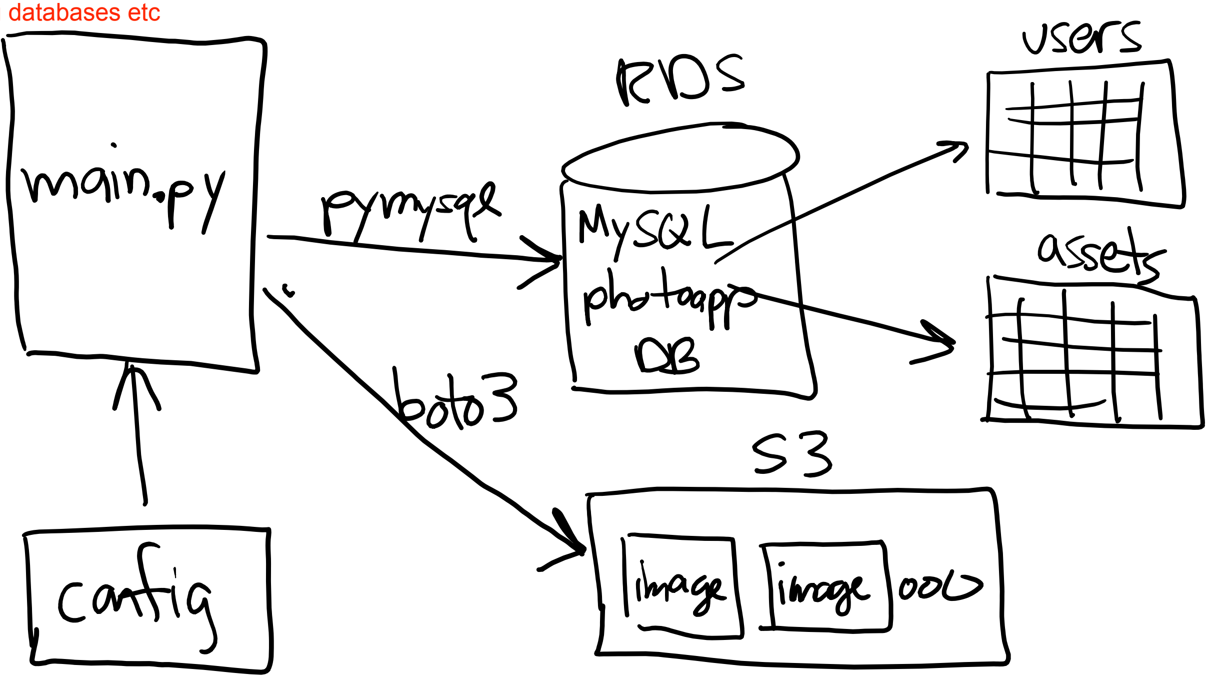
Server-side execution

- **Server-side execution of:**
  - *JavaScript*
  - *Node.js*
  - *Express*
- **Asynchronous execution**
- **Callbacks vs. Promises**

Project 01

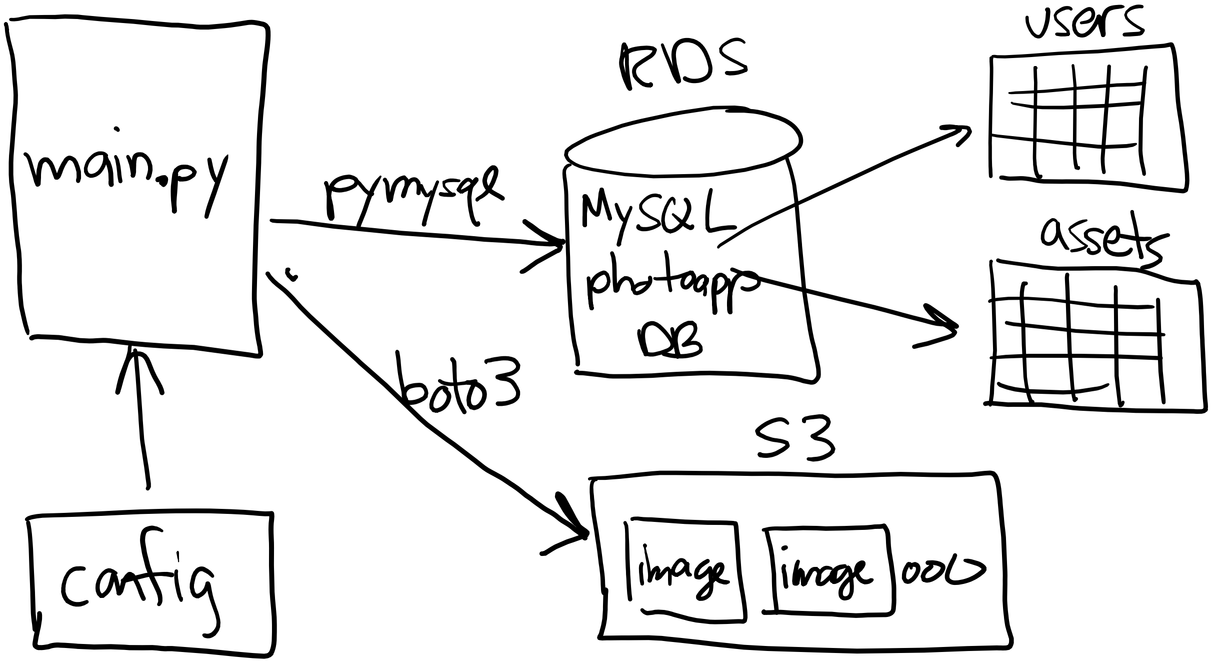
We call it thick client as
it was doing most of the work like handling
user interaction, handling databases etc



Goals?

- Experience with AWS
- Experience with Python and SQL
- Setup infrastructure for future projects

