

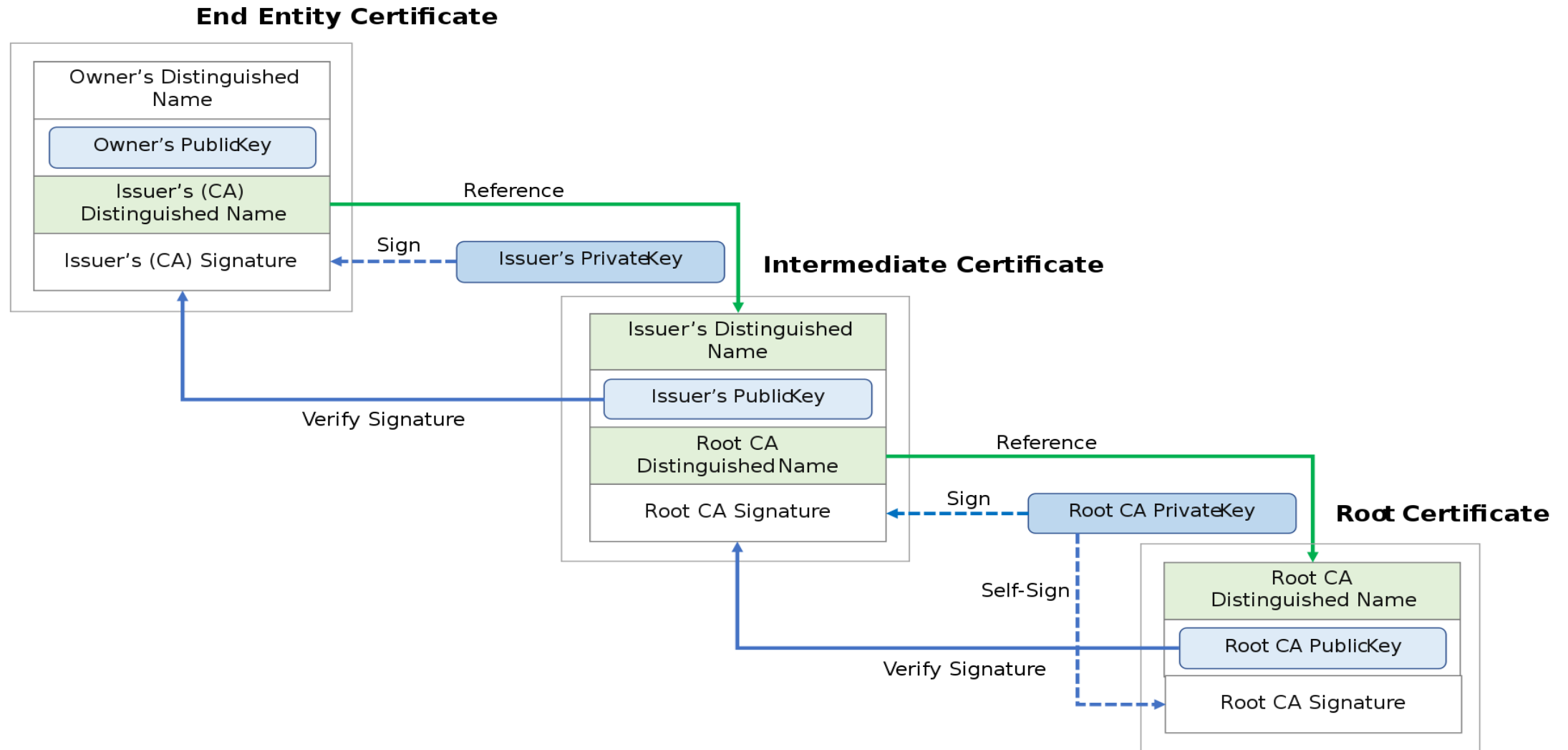


Transport Layer Security

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:

server_write_MAC_key:

client_write_key:

server_write_key:

Authentication and Integrity check:

Authentication and Integrity check:

Message encryption using symmetric key:

Message encryption using symmetric key:

