SENG 365 Week 5

GraphQL and API Testing

This week

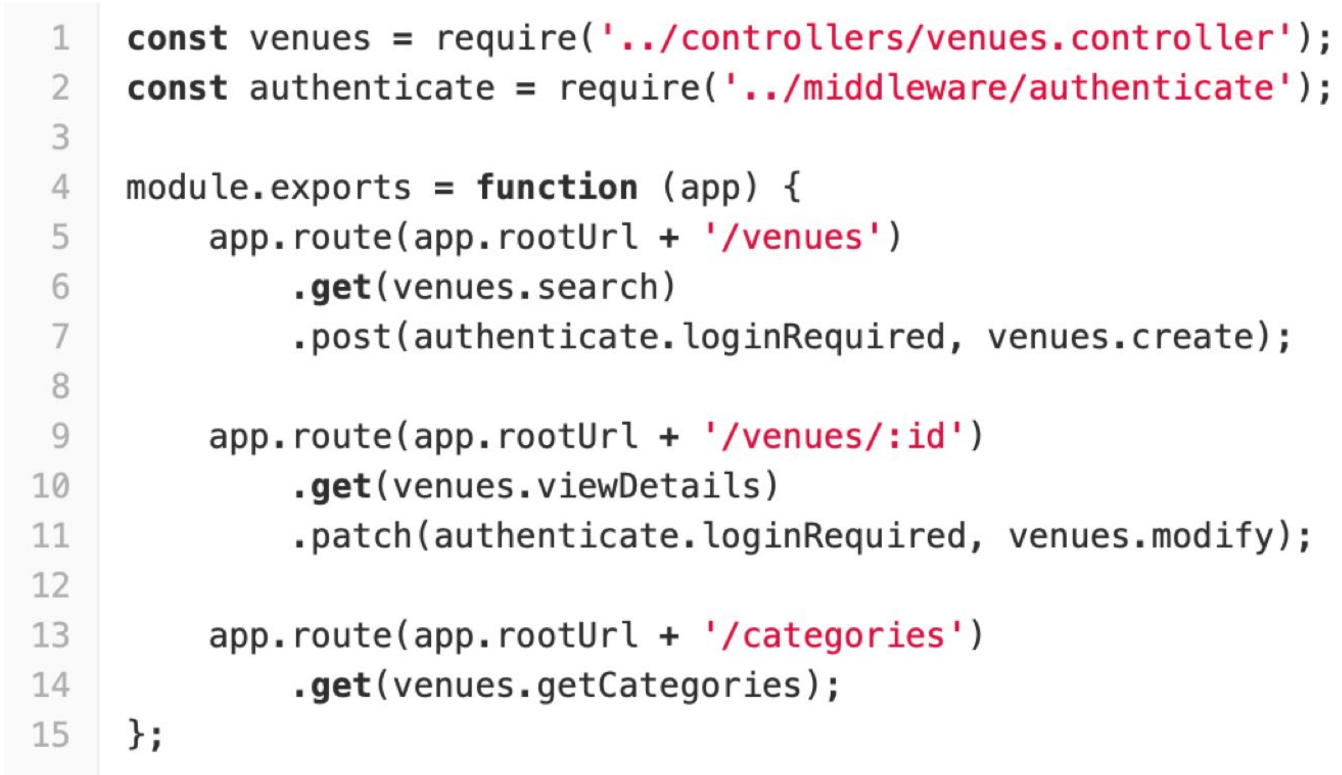
- More info on assignment
- GraphQL
- API testing