Project 01 => Project 02

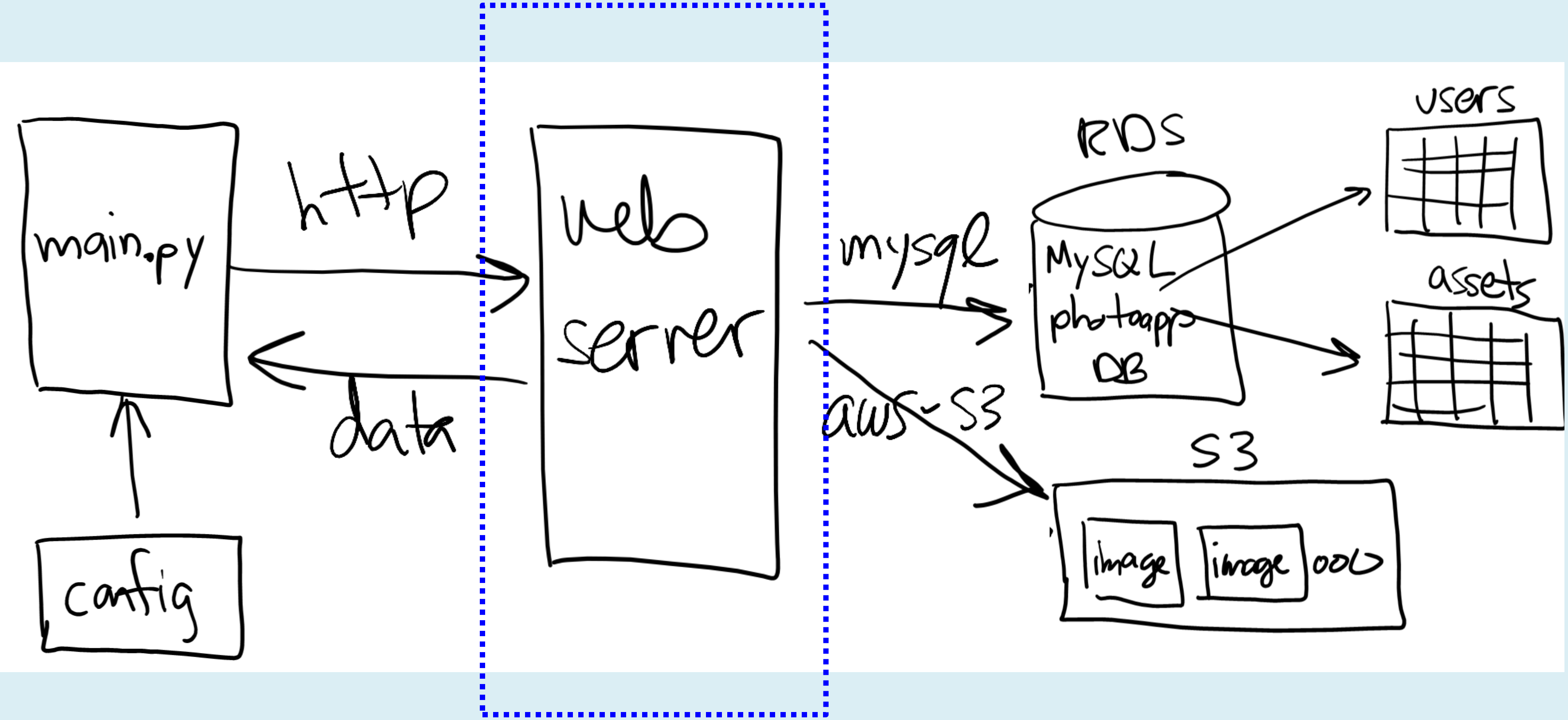


Why?

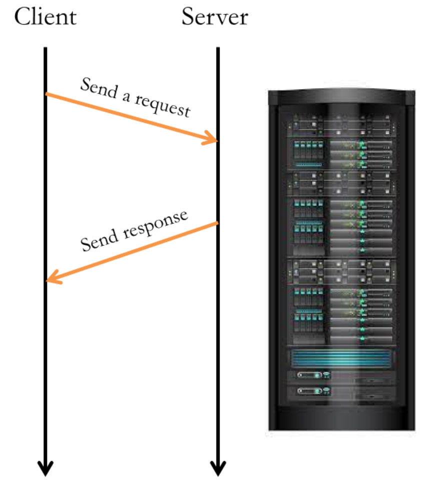
- **Project 01 works fine…**

But it requires configuring lots of ports to enable connection between client and server.

- **Why complicate things with another layer of software?**



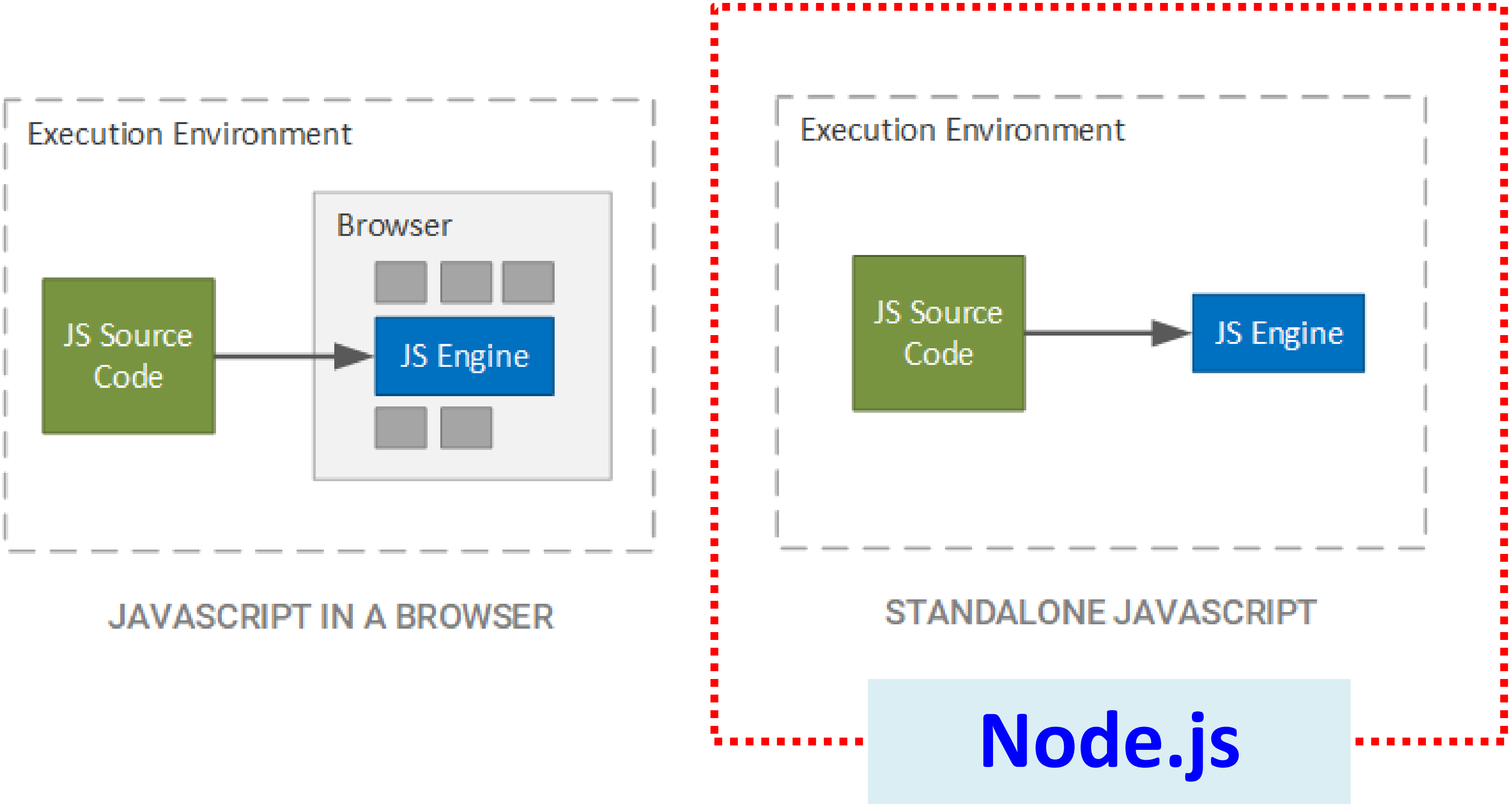
Server-side programming is a VERY DIFFERENT programming style…



