cscc01 (summer 2021)
introduction to software engineering

tutorial 2

RESTful APIs

tutorial outline

01 http requests

02 apis

03 restful

04 testing

info

- **team.md** is due may 21st 11:59 pm

after teams are finalized, we will release a google form for project demo sign ups

- you must register on github and jira by may 21st 11:59 pm

- sprint 0 deliverables are also due may 28th 11:59 pm

please read the handout for an overview

they aren’t difficult, but require thought as your design + software engineering practices will have a great impact on your project success

http requests

http (hypertext transfer protocol) requests are the basis of communication on the Internet!

when you type in the URL (uniform resource locator) of a website, your web browser sends a GET request to the server to provide the required files for displaying the site.

try opening the browser console, clicking the network tab, and loading in a page. You’ll be able to see all the network requests made!

http methods

there are five types of requests we can issue, which cover the basis of CRUDing (create, read, update, delete) data.

Method	Description	Arguments
GET	Used to **request** data from a source	Query params
POST	**Create/update** a resource with provided data	Request body
PATCH	**Update** a resource, providing the changes	Request body
PUT	**Update** a resource, providing the entire modified resource	Request body
DELETE	**Delete** a resource	Query params

request example

take for example, this endpoint from the github api: **GET /repos/{owner}/{repo}**

GET `https://api.github.com/repos/octocat/hello-world`

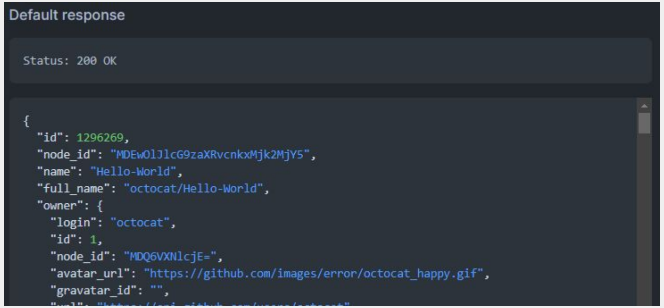
Q. what do we expect the response to be?

request example

take for example, this endpoint from the github api: **GET /repos/{owner}/{repo}**

GET https://api.github.com/repos/octocat/hello-world

Q. what do we expect the response to be?
A. information about the hello-world repository, under the octocat owner



Default response

```
Status: 200 OK
```

```
{
  "id": 1296269,
  "node_id": "MDEwOlJlcG9zaXRvcnkxMjk2MjY5",
  "name": "Hello-World",
  "full_name": "octocat/Hello-World",
  "owner": {
    "login": "octocat",
    "id": 1,
    "node_id": "MDQ6VXNlcjE=",
    "avatar_url": "https://github.com/images/error/octocat_happy.gif",
    "gravatar_id": "",
    "url": "https://api.github.com/users/octocat"
```

http statuses

after issuing an http request, we expect to receive a status code and response body (typically JSON). http statuses describe what the server did in response to the request.

200 OK: The response has succeeded!

201 Created: The request has succeeded, and the resource has been created (usually for POST)

400 Bad Request: The server could not understand the request due to invalid syntax

401 Unauthorized: The client is not allowed to get the requested response

404 Not Found: The server cannot find the requested resource

500 Internal Server Error: The server has encountered an issue while processing your request

apis

apis (application programming interface) are servers that define interactions between clients and databases. when people say “backend programming”, they’re referring to developing apis!

clients issue requests to apis, which are processed and evaluated according to the api’s business logic - including (but not limited to): querying/modifying/deleting data, authorization, authentication, and more

endpoints

apis consist of **endpoints**, which are defined methods to handle http requests. after the api receives a request, it performs request handling, then routes it to the proper endpoint. Endpoints are defined by a verb and a url.

POST /books/

GET /users/:uid

Q. What would the endpoint url be for deleting a book?

endpoints

apis consist of **endpoints**, which are defined methods to handle http requests. after the api receives a request, it performs request handling, then routes it to the proper endpoint. Endpoints are defined by a verb and a url.