More info on Assignment 1

Getting started

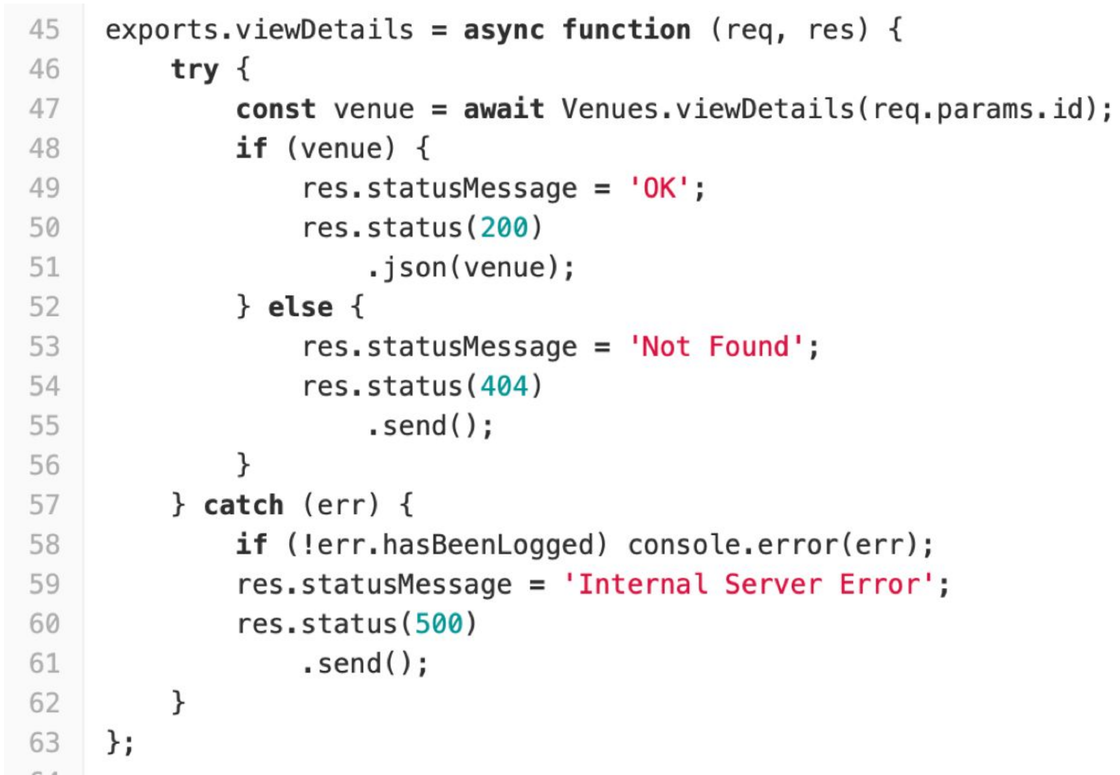
- Some of the endpoints rely on other endpoints
- E.g. you cannot do a POST request to create a new event until you have logged in
    - You will get a 401 unauthorized error
- Where to start?
    - Implementing user endpoints
    - Other GET requests that do not rely on user authentication

Example routes code (in JS)



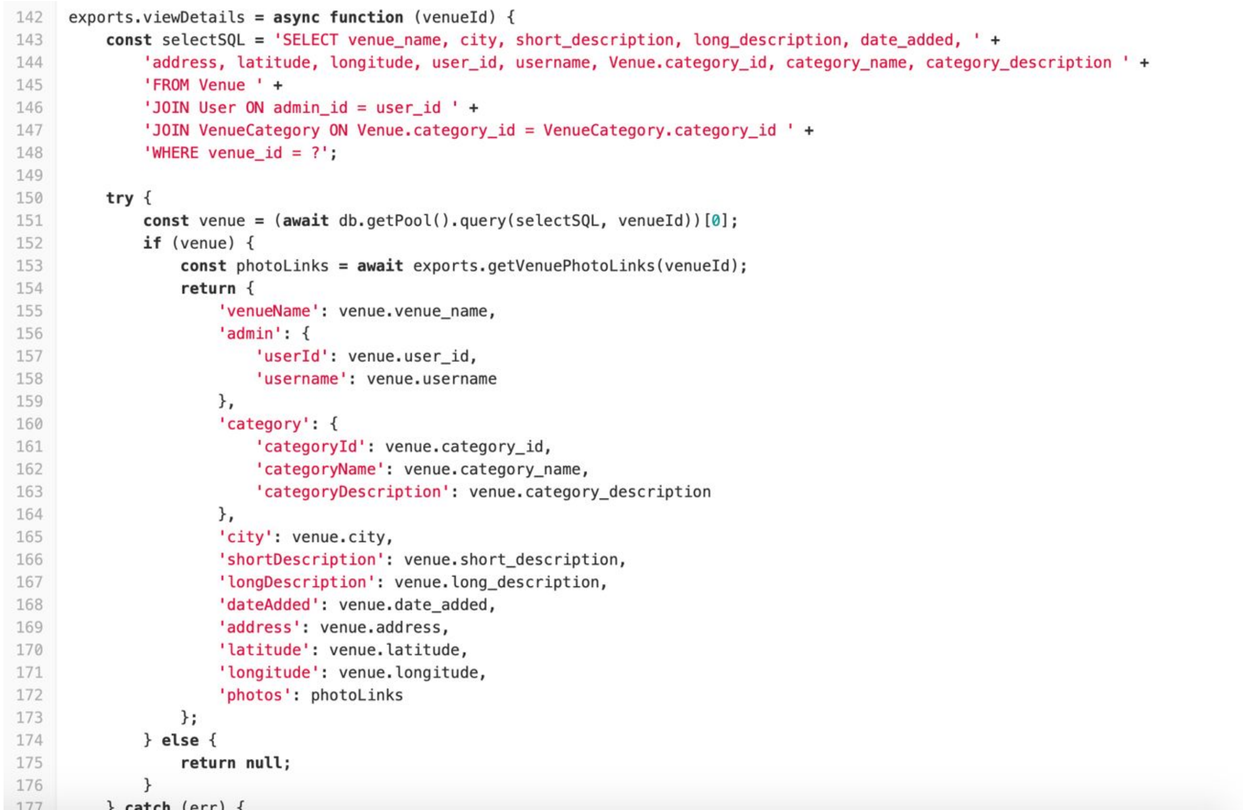
```javascript
const venues = require('../controllers/venues.controller');
const authenticate = require('../middleware/authenticate');

module.exports = function (app) {
    app.route(app.rootUrl + '/venues')
        .get(venues.search)
        .post(authenticate.loginRequired, venues.create);

    app.route(app.rootUrl + '/venues/:id')
        .get(venues.viewDetails)
        .patch(authenticate.loginRequired, venues.modify);

    app.route(app.rootUrl + '/categories')
        .get(venues.getCategories);
};
```

