▶ **Hash** can be used for **integrity** as it is a unique fixed length number very sensitive to data change
- ▶ **MAC / HMAC** used to provide security of the hash by mixing in a symmetric key so provides
  - ▶ **authenticity** and **integrity**
- ▶ **Public key encryption.** Keys are a pair and are different: private (secret) and public
  - ▶ Public key used to encrypt data for **confidentiality**
  - ▶ Private key used to decrypt confidential data encrypted with public key
- ▶ **Private key encryption** can be used to create a digital signature.
  - ▶ Provides **integrity, authenticity & non-repudiation**
  - ▶ Digital signature is the equivalent to symmetric key HMAC but better as it can provide non-repudiation
- ▶ RSA and Elliptic Curve are popular for asymmetric encryption - e.g. digital signature and digital certificates

# Digital Certificate

- ▶ We have almost nailed the security using the various approaches and digital signatures are simple, very effective and provide three of the security essentials we need: **integrity**, **authenticity** and **non-repudiation**. But we need a public key to be able to check the signature. How do we know where that came from?
- ▶ Digital certificates are the final piece in the puzzle
- ▶ Digital certs answer the question "Can we be sure that Bob's public key really is Bob's key?"
- ▶ Bob creates a public key then gets it embedded into a digital certificate by a trusted third party - certification authority
  - ▶ Bob needs to provide other info including the domain the certificate will be used e.g. bob.com. This is checked by the CA
- ▶ The CA, adds Bob's public key and his domain then signs the certificate with its digital signature to validate it
- ▶ When we want to use Bob's public key, we ask for the certificate. We can check the CA digital signature using the CA public key and if ok, we can trust Bob's public key so extract it from the certificate and use it
- ▶ In https, Bob would store his certificate on the server that is used to provide a service at the certified domain
- ▶ For practical purposes, there is more than one CA in the world, and the CA can delegate authority to intermediate CAs to make the process more manageable. This creates a chain of trust

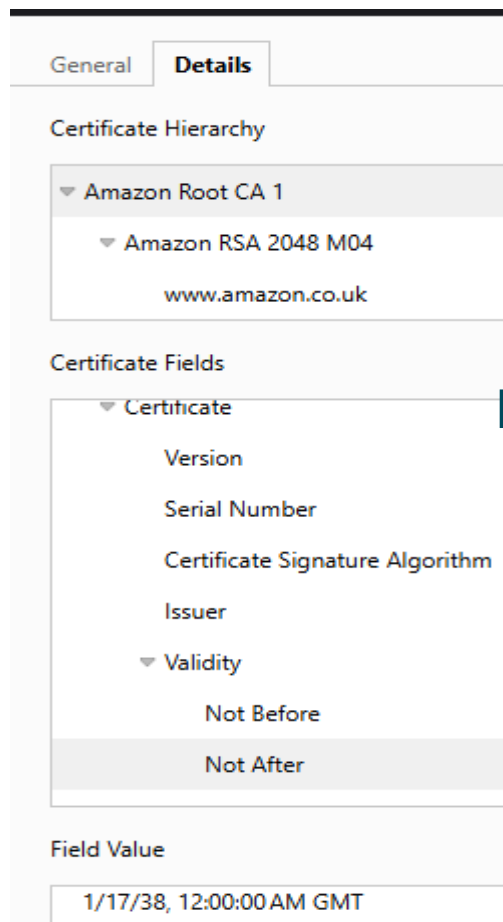
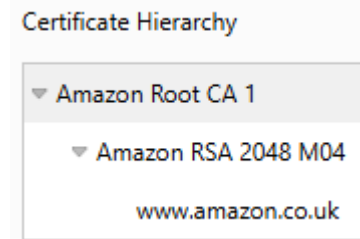


- ▶ **Root certificate**
  - ▶ A root certificate is self-signed as it must start somewhere. i.e. the CA digitally signs its certificate using its secret key
  - ▶ Root certificate authorities are known and trusted by most browsers - installed in OS and browser during updates
- ▶ **Intermediate certificates**
  - ▶ These can be created from the root and used by delegated organisations to sign user certificates. The lineage of certificate signing is held in the certificate
  - ▶ Having just a root CA is limiting in that it could be a bottleneck in providing certs and if its secret key is compromised then all its certs are worthless, so intermediates are where you tend to get your cert from
  - ▶ Intermediate certs and your cert should be installed on the server. The Root cert should be on the client in the OS trusted store

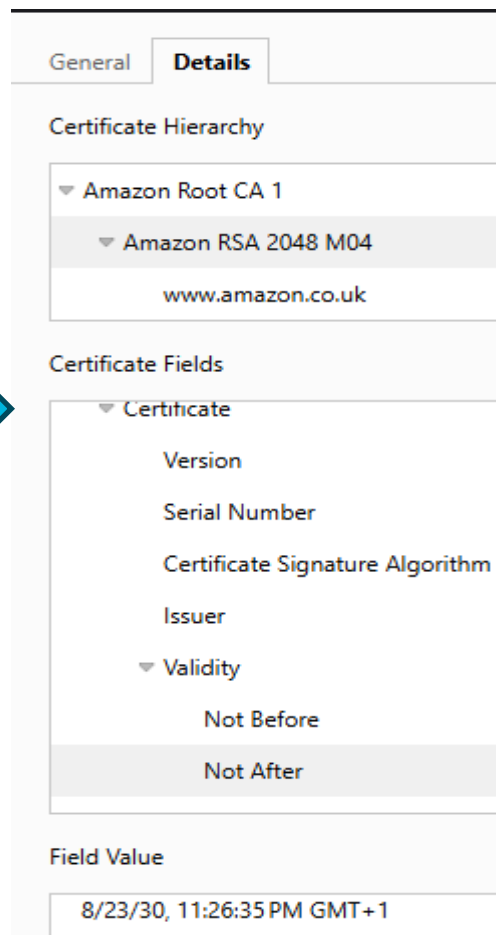


# Chain of trust

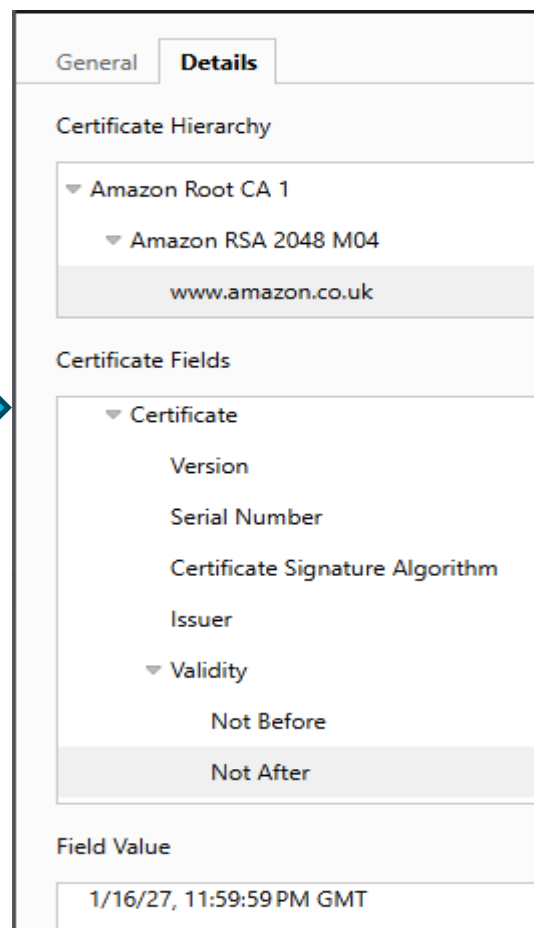
- ▶ Enter amazon.co.uk into your browser. It is secure by default
- ▶ Click on the padlock (now a different symbol) then certificate
- ▶ Certification path does what it says on the tin



Self signed root CA



Intermediate CA



Server Cert

Demo in browser

# How to get a certified digital certificate

- ▶ You need to make a Certificate Signing Request (CSR)
- ▶ X.509 is the most common standard used
- ▶ You need to provide information proving who you are, and the details associated to you:
  - ▶ Your name – generally the host name of the server you want to use – e.g. your domain `tonynicol.com` as this is unique to you
  - ▶ Organisation – if you have one. This can be verified by the Certification Authority if it is registered
  - ▶ Division within the organisation – if relevant
  - ▶ City, State/County, Country, Email Address – they may contact you on this
- ▶ You need to provide verifiable details
  - ▶ Depending on the cert type, they may check your domain name provider, your payment provider, email, phone etc.
  - ▶ Extended Validation (EV): - Legal business name, Jurisdiction of incorporation, Registration number etc (e.g. `digicert.com`)
  - ▶ Organisational Validation (OV): Organisational details such as verified name, location (e.g. `Microsoft.com`)
  - ▶ Domain Validation (DV): simply prove you own the domain. You could be anyone (e.g. `Lancashire.ac.uk`)
- ▶ You create public / private key pair
  - ▶ Keep the private key secure and private
  - ▶ Public key will be sent to CA in the CSR and will be embedded in the signed certificate

# Creating a simple DV cert

- ▶ Domain validation certs are very popular
- ▶ mkcert is a useful app from [mkcert.org](https://mkcert.org)
- ▶ mkcert creates a CA on your client machine which enables you to create your own certificates
- ▶ mkcert adds itself to the secure vault in the trusted CAs in your OS so any certificates created from it and deployed to your server are trusted
- ▶ Useful for quick local ssh testing. Can use an IP with it in the domain registration
- ▶ The certs won't be trusted by anyone else's browser but it's good for development and learning
- ▶ Instructions are available at [mkcert.org](https://mkcert.org)
- ▶ Get the app (Windows): `winget install mkcert`
- ▶ Once we have a CA trusted by our browsers we can create our own certificates
  - ▶ **`mkcert -key-file key.pem -cert-file cert.pem domain`**
  - ▶ this will create `key.pem` and `cert.pem` in the current directory. Domain needs to be `localhost`, `20.117.114.49` or wherever you will use the cert. The browser can check the IP it called and compare to this value
- ▶ If you want to look at the cert in a readable form:
  - ▶ **`certutil -decode cert.pem`** (on Windows)
  - ▶ Check the *X509v3 Subject Alternative Name: IP Address:20.117.114.49* or wherever, is correct if you are having problems
- ▶ We don't add certs into the container, or each container needs one and the secret key is in the image on Docker hub

# Creating a let's encrypt real cert

- ▶ Cannot create a let's encrypt cert using an IP address - you need a domain name
- ▶ You install certbot on your server then using it in a single command it will
  - ▶ Send a server to let's encrypt (LE) server POST request with the domain name you want to register
  - ▶ Server receives a token from LE - basically a random string - this is a challenge step
  - ▶ Server POSTS back the token and a hash of the public key as a response to the challenge
  - ▶ Server POSTS a "Please validate the challenge" to LE
  - ▶ LE makes a get request to a web server on your server (typically port 80) asking for the token to check it matches
  - ▶ If it matches, certbot creates a key pair and a CSR containing the public key, domain names, and a digital signature proving ownership of the private key associated to the public key in the CSR and server sends it to LE
  - ▶ LE creates and returns cert.pem, chain.pem (the intermediate cert) and chain.pem (both in one file). We tend to use this one
- ▶ All that is done for you but you should know the sequence of events behind it
- ▶ It does create a CSR and it is stored on your server but you never see it unless you look for it
- ▶ If use digicert or similar, you create a CSR on their web page, add your public key and they send you the certs

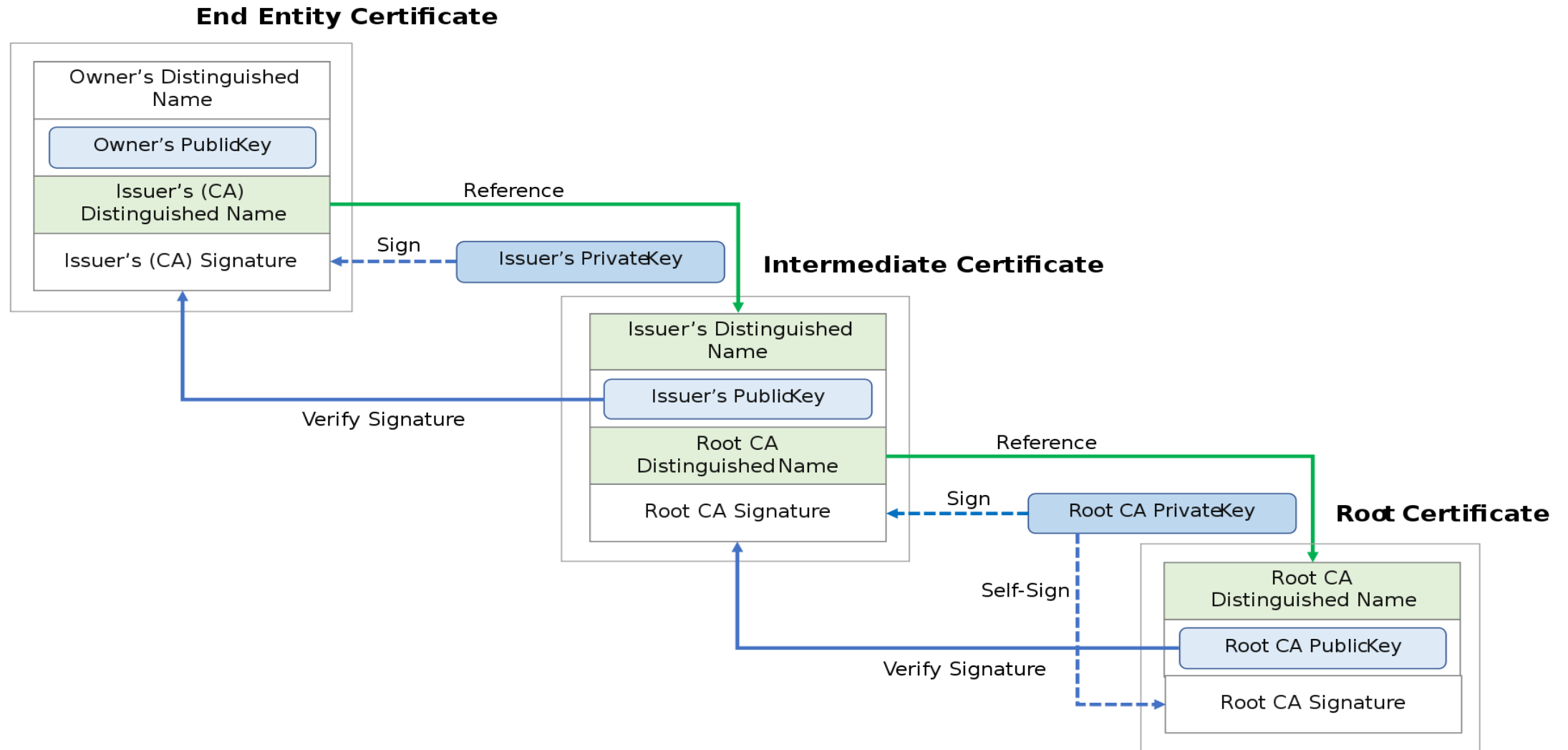


# Transport Layer Security

# Algorithm usage in SSL / TLS

- ▶ Hashing
- ▶ MAC / HMAC
- ▶ Symmetric key encryption
- ▶ Asymmetric key encryption
- ▶ Digital signature
- ▶ Key exchange
- ▶ Digital Certificate

# Reminder: How does the browser trust your certificate?



Client

# TLS handshake

Server

TLS Version  
Random Num  
Session ID  
Cipher Suite  
Compression  
Extensions

Client Hello

Server Hello &  
Public param  
Exchange

TLS Version  
Random Num  
Session ID  
Cipher Suite  
Compression  
Extensions  
Certificates  
DH pub param  
Server Done

