UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

SENG365 Web Computing Architecture: #5 GraphQL and automated API testing

Ben Adams
Course Coordinator & Lecturer
benjamin.adams@canterbury.ac.nz
310, Erskine Building

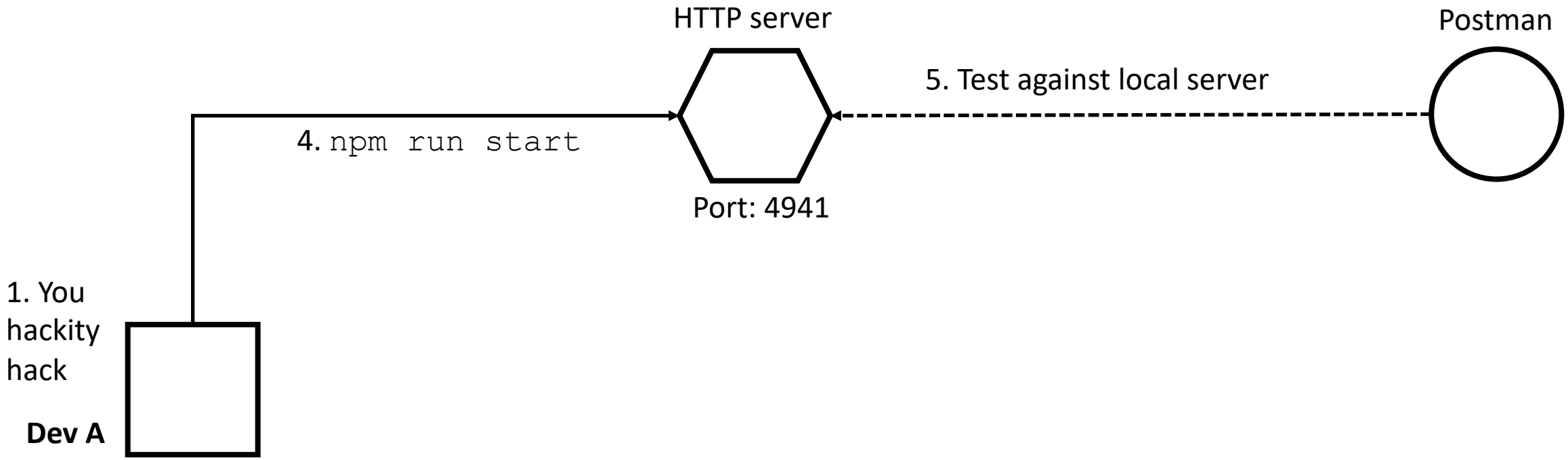
In the lecture this week

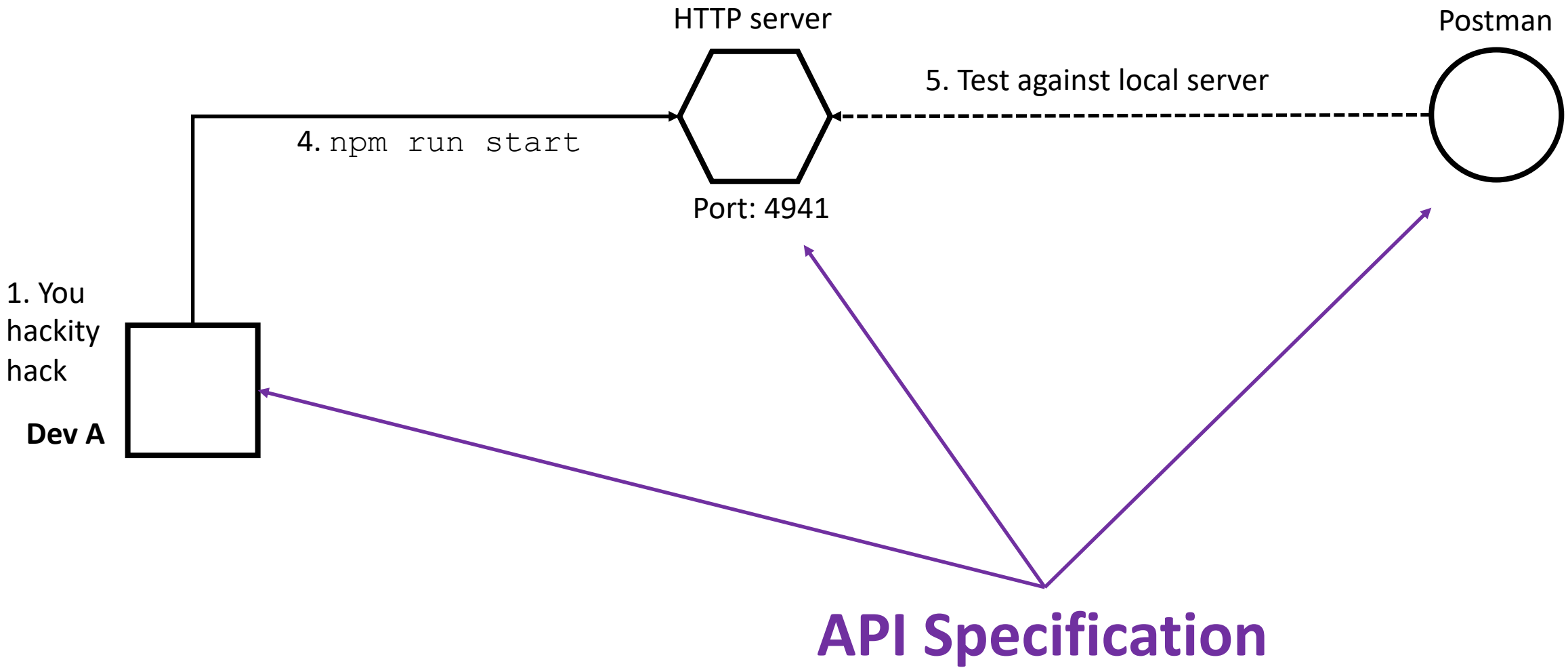
- Assignment 1
- GraphQL
- Automated API testing
- Questions about the mid-term