Example controller code (in JS)



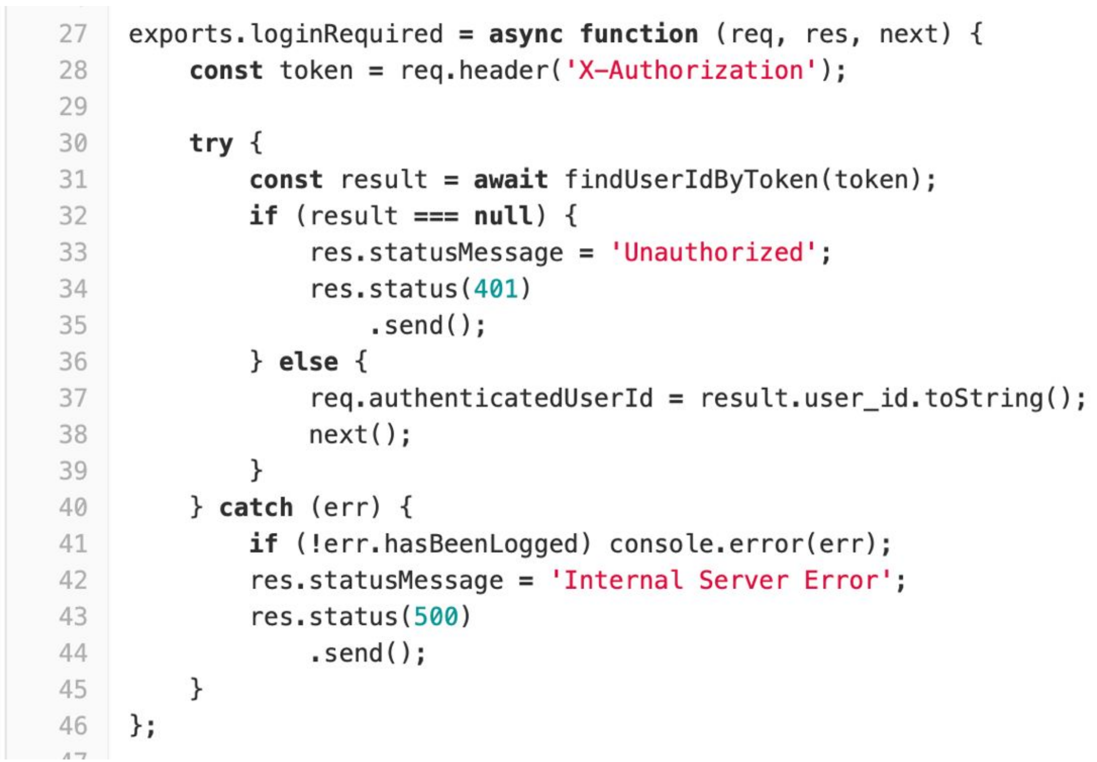
```
exports.viewDetails = async function (req, res) {
    try {
        const venue = await Venues.viewDetails(req.params.id);
        if (venue) {
            res.statusMessage = 'OK';
            res.status(200)
                .json(venue);
        } else {
            res.statusMessage = 'Not Found';
            res.status(404)
                .send();
        }
    } catch (err) {
        if (!err.hasBeenLogged) console.error(err);
        res.statusMessage = 'Internal Server Error';
        res.status(500)
            .send();
    }
};
```