Check certificates  
Creates symmetric PM key

Client pub param or PMS  
Change Cypher  
Encrypted finished

Client Key Exchange

Handshake Complete

Create symmetric keys

Change Cypher  
Encrypted finished



# Symmetric session key generation

Pre-master secret, "master secret", ClientHello.random + ServerHello.random

PRF (Pseudo Random Function)

Pre-master key now securely deleted from client and server

Master secret (48 bytes)

Master secret, "key expansion", ClientHello.random + ServerHello.random)

PRF (Pseudo Random Function)

Call multiple times to get enough bytes for all four session keys  
e.g. 4 x 32 bytes for AES 256. This example generates four x 10  
byte keys for simplicity

12AF349DD8ABBE73486DAB937FA5B3C81A0F6C024BBAC5C662FBA8E9D04CA7F4B274DDB2C87AA3BD2E992AAB34CAAB32

*client\_write\_MAC\_key:*

Authentication and Integrity check:

12AF349DD8ABBE73486D

*server\_write\_MAC\_key:*

Authentication and Integrity check:

AB937FA5B3C81A0F6C02

*client\_write\_key:*

Message encryption using symmetric key:

4BBAC5C662FBA8E9D04C

*server\_write\_key:*

Message encryption using symmetric key:

A7F4B274DDB2C87AA3BD

# Mutual TLS

- ▶ mTLS requires mutual authentication
- ▶ One server A tries to connect to a server B using https.
- ▶ Server B returns its cert as usual but is also configured for mTLS e.g. in Node, requestCert:true in initialisation of the https server
- ▶ Server B also returns a list of CA certs it holds in a directory or wherever
- ▶ Server A checks that list to make sure it is a CA that can validate its cert and if so, sends its cert to B
- ▶ Each side checks each others' certificates and if ok, an https session is established
- ▶ i.e. the only difference between TLS and mTLS is that server B tells server A it wants its cert by setting requestCert to true and adding a list of CA (typically one or a few) and intermediate certs it supports in client-ca.pem
- ▶ Server A sees the list, checks its cert is trusted by at least one of them, then if so, returns its certificate for authentication
- ▶ All this happens in the same handshake discussed for TLS but with a couple of extra steps but technically the same
- ▶ Typically used system to system on private networks for extra security

```
const fs = require("fs");
const https = require("https");
const express = require("express");

const app = express();

const server = https.createServer({
 key: fs.readFileSync("server.key"),
 cert: fs.readFileSync("server.crt"),

 // Trust store for validating client
 ca: fs.readFileSync("client-ca.pem"),

 // This makes the server send Certificate
 requestCert: true,

 // Optional but common: fail the handshake if not authenticated
 rejectUnauthorized: true
}, app);

server.listen(443);
```



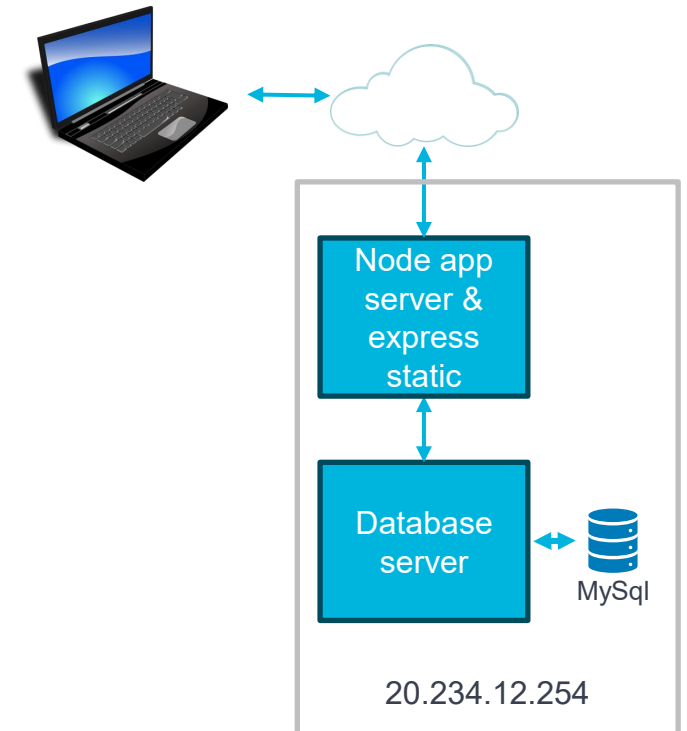
# Cross Origin Resource Sharing

# Origins and Cross Origin Resource Sharing (CORS)

- ▶ We work with same origin or cross origin
- ▶ Origin is **scheme://domain:port**
  - ▶ e.g. **http://tonynicol.com:3000** or **http://20.234.12.254:3000**

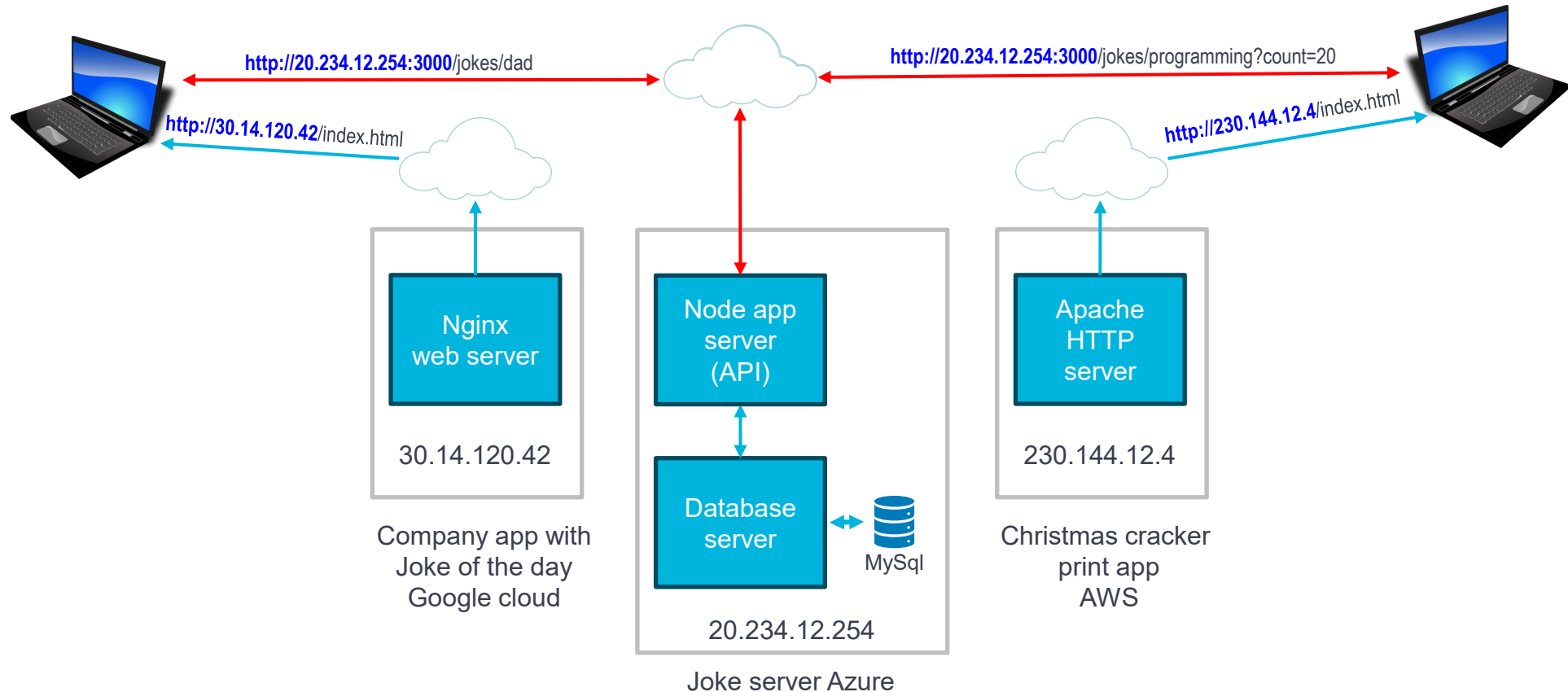
## Same origin

- ▶ The user makes a request to say **http://20.234.12.254:3000/jokes.html**
- ▶ `express.static` will return the web page
- ▶ the page requests css with `href="jokes.css"`. No need for origin as browser will assume **same origin** and add the **http://20.234.12.254:3000/jokes.css**
- ▶ page requests javascript with `src="jokes.js"` using same assumption
- ▶ With page displayed and formatted, press a button to invoke `fetch()`
- ▶ `fetch` will also assume **same origin** so we only need `fetch('/jokes/general')`
- ▶ You can write `fetch('http://20.234.12.254:3000/jokes/general')` if you want to
  - ▶ As long as the origin is the same as the web page it's fine
- ▶ `fetch()` from a different origin is a security issue and is blocked by default
  - ▶ page may be infected to point to a different origin to send information from your page



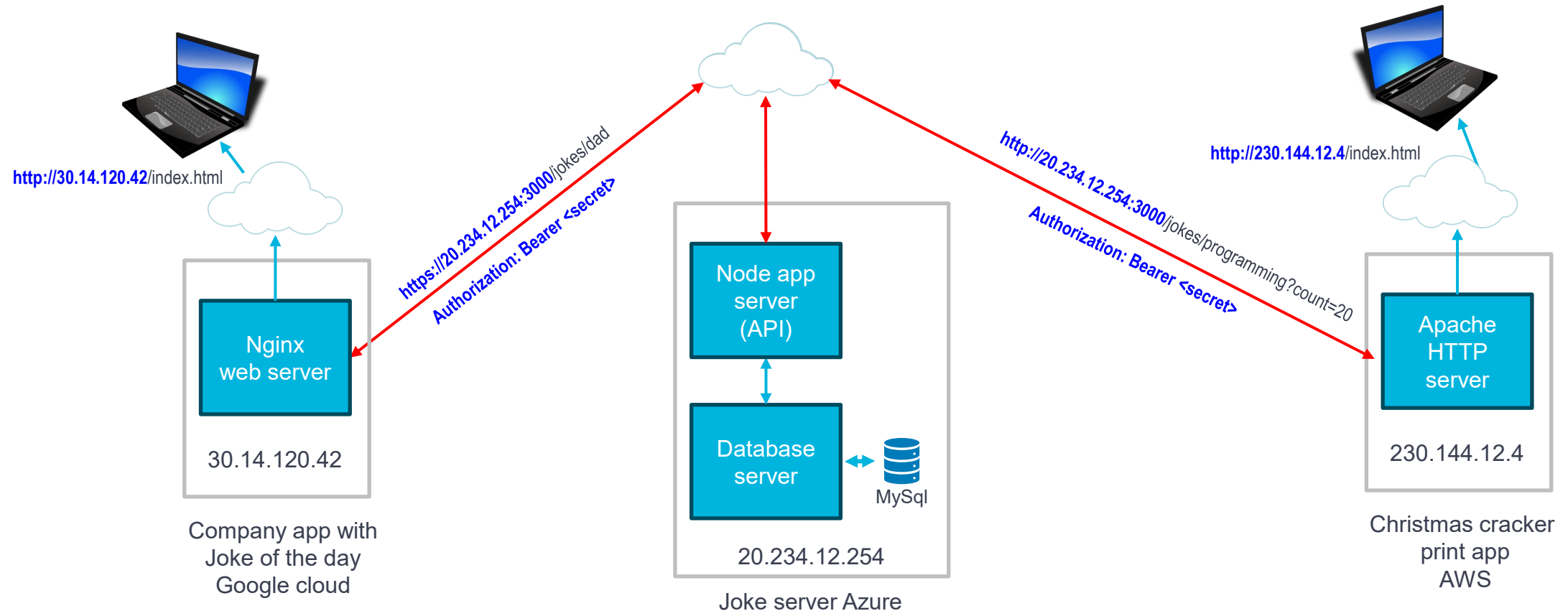
## Cross origin

- ▶ When we develop APIs, we don't usually provide a front-end
  - ▶ APIs are for use by many different users to get or change data
  - ▶ They provide their own web front-end
- ▶ The front-end files are requested from their own web server - API call is cross origin
  - ▶ API call is to a different (cross) origin. Blocked by default. Server needs to enable CORS policy
  - ▶ Server CORS Policy can include list of origins, http method and various other conditions



## Cross origin

- ▶ A better approach is for the communication to go from server to server
- ▶ Use https and send a pre-agreed bearer token such as an API key in a JWT or OAuth token for authorisation
- ▶ As the browser is not involved, there is no CORS issue or risk





OpenAPI



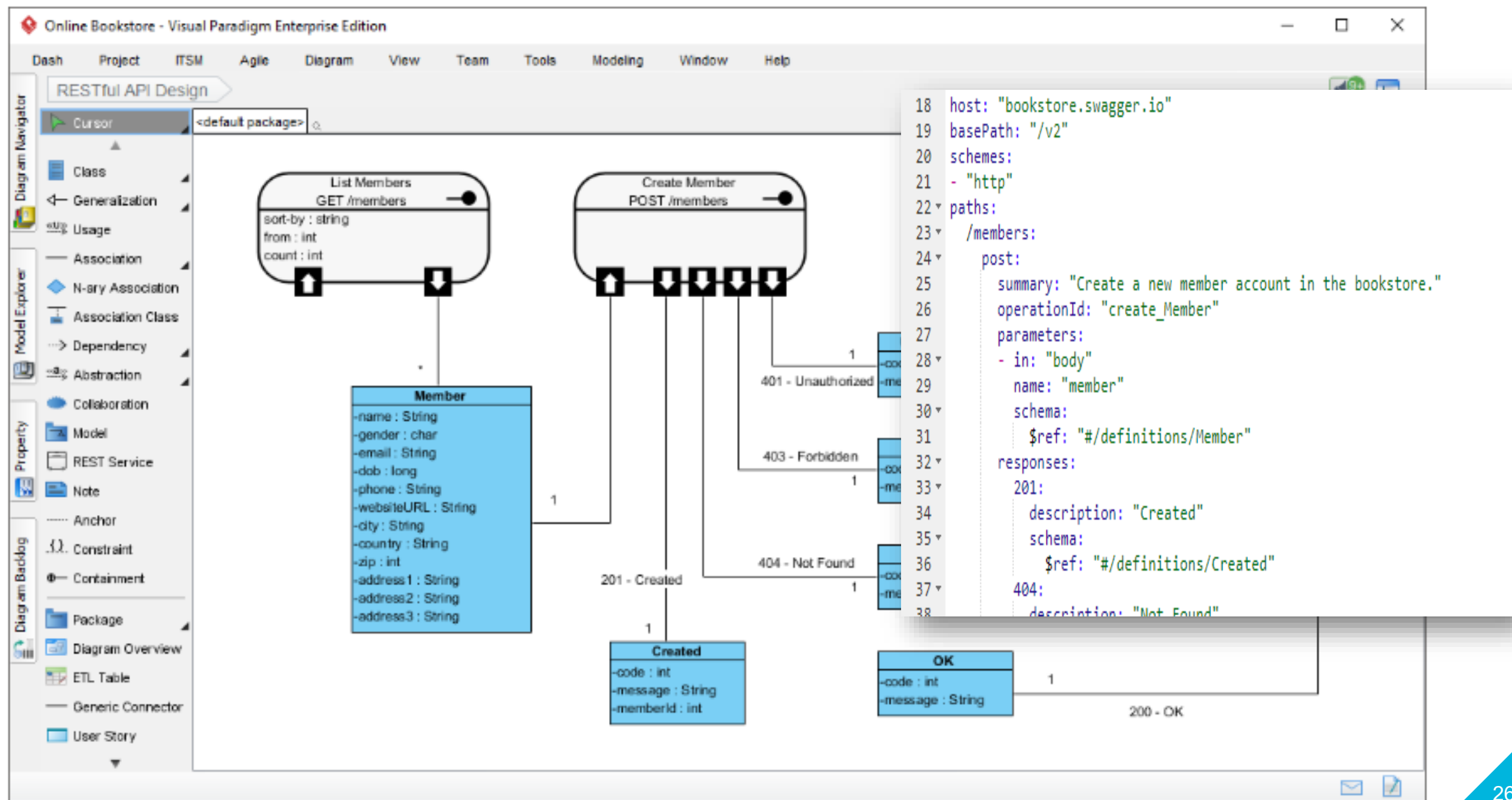
# Learning objectives

- ▶ Upon successfully completing this section, you should be able to:
  - ▶ Explain the three types of API documentation approaches
  - ▶ Explain when you may choose to use one approach over another
  - ▶ Explain the benefits of using the OpenAPI standard documentation
  - ▶ Implement OpenAPI documentation and demonstrate its interactivity with the API