They are different in the sense, they are based on request respond pattern, which is a package being sent over the network. Sometimes, network can go down or the server is busy, might not respond in time, we need to take care of many things like how fast server responds, if the network drops, we make sure to call the server again. etc

Node.js

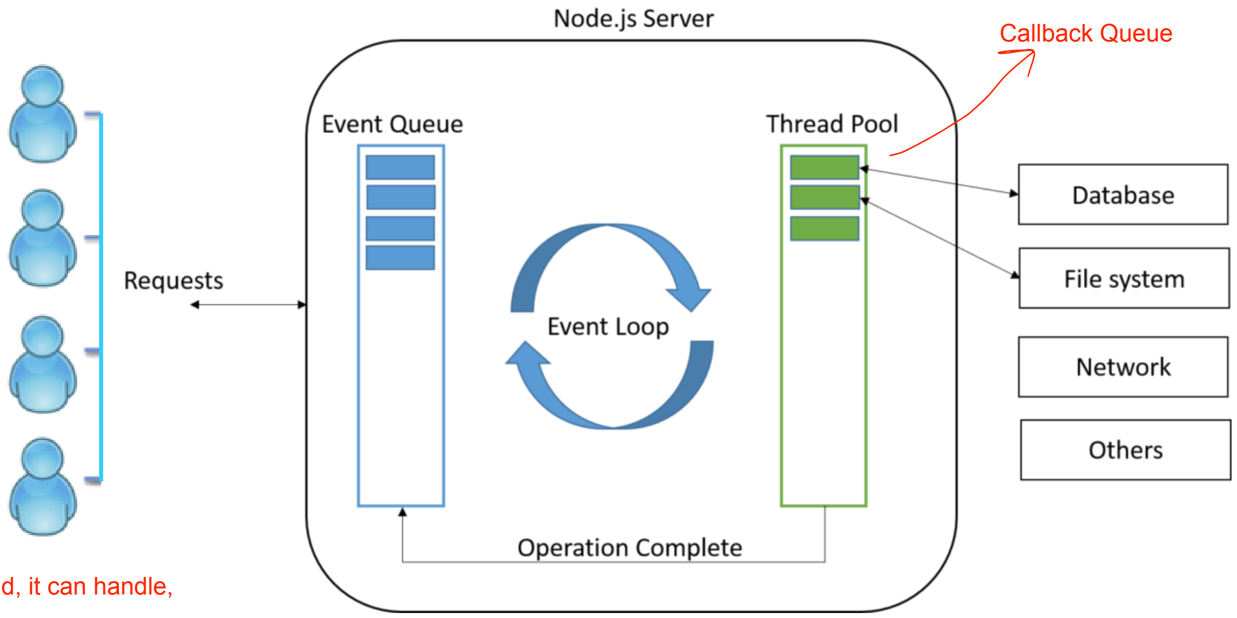
- JavaScript first appeared as client-side, scripting language
- **Node.js** is runtime engine for executing JS outside browser
- **Node.js** designed for server-side execution



Node.js

- **Single-threaded with large software library (MySQL, S3, …)**
  - *https://nodejs.org/api/documentation.html*

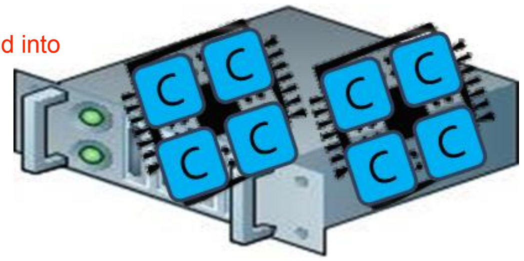
Node runs on single core and with one single thread. This is done to make the software simpler.
If we want to run more threads or scale it up, just scale up the software by replicating it on multiple cores.



Even though it runs on a single thread, it can handle,
multiple requests concurrently.

When a request comes in, it gets queued up in the event queue,
from where a single running thread picks up the tasks and start executing it.

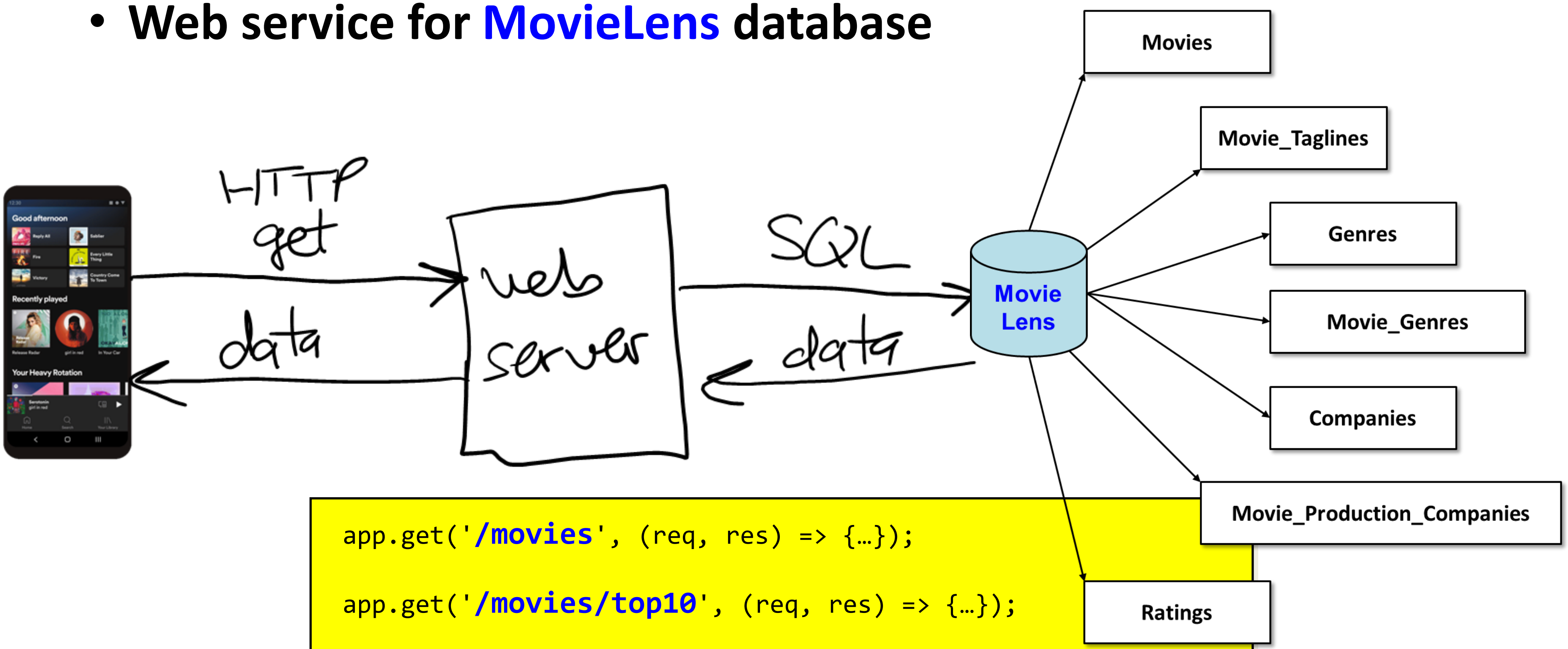
The work thats in progress gets piled up in the callback queue and gets pushed into
the event queue as soon as the task gets completed.



