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

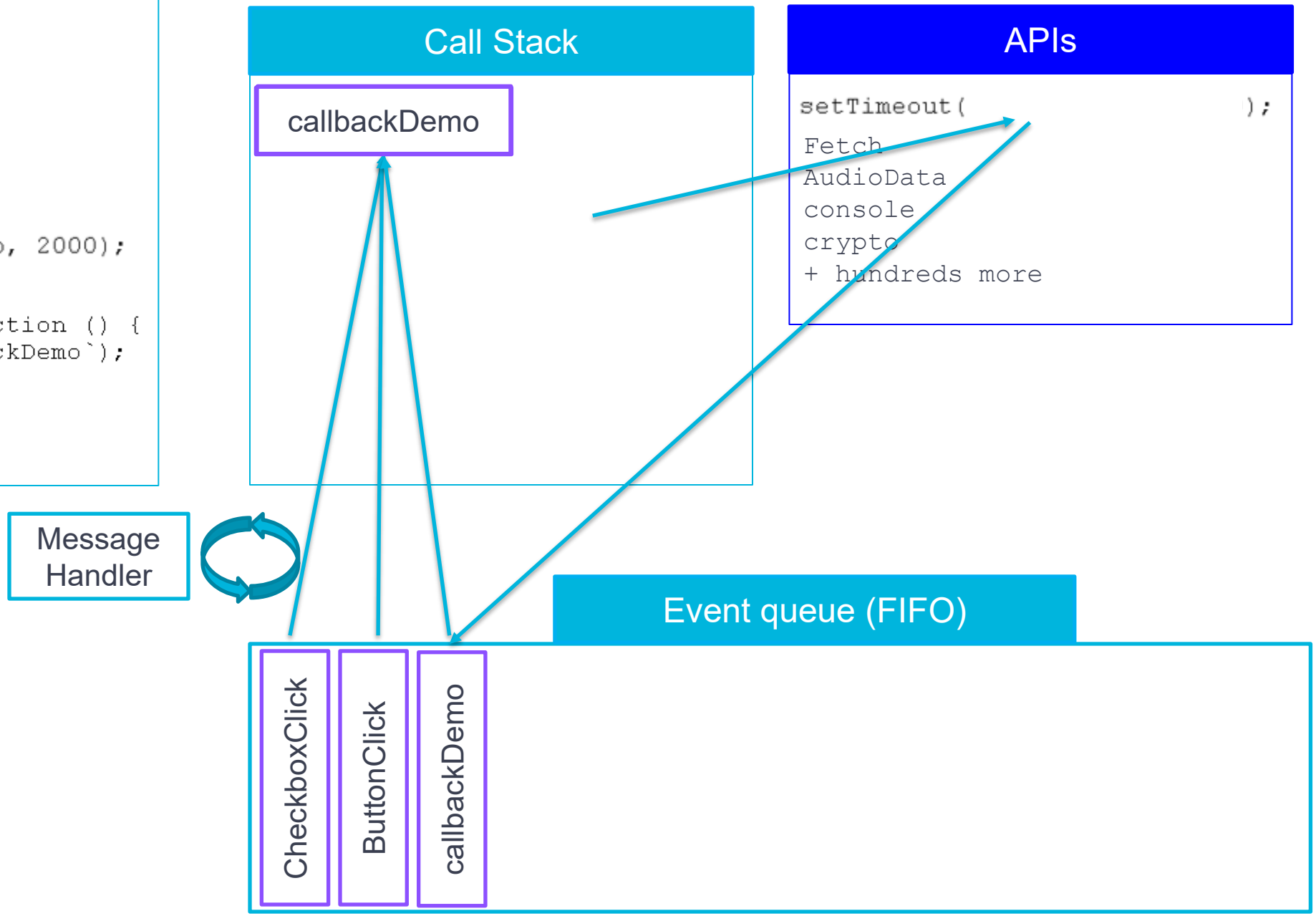
```
function level1() {
  level2();
}

function level2() {
  level3();
}

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

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


level1();
```