Mid-semester test

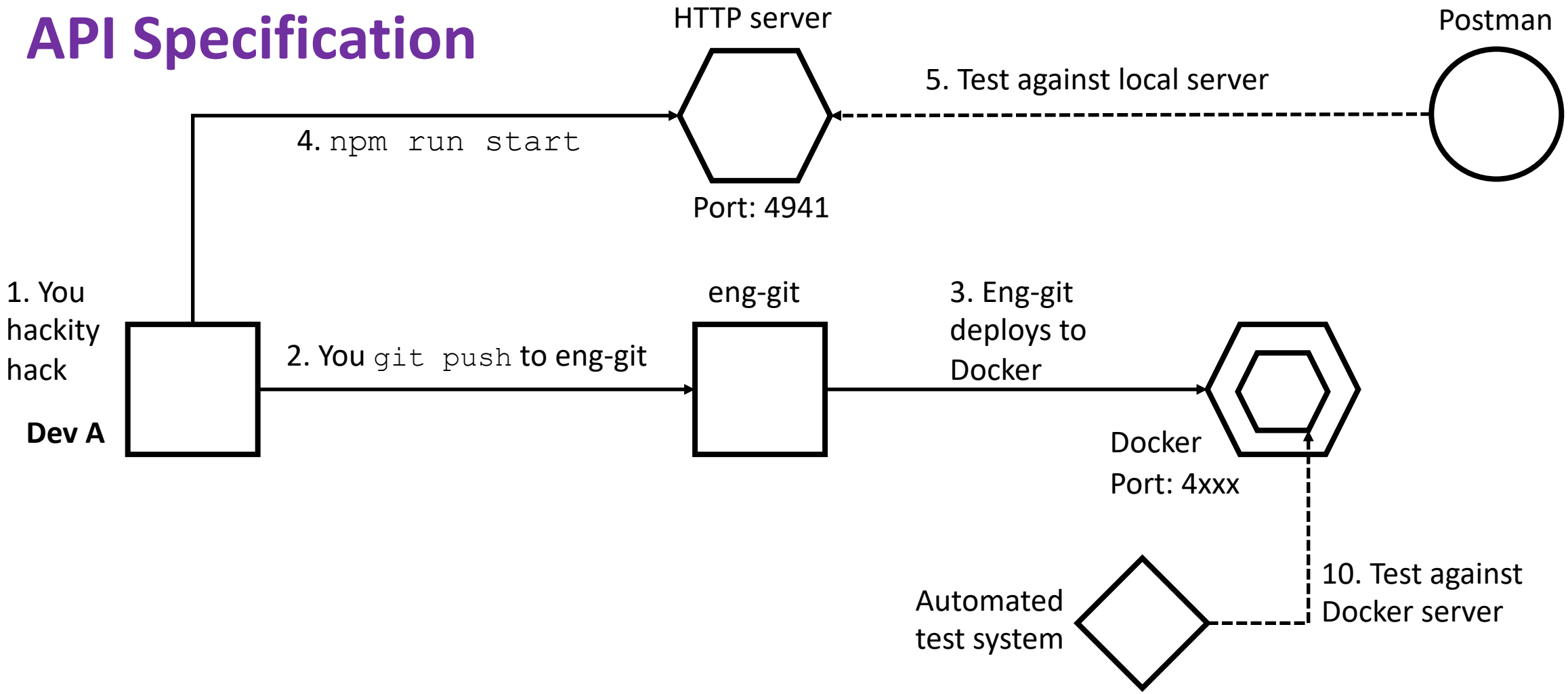
- **Tomorrow 7pm**
- **E8 Lecture Theatre**
- **Please bring pen/pencil**

Assignment 1


HTTP server
Postman
5. Test against local server
4. npm run start
Port: 4941
1. You hackity hack
Dev A


HTTP server
Postman
5. Test against local server
4. npm run start
Port: 4941
1. You hackity hack
Dev A
API Specification