EXAMPLE

Example from last time

- Web service for MovieLens database

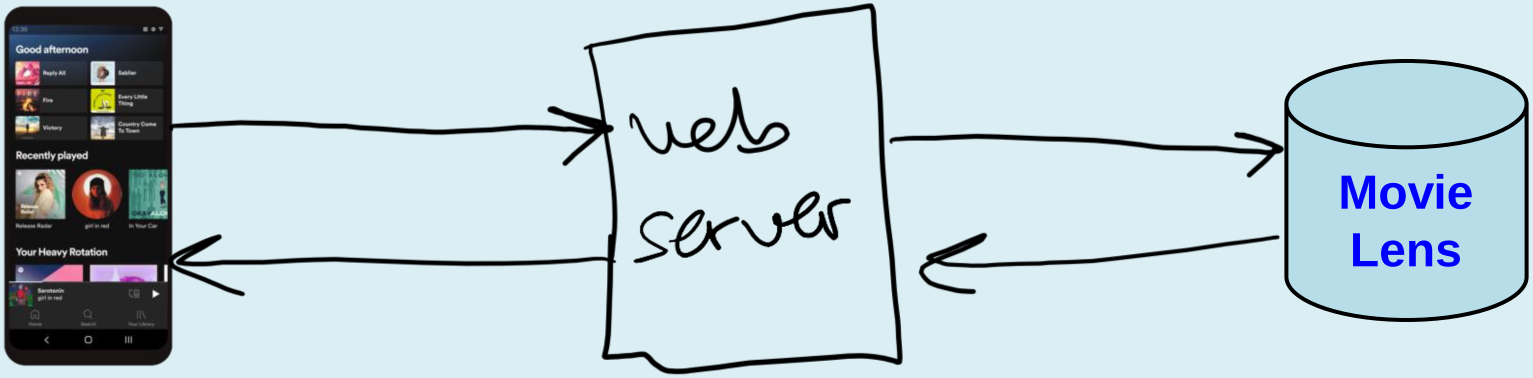


```
app.get('/movies', (req, res) => {…});

app.get('/movies/top10', (req, res) => {…});

// top N with at least M reviews:
app.get('/movies/topNwithM', (req, res) => {…});

// top N with at least M reviews, in given genre:
app.get('/movies/topNwithM/:genre', (req, res) => {…});
```

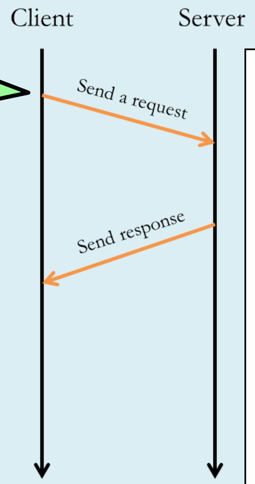
/movies

```
//
// Retrieve all movies in the database:
//
app.get('/movies', (req, res) => {
  try {
    console.log("call to /movies...");

    let sql = "Select * From Movies Order By Movie_ID;";
    let params = [];

    // execute the SQL:
    movielens.all(sql, params, (err, rows) => {
        if (err) {
          res.status(500).json( {"message": err.message, "data": []} );
          return;
        }

        // send response in JSON format:
        console.log("sending response");
        res.json( {"message": "success", "data": rows} );
    });

    console.log("about to return");
  }
  catch(err) { res.status(500).json({"message": err.message, "data": []}); }
});
```

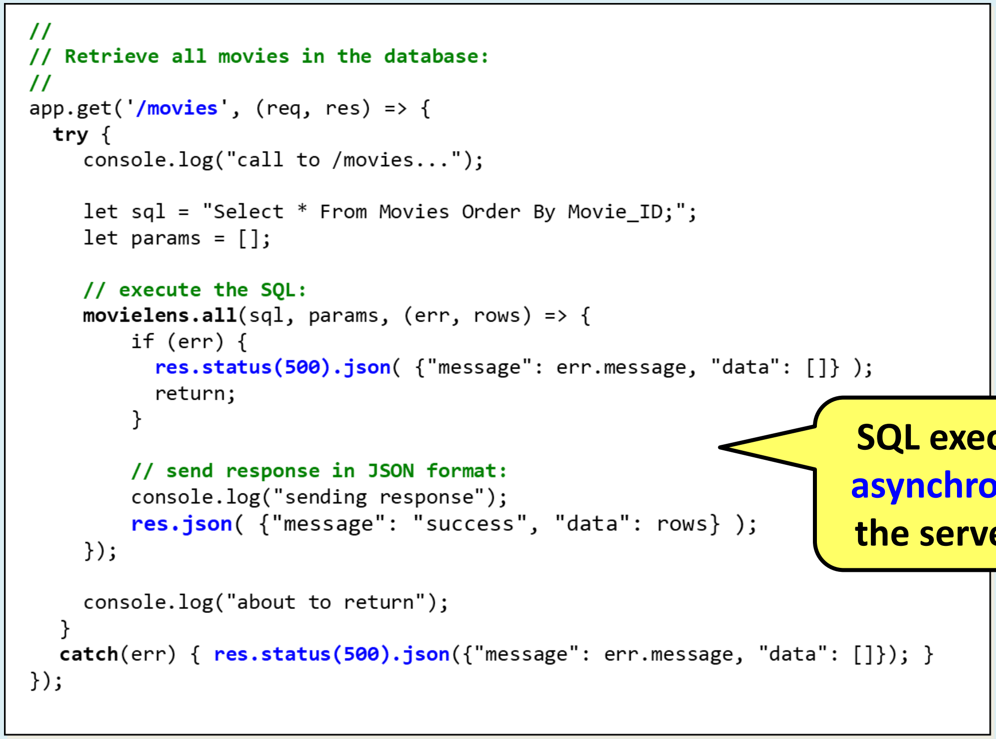
Execution of /movies --- sync vs. async



Client sees the API as synchronous, i.e. waiting for call to return…



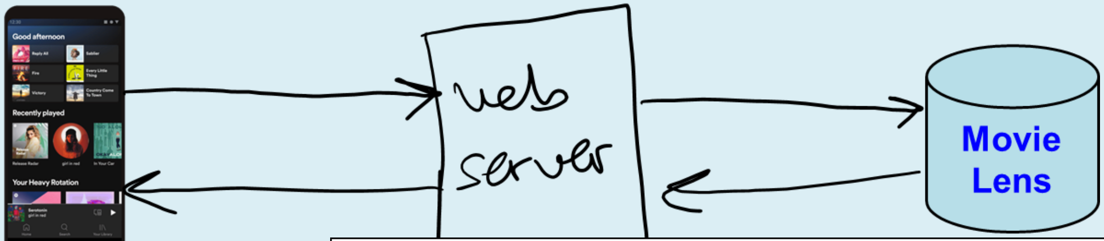
The javascript code is not synchronous,
The code execution is not always top to bottom.