- ▶ API documentation is essential, or no one would know how to use your API. There is a standard approach to documentation defined by the OpenAPI specification as a project of the Linux foundation
- ▶ This documentation approach was originally called Swagger until the specification was donated to the OpenAPI initiative under the Linux foundation. So OpenAPI maintain the current standard but some of the tools used in document creation are called Swagger.
- ▶ There are three common approaches to API documentation, Design-first, API-first and Code-first - each of these results in an OpenAPI specification document (formerly Swagger).
- ▶ **Design-first** use-case is more common in large organisations that have many complex APIs. Typically used when the business works closely with engineers to create a design that satisfies the business requirements either before code is written or potentially to create coding stubs to get a feel as to how the interfaces could be used
- ▶ Below is an example of a graphical tool used to create a class-like diagram of API endpoints. This tool can generate an openAPI specification and some boilerplate code

# Design First



# API First

- ▶ The **API-first** approach is where you formally define your endpoint documentation / contract, using an API specification language. There are various approaches but the OpenAPI specification, formally Swagger, is the standard so we will look at that
- ▶ The OpenApi documentation is written in yaml (or json). You can write it in a text editor or use the swagger web-based tool
- ▶ There is an online tool to help to write your specification: <https://editor.swagger.io/>
- ▶ We will use YAML as it is claimed to be more readable than JSON and is more popular for this
- ▶ These specifications could be written by product owners or engineers or ideally both together before coding starts

# Top Level Elements of an openAPI spec

- ▶ **openapi** - version
- ▶ **info** – used to inform users and make the API discoverable
- ▶ **servers** – a list of servers running the API can run on
- ▶ **paths** – a list of endpoints, methods, parameters
- ▶ **components** – data entities for reuse
- ▶ **tags** – used to group endpoints and other elements together so they appear as a group in the final documentation. e.g. users, payments etc
- ▶ **\$ref** - reference to reusable blocks (components)

- ▶ Design-first and API-first
  - ▶ Better understand the problem up front – a bit like test driven development
  - ▶ Parallel development between team members and teams – using something like swagger hub, developers can access the documentation as it is created and start generating the code in parallel
- ▶ The following is a demo of the API first approach which could also be the output from a design-first tool as it is basically a YAML document written to the openAPI standard
- ▶ Load the YAML into <https://editor.swagger.io/>
- ▶ This will syntax check and provide the interactive documentation to test the paths
- ▶ Note the Servers selectable from the YAML servers definition
- ▶ Note the groups based on the tags: general and people
- ▶ Note the section for componentised elements, in this case schemas
- ▶ Note the different POST options built into the YAML file are selectable
- ▶ Note: your server must be running at the selected domain and port
- ▶ You can create various outputs. e.g. a static page using generate client html2

# Code-first API development

- ▶ With the **code-first** approach, comments need to be added in swagger (openAPI) YAML style to enable document generation from the code comments
  - ▶ This keeps the documentation with the code and enables you to serve up the documentation as an API endpoint
  - ▶ The following demo is of a simple API and the associated openAPI documentation using the code first approach
  - ▶ You (or your organisation) choose the approach
  - ▶ You may find it easier to write your documentation in the openAPI editor an endpoint at a time, then insert it into your code as you need to embed your YAML into comment blocks
  - ▶ The demo will explain
- 
- ▶ Note: in vscode, to select a column, click top left, then shift-alt drag to create multiple cursors
  - ▶ To indent a block: ctrl-]
  - ▶ Set tab size: settings | editor tab
  - ▶ You could argue that this approach makes the code file messy - or a counter argument is that the code and documentation are in one place



# Authentication and Authorisation

# Learning Objectives

- ▶ On completion of this section, you will be able to:
- ▶ Explain the difference between authentication and authorisation - providing use cases
- ▶ Explain the construct and use-case of a Json Web Token (JWT)
- ▶ Explain the difference between base64 and base64url
- ▶ Provide some key attributes of a JWT when used in authentication and authorisation
- ▶ Explain at a detailed level, step by step:
  - ▶ Authorisation code flow in detail to illustrate , authorisation and authentication using OIDC



# First Party Authentication Approach

- ▶ Authentication is "am I sure you are who you say you are"
- ▶ Create an account
  - ▶ User enters a username and password into an html form to register an account
  - ▶ This is posted to the server using https POST
- ▶ Add the hashed password to a database (maybe salt and pepper)
- ▶ The user's roles and access authorisation are added to the database if required
- ▶ User logs in by sending credentials, server creates a hash of the password, searches for user by username, hashes the password and checks it against the stored hash
- ▶ If ok, the server creates a session object with a random session ID, user ID (primary key), roles/permissions and an expiry date and time. This is stored in memory or a memory-based database like Redis
- ▶ Server creates a session cookie to avoid re-authentication during the session
  - ▶ Opaque meaning the content is of no value to anyone as no user data is stored in it - it only makes sense to the server
  - ▶ Typical opaque session cookie: **connect.sid=s:<sessionId>.<signature>;**
  - ▶ sessionId: random number used by the server to lookup the session object from the database
  - ▶ signature: HMAC of the ID for symmetric digital signature. HMAC checked on return for tampering
  - ▶ Cookie is returned to client. HTTP headers are set: HttpOnly; Secure; SameSite=Lax
    - ▶ **HttpOnly**: protect from JavaScript. JavaScript cannot access the cookie stored on the browser
    - ▶ **Secure**: only send cookie from client if using https
    - ▶ **SameSite=Lax**: may be provided if necessary to allow get and redirect cross-site but not POST

- ▶ When something is clicked on the page and a request is made to same origin, cookie is sent automatically
- ▶ Server checks the cookie signature (HMAC) and if ok, retrieves the session object and checks for expiry
- ▶ If the session has expired (or has been auto deleted), you may want to extend it or redirect to a login page
- ▶ If not expired, the application code can check the access or role before allowing access to any protected resource
  - ▶ e.g. access to a specific API endpoint such as writing to the database or reading specific data
- ▶ **Benefits**
  - ▶ Nice and simple, full control, no external dependencies
- ▶ **Challenges**
  - ▶ Need to manage password resets, email verification, MFA, all aspects of credential security, account blocking, audit, responsible for any data breach regarding user credentials even if using salted hash as they may be brute forced if the salt is known
- ▶ When there are only a few hundred users, this approach is probably ok. However, when scaled to thousands or more, it's a big deal and best left to identity specialist so offload this responsibility to a third party service
- ▶ Third party authentication and authorisation is very popular with web applications

# JSON Web Token

- ▶ Security Tokens - used by OAuth and OIDC
  - ▶ JSON as opposed to SAML XML tokens so simpler - more lightweight
- ▶ JWT is an open standard
  - ▶ (RFC 7519) to securely exchange information using JSON objects
  - ▶ It's an encoded string, which is URL safe (base64url) so no + / = issue with base 64
- ▶ Token is cryptographically signed, similar to SAML
  - ▶ Data is not encrypted so use TLS if you need protection
  - ▶ It can be signed using symmetric HMAC or asymmetric RSA or ECDSA (Elliptical curve Digital Signature Algorithm)
  - ▶ The digital signature algorithm is fed all the base64-url data to calculate the signature
- ▶ JWT has 3 parts:
  - ▶ header - crypto algorithm used
  - ▶ payload - plain data. It contains no credentials
  - ▶ signature - digital signature. Generally using an asymmetric key method but symmetric is available
  - ▶ each is base64url encoded separated by a '.' (full stop)

# Example

- ▶ Header example:

```
{
 "alg": "RS256", # RSA is the asymmetric key type. SHA256 is the hash type for the signature
 "typ": "JWT",
 "kid": "R4HiYLxTu-b_fG3JxKBqZ" # A lookup for the public key needed to decode the signature
}
```
- ▶ Payload example – this is where the **claims** about an entity (typically a user) are made

```
{
 "sub": "1234567890",
 "name": "John Doe",
 "iat": 1516239022
}
```
- ▶ The final part is the signature which is also base64url encoded
- ▶ Above example JWT would look like this:  
  

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```



eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJnaXZlb19uYW11IjoieVVG9ueSIsImZhbWlseV9uYW11IjoieTmljb2wiLCJuaWNrbmFtZSI6InRvbnkiLCJuYW11IjoieVVG9ueSBOaWNvbCIiInBpY3R1cmUiOiJodHRwczovL2xoMy5nb29nbGV1c2VyY29udGVudC5jb20vYS9BQ2c4b2NKOjcwZ041RG5Xd0JHTlJzSzY0Z2ptNTU0dmE5UjdrSWpJVHVKQ09zS2pT

akeE9czk2LWMI6InRvbnluc2NhbGUiOiJlbi1HQiIsInVwZGF0ZWRfYXQiOiIyMDI0LTAYLTE0VDIwOjMxOjI3LjQ2MVoilCJlbWFpbCI6InRvbnluaWNvbEBnbWFpbC5jb20iLCJlbWFpbF92ZXJpZmllZCI6dHJ1ZSwiaXNzIjoiaHR0cHM6Ly90b255bmljb2wudWsuYXV0aDAuY29tLyIsImF1ZCI6Ikv5bkd3T1ROZGhIcEFXYkxzWDN3Y0lBbWUwMlNxaDYzIiwiaWF0IjoxNzA3OTQyNjg3LCJleHAiOjE3MDc5Nzg2ODcsInN1YiI6Imdvb2dsZS1vYXV0aDJ8MTAxNDQ1NzEzOTA0MjkmZmZzNzYxIiwic2lkIjoieUlJQbG9ZRWt3VHg0aUZPXzg0bGNWazVWIiwibm9uY2UiOiJaWUZhOXd4WkJqR3Fuc1o4Z1BOemZmb1NMBUNDQVFOUzdTc2RmeXZNa2NvIn0.X8mJaQpJ50tEBg7kbE1CHHiZdxJ1r4MsKqtFDgNtVVg

# JWT v JWS v JWE

- ▶ A confusing number of J things
- ▶ JWT is effectively an umbrella term for a base64 encoded token
- ▶ JWT is either a JWS or a JWE
- ▶ JWS is a JSON Web Signed token as described previously. Signed token but not encrypted
  - ▶ Provides integrity, authenticity and non-repudiation due to the signature if using asymmetric keys
  - ▶ Has 3 parts as shown previously - each base64 URL encoded
- ▶ JWE is a JSON Web Encryption token. JWT Payload is encrypted not signed
  - ▶ Provides confidentiality
  - ▶ Typical use case is application session tokens so they can only be read by the server on decryption
  - ▶ Has 5 parts, each base64 url encoded
    - ▶ Header: alg that protects the encryption key, enc - encryption algorithm (e.g. A256GCM")
    - ▶ encrypted key: if alg=dir, no key (default). Could be shared secret - encrypted with recipient pub key.
    - ▶ iv: initialisation vector. A random number used by the encryption algorithm
    - ▶ ciphertext: the encrypted payload
    - ▶ tag: value created by the encryption algorithm - like an HMAC value

## Claims list

- ▶ nonce: Number used once. Random number to avoid replay attack. Server can check it hasn't seen it before
- ▶ alg: Signature algorithm. e.g., HS256=HMAC + SHA256, RS256=RSA + SHA256, ES256=Elliptic curve + SHA256
- ▶ x5t: X509 certificate thumbprint. Hash of the issuer's / signer's public key. Used to uniquely identify the public key in a compact form
- ▶ kid: Key ID. Unique identifier for the crypto key used to sign the JWT. i.e. used to find the key in a key vault to use to verify the signature
- ▶ aud: Audience. Typically, the name of the resource server or services the JWT is intended for
- ▶ iss: Issuer. Typically, identifier or url of issuer.
- ▶ iat: Issued at. Num seconds after Unix epoch (Jan 1<sup>st</sup>, 1970, UTC). i.e. data and time of issue
- ▶ nbf: Not before. It can be issued before it can be used
- ▶ exp: When it expires
- ▶ acct: User account or profile
- ▶ acr: Authentication Context-class Reference: e.g. low is UN & PW. medium may mean 2fa. Level of auth strength of the process used to create the token
- ▶ aio: Azure AD object Id. Unique identifier of the user within the Azure AD environment
- ▶ amr: Authentication methods reference. Authentication methods used during authentication. e.g. pwd, 2fa,otp
- ▶ appid: UUID of the client that requested the token. 128 bits in 8-4-4-4-12 hex digits format
- ▶ appidacr: Strength of authentication of the application authentication. Similar to acr but for the app
- ▶ auth\_time: time the user was authenticated
- ▶ family and given name need no explanation
- ▶ ipaddr: IP address from where the authentication request came
- ▶ name.: Name to display - e.g. on a web page. Often given and family name
- ▶ oid: object ID. UUID of the user object in Azure active directory to identify the user in the directory
- ▶ onprem\_sid: User active directory ID in situations where on-prem AD is synchronised with Azure AD
- ▶ puid: persistent user ID. Single ID used across multiple accounts
- ▶ rh: Refresh history. How many times has this been refreshed. Usually at least a number. But this is Microsoft - they don't always follow standards
- ▶ scp: Scope. Specifies the permissions granted and what actions the app can perform on behalf of the user
- ▶ signin-state: sign in status. Values vary on ID provider. e.g. pending, success, failure .... kmsi means keep me signed in. Long session or refresh token
- ▶ sub: Subject. UUID of the subject, typically the user or whatever the token signifies within the scope of the issuer. In this case, Azure AD
- ▶ tid: Tenant ID. Azure tenancy the user belongs to.
- ▶ unique\_name: Could be unique username, email address or something else
- ▶ upn: User principal name. Basically, the unique user again.
- ▶ uti: Token ID. UUID of the token itself
- ▶ ver: Version of the token format / standard