POST /books/

GET /users/:uid

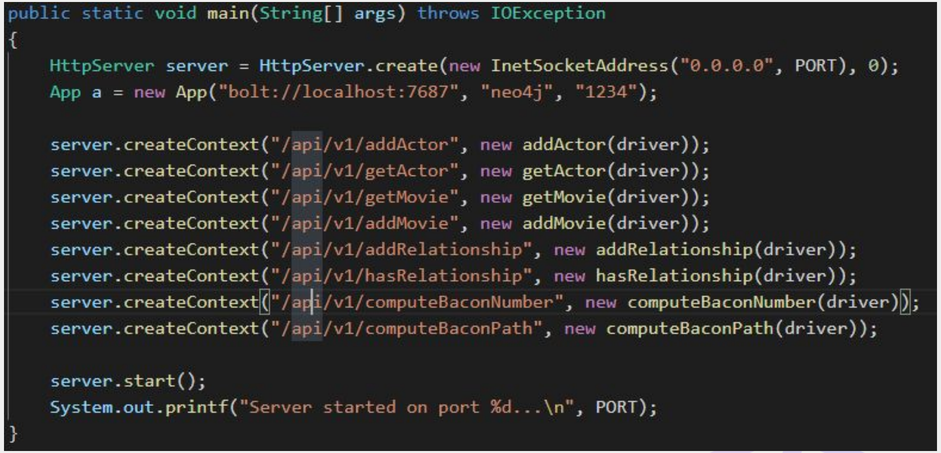
Q. What would the endpoint url be for deleting a book?

DELETE /books/:uid

case study: java apis

Let's take a look at a Java backend API to see how it's implemented.

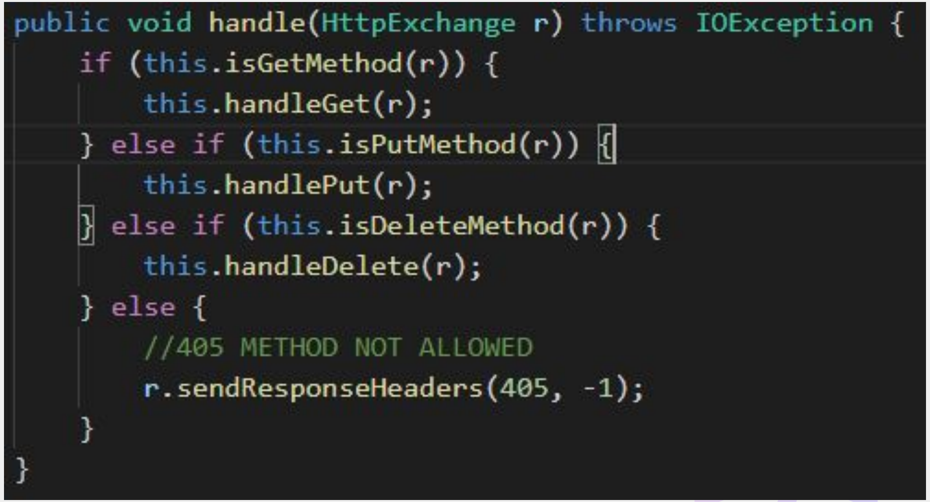
In App.java's main method, we connect to the database driver and define the paths.



```
public static void main(String[] args) throws IOException
{
    HttpServer server = HttpServer.create(new InetSocketAddress("0.0.0.0", PORT), 0);
    App a = new App("bolt://localhost:7687", "neo4j", "1234");

    server.createContext("/api/v1/addActor", new addActor(driver));
    server.createContext("/api/v1/getActor", new getActor(driver));
    server.createContext("/api/v1/getMovie", new getMovie(driver));
    server.createContext("/api/v1/addMovie", new addMovie(driver));
    server.createContext("/api/v1/addRelationship", new addRelationship(driver));
    server.createContext("/api/v1/hasRelationship", new hasRelationship(driver));
    server.createContext("/api/v1/computeBaconNumber", new computeBaconNumber(driver));
    server.createContext("/api/v1/computeBaconPath", new computeBaconPath(driver));

    server.start();
    System.out.printf("Server started on port %d...\n", PORT);
}
```

case study: java apis

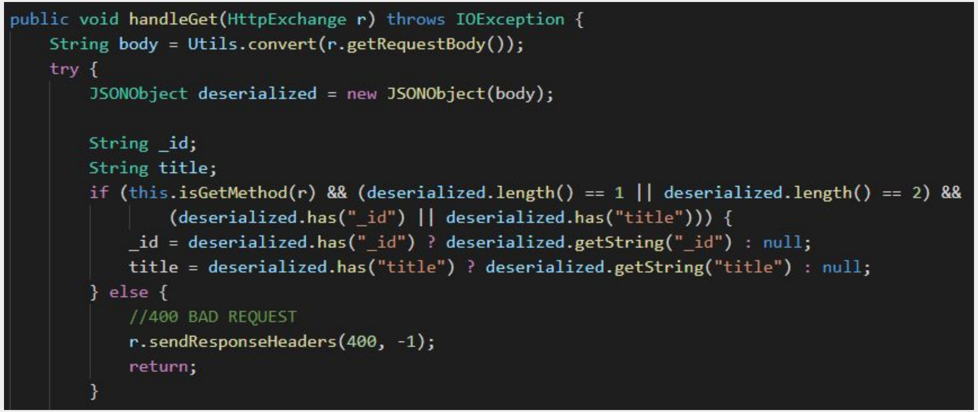
endpoint objects inherit a handle method, allowing them to route requests to their respective handlers



```java
public void handle(HttpExchange r) throws IOException {
    if (this.isGetMethod(r)) {
        this.handleGet(r);
    } else if (this.isPutMethod(r)) {
        this.handlePut(r);
    } else if (this.isDeleteMethod(r)) {
        this.handleDelete(r);
    } else {
        //405 METHOD NOT ALLOWED
        r.sendResponseHeaders(405, -1);
    }
}
```