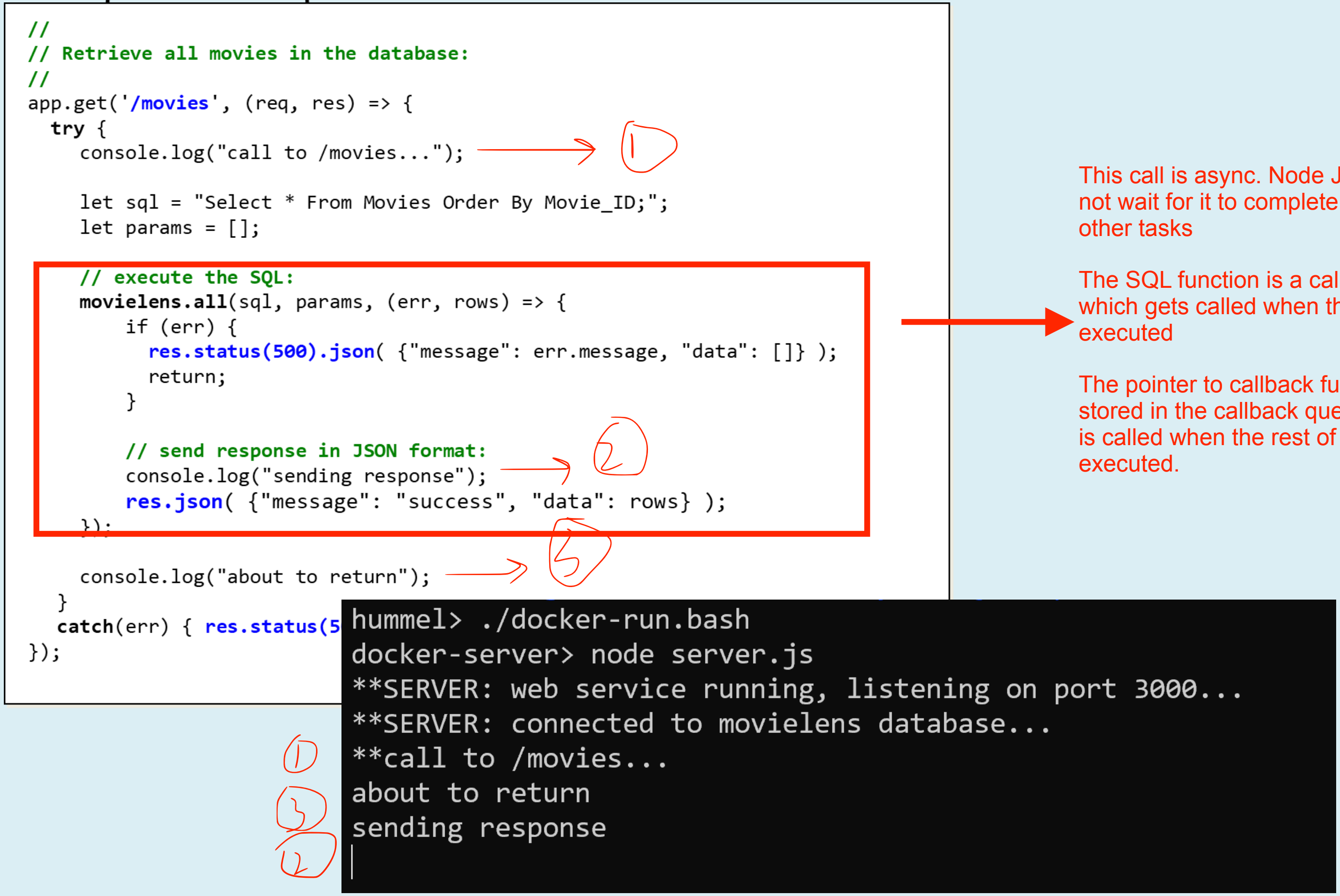


```
//
// Retrieve all movies in the database:
//
app.get('/movies', (req, res) => {
  try {
    console.log("call to /movies...");

    let sql = "Select * From Movies Order By Movie_ID;";
    let params = [];

    // execute the SQL:
    movielens.all(sql, params, (err, rows) => {
        if (err) {
          res.status(500).json( {"message": err.message, "data": []} );
          return;
        }

        // send response in JSON format:
        console.log("sending response");
        res.json( {"message": "success", "data": rows} );
    });

    console.log("about to return");
   }
   catch(err) { res.status(500).json({"message": err.message, "data": []}); }
});
```

SQL execution is asynchronous on the server side…

Look carefully at the output…





```
//
// Retrieve all movies in the database:
//
app.get('/movies', (req, res) => {
  try {
    console.log("call to /movies...");

    let sql = "Select * From Movies Order By Movie_ID;";
    let params = [];

    // execute the SQL:
    movielens.all(sql, params, (err, rows) => {
        if (err) {
          res.status(500).json( {"message": err.message, "data": []} );
          return;
        }

        // send response in JSON format:
        console.log("sending response");
        res.json( {"message": "success", "data": rows} );
    });

    console.log("about to return");
  }
  catch(err) { res.status(5
});
```

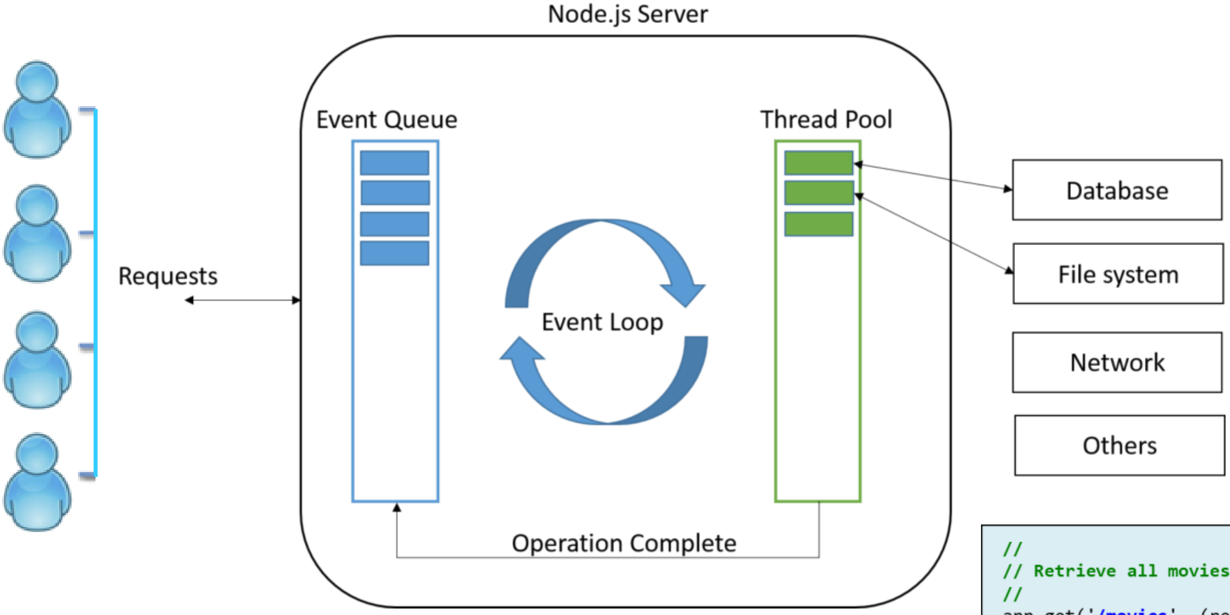
```
hummel> ./docker-run.bash
docker-server> node server.js
**SERVER: web service running, listening on port 3000...
**SERVER: connected to movielens database...
**call to /movies...
about to return
sending response
```