Example model code (in JS)



```
exports.viewDetails = async function (venueId) {
    const selectSQL = 'SELECT venue_name, city, short_description, long_description, date_added, ' +
        'address, latitude, longitude, user_id, username, Venue.category_id, category_name, category_description ' +
        'FROM Venue ' +
        'JOIN User ON admin_id = user_id ' +
        'JOIN VenueCategory ON Venue.category_id = VenueCategory.category_id ' +
        'WHERE venue_id = ?';

    try {
        const venue = (await db.getPool().query(selectSQL, venueId))[0];
        if (venue) {
            const photoLinks = await exports.getVenuePhotoLinks(venueId);
            return {
                'venueName': venue.venue_name,
                'admin': {
                    'userId': venue.user_id,
                    'username': venue.username
                },
                'category': {
                    'categoryId': venue.category_id,
                    'categoryName': venue.category_name,
                    'categoryDescription': venue.category_description
                },
                'city': venue.city,
                'shortDescription': venue.short_description,
                'longDescription': venue.long_description,
                'dateAdded': venue.date_added,
                'address': venue.address,
                'latitude': venue.latitude,
                'longitude': venue.longitude,
                'photos': photoLinks
            };
        } else {
            return null;
        }
    } catch (err) {
```

Authentication



```javascript
exports.loginRequired = async function (req, res, next) {
    const token = req.header('X-Authorization');

    try {
        const result = await findUserIdByToken(token);
        if (result === null) {
            res.statusMessage = 'Unauthorized';
            res.status(401)
                .send();
        } else {
            req.authenticatedUserId = result.user_id.toString();
            next();
        }
    } catch (err) {
        if (!err.hasBeenLogged) console.error(err);
        res.statusMessage = 'Internal Server Error';
        res.status(500)
            .send();
    }
};
```

Some advice #1

- We are testing against the **API specification**
- Be **clear** about what you are trying to achieve with **each function**
- Ensure npm packages have been added to **package.json**
- Remember to do an **npm install** when doing a clean test deploy
- Be aware of your **npm dependencies**
  - Dependencies in **dev** vs dependencies for **prod** e.g. nodemon
- Remember the prefix to the URL, **/api/v1**
- Check against the **latest version** of the API specification
  - Am I using the correct parameters? Are they formatted correctly?

Some advice #2

- How are you handling photos?
  - Do you need to add a photo directory to git?
  - `/storage/photos` is tracked, but the contents are not...
  - Make sure that you use correct mime type for images, e.g. `image/png`
  - Use `mz/fs` to handle file reading and writing of image files from filesystem: `https://www.npmjs.com/package/mz`

- Test against the reference server

Some advice #3

- Encrypting password in database
  - Best practice to use existing library, e.g. `bcrypt`
  - https://www.npmjs.com/package/bcrypt
  - We will test that you are not storing the password in plain text
- Generate authentication token
  - Several options: e.g. `rand-token`:
  - https://www.npmjs.com/package/rand-token

{REST}

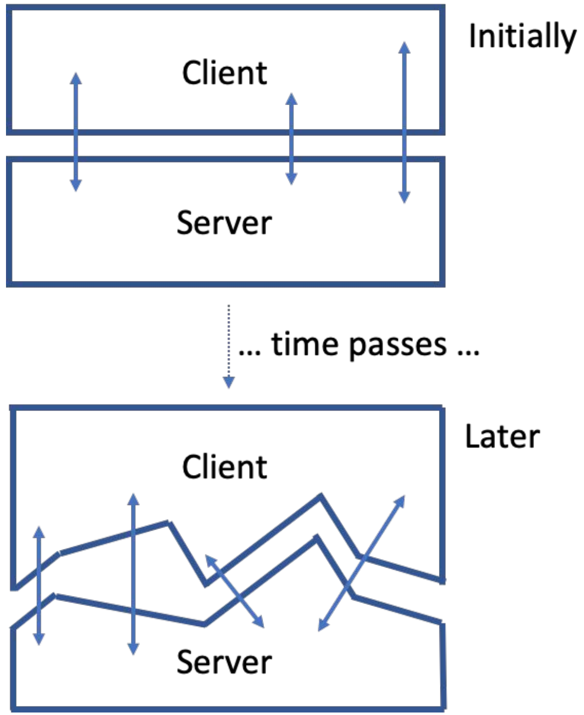
GraphQL

Endpoints and client views

- Endpoints tend to be designed and structured according to the views expected to be needed on the front-end
  - e.g. we design request parameters (query & body) and the response's JSON structure to fit the view
- That's an efficient design…

… EXCEPT THAT…

- Views change
- Users want different information, more information, less information, more and less views
- The fit between endpoint/s and view/s therefore disintegrates



RESTful APIs and their limitations

- Fetching complicated data structures requires **multiple round trips** between the client and server.
- For mobile applications operating in **variable network conditions**, these multiple roundtrips are highly undesirable.

An example set of requests

```
/auctions/{id}
/auctions/{id}/bids
/users/…
/auctions/{id}/photos
```

Overfetching and underfetching

Overfetch: Download more data than you need

- e.g. you might only need a list of usernames, but `/users` downloads
- (as a JSON object) more data than just usernames
- And endpoint provides more than you need

Underfetch: download less than you need so must then do more (the n+1 problem)

- e.g. you need a list of most recent three friends for a username, so for each item in `/users` you need to get information from `/user/friends`, but then only take the first three entries

RESTful APIs and their limitations cont.