case study: java apis

within the handle methods, we perform input validation, access the database, and return the appropriate response.



```
public void handleGet(HttpExchange r) throws IOException {
    String body = Utils.convert(r.getRequestBody());
    try {
        JSONObject deserialized = new JSONObject(body);

        String _id;
        String title;
        if (this.isGetMethod(r) && (deserialized.length() == 1 || deserialized.length() == 2) &&
                (deserialized.has("_id") || deserialized.has("title"))) {
            _id = deserialized.has("_id") ? deserialized.getString("_id") : null;
            title = deserialized.has("title") ? deserialized.getString("title") : null;
        } else {
            //400 BAD REQUEST
            r.sendResponseHeaders(400, -1);
            return;
        }
```

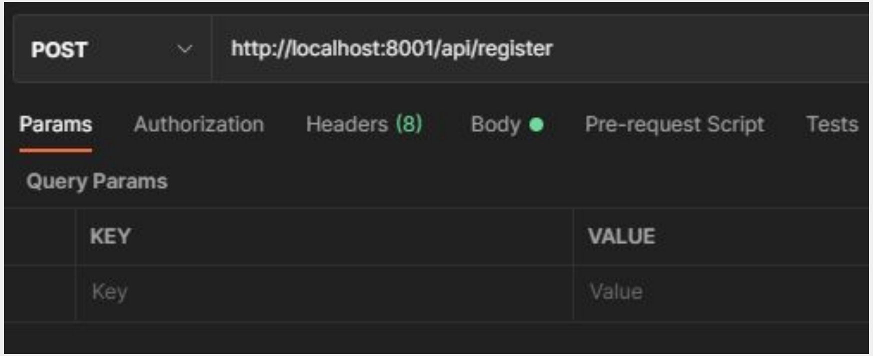
{ api } restful services

REST (REpresentational State Transfer) is a design philosophy widely employed and suited for the web.

- **stateless:** the client and server do not need to know anything about each other
- **separation of client and server:** as long the format of the messages is known, they can remain modular and separate
- **resource identification through URI:** resources are accessed via paths that are self-explanatory i.e. **/repos/{owner}/{repo}** from the previous example

manual testing with postman

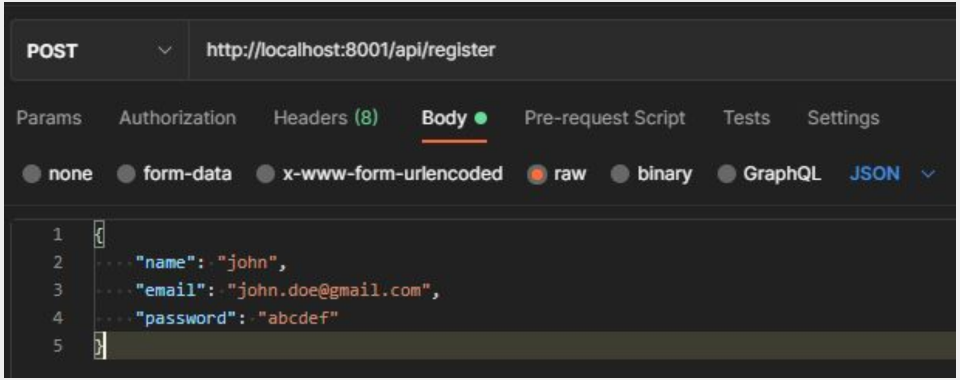
postman is an API client enabling you to send HTTP requests to your API. this is very useful during development as it allows you to sanity check your routing, input sanitization, and more. however, like all manual testing it should not be depended on over automated testing.