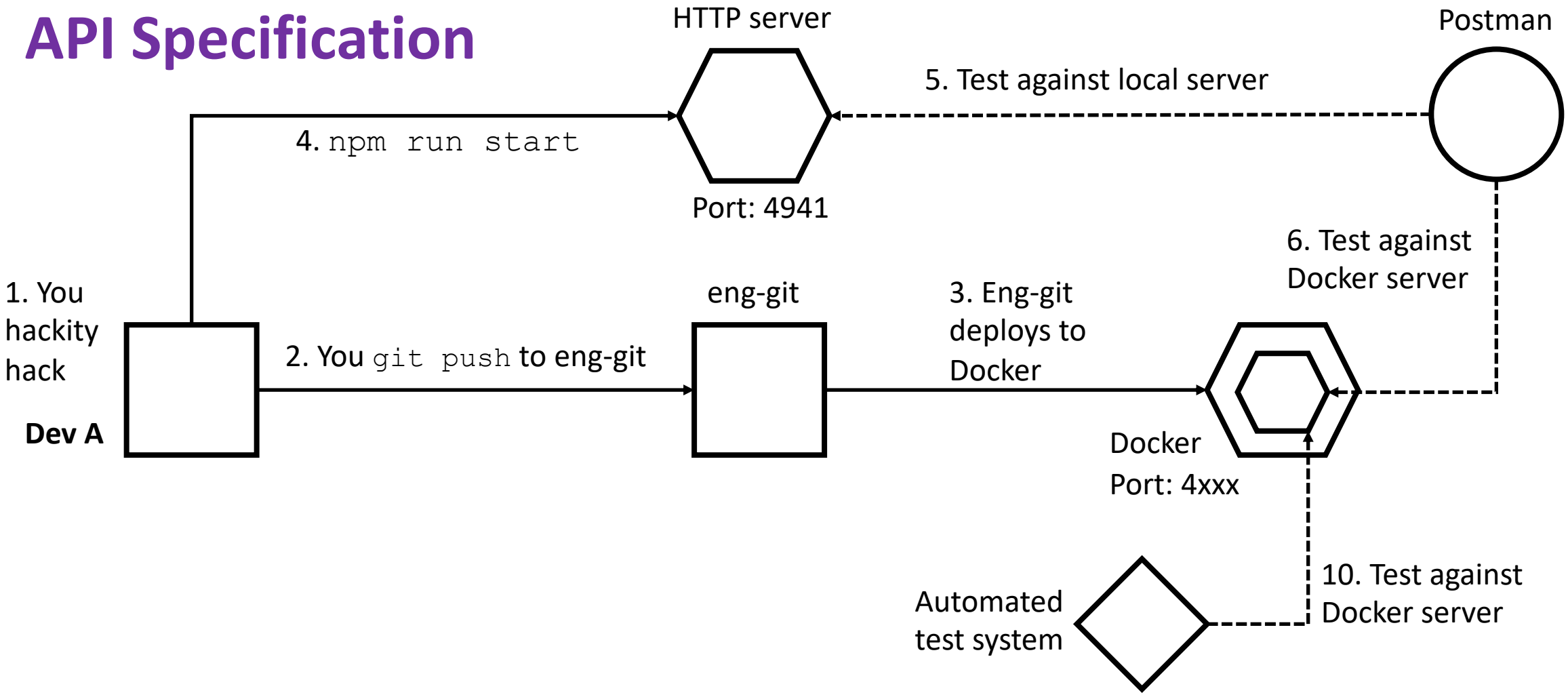
API Specification



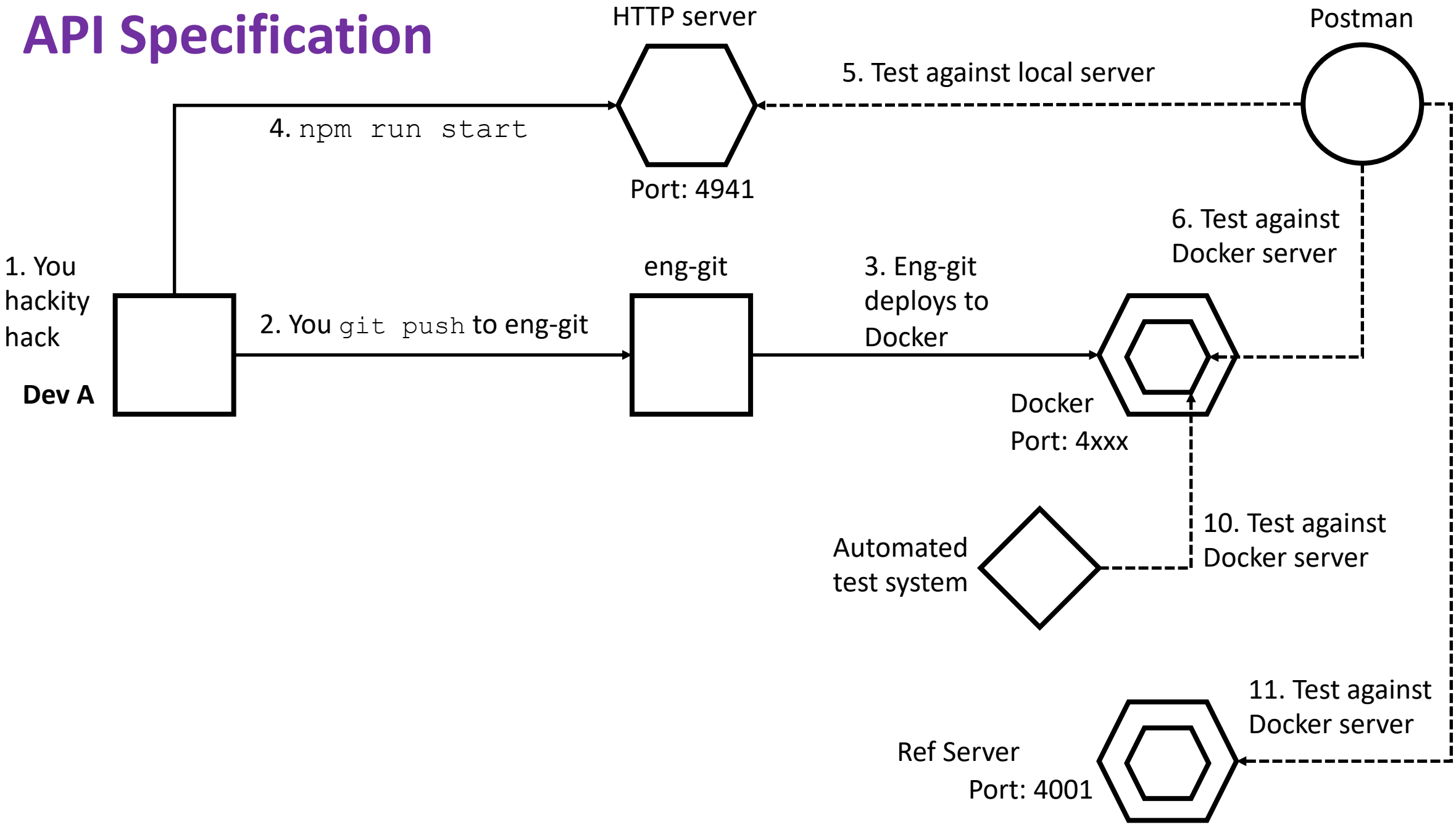
Why may there be differences?

1. Why can your local HTTP server **pass** your Postman tests, and **fail** the automated test server's tests?
2. What problems arise when you use the automated test server?
3. How can we respond to those problems?