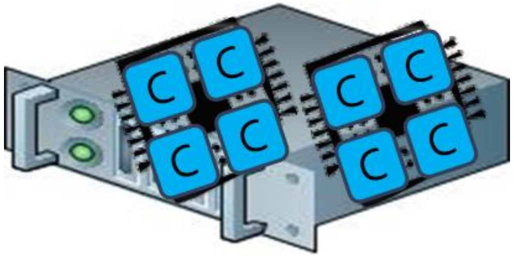
This call is async. Node Js does not wait for it to complete but finish other tasks

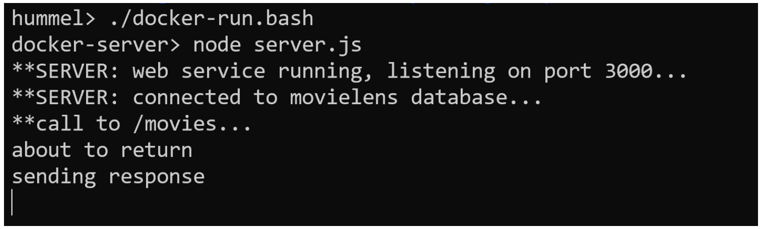
The SQL function is a callback function which gets called when the query gets executed

The pointer to callback function, is stored in the callback queue, which is called when the rest of the tasks are executed.

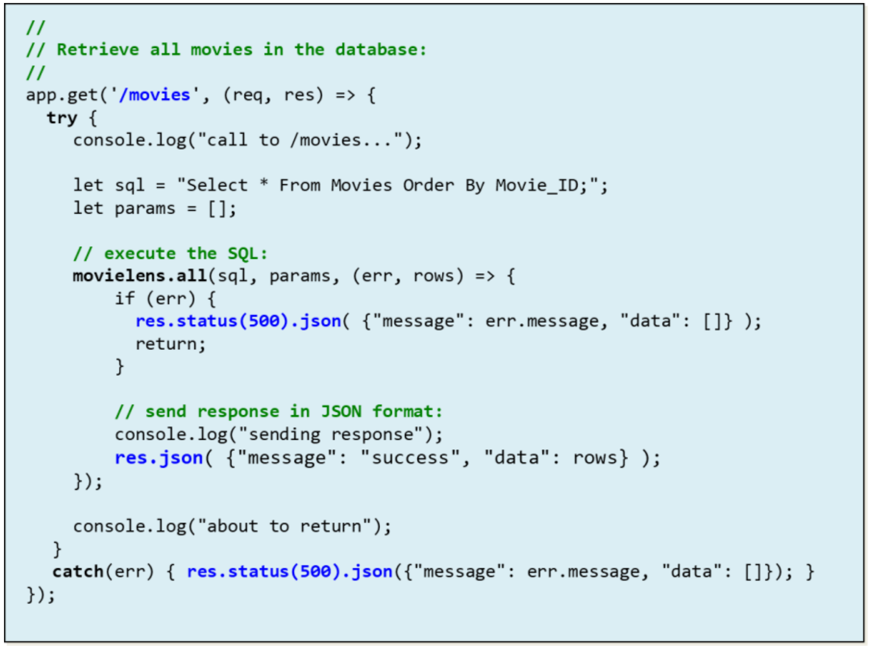
/movies running on Node.js





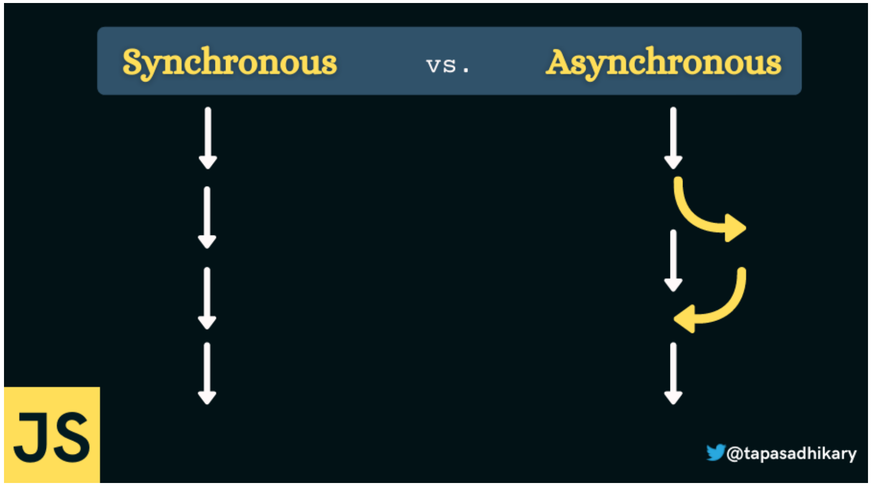


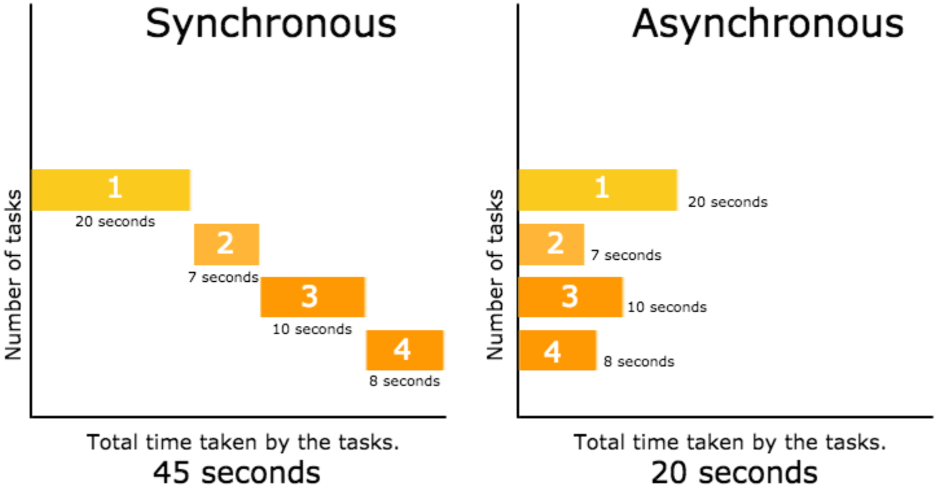
```
hummel> ./docker-run.bash
docker-server> node server.js
**SERVER: web service running, listening on port 3000...
**SERVER: connected to movielens database...
**call to /movies...
about to return
sending response
```



```
//
// Retrieve all movies in the database:
//
app.get('/movies', (req, res) => {
  try {
     console.log("call to /movies...");

     let sql = "Select * From Movies Order By Movie_ID;";
     let params = [];

     // execute the SQL:
     movielens.all(sql, params, (err, rows) => {
         if (err) {
           res.status(500).json( {"message": err.message, "data": []} );
           return;
         }

         // send response in JSON format:
         console.log("sending response");
         res.json( {"message": "success", "data": rows} );
     });

     console.log("about to return");
   }
   catch(err) { res.status(500).json({"message": err.message, "data": []}); }
});
```