# Authentication and Authorisation

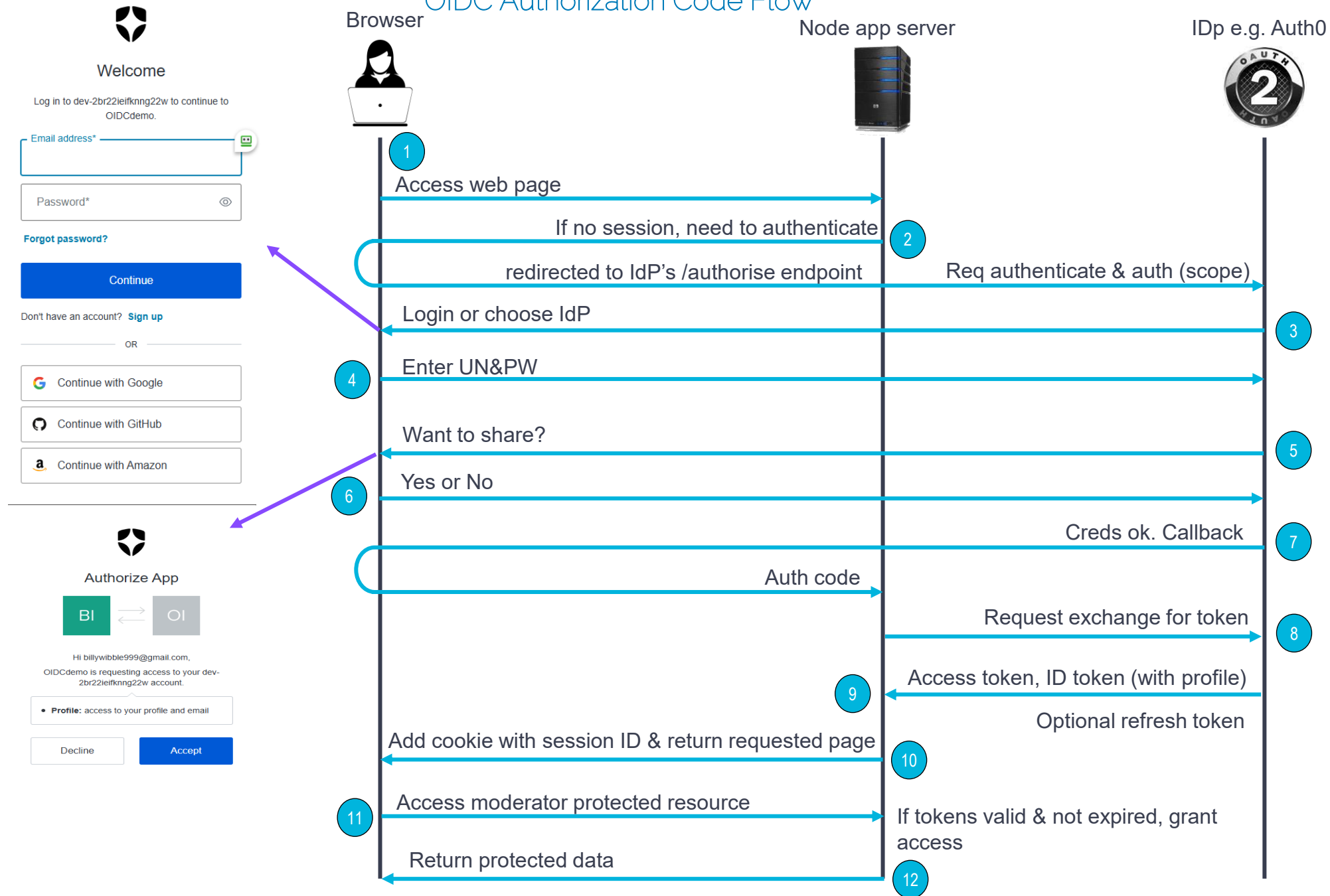
- ▶ Authentication is "am I sure you are who you say you are"
- ▶ Authorisation is "once authenticated what are you allowed access to"
- ▶ The most modern approach to authorisation is Open Authorization (Oauth), now at version 2 so OAuth2
  - ▶ Open standard for secure authorisation to protected resource
  - ▶ Used JWT to grant access
  - ▶ There are various flows (process), we will focus on the popular **Authorisation Code Flow**
  - ▶ Oauth only creates an authorisation token and optionally a refresh token - NOT an authentication token
- ▶ For authorisation, Auth0 does require authentication but doesn't care how its done but as the process does ask for authentication, only minor additions are required to perform authentication as well
- ▶ OpenIdConnect (OIDC) is a layer on top of OAuth enabling authentication
- ▶ As OIDC is an enhancement to OAuth, just using OIDC means you are also using Oauth so OIDC can be used for authentication and authorisation
- ▶ I'll talk through the Authorisation code flow and explain how we authenticate as well



# Prerequisites to Delegated Authentication & Authorisation

- ▶ An application developer creates an account with an Identity Provider (IdP) such as Auth0 (an Okta company) and registers their application to use OAuth2/OIDC flows.
  - ▶ Others are available such as Microsoft, Google etc. Same sort of approach
- ▶ Register your app and provide some details such as:
  - ▶ You will be given a URL to direct your server to, to invoke authentication at the IdP
  - ▶ You will be provided with a client ID and secret specific to your app so it can authenticate to the IdP
  - ▶ You provide a callback URL: which endpoint do you want the IdP to return to your app
  - ▶ Maybe provide Logout URL: e.g., goodbye page
  - ▶ Choose which additional IdP services you want to provide as options e.g. Microsoft, Google, Github ...
- ▶ Once the app is registered and operational, you can use it to create an identity token which represents an authenticated user, and an access token which contains user roles and / or access permissions

# OIDC Authorization Code Flow



# OIDC Authorisation Code Flow

- ▶ 1. User requests a protected page so app requires user authentication and maybe profile information (e.g., name, email, photo) to personalise the page
- ▶ 2. The server checks for an existing application session. If none exists, it redirects the user to the Authorisation Server (IdP) using HTTP 302 and a Location header pointing to the IdP's `/authorize` endpoint
- ▶ The redirect includes: - `response_type=code`; `client_id` (issued when the app was registered); `redirect_uri` (where the IdP should send the user back); `scope` (permissions requested; include "openid" if identity is required) and a couple of additional security values
- ▶ 3. The Authorisation Server displays its login page. The user chooses how to authenticate (username/password, social login, enterprise login, etc.)
- ▶ 4. The user submits credentials. The IdP verifies them, loads the user's profile, and checks whether the user has previously granted consent for the requested scopes
- ▶ 5. If consent has not been granted before, the IdP displays a consent page asking the user to allow the application to access the requested data
- ▶ 6. If the user approves, the IdP records the consent
- ▶ 7. IdP redirects the browser back to the application's `redirect_uri` with an authorisation code. If the user denied sharing, the IdP redirects back with an error
- ▶ 8. The application backend exchanges the authorisation code for tokens by calling the IdP's token endpoint. It sends the code, `client_id`, `client_secret`, `redirect_uri`. The `redirect_uri` enables the IdP to check confirm the created token contains the same URI as that redirected via the browser
- ▶ 9. The IdP returns tokens: - access token (OAuth2); ID token (OIDC, only if "openid" was requested); refresh token (optional, depending on IdP settings)
- ▶ 10. The application validates the tokens (signature, issuer, audience, expiry) and extracts identity information from the ID token. It creates an application session and stores the tokens securely (e.g., in a server-side session in memory, Redis or whatever)
- ▶ 11. The application returns the protected page to the user. The browser stores the session cookie automatically
- ▶ 12. When the user later accesses another protected resource, the browser sends the session cookie. The application checks token validity (e.g. expiry) and authorisation (roles/permissions) and returns the protected data or an error
- ▶ If tokens have expired, either: the user is redirected back to the IdP to authenticate again, or, if a refresh token is available, the application uses it to request new tokens without user interaction

# Summary

- ▶ Authentication: are you who you say you are
- ▶ Authorisation: when we know you are who you say you are, what are you authorised to access
- ▶ OAuth, now at version 2 is an **authorisation** protocol
- ▶ Although OAuth2 flow does include authentication, but it doesn't retain any identification data. I.e., using OAuth, we don't know who authenticated or when
- ▶ OpenID connect extends the authorisation capability of OAuth2 to also enable **Authentication**
- ▶ Active Directory works within one domain to hold user credentials, resource access and SSO
- ▶ Active Directory Federation Service enables AD to be accessed across domains using SAML
- ▶ SAML is an XML file used to securely exchange information between an IdP and an SP across domains for authentication - typically using ADFS / AD
- ▶ OAuth2 is used for delegated authorisation - can be used with SAML and JWT
- ▶ JWT is a JSON web token used to securely exchange data. Used typically in place of SAML as it's lightweight based on simple JSON structure rather than a complex and verbose XML structure



NoSQL database

# Learning objectives

- ▶ Upon successfully completing this section, you should be able to:
  - ▶ Explain the difference between a SQL and a NoSQL database
  - ▶ Explain the differences between a relational database and a document database
  - ▶ Create a MongoDB Mongoose schema, database and collection
  - ▶ Create an application server with routes to carry out CRUD operations on a MongoDB collection
  - ▶ Explain how http methods can be used to implement CRUD operations
  - ▶ View the data in a collection using MongoDB Compass

- ▶ A NoSQL database is technology that stores data but not in a tabular format
- ▶ With a structured / relational database, each row has the same number of columns. If you have a million rows and want a new column, all million rows will have a new column added whether that makes any sense or not
- ▶ Tables must have a schema defining the type allowed in each column
- ▶ The tabular structure is great when you know in advance what your data content will look like and every entry in the table needs the same data. i.e. the data is structured
- ▶ NoSQL databases, data are not stored in a tabular form - more flexible for semi-structured and unstructured data
- ▶ NoSQL doesn't mean that you can't use SQL but not only SQL. Any database that doesn't store data in a tabular form can be considered NoSQL. Popular products currently include MongoDB, Apache Cassandra, Apache Hbase, Apache CouchDB, Neo4j, RavenDB, Redis and OrientDB
- ▶ MongoDB stores data in a JSON-like format. It is a distributed database so has high availability through horizontal scaling. This is called a document database
- ▶ Cloud hosts provide their own proprietary databases and open-source databases with their own support wrapper
- ▶ I'll focus on a brief intro to MongoDB
- ▶ Available at <https://www.mongodb.com/try/download/community> for local install. It is available as a container too

# MongoDB

- ▶ MongoDB is a very popular highly scalable, distributed NoSQL document-based database
- ▶ Document databases store data in collections which can be likened to tables in a database but not tabular
- ▶ It stores each item as a document which can be likened to a row in a tabular database but each document doesn't need the same data entries (columns) as in a tabular database. It stores the data in a json-like format

## MongDB Document

```
{
 "_id": {
 "$oid": "635c33304f31ed7bbb702927"
 },
 "id": 23,
 "type": "programming",
 "setup": "Why do programmers always mix up Halloween and Christmas?",
 "punchline": "Because Oct 31 == Dec 25"
}
```

## MySQL Table

| id | type | setup                                                     | punchline                |
|----|------|-----------------------------------------------------------|--------------------------|
| 23 | 3    | Why do programmers always mix up Halloween and Christmas? | Because Oct 31 == Dec 25 |



- ▶ MongoDB can be accessed from the command line or from a GUI called compass – Mongo equivalent of Workbench
- ▶ Using compass, use the default connection string: `mongodb://localhost:27017` where host and port depend on where your mongo instance is
- ▶ We can use this to view and modify data directly or check if our app has updated data correctly. Tip: to sort documents in reverse order in compass, use: `options | sort | { _id: -1 }`
- ▶ You can also try out commands on the compass command line
- ▶ In a similar way to loading a MySQL database driver for node, there are a few options for MongoDB too. We will use a package called Mongoose. This provides a driver and an object data modelling (ODM) approach to map code to a database
- ▶ An object relational mapping ORM extension is similar(ish) for relational databases e.g. sequelize for node - use JavaScript objects to interact with the database rather than SQL as sequelize converts the JavaScript to SQL
- ▶ MongoDB doesn't need a schema so you can add pretty much anything. For example: create database e.g., `junk`, create collection e.g., `junkCollection`, add data. Note MongoDB will automatically give you an id, denoted `_id`, for every document added as the default primary key. Each document need not follow a defined structure (see example =>).
- ▶ However, if you want some structure, an object relational modelling extension can be used, we will use Mongoose. This can provide a full or partial schema

```
{
 "_id": {
 "$oid": "637fd added"
 },
 "name": "Tony",
 "car": "BMW 320D"
}

{
 "_id": {
 "$oid": "637fe05f60ab06607b055cf6"
 },
 "name": "phil",
 "dog": {
 "breed": "poodle",
 "name": "cuddles"
 }
}

{
 "_id": {
 "$oid": "637fe14660ab06607b055cf7"
 },
 "aNumber": 27
}
```

- ▶ In an object data mapping, you define a structure in your code that maps onto the collection. This way you can create an object in your code with similar or same names as in the collection, read or write the object's properties reads or writes the object to or from the database collection
- ▶ Using a ODM, enforces some structure that can be checked and an error generated if you deviate from it – if that's what you want to do of course; you have the choice. If a type is wrong, Mongoose will throw an error. If you try to save data and a mandatory property is missing, you get an error
- ▶ Mongoose provides several functions for interacting with the database. These are here: <https://mongoosejs.com/docs/>
- ▶ I'll use the local and container versions

# Mongoose Schema

- Mongoose requires a schema; a schema lists properties and types that will map to the document in the collection

```
const bookSchema = new mongoose.Schema({
 author: {
 type: String,
 required: true
 },
 title: {
 type: String,
 required: true
 },
 pages: {
 type: Number,
 required: false
 },
 keywords: {
 type: [],
 required: false
 },
 rating: {
 type: Number,
 required: false
 },
 reviews: {
 type: [
 {
 name: { type: String, required: true },
 comments: { type: String, required: true }
 }
],
 default: [] // Can be empty
 }
})
```



```
_id: ObjectId('635aecf40fccd98b2c56c617')
author: "Liz Truss"
title: "Memoirs of a Prime Minister"
pages: 1
key-words: Array
 0: "fail"
 1: "deluded"
 2: "cost"
 3: "shortest"
rating: 1
reviews: Array
 0: Object
 name: "Alan Sugar"
 comments: "You're fired"
 1: Object
 name: "Boris Johnson"
 comments: "I may have had a few beers but didn't wreck the economy"
```

# Create a model of the schema

- ▶ A model is an instance of the schema for a document. i.e. the schema is a definition and the model is an object with lots of methods we can use which is based on the schema definition

```
const bookModel = mongoose.model('book', bookSchema)
```
- ▶ An instance of a model is called a document as it is an object that maps directly onto a collection document. i.e. Write to the document object then simply save it into the collection
- ▶ e.g. `bookModel.find()` will find all documents in the books collection and return them as an array of model objects. That is, the documents **with all the methods too**. `res.json(result)` will work out that it needs to access the `_doc` elements of the objects but if you just want the document info, you can use `bookModel.find().lean()`
- ▶ `const books = await bookModel.find()` would return all the documents in the collection in the books array
- ▶ There are many functions available in Mongoose most of the ones we need are illustrated in the demonstration application. For example, you can search for data using regular expressions:
- ▶ `books = await bookModel.find({ author: { $regex: like, $options: 'i' } })` This emulates the SQL LIKE option As it uses the regex in 'like' to hold the regular expression and options in this case 'i' means case insensitive
- ▶ More information can be found in the Mongoose documentation. You can write code that interacts directly with MongoDB but Mongoose makes it a bit easier and provides the ODM capability. They have tried to make the functions as similar to MongoDB functions as they can
- ▶ Note: Mongoose functionality is promise-based so we don't need to wrap any of the functions in a promise, you can use `async / await`, or `.then()` which makes things a bit easier for us

# Deploy into the cloud

- ▶ You could create an IaaS VM and install MongoDB or
- ▶ You could create an IaaS VM, install docker and pull a MongoDB container or
- ▶ You could deploy MongoDB as a PaaS docker container in web apps
- ▶ Azure Cosmos supports multiple dbms including MongoDB
- ▶ You could deploy MongoDB at the SaaS level – but rather than call it Software as a Service, it's referred to as Database as a Service – DBaaS but is at the same cloud level as SaaS but more descriptive for databases
- ▶ The DBaaS offered by MongoDB is available on all major cloud platforms including Azure