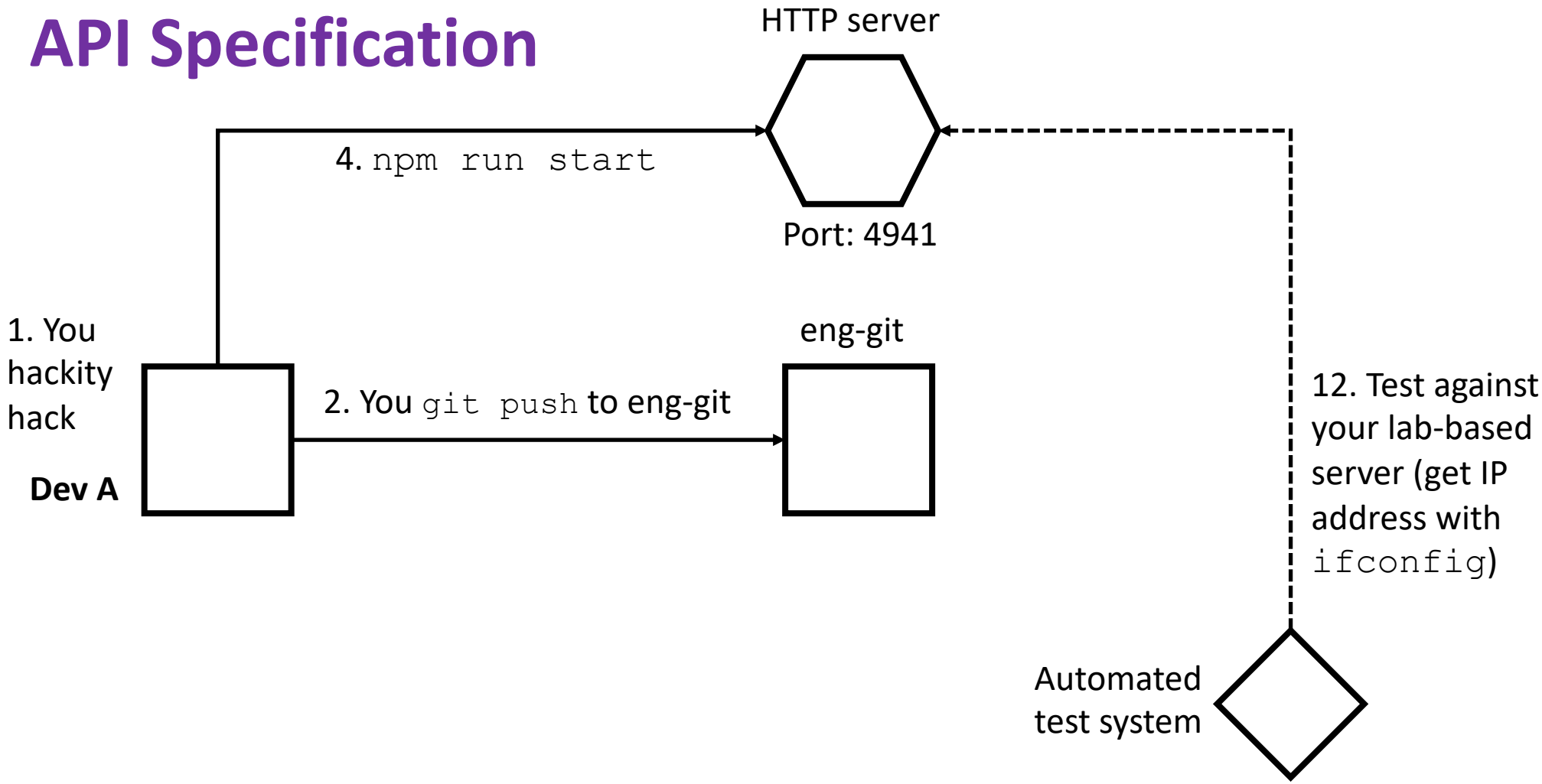
API Specification



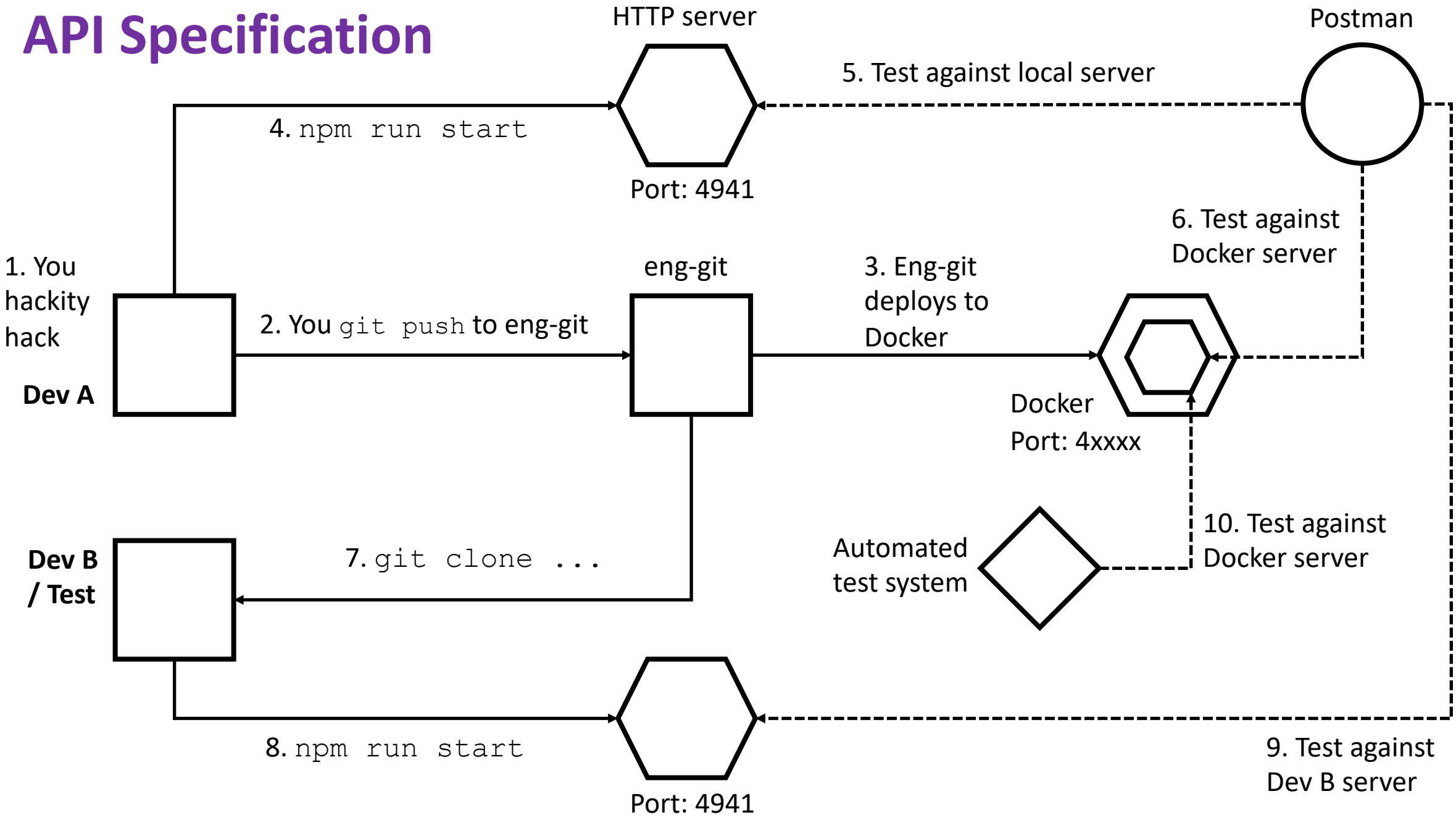
API Specification



API Specification



API Specification



Deploying a second server to a dev/test environment…

- What do I need to remember to do when I manually deploy my project and run my server?
- Why is a test server of my own (rather than the Docker contained version) helpful?

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

```
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

```
45  exports.viewDetails = async function (req, res) {
46      try {
47          const venue = await Venues.viewDetails(req.params.id);
48          if (venue) {
49              res.statusMessage = 'OK';
50              res.status(200)
51                  .json(venue);
52          } else {
53              res.statusMessage = 'Not Found';
54              res.status(404)
55                  .send();
56          }
57      } catch (err) {
58          if (!err.hasBeenLogged) console.error(err);
59          res.statusMessage = 'Internal Server Error';
60          res.status(500)
61              .send();
62      }
63  };
```

Example model code

```
142 exports.viewDetails = async function (venueId) {
143     const selectSQL = 'SELECT venue_name, city, short_description, long_description, date_added, ' +
144         'address, latitude, longitude, user_id, username, Venue.category_id, category_name, category_description ' +
145         'FROM Venue ' +
146         'JOIN User ON admin_id = user_id ' +
147         'JOIN VenueCategory ON Venue.category_id = VenueCategory.category_id ' +
148         'WHERE venue_id = ?';
149 
150     try {
151         const venue = (await db.getPool().query(selectSQL, venueId))[0];
152         if (venue) {
153             const photoLinks = await exports.getVenuePhotoLinks(venueId);
154             return {
155                 'venueName': venue.venue_name,
156                 'admin': {
157                     'userId': venue.user_id,
158                     'username': venue.username
159                 },
160                 'category': {
161                     'categoryId': venue.category_id,
162                     'categoryName': venue.category_name,
163                     'categoryDescription': venue.category_description
164                 },
165                 'city': venue.city,
166                 'shortDescription': venue.short_description,
167                 'longDescription': venue.long_description,
168                 'dateAdded': venue.date_added,
169                 'address': venue.address,
170                 'latitude': venue.latitude,
171                 'longitude': venue.longitude,
172                 'photos': photoLinks
173             };
174         } else {
175             return null;
176         }
177     } catch (err) {
```

Authentication