- REST endpoints are usually **weakly-typed** and lack machine-readable metadata.

An example of the confusion

`eventStartTime integer`

Why integer and not Date?

Mapping from integer to date and time?

`POST /events` API, is `startingTime` the same as the `event_startingtime` in the `events` table?

GraphQL

- A specification for:
  - How you specify data (cf. strong-typing)
  - How you query that data
- There are reference implementations of the GraphQL specification
  - https://github.com/graphql/graphql-js (Node.js)
- Extra lab on LEARN (not pre-req for assignment)

GraphQL simple example

Comments

- `Character` is a GraphQL Object Type that has fields
- `name` and `appearsIn` are the fields
- `String` is a scalar type (a base type that's irreducible)
- `[Episode]!` is an array [] that's non-nullable (due to the !)
- Each `type Query` specifies an entry point for every GraphQL query.

Example (of API)

```
type Character {
  name: String!
  appearsIn: [Episode]!
}
type Query {
  hero: Character
}
```

GraphQL vs REST

GraphQL

- Define objects and fields that can be query-able
- Define **entry points** for a query
- The client application can dynamically ‘compose’ the content of the query
- A much more flexible interface to the server side.

REST

- **Endpoints** that are set and inflexible
- Pre-defined fixed endpoints that
  - Require pre-defined inputs
  - Return pre-defined data structures
- Those endpoints are then ‘set’…
  - … until version x.y.z of the API

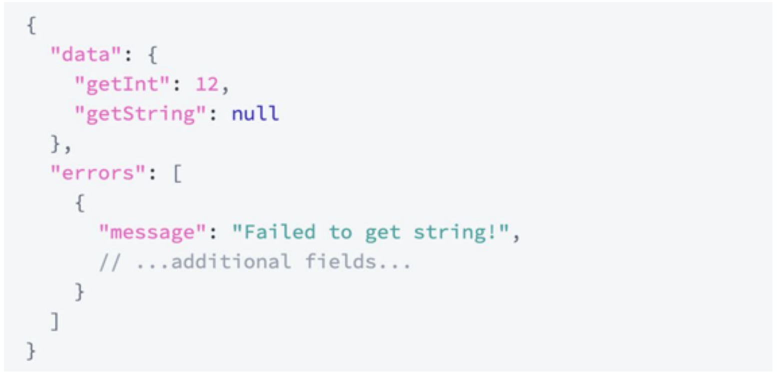
GraphQL vs REST response codes

GraphQL

- All GraphQL queries return 200 response code, even errors.
    - E.g. malformed query, query does not match schema, etc.
- Errors are returned in user-defined field
- Network errors can still return 4xx/5xx
    - E.g. GraphQL server is down