# Create Atlas account and connect

- ▶ You create a free account and deploy into a fully managed, high performance, scalable MongoDB database:
  - ▶ <https://www.mongodb.com/home>
  - ▶ Start free | Fill in details & create | Verify email | login
  - ▶ Fill in the tell us about you
  - ▶ Select the FREE offering
  - ▶ Keep the default region, Ireland in my case, or it may move you onto a dedicated plan which will cost
  - ▶ Leave cluster tier, additional settings and Cluster name as is. Double check you're on the free shared plan before pressing "Create Cluster"
  - ▶ Choose Username and Password for authentication and fill in the boxes with what you want BUT DON'T FORGET THEM
  - ▶ Create user
  - ▶ Connect from: local environment
  - ▶ IP List: click add my current IP address. You can add others later if you need (It doesn't fill in the boxes, it shows them at the bottom of the box)
  - ▶ Finish and close
  - ▶ Termination protection: Setup if you want, or click close. If you pressed setup, just scroll down and enable
  - ▶ We are using the free M0 cluster which is free so don't change that. Press review changes | apply changes
  - ▶ That's it. Click connect (half way down the page) then Connect your application
  - ▶ Choose node.js driver, newest version they have, and copy the connection string for your application. It will look like:  
mongodb+srv://yourname:<password>@cluster0.xwgfgyz.mongodb.net/?retryWrites=true&w=majority
  - ▶ A big GOTCHA here is that it doesn't say you need to define your database here so the connection string is incomplete. It should be:  
mongodb+srv://yourname:<password>@cluster0.xwgfgyz.mongodb.net/**databasename**?retryWrites=true&w=majority
  - ▶ So in the demo case, it's bookstore
  - ▶ In compass, connect | new connection | add your connection string and you should connect to your atlas database environment
  - ▶ Create database | bookstore (or whatever you want)
  - ▶ Collection name | books (or whatever – but stick to these if you want to use the example code as-is)
  - ▶ Create database
  - ▶ Click bookstore | books | import data | JSON | find your json file
  - ▶ Add the connection string to your code. Bear in mind that in this basic approach, your username and password are in the string. We can change that later using .env but just check it works first
  - ▶ If you have issues with IP settings due to using VPN or logging in from different locations, go to your atlas account | network Access | edit | add any IP or manually type in 0.0.0.0/0



# Scaling & Resilience

# Learning Objectives

- ▶ Upon successfully completing this section, you should be able to:
  - ▶ Explain the term "elastic compute" and cite its benefits
  - ▶ Explain auto-scaling, draw an auto-scaled architecture with load balancer
  - ▶ Explain how you can make your solutions highly available and resilient
  - ▶ Compare and contrast vertical and horizontal scaling in "on-premises" and on cloud
  - ▶ Explain the key components that make an availability zone resilient to failure
  - ▶ Explain how areas above, can be used to make your distributed systems more resilient



- ▶ We have spent a lot of time on virtualisation and containerisation as these are the most popular modern methods of software deployment in enterprise systems.
- ▶ We will consider how these approaches can help in scaling our systems and improve their resilience and availability
- ▶ High availability - focusses on the system staying up in the event of expected system failure
  - ▶ e.g. a container or VM fails, this is detected, deleted and a new one created
  - ▶ during that short period, the system may be a little under-powered but it will recover
  - ▶ Employs redundancy to cover failures - e.g. dual power, dual NICs, dual network switches
  - ▶ Spread VMs across multiple racks, data replication - typically x 3 in the same data centre
  - ▶ A primary goal is for continuous up-time - i.e. no unrecoverable failure in business processes
- ▶ Resilience
  - ▶ An overarching term and includes high availability
  - ▶ A system may be highly available but in one data centre, resilience would spread the load across data centres, even regions
  - ▶ A system may be resilient in that disks and data are available, but corrupt data would be replicated so available but useless. A resilient system would detect the corruption before replication, stop the replication, promote a clean replica if there is one or try to rollback to a known clean state
- ▶ These are often used interchangeably but they are different
  - ▶ Well designed systems are likely to be resilient and highly-available

- ▶ The term "Elastic compute" is popular today and is probably a key reason cloud hosting took off. It refers to:
  - ▶ the ability of a computing infrastructure to dynamically adapt and scale its resources based on demand
  - ▶ cloud computing environments where the allocation and deallocation of computational resources, such as virtual machines, can be adjusted automatically in response to varying workloads
  - ▶ the ability to efficiently handle fluctuations in demand, ensuring optimal resource utilisation & performance yet minimising costs
  
- ▶ Key components of elastic compute include:
  - ▶ **Virtualisation.** This is the foundation of elastic compute. It allows rapid deployment and scaling of VM instances as needed
  - ▶ **Auto-scaling.** Automatically adds or deletes VMs based on predefined policies or triggers such as CPU, number of requests
  - ▶ **Load Balancing.** This distributes incoming network traffic across multiple computing resources to ensure even utilization and prevent overloading of any single instance
  - ▶ **Containerisation.** Containers can be rapidly deployed, scaled, and moved between environments in response to varying workloads - particularly if used with an orchestration service such as Kubernetes
  - ▶ **Orchestration.** This deploys containers and VMs, distributes them automatically for elasticity to provide resilience and high-availability - e.g. Kubernetes

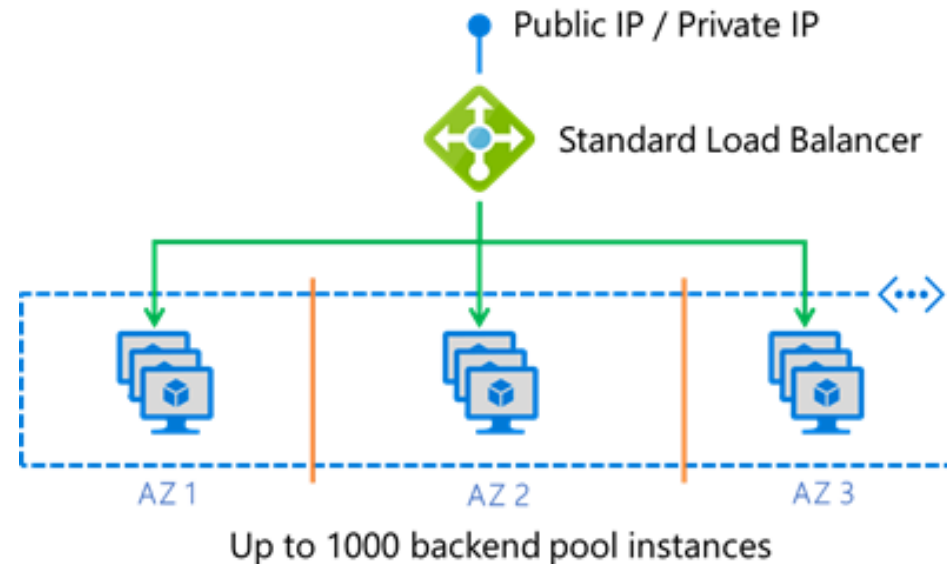
## Scaling - on prem

- ▶ **Vertical** scaling is making the current machines bigger or smaller (unlikely)
  - ▶ what you do when you find that your PC can't keep up with the latest application or game for example. You get a graphics card with a better GPU or maybe you are keen on creating videos and it takes ages to render them – you basically buy a new machine with more RAM, storage, faster CPU or whatever
- ▶ In business you see an increase in demand. What do you do? Write off all your servers and buy bigger ones?
- ▶ **Horizontal** scaling. Don't increase the size of the machine but provide more of them and load balance across them. This avoids the server write-off but still requires servers to be bought - this is not elastic
- ▶ This can work well, but in large organisations you need to:
  - ▶ create a business case
  - ▶ secure funding - for larger sums this needs to go to the treasury for approval in government
  - ▶ work with finance and commercials. Commercials will want to put the requirement out to tender for the best price
  - ▶ submit the order, finance have to pay the bill, the machines are shipped
  - ▶ arrange installation and configuration: racked, cabled, network configured, firewalls set up, VMs built, etc
- ▶ All this can take months - so scalability is possible on-prem but can be very slow
  - ▶ If the workload reduces, you can't get your money back on the servers, they just sit there getting warm. This is not elastic
- ▶ Buying servers is an up-front cost called Capital Expenditure (CapEx)

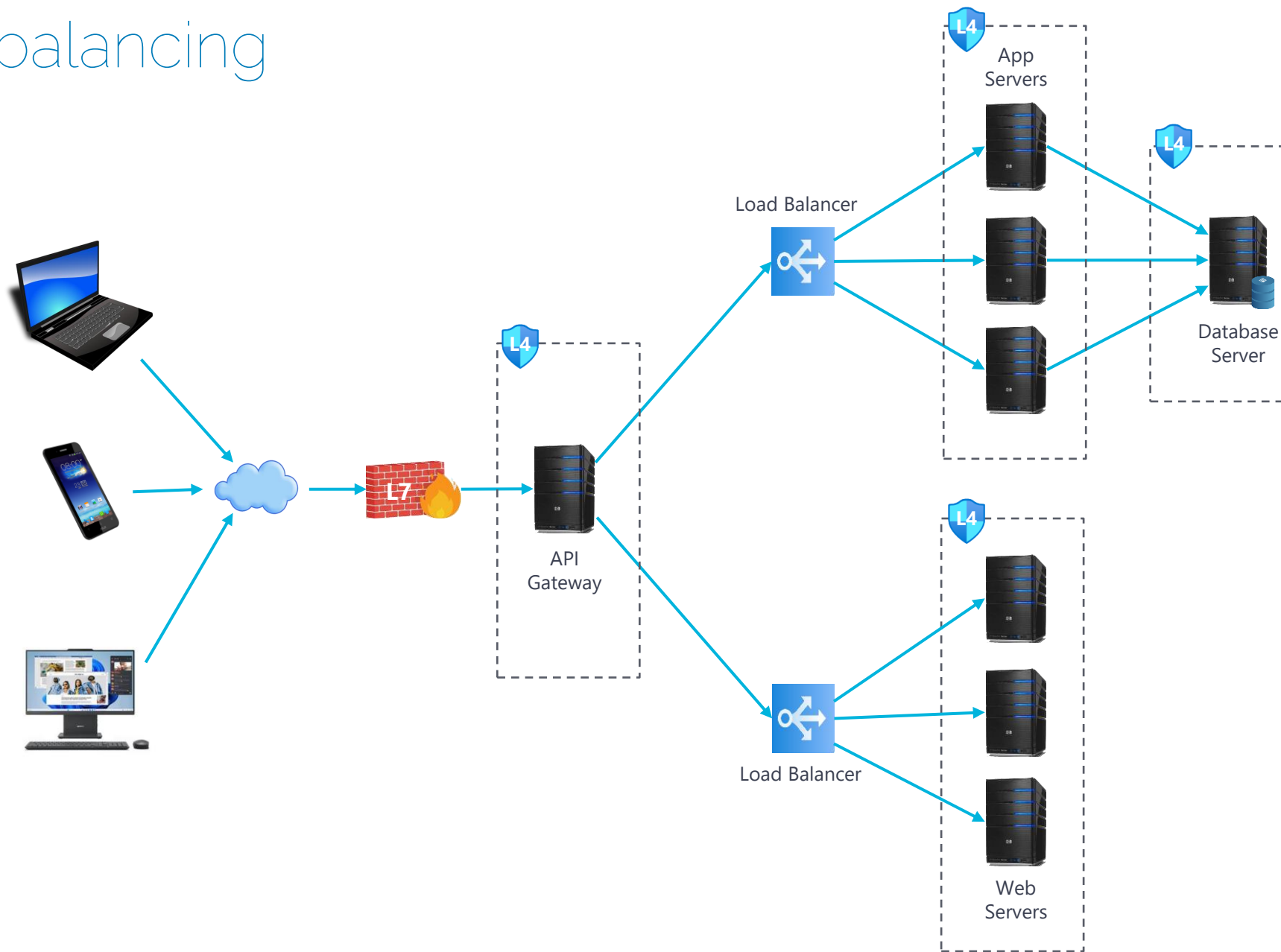
## Scaling in the cloud

- ▶ This is very easy and quick in comparison to on premises
  - ▶ Cloud providers have invested in huge resources that they rent out
  - ▶ They manage the variable supply and demand. As one person releases a machine, they can accommodate someone else
  - ▶ Vertical scaling is possible. Choose a different machine size then migrate your applications, databases etc
- ▶ Scaling horizontally, elastically, is far more popular. Same type VMs created within minutes using autoscaling
  - ▶ Azure uses scale sets, AWS uses auto scaling groups
    - ▶ scale on criteria such as %cpu, %memory, number of requests etc
    - ▶ Video: Create an Azure scale set.mp4
    - ▶ Video: AWS elastic load balancing part 1-3.mp4
      - ▶ Shows autoscaling in action in response to changing client access demand
- ▶ You can also scale by replicating containers within or across vms
- ▶ However, adding 10 more VMs with 10 more IP addresses is impractical. Which would a user access?
  - ▶ Use a load balancer. This device has one IP address, but it knows the private IP addresses of the new machines
  - ▶ All traffic comes into the load balancer, and it distributes traffic according to a distribution algorithm or as simple as round robin

- ▶ A data centre offers high availability through redundancy including:
  - ▶ mains, generator, battery, PSU, NICs, vm rack deployment, replicated data - employs redundancy
- ▶ Deploy VMs across AZs in a region for resilience
  - ▶ AZs are located away from each other for disaster resilience
  - ▶ A load balancer can balance across AZs and scale sets
- ▶ VNet stretches across all AZs in a region
  - ▶ Subnets also stretch across all AZs in a region. i.e. it's like they are in the same building but geographically disperse

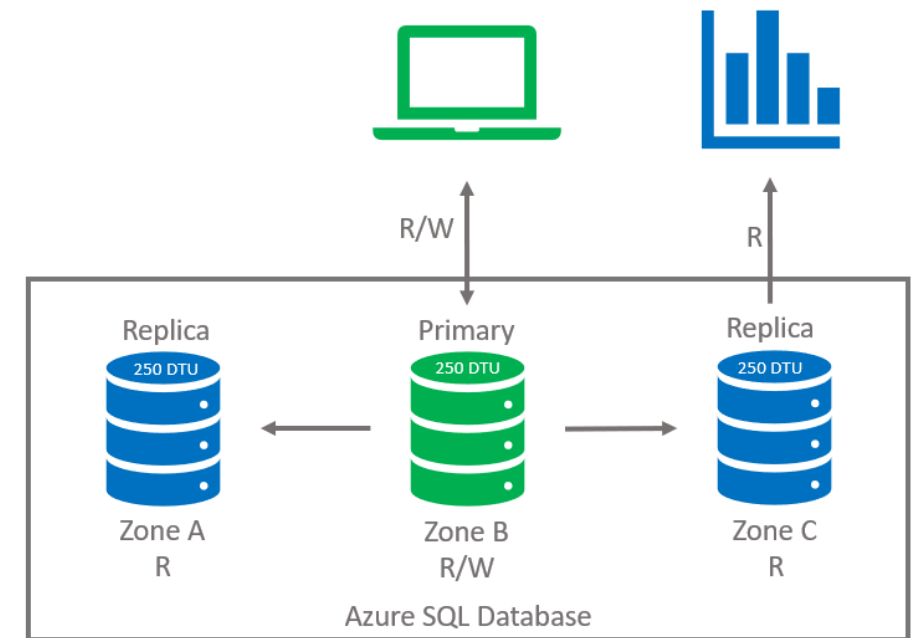


# Load balancing



# A Relational Database Scaling Pattern

- ▶ **Read-replicas:**  
Typically there are many more reads from a database than writes so one approach to try to overcome this is to create read replicas
- ▶ That is where the master server is the only one allowed to perform updates but those updates can be copied to read-only copies for others to access
- ▶ If these replicas are on different machines, they will be connected via a network so there will be some latency in updating the replicas
- ▶ They could be in different countries to keep read latency lower but update latency is longer
- ▶ This latency will create data inconsistencies, which may be acceptable as they may only be wrong for a very short time but it needs to be considered. i.e., what is the business impact
- ▶ The eventual consistency of the data over a short period of time is called **eventual consistency**
- ▶ This approach provides high availability as the read replicas can be in different data centres - or different regions but it comes at the cost of data consistency