```
27  exports.loginRequired = async function (req, res, next) {
28      const token = req.header('X-Authorization');
29
30      try {
31          const result = await findUserIdByToken(token);
32          if (result === null) {
33              res.statusMessage = 'Unauthorized';
34              res.status(401)
35                  .send();
36          } else {
37              req.authenticatedUserId = result.user_id.toString();
38              next();
39          }
40      } catch (err) {
41          if (!err.hasBeenLogged) console.error(err);
42          res.statusMessage = 'Internal Server Error';
43          res.status(500)
44              .send();
45      }
46  };
```

Some advice (not exhaustive) #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?
- Use the correct ports: 4941, 4xxx, 3306

Some advice (not exhaustive) #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
- Check the eng-git deployment log
- Test against the reference server

Some advice (not exhaustive) #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}

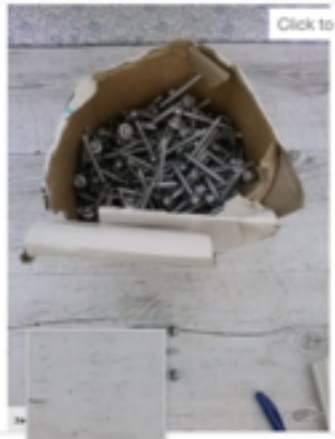
GraphQL

6cm Screws

Listing #: 1570425811
Hastings, Hawke's Bay, NZ

40 people added this to their Watchlist

Pay Now



Closes: 1 min
Tue 20 Mar, 3:34 pm
This auction may auto-extend.

Add to Watchlist

Email reminders: None

Current bid $14.04
Total Bids: 18 Reserve met

Next minimum bid $14.54

Auto-bid Place bid

Hide bid history

Tue 20 Mar

$14.04	baxter04 (1,708)	3:32 pm
$13.54	baxter04 (1,708)	3:32 pm
$13.04	stonepen... (348)	3:32 pm
$12.54	baxter04 (1,708)	3:22 pm
$12.04	rhyco1 (257)	2:47 pm
$11.54	ikipike (198)	9:04 am
$11.04	goodbusi... (244)	7:12 am

Mon 19 Mar

$10.54	ikipike (198)	8:26 pm
$10.50	jimmyjam... (298)	8:26 pm
$6.00	jimmyjam... (298)	8:25 pm

Only the latest 10 bids are shown

Shipping
From $6.50 More...

Buyer Protection Learn More

Description | Questions & answers 2

Maybe around 300 screws

RULES OF ENGAGEMENT---

No warranty implied or given.
All items are sold "as is" and the $1 reserve reflects this.
*Please let us know if you would like more photos on any of our listings.

