Web Application Programming Interface (API)

Software Architecture

February 21, 2024

Brae Webb & Evan Hughes

Aside

Github Classroom links for this practical can be found on Edstem https://edstem.org/au/courses/15375/discussion/1753712

1 This Week

This week our goal is to:

- initialise our GitHub repositories where we will be working on practical exercises; and
- build a minimal HTTP API of a todo app using the Flask framework.

2 Practicals

These practicals are designed to prepare you with the technical skills required for the scalability and capstone projects. We will normally spend the first section of the practicals gaining the relevant conceptual background for the practical. The second section will be a practical exercise where you will need to write and run some code.

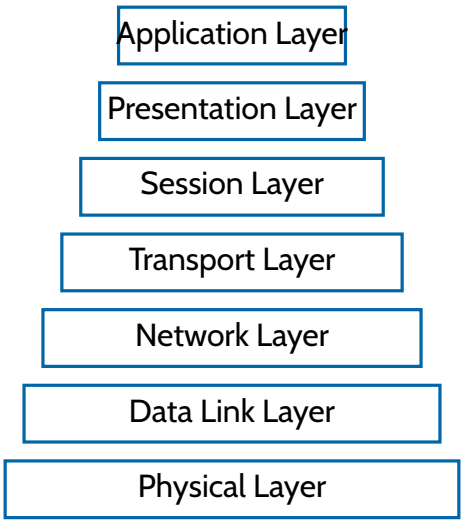
This semester we will be working on the creation of a scalable and fault-tolerant *todo list* application. You should aim to keep up with the practicals or you will not be able to complete the projects.

Info

In this practical we will build a RESTful API that communicates over HTTP. It is worth noting that while this is a common way to build APIs, it is far from the only way. We will briefly explore some alternatives in subsequent weeks.

3 Concepts

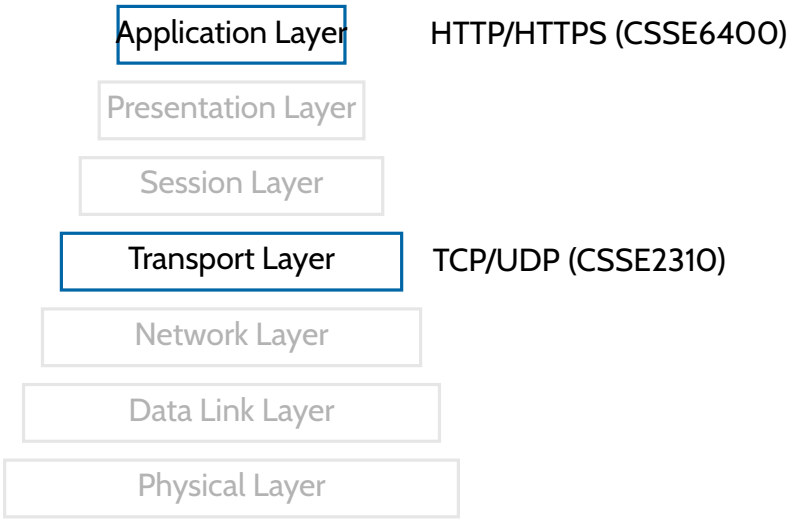
3.1 Networking



The above diagram shows the layers of the *OSI* model. These are the layers of abstraction that comprise the Internet Protocol Suite (or, how computers communicate over the Internet).

At the transport layer, we have the *Transmission Control Protocol* (TCP) and *User Datagram Protocol* (UDP). This is a low-level protocol that you may have already used in your previous studies, this is the level of networking taught in *CSSE2310*. While it is possible to develop applications that use this protocol directly, it is not very practical.

In this course we will be using the *Hypertext Transfer Protocol* (HTTP). HTTP is a higher-level protocol that is built on top of TCP, it sits in the *Application Layer* of the OSI model. HTTP is the protocol that is used to transfer web pages over the Internet.