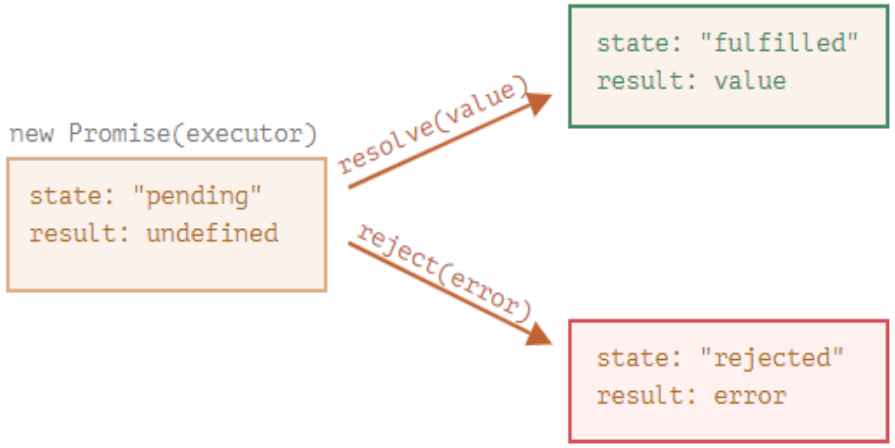


Asynchronous approaches

- 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 no 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)
```

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

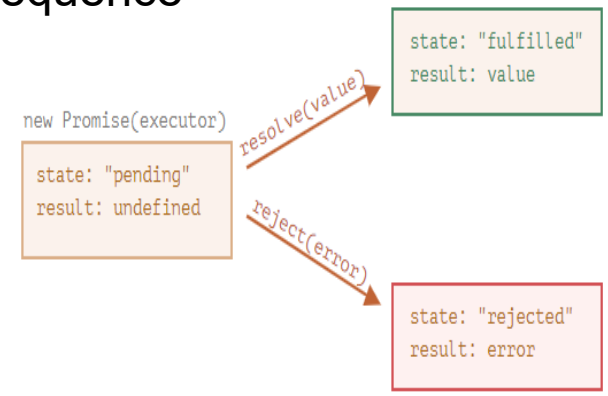
```
function cbTimeout(resolve, reject) {
  let num = 6
  if (num < 5) resolve(`${num} is good`)
  else
    reject(`${num} is too big`)
}
```

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

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

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

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



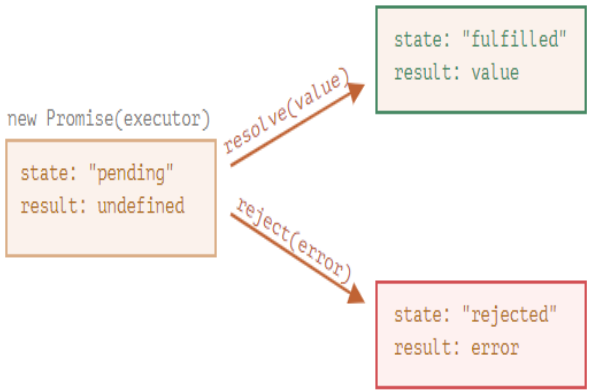
demos/async/promise1.js

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")
```




```
function getSmallNum(num) {
  const prom = new Promise((res, rej) => {
    setTimeout(() => {
      if (num < 5) res(`${num} is good`)
      else rej(`${num} is too big`)
    }, 2000)
  })
  return prom
}

async function checkSmallNum(num) {
  try {
    const result = await getSmallNum(num)
    console.log(`${result}`)
  } catch (err) {
    console.log(`${err}`)
  } finally {
    console.log(`Promise Finally`)
  }
}
```

Immediate execution of the callback
Reg timer callback - not executed
Call function returning promise
Return Promise
Needs to be async to use await
Pending
Resolved
Rejected
Do this regardless
Continue with main whilst waiting
When promise resolved, continue with suspended code