Pick up: Whakatu, Hastings (Mon-Fri 8.00am-4.30pm)
Items must be collected within 7 days after the Auction closes. Items that have not been collected within two weeks will be relisted unless special arrangements have been made.

Postage: Due to the high volume of sales, we will post winning items out on Wednesdays and Fridays. Once payment is received we will send the items on the next scheduled postage date.

We add items daily so be sure to save us as a seller "3scavengers" and while you're here, check out our other listings.

*Please read the **questions and answers** for this listing.*

Shipping options	
Nationwide	$6.50
Rural	$10.20

Seller allows pick-ups
Seller location: **Hastings, Hawke's**

Payment options

Use your debit or credit card.
What's Pay Now?

14 Classic Fire Engines and memorabilia collection

Listing #: 1571234651
Wellington City, Wellington, NZ

825 people added this to their Watchlist

Closes: Wed 28 Mar, 12:00 pm
(7 days, 20 hours, 30 minutes)

Add to Watchlist

Email reminders: 12 hrs

Start price $150,000.00
No reserve

Starting bid $150,000.00

Auto-bid Place bid

Shipping
Buyer must pick-up More...

Key details

On road costs Excluded
Additional costs may apply

Description | Questions & answers 25

Have you ever wanted to be a fire fighter or even own your own fire engine? Well I did and after 35 years of collecting I have decided to sell my personal collection and find a new passion in life!

I purchased my first fire engine back in 2000 and started a collection now known as the "Wellington Fire Museum" I have many items including fire helmets and uniforms, waterway equipment, Ladders. As there is too much to list individually I would recommend viewing. Please also see the photographs to give you an idea of what is available.

Featured on Stuff, paste link below to find out more
https://www.stuff.co.nz/national/101973760/passion-loses-its-spark-as-fire-engine-collector-puts-his-museum-up-for-sale

For more details check out the Wellington Fire Museum Facebook page

Fire engines as picture (2 -15)

One Business Mobile:
3 months money back guarantee