REST

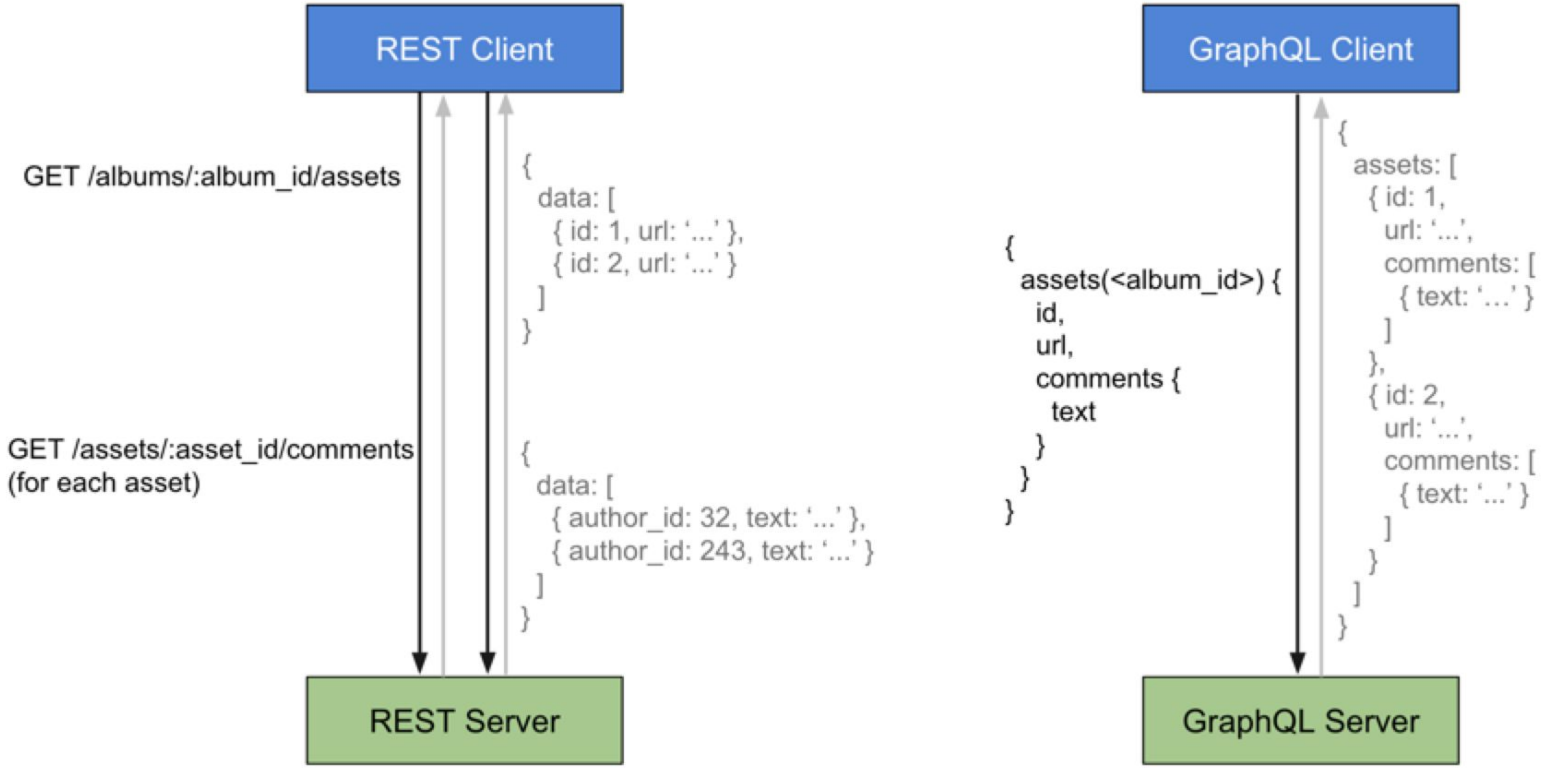
- HTTP response code indicates success / error
- 2xx, 4xx, 5xx, etc.



```
{
  "data": {
    "getInt": 12,
    "getString": null
  },
  "errors": [
    {
      "message": "Failed to get string!",
      // ...additional fields...
    }
  ]
}
```

GraphQL and data

- Does not require you to think in terms of graphs
  - You think in terms of JSON-like structures for a query (see earlier slide)
- Is not querying the database directly
  - Rather is a ‘language’ (specification) for composing queries to a server
- Still requires some kind of pre-defined data and queries on the server-side
  - Objects, fields and allowable queries
  - But these pre-definitions are more ‘atomic’ in their nature


REST Client
GET /albums/:album_id/assets
GET /assets/:asset_id/comments (for each asset)
REST Server
GraphQL Client
GraphQL Server

```
{
  data: [
    { id: 1, url: '...' },
    { id: 2, url: '...' }
  ]
}
```

```
{
  data: [
    { author_id: 32, text: '...' },
    { author_id: 243, text: '...' }
  ]
}
```

```
{
  assets(<album_id>) {
    id,
    url,
    comments {
      text
    }
  }
}
```

```
{
  assets: [
    { id: 1,
      url: '...',
      comments: [
        { text: '...' }
      ]
    },
    { id: 2,
      url: '...',
      comments: [
        { text: '...' }
      ]
    }
  ]
}
```