# Container Orchestration using Kubernetes



# Learning Objectives

- ▶ Upon successfully completing this section, you should be able to:
  - ▶ Explain what Kubernetes is
  - ▶ What benefits does it offer over Docker Compose
  - ▶ Describe the K8s architecture
  - ▶ Explain the operation and key components of the control plane
  - ▶ Explain the following terms: Cluster, Node, Pod
  - ▶ Explain the key properties of the following objects:
    - ▶ Replica Set
    - ▶ Deployment
    - ▶ Service
    - ▶ ClusterIP
    - ▶ NodePort
    - ▶ LoadBalancer
    - ▶ ConfigMap
    - ▶ Secret
  - ▶ Explain how Kubernetes enables deployment of an application upgrade to multiple pod replicas with no downtime
  - ▶ Explain how we can rollback from a failed upgrade

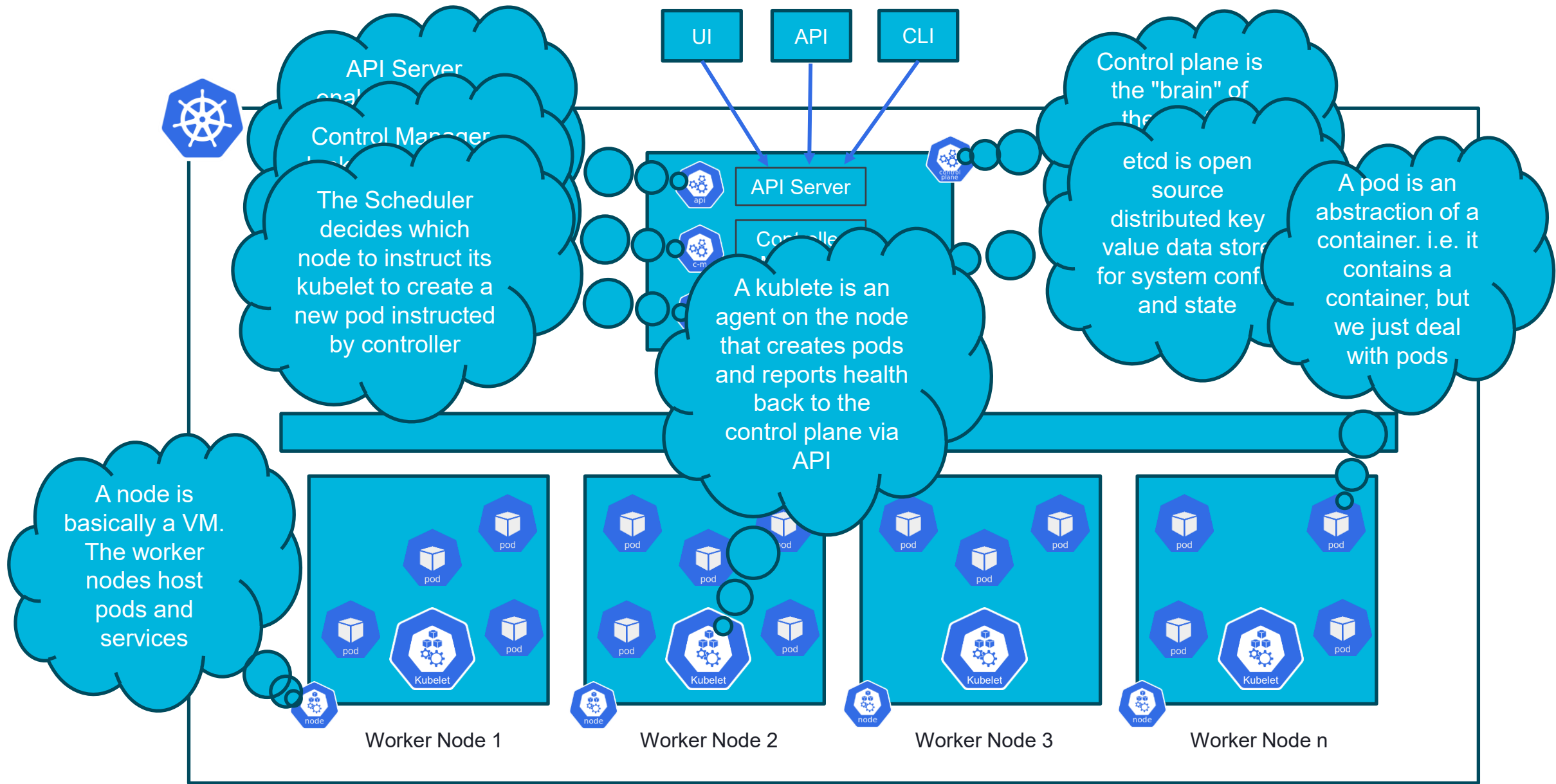
# Kubernetes

- ▶ To manage, monitor and scale many docker containers requires an orchestration application. Docker provides Docker Swarm, but the most popular container orchestration application is Kubernetes
- ▶ Kubernetes is free and open source. User is responsible for all infrastructure at an IaaS level. Responsibility is on the client, but they have full control
- ▶ To simplify container deployment and management, the major cloud providers offer container and Kubernetes services; these are PaaS services so they deploy and manage the infrastructure, i.e. network, VMs, storage, Operating system etc., for you
- ▶ PaaS can be a benefit in that a lot of responsibility is passed over to the cloud provider. A disadvantage is that it likely costs more, you have less control, and you are effectively locked-in to that supplier to some extent
- ▶ In AWS, various services are available
  - ▶ ECS - Elastic Container Service. This PaaS service is to some extent Amazon's bespoke version of Kubernetes or Docker swarm orchestration so is a vendor lock-in option. It can enable full control by deploying containers to EC2 IaaS, so you have detailed control, or Fargate which is the PaaS option for running containers
  - ▶ EKS - Elastic Kubernetes service. This PaaS service provides a Kubernetes orchestration platform. Deployment is via kops or CloudFormation templates so it's a bit more complex but less lock-in. AWS manages the control plane (which you pay for) and you pay for the EC2 VMs running in the cluster
- ▶ In Azure, the Kubernetes service is AKS (Azure Kubernetes Service) and the container service is ACI (Azure Container Instances). Other services will accept container images such as Azure webapps

# Kubernetes

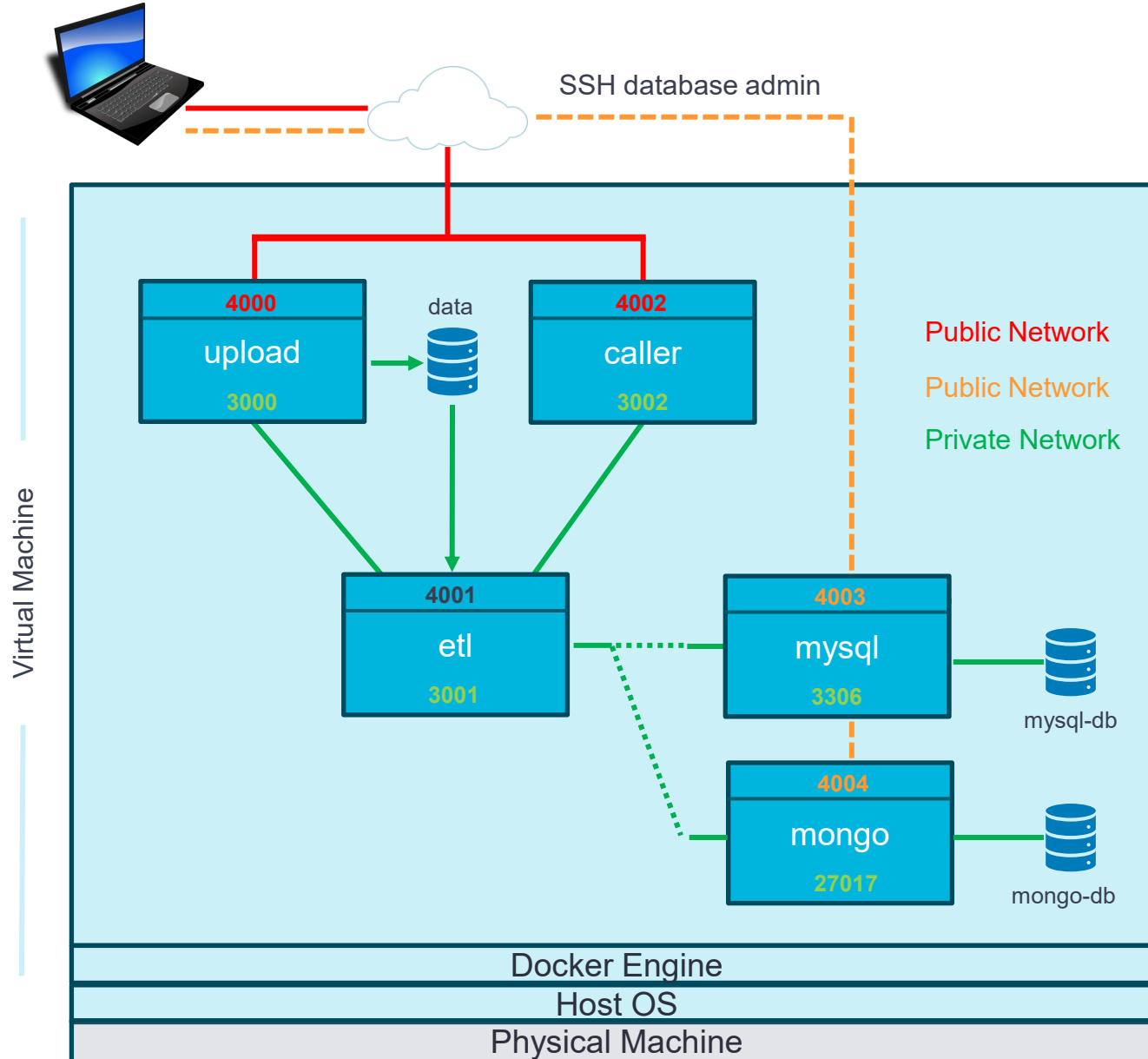
- ▶ A system could consist of hundreds or thousands of containers, so we need a means of being able to automate the creation, start, stop, delete and scaling of microservices; i.e., a container orchestrator
- ▶ We may want to scale containers in a vm and load balance, or scale vms, or both
- ▶ Kubernetes, coupled with the right architectural design, can enable system operation with basically zero downtime even during application upgrades
- ▶ Docker is used to create images - Kubernetes works with Docker to deploy and manage containers
- ▶ For simplicity Kubernetes is shortened to k8s (k, 8 letters, and s)
- ▶ The architecture consists of three key components:
  - ▶ A **cluster**: this encapsulates all other components
  - ▶ **Nodes**: these are typically VMs. There is a master node (control plane) that controls worker nodes
  - ▶ A **pod**: a pod lives on a node - there can be several. A pod is an abstraction of a container
- ▶ So, think of a worker node as a vm onto which we deploy pods and in a pod is a container - usually one
- ▶ The master node is responsible for deploying and managing pods on a node - e.g. if a pod dies, the master can detect it, delete it and create a new one
- ▶ As a vm (or the physical machine) is a SPOF, the master is capable of creating and managing multiple pods across multiple nodes across multiple AZs for high availability and resilience
- ▶ Using a Cluster Autoscaler open source component, K8S can interact with cloud services to create and destroy nodes on demand for higher level scalability

- 
- ▶ Kubernetes is popular for managing distributed systems as it enables:
    - ▶ High availability (no downtime) and resilience
    - ▶ Elastic Scalability - easily and automatically horizontally scale
    - ▶ Disaster recovery - can backup and restore data
    - ▶ Load balancing traffic across pods
    - ▶ Rolling updates and rollbacks - gradually rolls out update to each pod to avoid downtime
    - ▶ Can set resource limits to prevent containers from consuming excessive resources
    - ▶ The system architecture, resource limits etc are declaratively specified in YAML or JSON
    - ▶ Self-healing - if a pod fails, it is automatically detected, and a new one is created and the old one deleted
    - ▶ A deleted pod is recreated with a different name
    - ▶ Works on-prem and in all the cloud providers
    - ▶ Built-in security - network and pod policies and RBAC



Kubernetes cluster high level logical architecture

# Data Engineering Solution Architecture



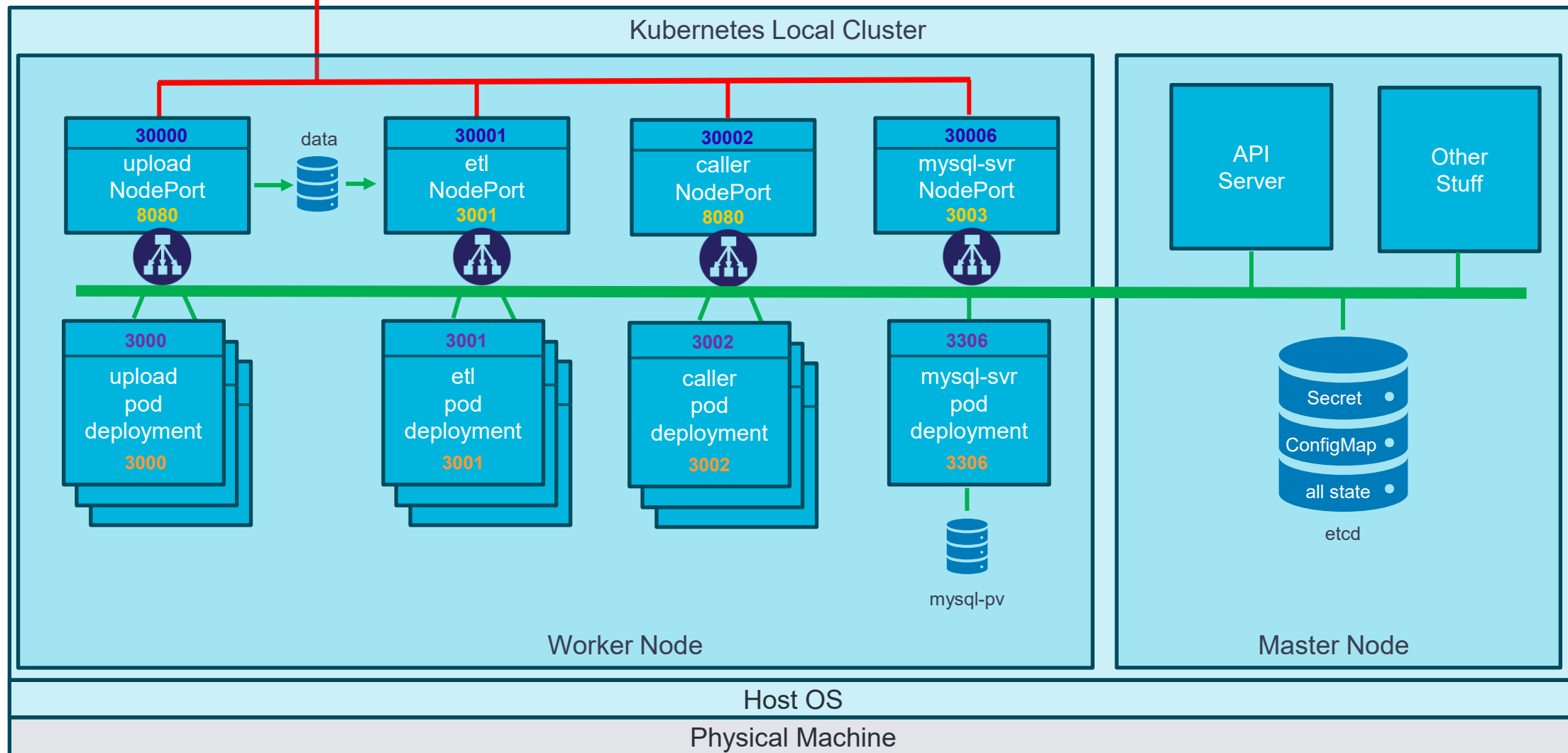
- ▶ We will deploy this architecture to Kubernetes on the local machine
- ▶ Kubernetes uses YAML to declaratively state the desired system configuration, i.e., pods, services, volumes etc
- ▶ I'll only deploy one database, but you have seen how easy it is to exchange them
- ▶ This can be redrawn to better see the k8s architecture

browser (localhost:nodePort) -> service(port) -> pod(targetPort) -> node app(env var)