One month term
$15 per user, per month on selected Red+ Business plans
Find out more >
T&Cs apply
vodafone

Specialist Cars buyer's checklist
- Check the history and ownership with MotorWeb
- How much to insure this classic vehicle?
- How much to get this delivered?
- Check out the Motorsport NZ regulations

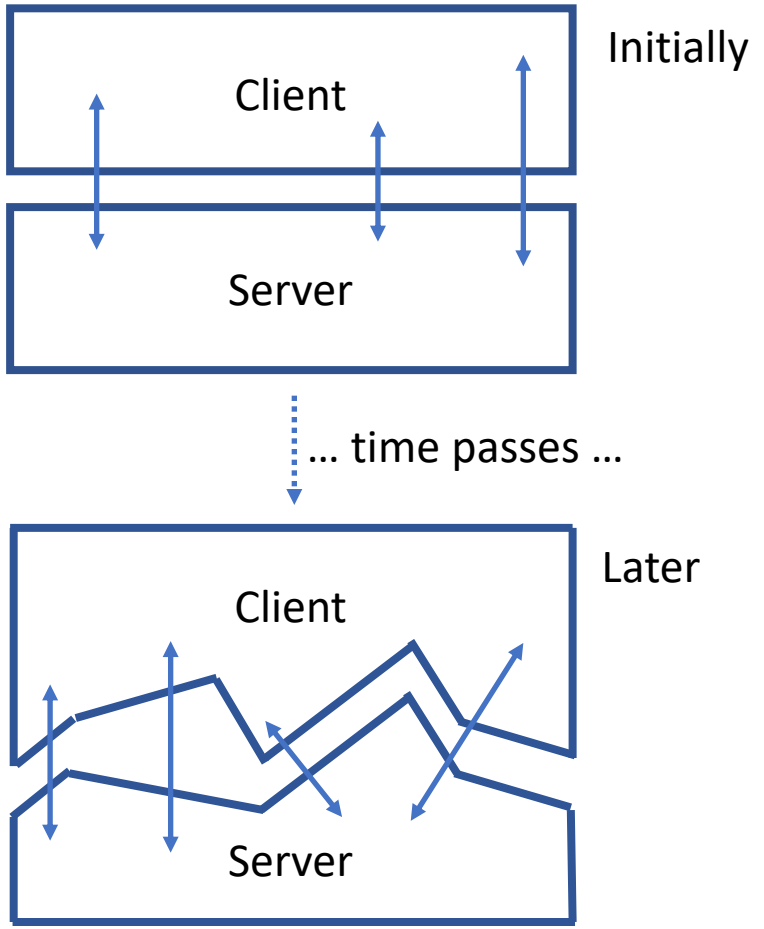
ecostore
safer for you and our world

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

1. Fetching complicated data structures requires multiple round trips between the client and server.
2. 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

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

(JavaScript is weakly typed too…)

An example of the confusion

`auctionStarttime integer`

Why integer and not Date?

Mapping from integer to date and time?

`POST /auctions` API, is `startingBid` the same as the `auction_startingprice` in the `auction` 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)

A *very* 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 code and errors

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.

GraphQL vs REST response code and errors

GraphQL

- All GraphQL queries return 200 response code, even errors.

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

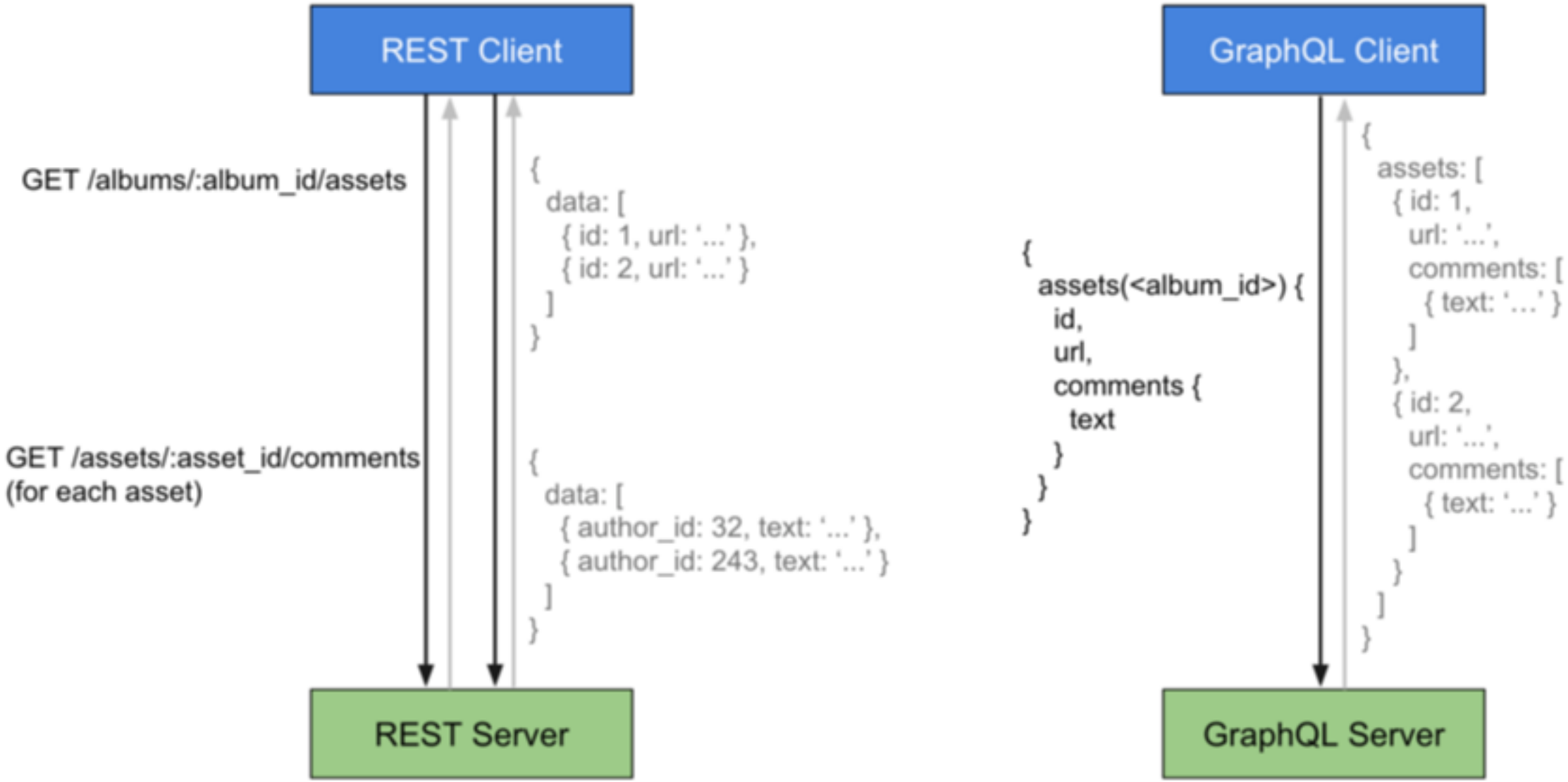
REST

- HTTP response code indicates success / error

== 400 BAD REQUEST

GraphQL

- Does not require you to think in terms of graphs
  - (Though relational tables are a type of graph…)
  - 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



https://medium.com/chute-engineering/graphql-in-the-age-of-rest-apis-b10f2bf09bba

GraphQL uses GETs and POSTs

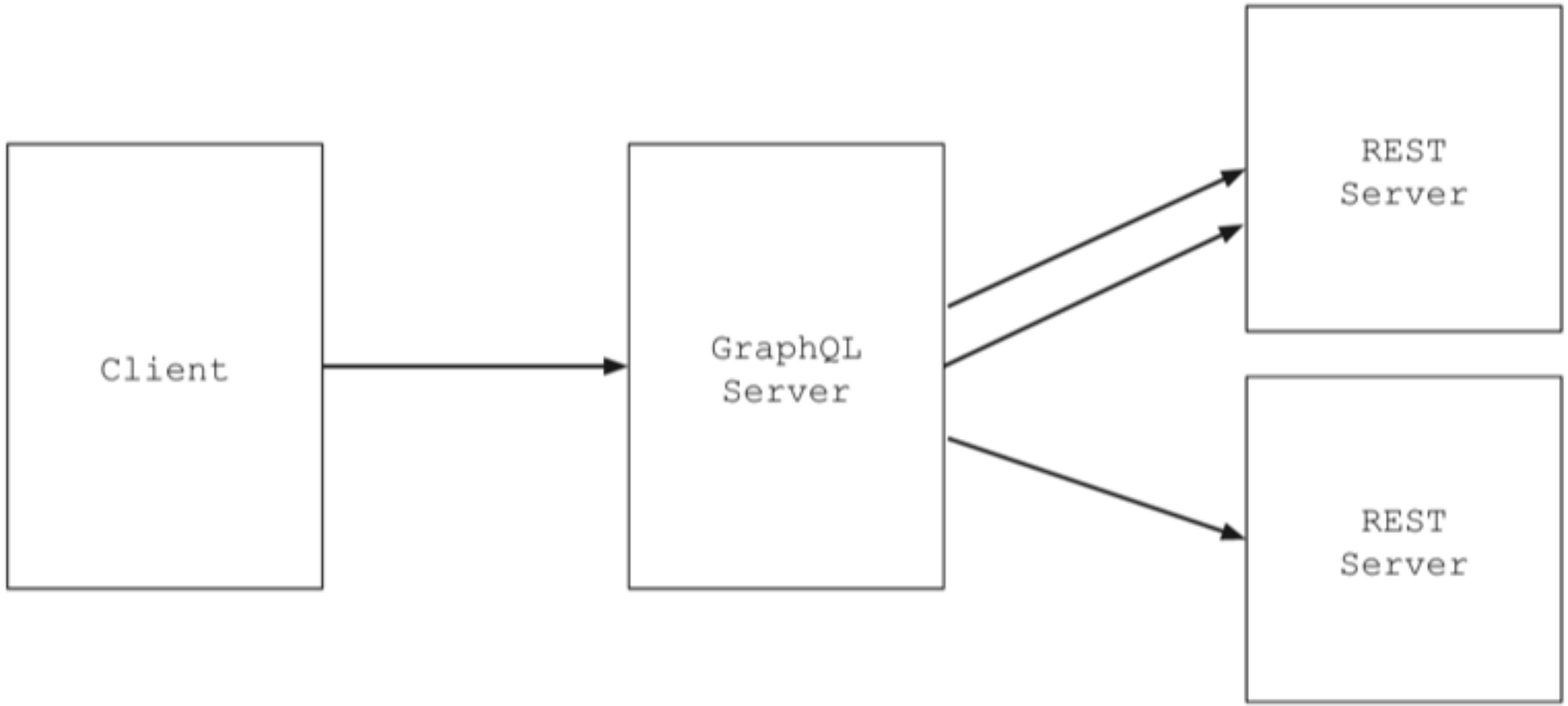
For GETs

- `http://myapi/graphql?query={me{name}}`
- GraphQL query is specified using the URL query parameters (JSON templating…)

For POSTs:

- `http://myapi/query`
- Specify the query in the HTTP body, using JSON, e.g.

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



https://medium.com/chute-engineering/graphql-in-the-age-of-rest-apis-b10f2bf09bba

https://www.npmjs.com/package/express-graphql