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

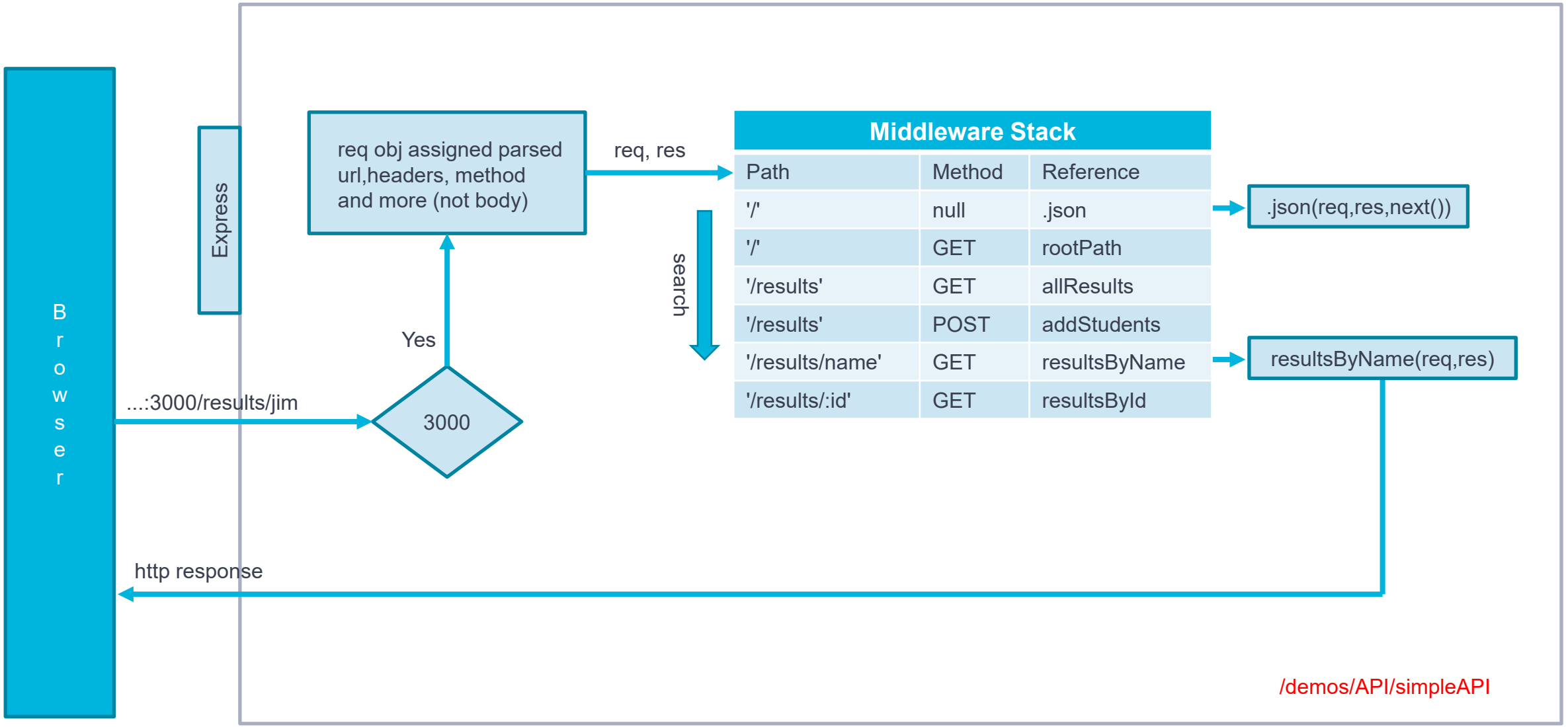
demos/async/asyncAwait1b.js

- 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

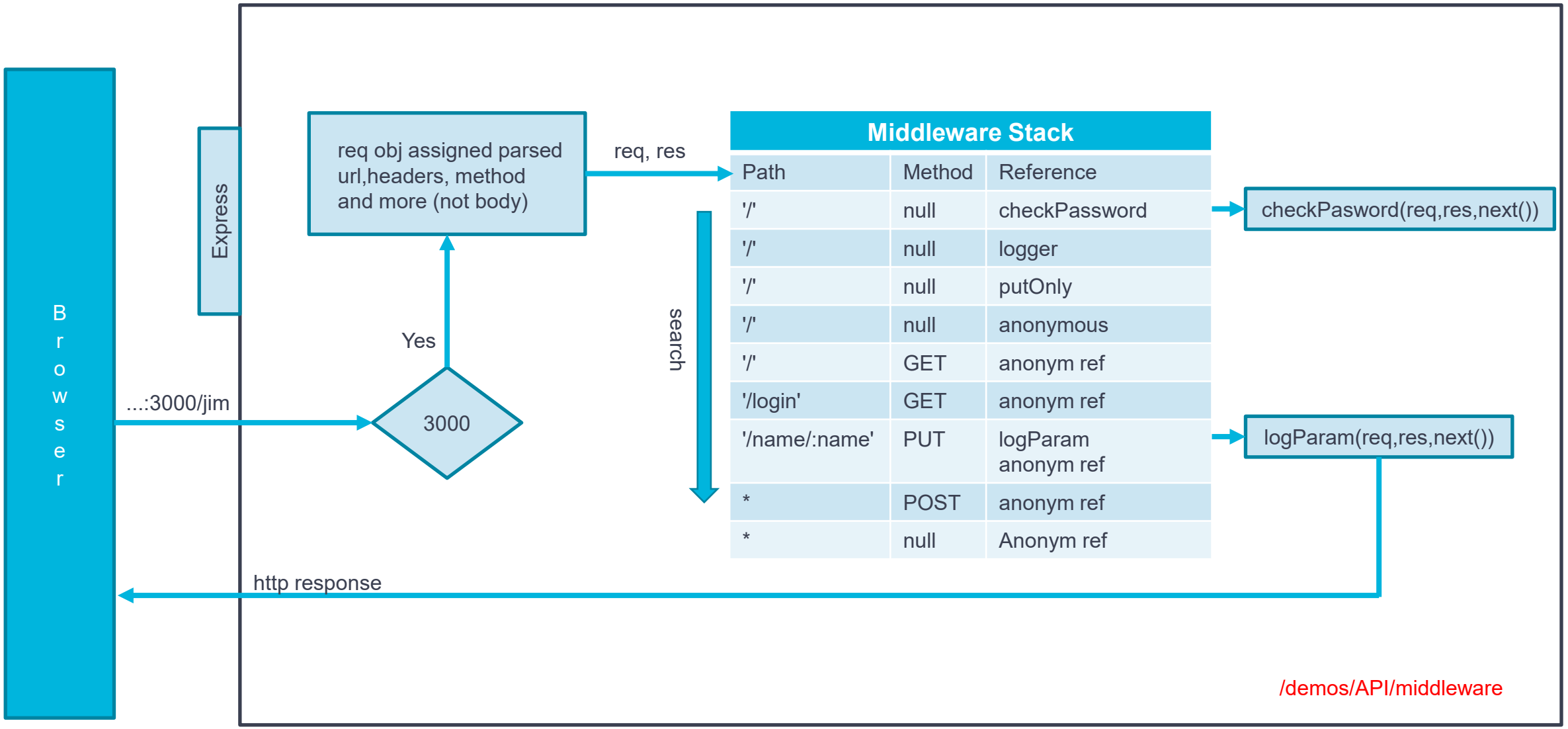
  - **Microtask queue**:
  - Includes: Promise callbacks (.then, catch, finally) and a few others.
  - These are processed **immediately after the call stack clears**, **before** any macrotasks

Express and Middleware


Browser
Express
req obj assigned parsed
url,headers, method
and more (not body)
req, res
Middleware Stack
Path
Method
Reference
'/'
null
.json
'/'
GET
rootPath
'/results'
GET
allResults
'/results'
POST
addStudents
'/results/name'
GET
resultsByName
'/results/:id'
GET
resultsById
.json(req,res,next())
resultsByName(req,res)
search
Yes
3000
...:3000/results/jim
http response
/demos/API/simpleAPI