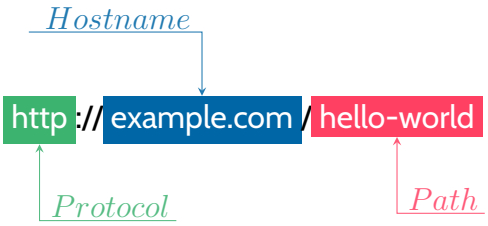
3.2 URLs

A *Uniform Resource Locator* (URL) is a string that identifies a resource on the Internet.
There are three main components of a URL:

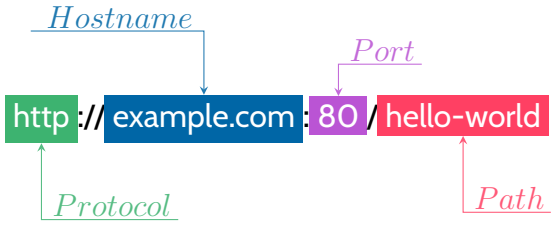
Protocol The protocol used to access the resource, e.g. *http* or *https*.

Host The host name of the server that hosts the resource, e.g. *example.com* or *localhost*.

Path The path to the resource on the server, how the server identifies the resource.

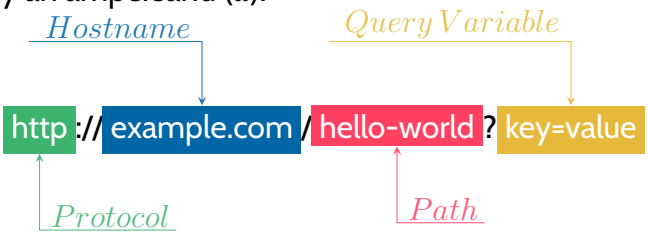


A URL can also contain a *port* number, which is the port number that the server is listening on. If the port number is not specified, the default port number for the protocol is used. For example, the default port number for *http* is 80, and the default port number for *https* is 443.



The URL *http://example.com/hello-world* is equivalent to *http://example.com:80/hello-world*.

URLs can also contain *query parameters*, which are key-value pairs that are used to pass information to the server. Query parameters are separated from the path by a question mark (?). Each query parameter is separated from the next by an ampersand (&).



3.3 HTTP

HTTP is a request-response abstraction for networking.

3.3.1 Request

The HTTP request is a message sent to the server. It contains the following information:

URL An endpoint to which the request is sent.

Method Described later.

Headers Specify type of data, e.g. JSON, HTML, etc. and other metadata about the request.

Body The optional data to send to the server.

3.3.2 Response

The HTTP response is a message sent from the server. It contains the following information:

Status code A number between 100 and 599 giving details about the response.

Headers Specify type of response data, e.g. JSON, HTML, etc. and other metadata about the response.

Body Content of the response.

Status Codes

200s Indicate the request was successful, 200 is the most common.

300s Redirects the requester to another location.

400s Indicates that the request was wrong, e.g. 404 meaning that the request was for something that does not exist.

500s Indicates that the server had a problem fulfilling the request.

Methods

GET Queries the server for information.

POST Creates a new resource on the server.

PUT Updates an existing resource on the server.

DELETE Deletes an existing resource on the server.

3.4 JSON

JavaScript Object Notation (JSON) is a data format commonly used to pass data to an API. It is fairly succinct and communicates the important points to a human reader better than some alternative formats. The popularity of JSON is largely due to its compatibility with JavaScript which has taken over as the defacto web development language. JSON is the map-esque data type in JavaScript. Detractors of JSON claim that its main disadvantage compared to XML (an alternative data format) is that it lacks a schema. However, schemas are possible in JSON[1], they are optional, just as in XML, but are used much less than in XML.

```
» cat csse6400.json
{
    "Course Code": "CSSE6400",
    "Course Title": "Software Architecture"
}
```

[1] https://json-schema.org/

3.5 REST

REST is an architectural style guided by a set of architectural constraints that allows us to build flexible APIs. In this course we do not dive too deep into the architectural style and instead opt for a more surface level understanding. It is a common mistake for people to refer to REST as a HTTP based web service API, they are different. In this course we chose to embrace this mistake and often refer to a HTTP based web service API when saying REST.
An example of this type of API might be:

GET /api/v1/todo List all tasks todo

POST /api/v1/todo Create a task todo

GET /api/v1/todo/id List all details about a certain task

PUT /api/v1/todo/id Update the fields of an existing task

DELETE /api/v1/todo/id Delete a specific task

Note that the API specification does not include details of the port or hostname, as these may change frequently.