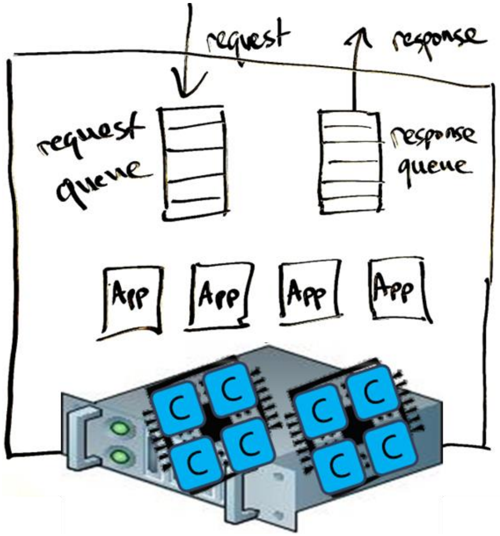
Sync vs. Async

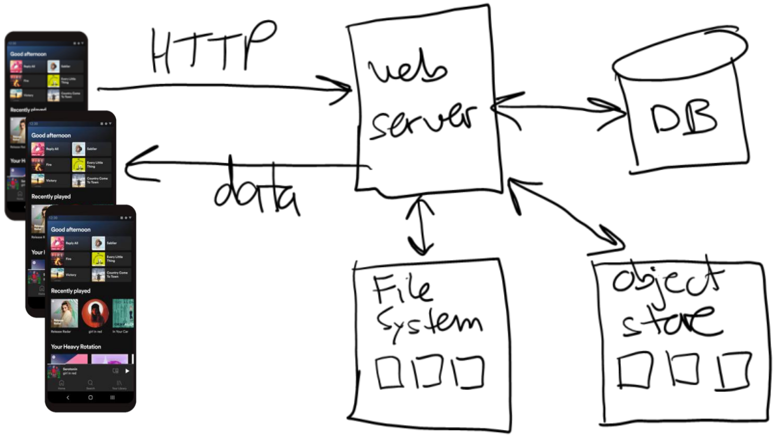




Asynchronous execution

- Asynchronous execution on the server frees up resources (threads) to process more requests & responses
  - Server maintains "in-progress" work queue
  - Async call is initiated, call-back function added to work Q
  - When call completes, call-back function executes

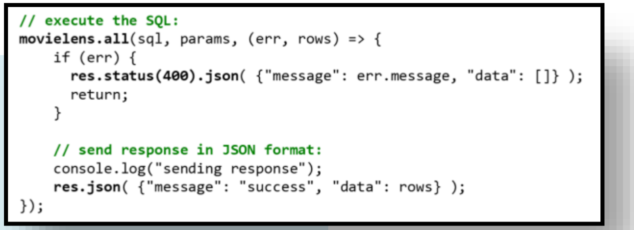




Networking

- **Networking is inherently unpredictable**
  - *You cannot predict how long a network operation will take*
  - *Network instability, traffic load, server load, …*

- **Solution?**
  - *Most networking libraries / functions are **asynchronous***



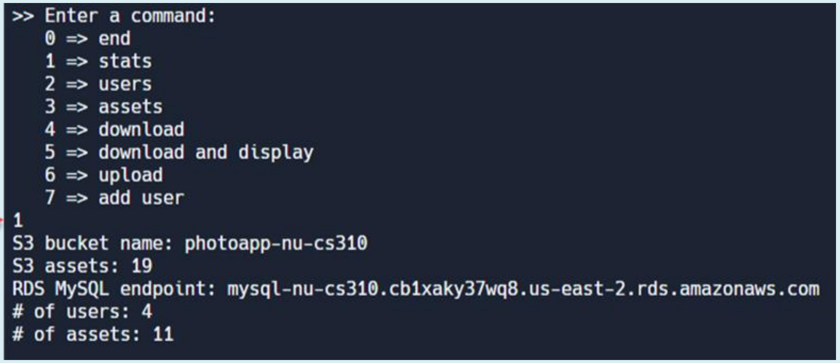
```
// execute the SQL:
movielens.all(sql, params, (err, rows) => {
    if (err) {
      res.status(400).json( {"message": err.message, "data": []} );
      return;
    }

    // send response in JSON format:
    console.log("sending response");
    res.json( {"message": "success", "data": rows} );
});
```

1. You **call** to *start* the operation
2. You go off and do other things…
3. You are **notified** when the operation completes (or you can **wait** for completion)
4. Grab result

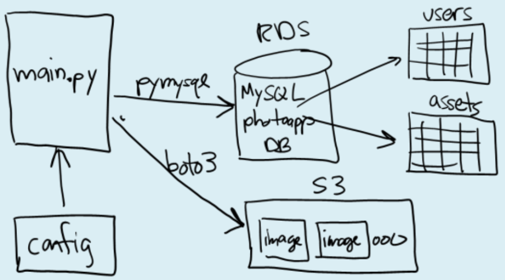
EXAMPLE

Accessing S3

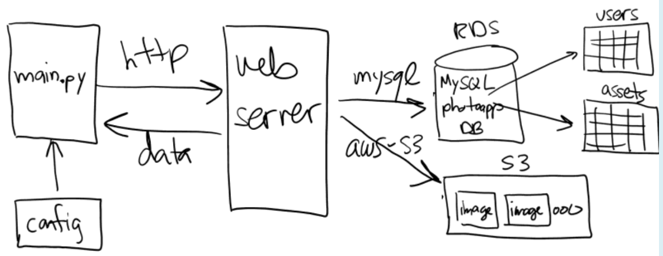
- **Recall the "stats" command from project 01…**



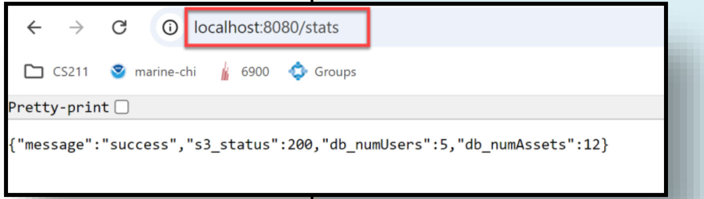
```
>> Enter a command:
   0 => end
   1 => stats
   2 => users
   3 => assets
   4 => download
   5 => download and display
   6 => upload
   7 => add user
1
S3 bucket name: photoapp-nu-cs310
S3 assets: 19
RDS MySQL endpoint: mysql-nu-cs310.cb1xaky37wq8.us-east-2.rds.amazonaws.com
# of users: 4
# of assets: 11
```