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


Browser
Express
req obj assigned parsed
url,headers, method
and more (not body)
req, res
Middleware Stack
Path
Method
Reference
'/'
null
checkPassword
checkPasword(req,res,next())
'/'
null
logger
'/'
null
putOnly
'/'
null
anonymous
'/'
GET
anonym ref
'/login'
GET
anonym ref
'/name/:name'
PUT
logParam
anonym ref
logParam(req,res,next())
*
POST
anonym ref
*
null
Anonym ref
search
Yes
3000
...:3000/jim
http response
/demos/API/middleware

- 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 it 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

Adding a Database

Public internet
Azure private network
Docker private network
Virtual machine
Application container
Rabbitmq combined container

80 / 443
kong
8445
public IP:
private IP:

/categories
/question/:category
/moderate
/categories
/categories
/submit
/docs

4000
question
3000
/categories

4100
moderate
3100
/categories
submitted question

4200
submit
3200
categories

4002
MongoDB
27017

4002
MySQL
3306
mysql
vol

categories

4102
rmq-admin
15672

4201
SUBMITTED_QUESTIONS
5672

4202
rmq-admin
15672

public IP:
private IP:

mongo
vol

4001
etl
3001

moderated questions

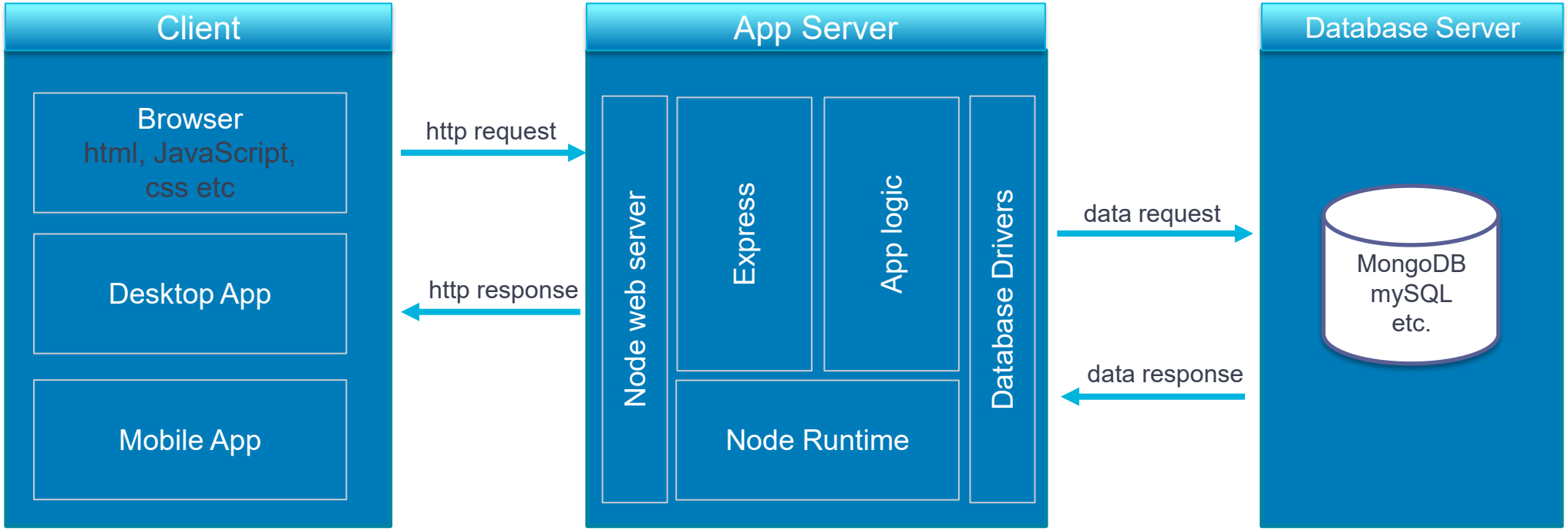
4101
MODERATED_QUESTIONS
5672

public IP:
private IP:

public IP:
private IP:

Container port number
Application port number

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