```
const express = require('express');
const graphqlHTTP = require('express-graphql');

const app = express();

app.use('/graphql', graphqlHTTP({
  schema: MyGraphQLSchema,
  graphiql: true
}));

app.listen(4000);
```

Further resources

GraphQL Introduction

- https://graphql.org

Apollo GraphQL Server

- https://www.apollographql.com/docs/apollo-server/

From REST to GraphQL

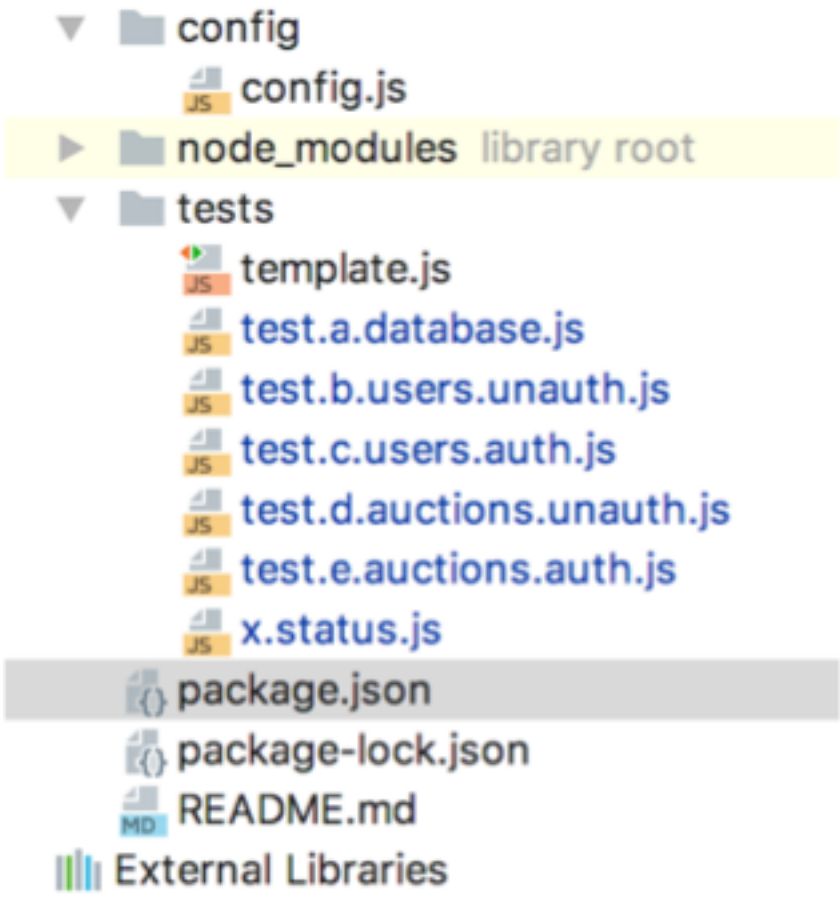
- https://0x2a.sh/from-rest-to-graphql-b4e95e94c26b

GraphQL in the age of REST APIs

- https://medium.com/chute-engineering/graphql-in-the-age-of-rest-apis-b10f2bf09bba

Automated software testing

- 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
- You can setup pre- and post-conditions
  - `before(), beforeeach(), after()` etc



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)

Asychronous behavior when testing

- 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-school 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 be 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 *un*successful *as expected* – the user is not logged in
- Need to think carefully about:
  - `.then(), catch(), done(), done(err),` and/or `throw err;`

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

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

SENG365 Web Computing Architecture: #5 GraphQL and automated API testing

Ben Adams
Course Coordinator & Lecturer
benjamin.adams@canterbury.ac.nz
310, Erskine Building