12AF349DD8ABBE73486D

AB937FA5B3C81A0F6C02

4BBAC5C662FBA8E9D04C

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);
```



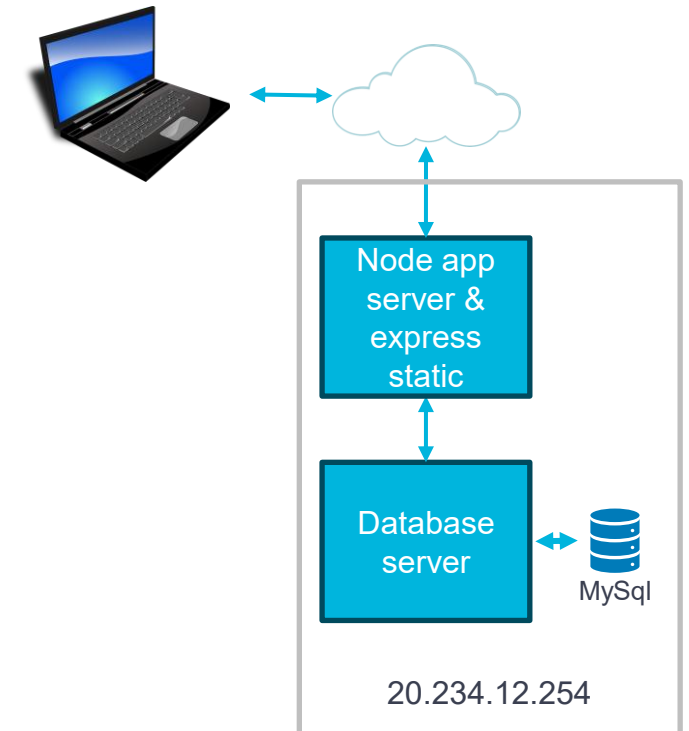
Cross Origin Resource Sharing

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**

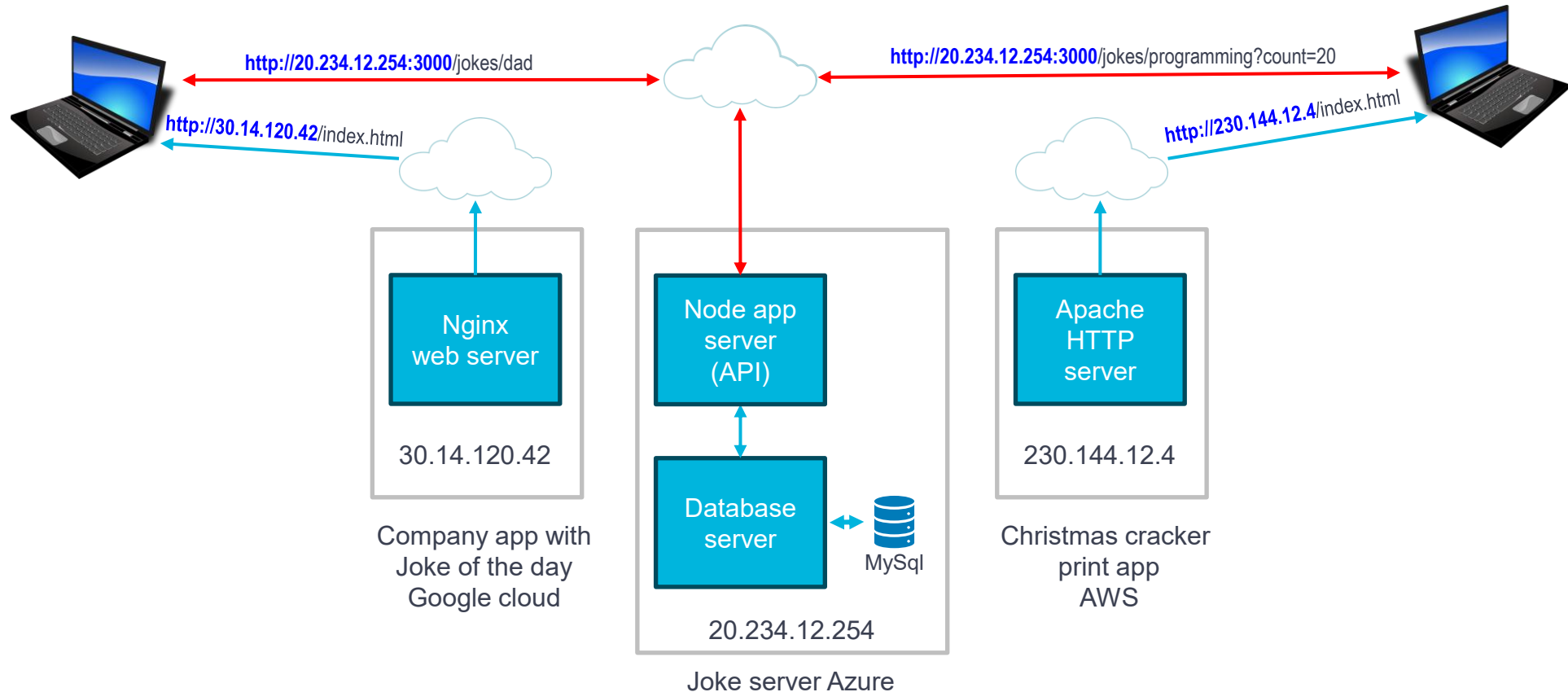
Same origin

- ▶ 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