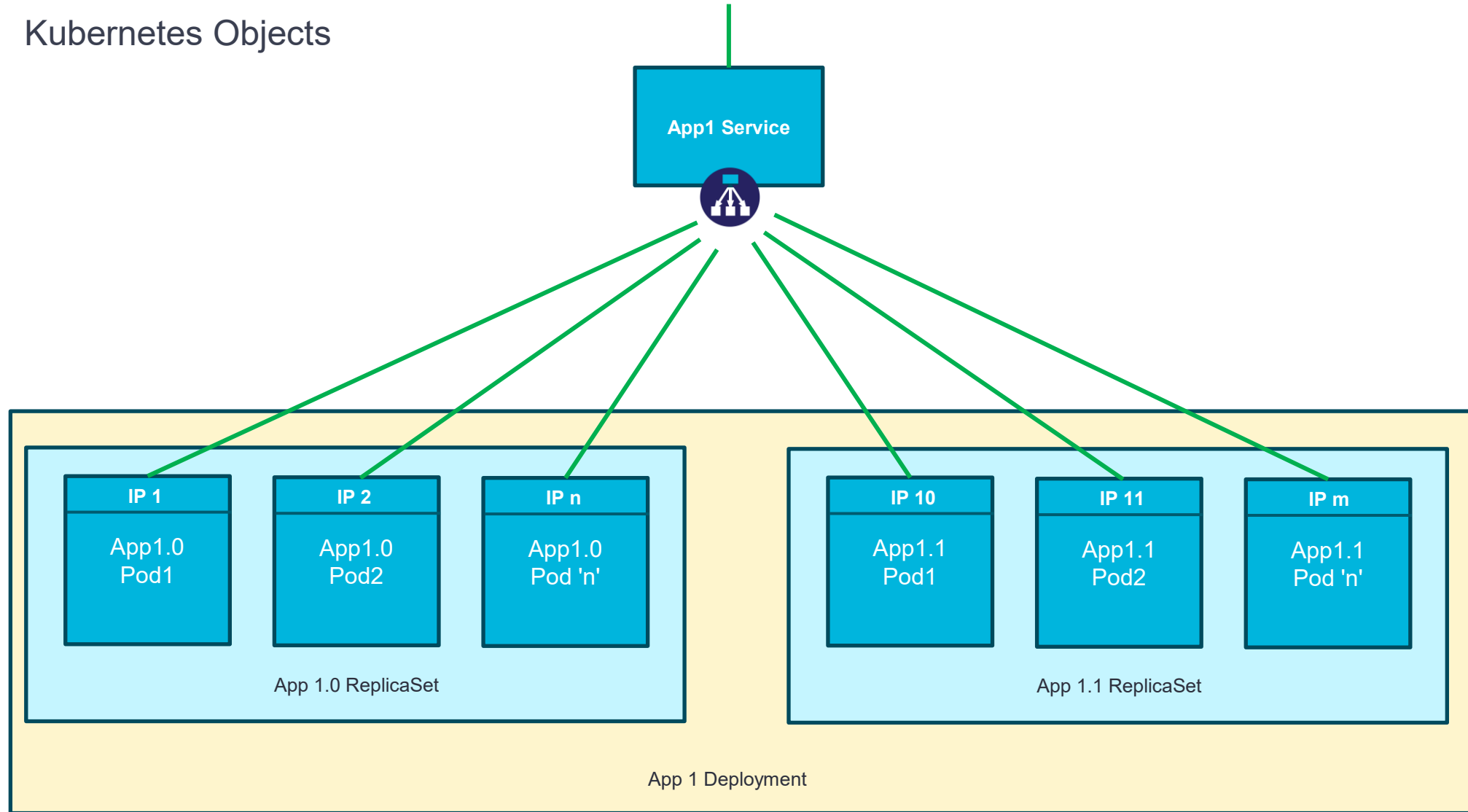


Public Network

Private Network



## Kubernetes Objects



Rolling update



# Kubernetes Objects

## ▶ Pod

- ▶ A pod is an abstraction of, generally a single container, or multiple containers
- ▶ Pods are the smallest deployable units of compute that you can create and manage in Kubernetes
- ▶ It could host a whole application or part of one
- ▶ K8S decides which node the pod will run on
- ▶ K8S automatically recreates a deleted pod but with a different name
- ▶ K8S runs one instance of each pod you define and allocates it a single IP address even if it hosts more than one container
- ▶ We don't interact with pods directly as they are ephemeral, e.g., memory leak may cause a pod crash. It is replaced, but the new pod may have a different IP address, so maintenance is difficult. We interact with services which access pods

# Kubernetes Key Objects

## ▶ ReplicaSet

- ▶ A single pod is unreliable. If it dies, you lose your service - at least for some time
- ▶ A replica set object can sit above a pod and is used to create multiple pod replicas - like a scale set or autoscale group
- ▶ You can specify the number of pod replicas you need then load balance across them
- ▶ A limitation to ReplicaSet is that an upgrade is a change to a container, so all the pods need to be deleted, upgraded and recreated. So, the ReplicaSet object will no longer be useful as it was created to manage specific pod version. We need a new replica set to manage the auto scaling of the new pods with the new containers - i.e. we have some downtime for an upgrade
- ▶ We can request replicas to be spread across nodes to protect against node failure

## ▶ Deployment

- ▶ A deployment sits above a replica set so can control, create and delete them. This enables rolling updates with no downtime
- ▶ Say we have a new image and apply it to the cluster. The deployment creates a new additional ReplicaSet with the desired number of pods and containers based on the new image. When it determines the pods are healthy, it will set the replica count of the old ReplicaSet to zero, so it can delete the old pods as the new ones are brought into service
- ▶ The ReplicaSet isn't deleted, so if something goes wrong, you can switch back to the old ReplicaSet and resurrect the old pods with a single command and the state is auto-rolled back
- ▶ By default, k8s will save the last 10 deployments if multiple rollbacks are required

## ▶ Service

- ▶ A service sits above a ReplicaSet or deployment and is the way in to access these components and pods. They route traffic to one or more pods. A service (not a good descriptive name) has a fixed IP address and is not deleted when its associated pods are, so when a new pod is created, it will be immediately contactable as the load balancer manages the pod IPs
- ▶ If we use a single pod for an app and it dies, users will have no access to the app until the pod is recreated
- ▶ If we create multiple replicas of a pod, the service can load balance across them - ensuring no downtime if a single pod dies
- ▶ If pods are spread across multiple nodes and nodes are spread across multiple AZs, the service can reach them all as it uses a common network

## ▶ There are four service types

- ▶ **NodePort** - this exposes the service on a static port on each node's IP. i.e., a service is accessible from outside the cluster. This is useful in development - like when we exposes container ports to the internet during testing
- ▶ **ClusterIP** - this is default. Service is only accessible inside the cluster on its internal network. It can load balance across multiple pods identified by a Selector Label
- ▶ **LoadBalancer** - a ClusterIP service will load balance internally, but this service is to provide integration to a cloud service load balancer. i.e. the cloud load balancer can distribute traffic to this service and is typically used in production
- ▶ **ExternalName** - unlike the others, this is not a load balancing service. It enables the provision of a dns name that services outside the cluster that wish to access a service within the cluster, can do so by using its dns name rather than needing to know an IP address. It can also be used to provide access to external services via a dns name from within the cluster

## ▶ ConfigMap and Secret

- ▶ These objects provide environment variables like .env

# Config maps

- ▶ Many applications need config values that are not secret. These can be provided statically using env vars in each pod definition, but this tightly couples the component because if, for example, the port of one container app is changed then any callers of it would need their .yaml files editing
- ▶ A config map is an API object where values are key value pairs
- ▶ Values can be entered imperatively at the command line and they will be added to the ConfigMap object that can be referenced in the pod definition. e.g. **kubectl create configmap configmap-name --from-literal=key=value**
- ▶ Alternatively, the config map object can be created declaratively in a .yaml file then applied as usual. This will also inject config values into the pods. Used instead of .env

```
kind: ConfigMap
apiVersion: v1
metadata:
 name: config-vals
data:
 UPLOAD_PORT: "3000"
 ETL_PORT: "3001"
 CALLER_PORT: "3002"

 STAGING_AREA: ../data

 MYSQL_HOST: mysql
 ETL_CONT_SERVICE: etl
 DB_TYPE: mysql
```

# Secrets

- ▶ Basically .env for credentials and keys
- ▶ Values can be stored in base64 (as keys tend to be in) or as strings

```
apiVersion: v1
kind: Secret
metadata:
 name: mysql
type: kubernetes.io/basic-auth
data:
 username: YWRtaW4=
 password: YWRtaW4=
 database: dHJpdg==
 root-pass: cm9vdA==
```

```
Alternative not in base64
stringData:
username: admin
password: admin
database: triv
root-pass: root
```

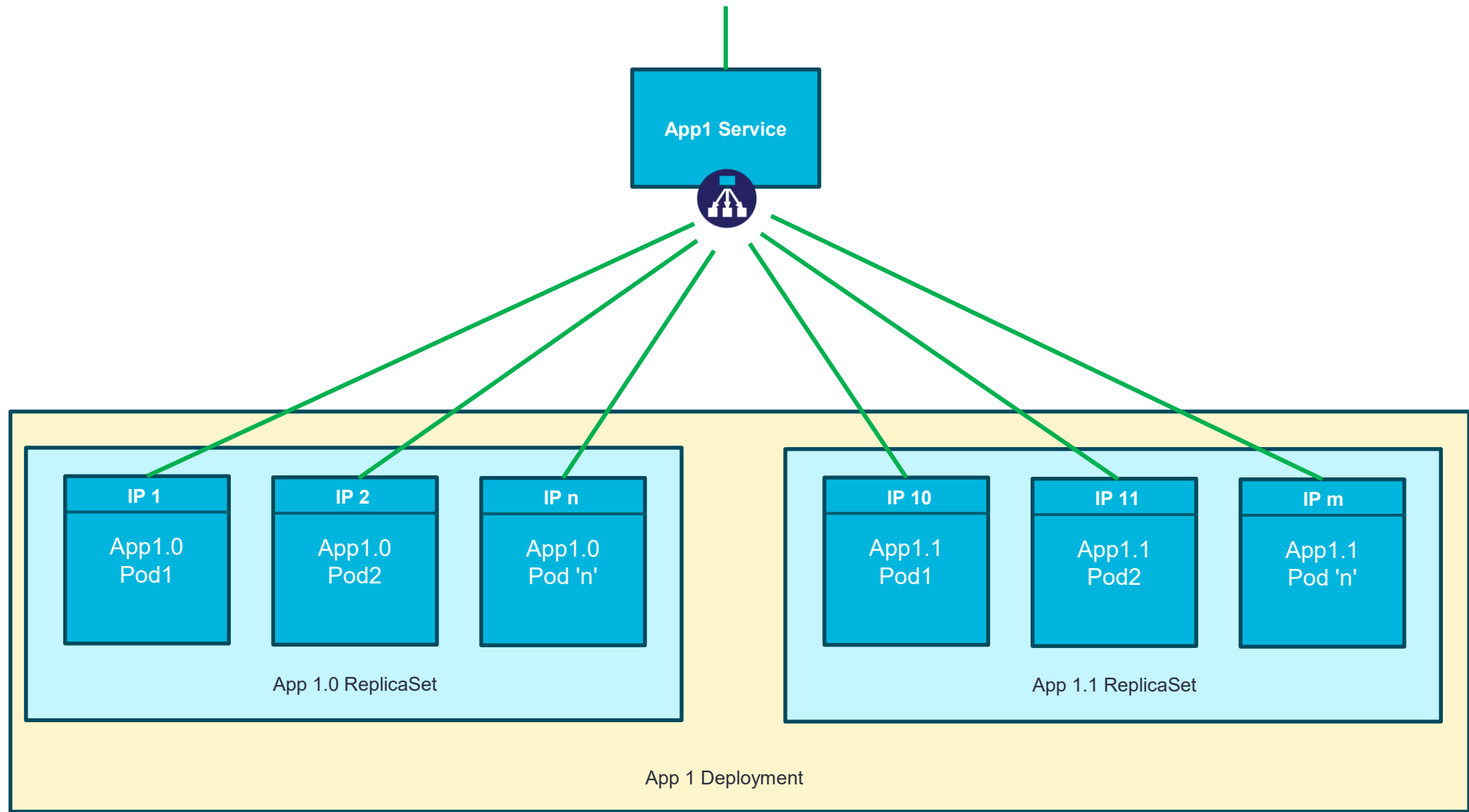
# Volumes & HPA

## Volumes

- ▶ Volumes need to be defined
- ▶ We could use local storage or specify external storage like Azure BLOB or AW S3
- ▶ Specify type, size, etc similar to Docker
- ▶ We specify the size of disk on the node with a persistent volume type
- ▶ We create a PersistentVolumeClaim type to specify our storage needs
- ▶ This is static binding. Dynamic binding is also available for use with say BLOB storage - the claim is made and that amount of BLOB is allocated

## HorizontalPodAutoscaler

- ▶ This object type enables you to specify min and max replica values and it will autoscale between those values depending on various metrics such as traffic load, cpu use, memory
- ▶ If another VM is needed to scale up to the max pods, it will be auto-created