- **Project 02 has a similar command…**



```
app.get('/stats', (req, res) => {

  call S3, get status code of bucket (200 => okay)
  .
  .
  .
  res.json({ "message": ...,
             "s3_status": ...,
             "db_numUsers": ...,
             "db_numAssets": ... });
});
```



```
{"message":"success","s3_status":200,"db_numUsers":5,"db_numAssets":12}
```

Attempt #1

```
app.get('/stats', (req, res) => {

  console.log("**Call to get /stats...");


  let input = {
    Bucket: s3_bucket_name
  };

  let command = new HeadBucketCommand(input);
  let s3_response = photoapp_s3.send(command);
```

It does not return a response but a promise object of response, since it is async call.

```
  res.json({ "message": "success",
             "s3_status": s3_response["$metadata"]["httpStatusCode"],
             "db_numUsers": -1,
             "db_numAssets": -1 });
});
```

S3 call is asynchronous, you have to wait for response…

Promises

- **The modern way to wait…**

- **A promise is an object that eventually resolves to a value**
  - *When you need the value, you "await" for it*
  - *Example: s3.send(…)*

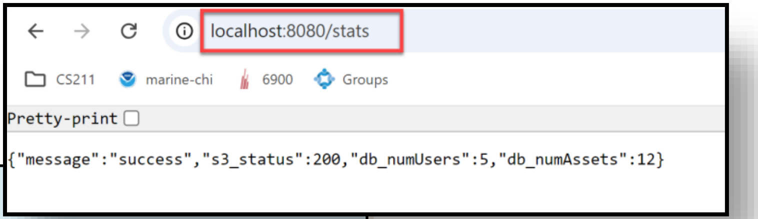
```
app.get('/path', async (req, res) => {
  try {
    let response = F(params);  // F returns a promise

    let result = await response;

    res.json(result);
  }
  catch(err) { res.status(500).json(…); }
});
```

Solution

```
app.get('/stats', async (req, res) => {

  console.log("**Call to get /stats...");

  let input = {
    Bucket: s3_bucket_name
  };

  let command = new HeadBucketCommand(input);
  let s3_response = photoapp_s3.send(command);

  let s3_result = await s3_response;

  res.json({ "message": "success",
             "s3_status": s3_result["$metadata"]["httpStatusCode"],
             "db_numUsers": -1,
             "db_numAssets": -1 });
});
```

Accessing MySQL

- The /stats function is also supposed to get the # of users and # of assets in the database…



```
{"message":"success","s3_status":200,"db_numUsers":5,"db_numAssets":12}
```

```
app.get('/stats', (req, res) => {

  call S3, get status code of bucket

  call MySQL to get # of users in the users table

  call MySQL to get # of assets in the assets table

  res.json({ "message": ...,
             "s3_status": ...,
             "db_numUsers": ...,
             "db_numAssets": ... });
});
```

Callbacks

- **MySQL library is based on callbacks, not promises…**
- **In this case, the result is ONLY available inside the callback**
  - *Example: db.query(…)*

```
app.get('/path', (req, res) => {
  try {

    db.query(sql, (err, result, …) => {
       try {
         if (err)
           res.status(500).json(err.message);
         else
           res.json(result);
       }
       catch(err) {…}
    });

  }
  catch(err) {…}
});
```

Solution

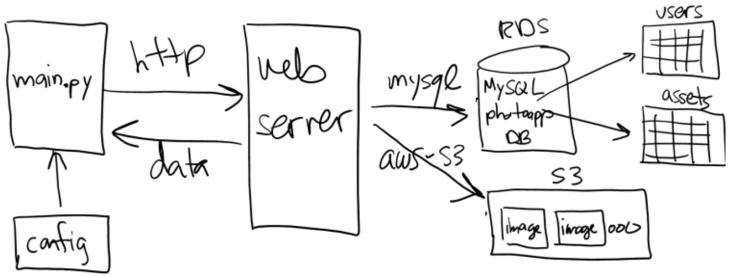
```
app.get('/stats', async (req, res) => {

  console.log("**Call to get /stats...");

  let input = {
    Bucket: s3_bucket_name
  };

  let command = new HeadBucketCommand(input);
  let s3_response = photoapp_s3.send(command);

  let sql = "select count(*) as NumUsers from users;";

  photoapp_db.query(sql, async (err, db_result, _) => {

    if (err) {
      res.status(500).json({ … });
    }
    else {
      let s3_result = await s3_response;
      let row = db_result[0];  // we got one row back, extract it

      res.json({ "message": "success",
             "s3_status": s3_result["$metadata"]["httpStatusCode"],
             "db_numUsers": row["NumUsers"],
             "db_numAssets": -1 });
    }//else

  });
});
```

We await for S3 inside the callback so it runs concurrently with MySQL…

This is very bad and complicated execution. If we need to add one more query execution, then we have to do another level of nesting. So we need to take some design decisions to execute this better.

Design question

- The /stats function needs to perform 4 steps:

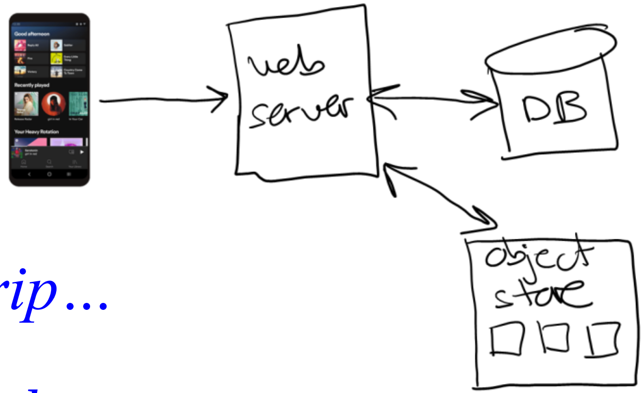


```
app.get('/stats', (req, res) => {

  (1) call S3, get status of bucket

  (2) call MySQL, get # of users in the users table

  (3) call MySQL, get # of assets in the assets table

  (4) res.json({ "message": ...,
                 "s3_status": ...,
                 "db_numUsers": ...,
                 "db_numAssets": ... });
});
```

To minimize response time to the client, what's the best way to execute these 4 steps?

Most efficient approach

- **Minimize trips across the network, and work concurrently when possible…**



1. *Put the SQL queries together into one trip…*
2. *Issue S3 and MySQL calls asynchronously so they execute concurrently…*

```
app.get('/stats', async (req, res) => {

  F1 = async () => s3.send("HeadBucketCommand"));

  F2 = async () => db.query(`Select count(*) from users;
                             Select count(*) from assets;`);

  wait(F1, F2);

  res.json(...);
});
```

pseudo-code…

Visualizing execution…

```
app.get('/stats', (req, res) => {

  F1 = async () => s3.send("HeadBucketCommand"));

  F2 = async () => db.query(`Select count(*) from users;
                             Select count(*) from assets;`);

  wait(F1, F2);  // pseudo-code…

  res.json(...);
});
```

Fork-join…

That's it, thank you!