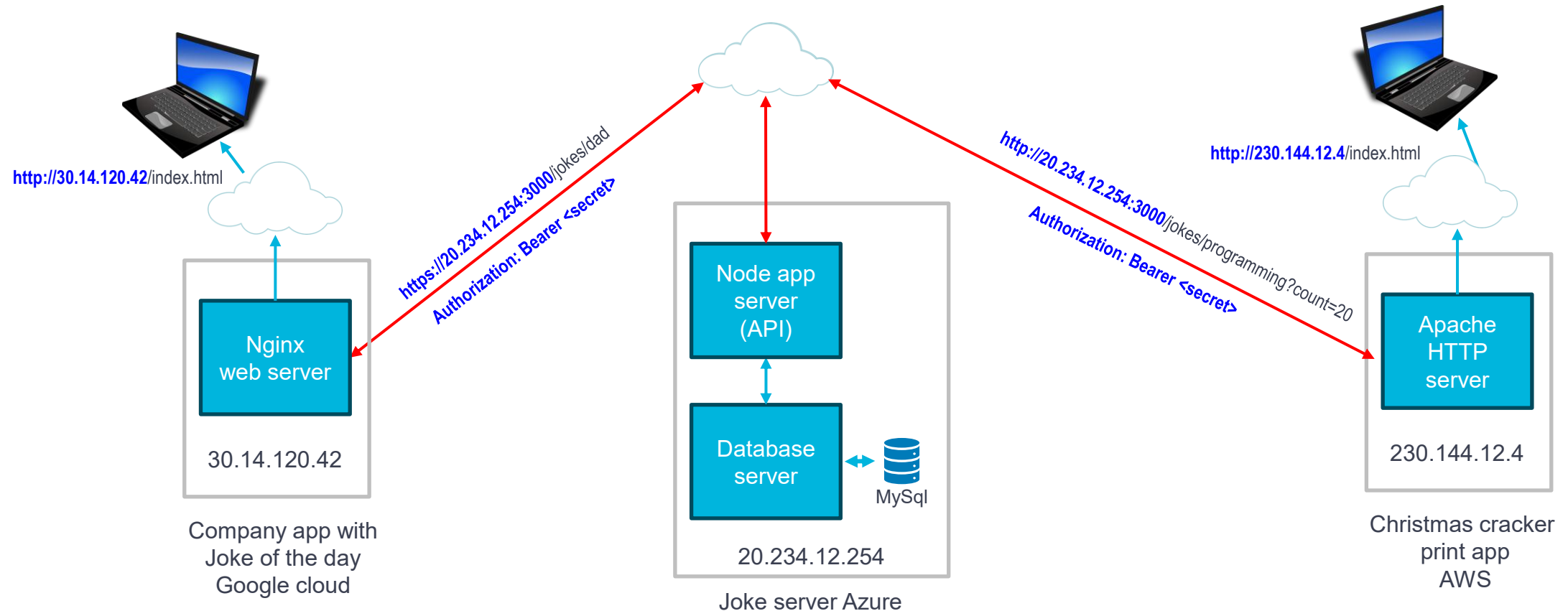
Cross origin

- ▶ 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

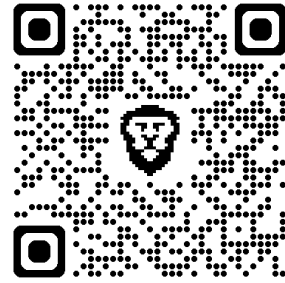


Cross origin

- ▶ 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



Reminder



University of
Lancashire

Complete the National Student Survey (NSS)