REST vs GraphQL requests

GraphQL uses GET and POST

GET

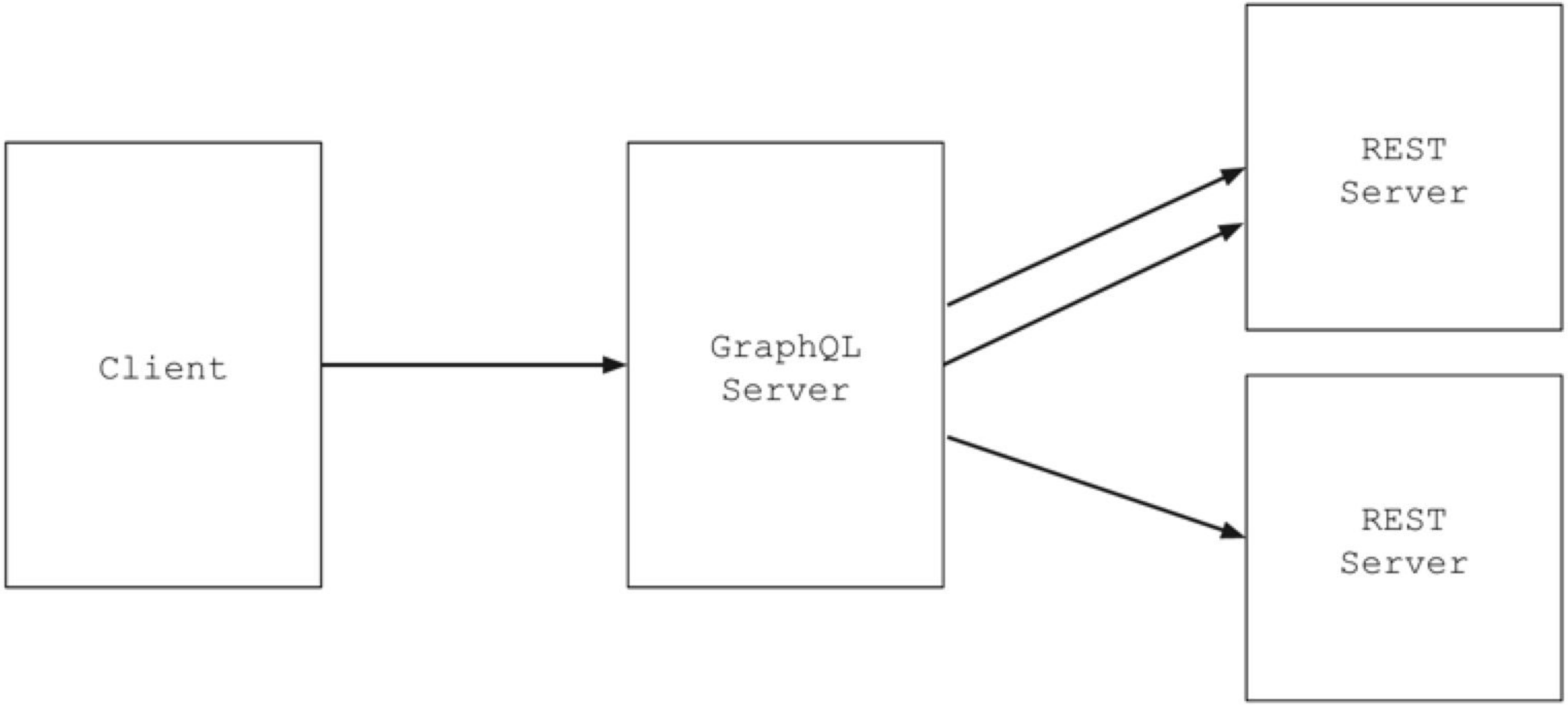
GraphQL query is specified using the URL query parameters

```
http://myapi/graphql?query={me{name}}
```

POST

Specify the query in the HTTP body, using JSON

```
"query": "...",
"operationName": "...",
"variables": {
  "myVariable": "someValue",
  ...
}
```



GraphQL can sit in front of REST API(s)

GraphQL additional resources

- GraphQL Introduction
  - https://graphql.org
- Express + GraphQL
  - https://www.npmjs.com/package/express-graphql
- Apollo GraphQL Server
  - https://www.apollographql.com/docs/apollo-server/
- From REST to GraphQL
  - https://0x2a.sh/from-rest-to-graphql-b4e95e94c26b