POST http://localhost:8001/api/register

Query Params

KEY	VALUE
Key	Value



POST http://localhost:8001/api/register

○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary ○ GraphQL JSON

```
{
    "name": "john",
    "email": "john.doe@gmail.com",
    "password": "abcdef"
}
```

automated testing

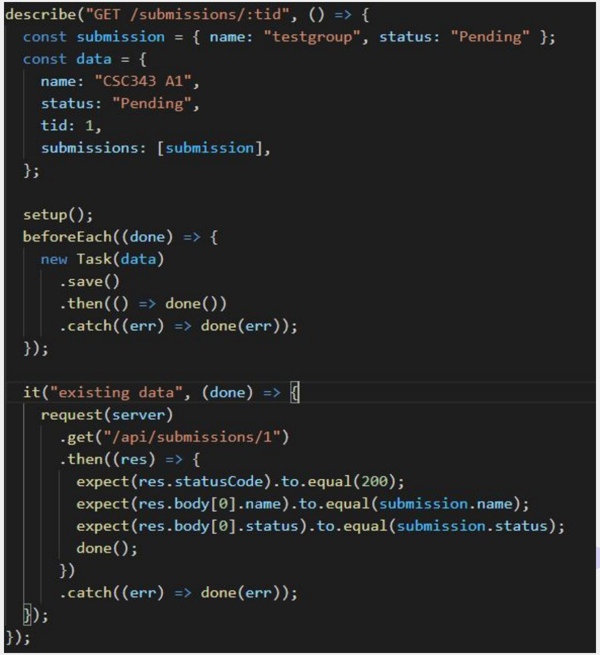
for automated testing in general, it is recommended to have tests that focus on a single responsibility rather than testing multiple behaviours within one test case. automated backend testing usually follows a typical flow:

- set up required mock data
- issue a request with that data
- verify the following:
  - status code
  - response
  - check database to verify changes have been made

we'll take a look at this for java, javascript, and django backend apis.

case study: javascript

for node.js/express backends, we can use chai to perform automated testing against our backend API.

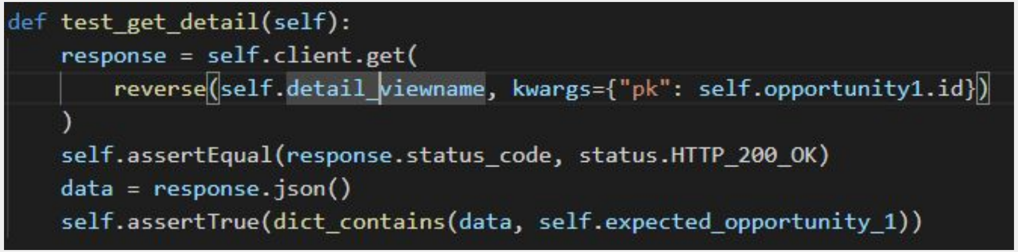


```
describe("GET /submissions/:tid", () => {
  const submission = { name: "testgroup", status: "Pending" };
  const data = {
    name: "CSC343 A1",
    status: "Pending",
    tid: 1,
    submissions: [submission],
  };

  setup();
  beforeEach((done) => {
    new Task(data)
      .save()
      .then(() => done())
      .catch((err) => done(err));
  });

  it("existing data", (done) => {
    request(server)
      .get("/api/submissions/1")
      .then((res) => {
        expect(res.statusCode).to.equal(200);
        expect(res.body[0].name).to.equal(submission.name);
        expect(res.body[0].status).to.equal(submission.status);
        done();
      })
      .catch((err) => done(err));
  });
});
```

case study: django

for django backend development, we highly recommend django-rest-framework.

using it, we create TestCase classes that inherit from rest_framework’s APITestCase that provides useful tooling as pictured.



```
def test_get_detail(self):
    response = self.client.get(
        reverse(self.detail_viewname, kwargs={"pk": self.opportunity1.id})
    )
    self.assertEqual(response.status_code, status.HTTP_200_OK)
    data = response.json()
    self.assertTrue(dict_contains(data, self.expected_opportunity_1))
```

api design tips

- **make sure your route names are descriptive and concise**
    - GET /getAllCars, POST /addCar -> GET /cars, POST /cars
- **consider future external usage:**
    - while your api starts off being used internally for a specific process, think about how external services may want to consume e.g. jira + github integration
- **version your apis**
    - changes are bound to happen as your api evolves, so to support customers on the old version you'll need to deploy and maintain multiple versions

api design tips

- **setup the infrastructure early so you can reap the rewards later**

It's tempting to put off features like ci/cd, security, and logging - but the longer you prolong them the more difficult it is to integrate into your codebase.

- **log your errors**

print statements are great and all, but what happens when the server is deployed and it crashes? to trace the issue, we'll need to view the logs

- **analytics**

usage metrics provide crucial insight into how your product is being used

api design tips

THINK ABOUT SCALABILITY!

while your api may initially only have a limited amount of data, it is important to think about what happens as your user base grows.

an endpoint **GET /posts** that returns all posts in a database may function fine for 100 posts, but as that number grows you don’t want to retrieve everything in the database.

one solution: add pagination e.g. **GET /posts?page=1** which returns the first 100 posts.