- ✓ You could win one of 50 graduation photography packages
- ✓ It only takes 10 minutes to complete
- ✓ Give your views on what was good and what might be improved
- ✓ Help us deliver the best experience for students



Visit thestudentsurvey.com





OpenAPI

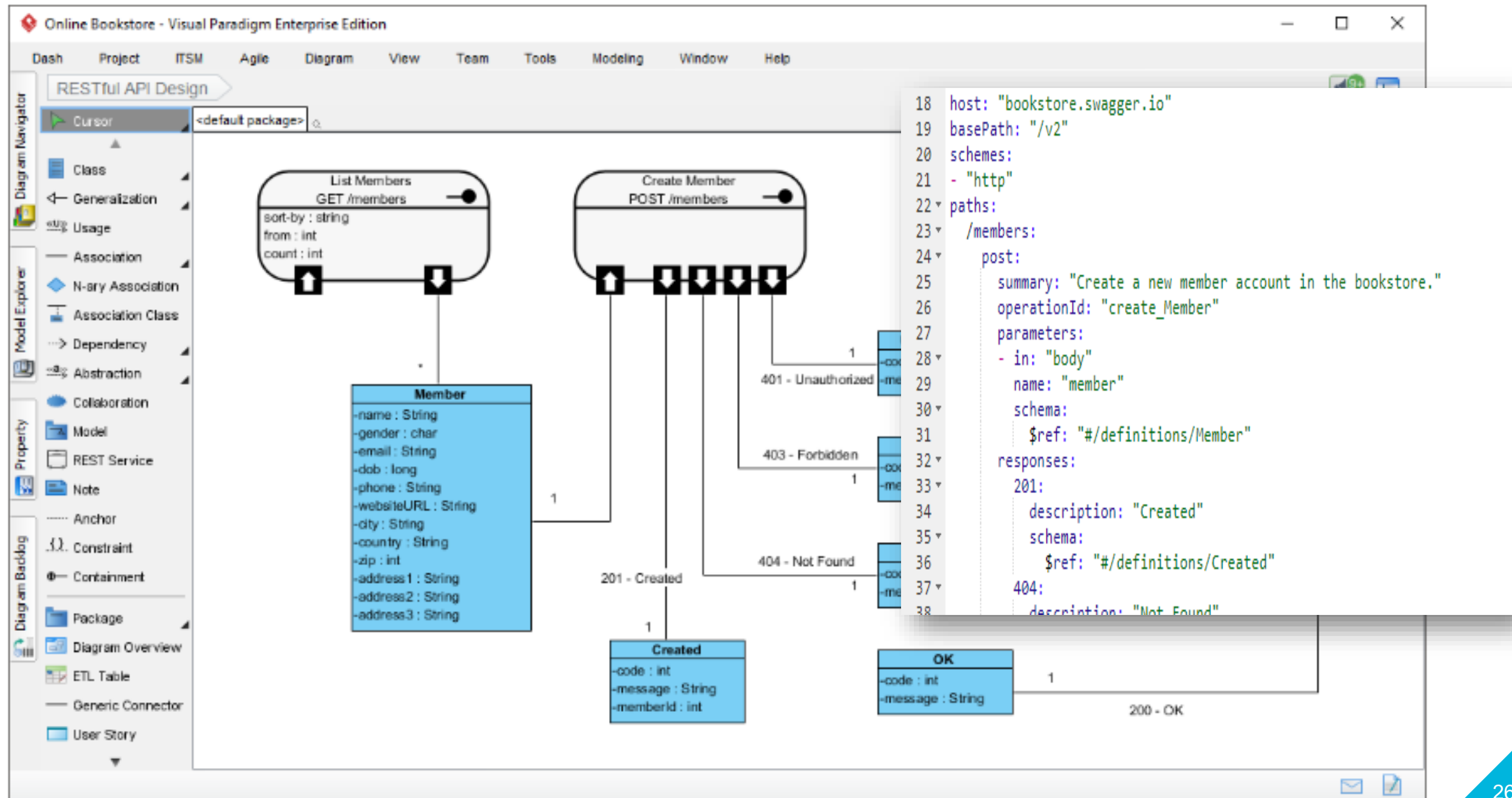


Learning objectives

- ▶ 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