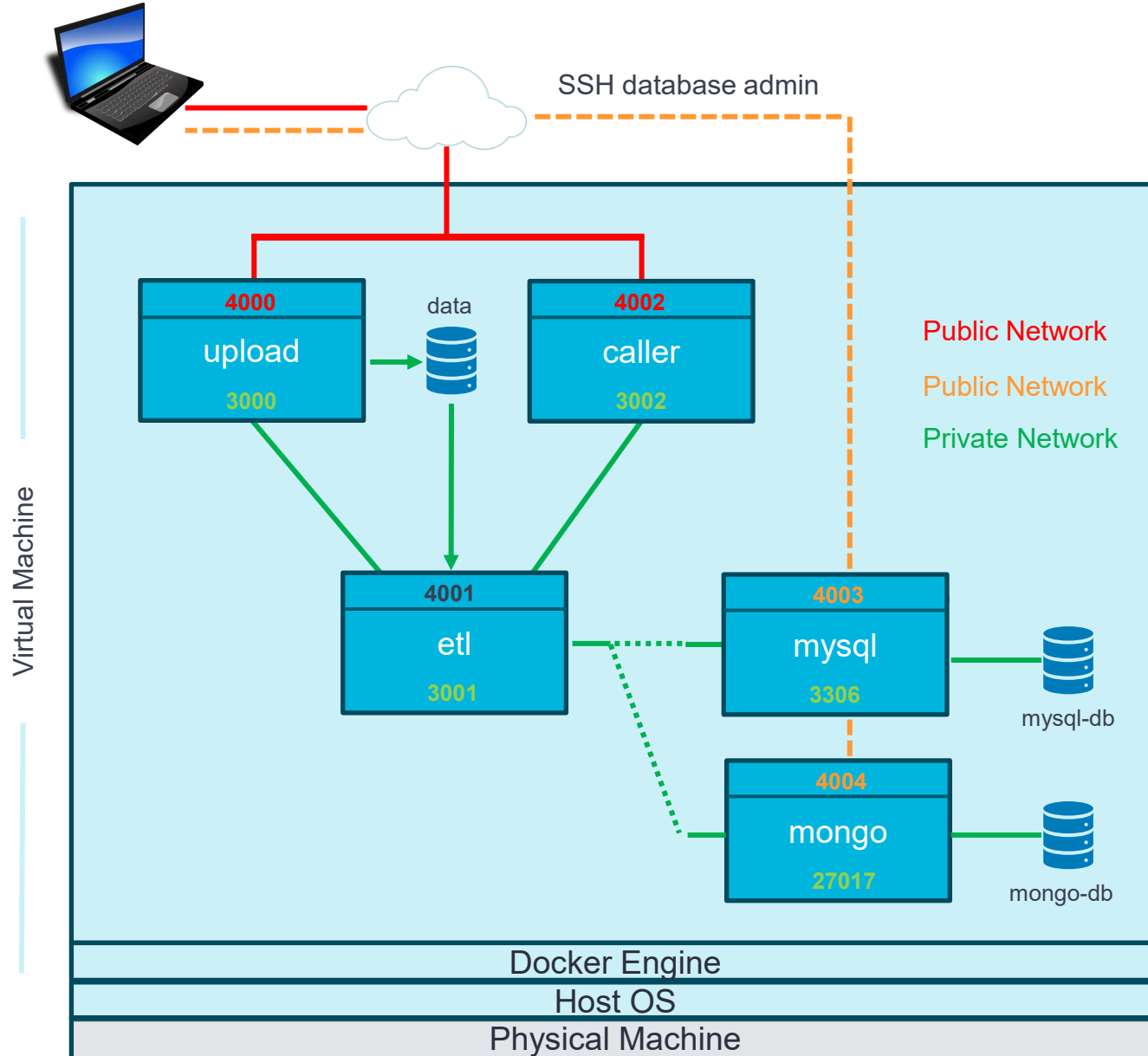
Rolling update

# Deploying Kubernetes

- ▶ There are a number of options
  - ▶ minikube - a lightweight version for local use. Requires a local VM such as docker, virtual box or similar
  - ▶ docker desktop - this now has a version built-in
  - ▶ kops - this is a k8s deployment onto cloud VMs so is cloud agnostic, but you need to support it
  - ▶ cloud services - all the big cloud hosts provide IaaS and PaaS options to deploy k8s
- ▶ If you want to try k8s
  - ▶ If you are using a modern version of Windows - win 10 pro or win 11, you can try docker desktop (I'll use that)
  - ▶ If you are using something else, or desktop doesn't work for you, you can try to use docker's vm
  - ▶ If that doesn't work, you can download a hypervisor such as virtual box, load minikube onto it and try that



# Data Engineering Solution Architecture



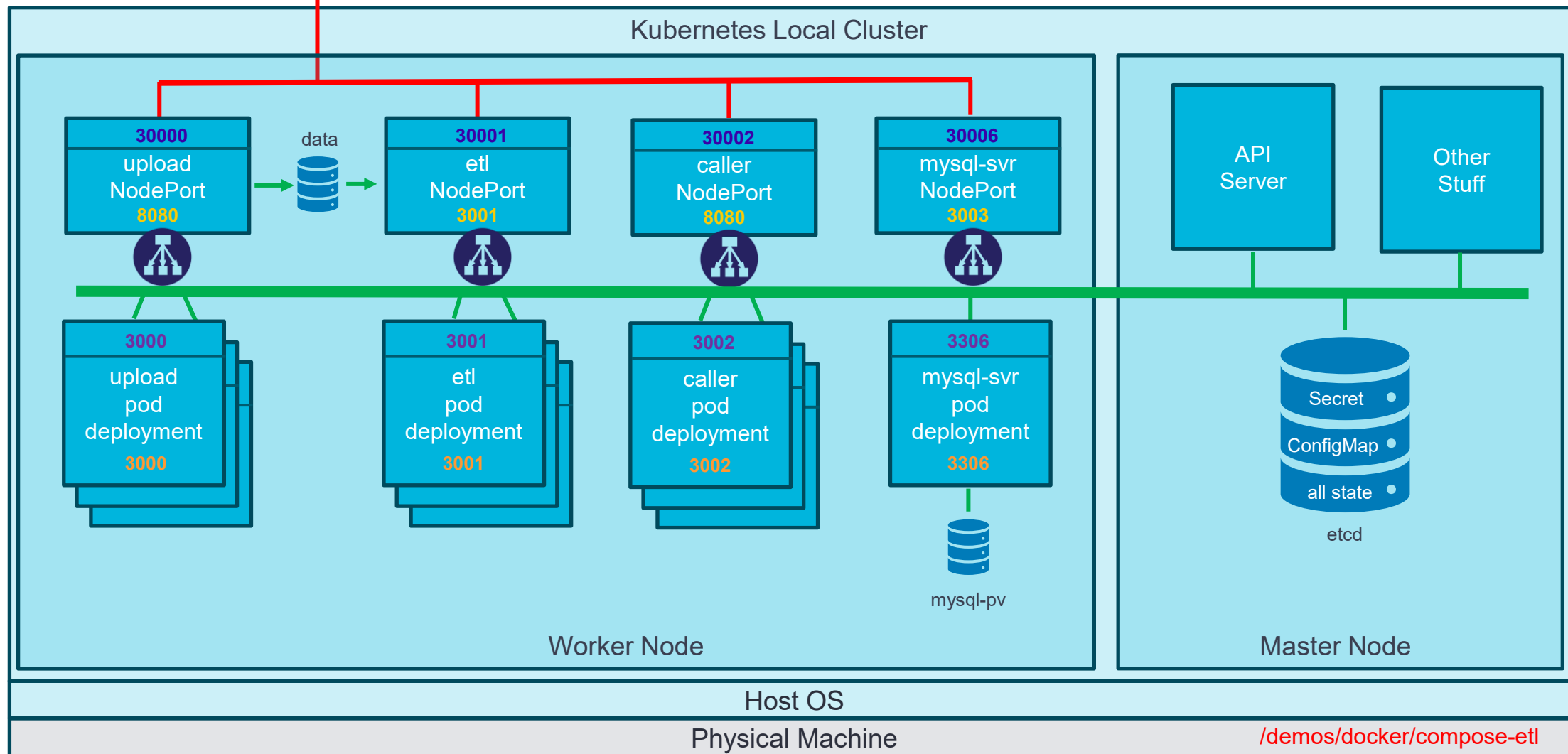
- ▶ This architecture is redrawn in terms of Kubernetes pods and services next
- ▶ It will be deployed locally in a Kubernetes cluster
- ▶ It will then be deployed to the Azure Kubernetes service and made available on the internet

browser (localhost:nodePort) -> service(port) -> pod(targetPort) -> node app(env var)



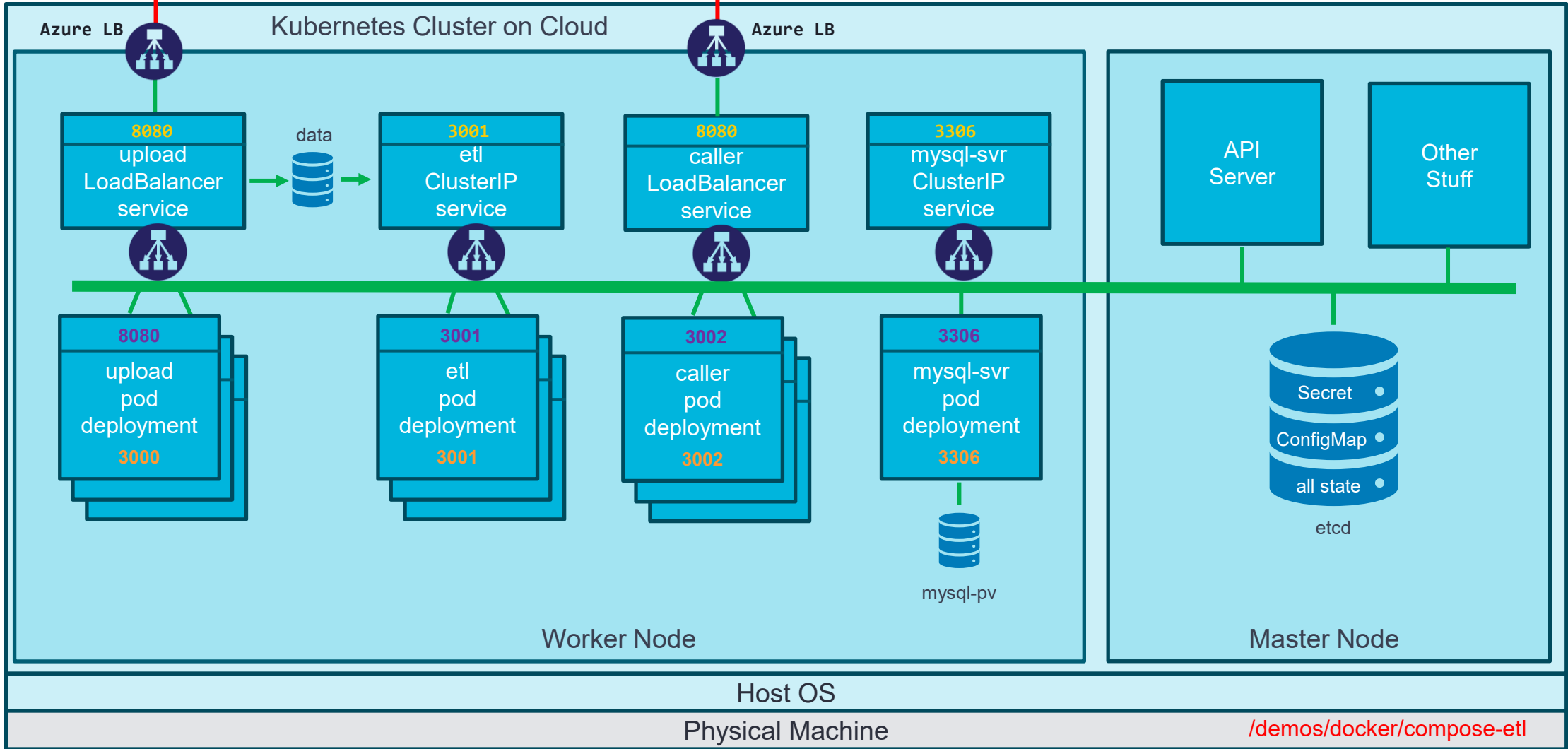
Public Network

Private Network



browser (localhost: **port**) -> pod(**targetPort**) -> node app(**env var**)

Public Network  
Private Network



# Some useful Kubernetes commands

- ▶ `kubectl get all` # this shows all pods and other components such as services
- ▶ `kubectl get pods` # just gets pods
- ▶ `kubectl get po` # short version of above - hardly though
- ▶ `kubectl get po --show-labels` # lists pods and labels
- ▶ `kubectl apply -f first-pod.yaml` #create component based on the script in the file and add to the cluster
  
- ▶ `minikube start` #what it says
- ▶ `minikube stop` #what it says. Mine defaults to using docker as the vm
- ▶ `minikube ip` #get the cluster ip address
  
- ▶ `kubectl describe pod/webapp` # (use pod name) creates a load of ;info about the component
- ▶ `kubectl exec webapp ls` is now `kubectl exec webapp -- ls` #execute command in pod / container. In this case list the files. e.g. `kubectl exec webapp -- ls -la`
  
- ▶ `kubectl -it exec webapp -- sh` #get shell like in docker
- ▶ `kubectl apply -f .` #apply all files in the current dir without naming them
- ▶ `kubectl delete pod podname` **Note: if you change an env var** in the config.yaml you need to restart the pod. Just delete it and it will restart
- ▶ `kubectl delete component component_name` # e.g. `kubectl delete pod webapp`
- ▶ `kubectl delete component component_name --force` #force it to stop
  
- ▶ # to delete if part of a deployment, i.e. stop it keep coming back under a different name, delete the deployment. Deletes pod but doesn't delete service
- ▶ `kubectl get deployment` # see what they are called
- ▶ `kubectl delete deployment` # e.g. `kubectl delete deployment mysql`
  
- ▶ `kubectl delete pods --all` # to delete all pods. Can do the same for other components
- ▶ `kubectl delete all --all --all-namespaces` # deletes everything or in docker desktop reset kubernetes cluster

- ▶ `kubectl describe replicaset name`
- ▶ or
- ▶ `kubectl describe rs webapp`
- ▶ `kubectl delete rs webapp` # to delete a replica set as deleting nodes will cause them to come back. This also deletes the pods in the set
- ▶ `kubectl rollout status deployment webapp` # watch the deployment happen
- ▶ `kubectl rollout history deployment webapp` # list the history of deployments
- ▶ `kubectl rollout undo deployment webapp` # rollback to previous version
- ▶ `kubectl rollout undo deployment webapp --to-revision=x` # if there are several you can go back to whichever number you want to with this switch. By default k8s will remember the last 10
  
- ▶ `kubectl get namespace` # list all namespaces. Use ns for short
- ▶ `kubectl get pods -n namespace` # e.g. `kubectl get pods -n kube-system`
- ▶ `kubectl get all -n namespace` # get all components
  
- ▶ `nslookup` # to resolve dns name to ip address or visa versa
  
- ▶ `kubectl delete -f filename.yaml` # delete all resources created by that file.
- ▶ `kubectl delete -f .` # delete from all files in the current folder
- ▶ or
- ▶ if using docker desktop, press the reset button - we are left with just the kubernetes API service which we need
- ▶ `kubectl get all -o wide` # gets a bit more info
- ▶ `kubectl logs podName` # what it says
- ▶ `kubectl logs -f podName` # -f is follow i.e. it keeps updating but it takes over the shell
- ▶ `kubectl get pv` # as get all doesn't list them pv will list the persistent volumes
- ▶ `kubectl get nodes` #to list how many and their names
  
- ▶ `kubectl create secret generic ...` # look up details how to fully create. e.g. `kubectl create secret generic mysql-root-pass --from-literal=password=root`
- ▶ `kubectl delete secret name` #delete the name. e.g. `kubectl delete secret mysql-user-pass`
- ▶ `kubectl get secrets` # to list them
- ▶ `kubectl describe secrets` # for more info
  
- ▶ # To switch contexts say from Azure back to client
- ▶ `kubectl config get-contexts` # List the contexts so you can choose what to switch to
- ▶ `kubectl config use-context <new context>` # e.g. `kubectl config use-context docker-desktop` or `kubectl config use-context junk` to go back