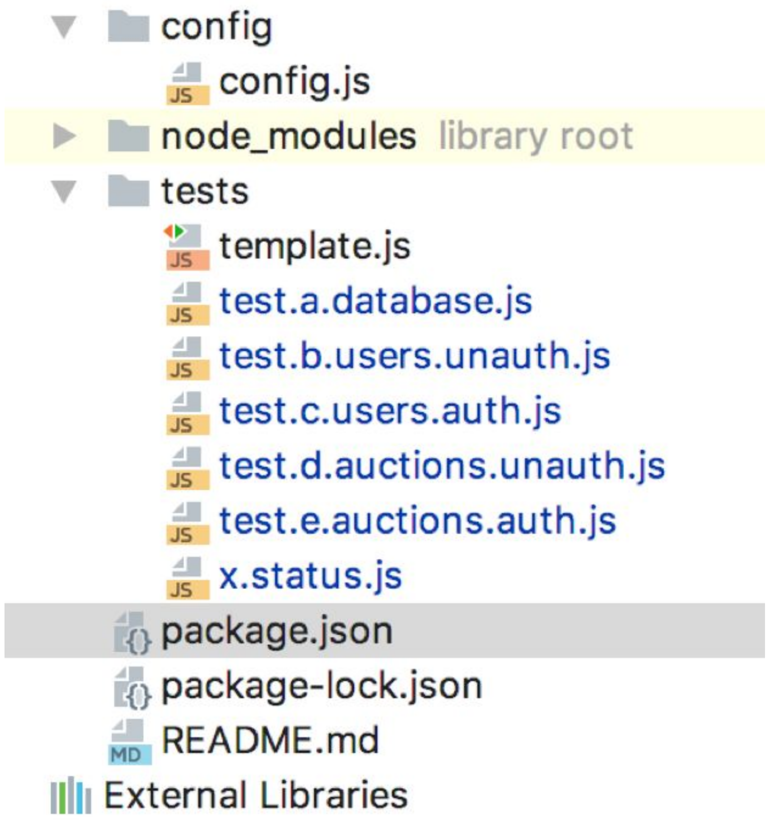
Automated API Testing

API testing

- **Postman** tests – Javascript console for testing API endpoints
- **Mocha + Chai**
  - Packages for automated testing Node.JS code
  - Can be used with continuous integration/deployment (CI/CD) environments, such as GitLab Runner
  - Mocha – Asynchronous testing environment
  - Chai – Assertion library
  - https://www.digitalocean.com/community/tutorials/test-a-node-restful-api-with-mocha-and-chai

Mocha + Chai setup

- Can have one test file
- For multiple test files:
  - Mocha runs test files in order of occurrence (depends on OS's file systems)
  - Depends on how defined in `package.json`
- Each test (even multiple tests in one test file):
  - Is intended to be independent
  - Runs asynchronously



```
▼ config
    config.js
▶ node_modules  library root
▼ tests
    template.js
    test.a.database.js
    test.b.users.unauth.js
    test.c.users.auth.js
    test.d.auctions.unauth.js
    test.e.auctions.auth.js
    x.status.js
  package.json
  package-lock.json
  README.md
External Libraries
```

Separate test project

In `package.json`

```
...
"scripts": {
    "start": "mocha ./tests/test.*.js --reporter spec --log-level=warn",
    "test": "mocha ./tests/test.a.file.js --reporter spec
--log-level=warn",
},
...
```

Given the above:

`npm start` will run all my test files

`npm test` will run a particular test file (that I have specified)

Asynchronous behavior when testing

- You can setup pre- and post-conditions
  - `before()`, `beforeeach()`, `after()`, etc
- Mocha, Chai and Chai-HTTP can handle callbacks, and Promises (and async/await)
  - Don’t get these mixed up in a given test
  - Avoid the use of `return` together with `done()`

A single test using a Promise

```
describe('Test case:/POST/login with parameters in query string', () => {
    it('Should return 200 status code, id and authorisation token', function () {
        return chai.request(server_url)
            .post('/users/login')
            .query(
                {
                    username: 'testUsername4',
                    email: "user4@testexample.com",
                    password: "testpassword"
                }
            )
            .then(function(res){
                expect(res).to.have.status(200);
                expect(res).to.be.json;
                expect(res.body).to.have.property('id');
                expect(res.body).to.have.property('token');
                authorisation_token = res.body['token’]; //use in subsequent test
                user_id = res.body['id’]; //use in subsequent test
            })
            .catch(function (err) {
                expect(err).to.have.any.status(400, 500);
                throw err; // there is any error
            });
    });
});
```

A single test using old-style callbacks

```
describe('Test case: ' + test_case_count + ': POST /users', () => {
    it('Callback with done(): Should return 400 or 500 as there was a duplicate entry', (done) => {
        chai.request(server_url)
            .post('/users')
            .send(
                {
                    username: "testUsername4",
                    givenName: "testGivenName",
                    familyName: "testFamilyName",
                    email: "user@testexample.com",
                    password: "testpassword"
                }
            )
            .then(function (res) {
                expect(res).to.have.any.status(201); // is this line really needed?
                done(new Error("Status code 201 returned unexpectedly")); //test completed but failed
            })
            .catch(function (err) {
                expect(err).to.have.any.status(400,500);
                done(); // test completed as it should / as it was expected to complete
            });

    });
});
```

Tests are asynchronous

- With the assignment, for example, you would be testing a **network request** to a server that is then making a **database request**
- **You don't know when** the network request or the database request will **complete**
  - Therefore you don't know when the test will complete
- You **shouldn't assume** that **test n+1** will complete before **test n+2** starts
  - Which is why you have `before()`, `beforeeach()`, `after()` etc.
- Need to be careful with the **dependencies between tests**
- Need to be careful on how you **report the progress of tests**, because the report **may not output synchronously** with completion of the test itself

Testing for expected success and expected failure

- Often we test to corroborate that something completes as we expected
  - e.g. that `user/login` is successful as expected: the user logs in
- We also need to test that the system rejects/doesn’t complete as expected
  - e.g. that `user/login` is unsuccessful as expected: the user is not logged in
- Need to think carefully about:
  - `.then()`, `catch()`, `done()`, `done(err)`, and/or `throw err;`

Passing tests does not always mean intended behavior

	Actual behavior: successful	Actual behavior: failed
Intended behavior: successful	The test passed	The test failed
Intended behavior: failure	The test failed	The test passed