4 GitHub

We will use GitHub to host our practical work. This is strongly encouraged as it will help you to get experience with the assessment submission process. Additionally, committing your work is a good habit to get into and will be useful for your future career.

4.1 Creating a GitHub Account

If you do not already have a GitHub account, you will need to create one. You can do this by visiting https://github.com/join.

4.2 Joining the Course Organisation

Once you have created an account, you will need to join the course organization. If you have not yet filled out the Google Form, you will need to do so before you can join the organization. The link to the Google Form can be found on Blackboard.

Once you have filled out the form, tell your tutor your GitHub username and they will add you to the organisation.

4.3 Joining the GitHub Classroom

Once you have joined the organization, you will need to join the GitHub Classroom. Follow the link provided by your tutor to join the classroom.

4.4 Creating a Practical Repository

Navigate to the GitHub Classroom link provided by your tutor. You should see a list of practicals, click on the week one practical. This will create a new repository for you in the course organisation. You can now clone this repository to your local machine or work directly in the browser with GitHub Codespaces.

5 TODO App

5.1 The API design

5.1.1 GET /api/v1/health

This endpoint should return a 200 status code and a JSON object with a single field, **status**, which should be set to **ok**.

```
GET /api/v1/health HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "status": "ok"
}
```

5.1.2 GET /api/v1/todos

This endpoint should return a list of all the tasks in the todo list.
Optional query parameters:

- **completed** A boolean value indicating whether to return completed tasks or not. Valid values are `true` or `false`.
- **window** An integer value indicating how many days past today's date a task should be due by.

```
GET /api/v1/todos?completed=true&window=7 HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
      "id": 1,
      "title": "Watch CSSE6400 Lecture",
      "description": "Watch the CSSE6400 lecture on ECHO360 for week 1",
      "completed": true,
      "deadline_at": "2023-02-27T00:00:00",
      "created_at": "2023-02-20T00:00:00",
      "updated_at": "2023-02-20T00:00:00"
    },
    ...
]
```

5.1.3 GET /api/v1/todos/{id}

This endpoint should return a single item from the todo list.

```
GET /api/v1/todos/1 HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": 1,
    "title": "Watch CSSE6400 Lecture",
    "description": "Watch the CSSE6400 lecture on ECHO360 for week 1",
    "completed": false,
    "deadline_at": "2023-02-27T00:00:00",
    "created_at": "2023-02-20T00:00:00",
    "updated_at": "2023-02-20T00:00:00"
}
```

5.1.4 POST /api/v1/todos

This endpoint should create a new task in the todo list. The title field must be included in the request and all other values are optional. The `created_at`, `updated_at` cannot be set by this method.

Attempting to post any other fields than `title`, `description`, `completed`, `deadline_at` will cause a 400 error to be returned.

```
POST /api/v1/todos HTTP/1.1
Content-Type: application/json

{
    "title": "Watch CSSE6400 Lecture",
    "description": "Watch the CSSE6400 lecture on ECHO360 for week 1",
    "completed": false,
    "deadline_at": "2023-02-27T00:00:00",
}
```

```
HTTP/1.1 201 Created
Content-Type: application/json

{
    "id": 1,
    "title": "Watch CSSE6400 Lecture",
    "description": "Watch the CSSE6400 lecture on ECHO360 for week 1",
    "completed": false,
    "deadline_at": "2023-02-27T00:00:00",
    "created_at": "2023-02-20T00:00:00",
```

```
    "updated_at": "2023-02-20T00:00:00"
}
```

5.1.5 PUT /api/v1/todos/{id}

This endpoint should update a task in the todo list. The `created_at`, `updated_at` cannot be set by this method.

Attempting to put any other fields than `title, description, completed, deadline_at` will cause a 400 error to be returned.

Attempting to put a task id that does not exist will cause a 404 error to be returned.

```
PUT /api/v1/todos/1 HTTP/1.1
Content-Type: application/json

{
    "title": "Join the Richard Thomas fan club",
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": 1,
    "title": "Join the Richard Thomas fan club",
    "description": "Watch the CSSE6400 lecture on ECHO360 for week 1",
    "completed": false,
    "deadline_at": "2023-02-27T00:00:00",
    "created_at": "2023-02-20T00:00:00",
    "updated_at": "2023-02-20T00:00:00"
}
```

5.1.6 DELETE /api/v1/todos/{id}

This endpoint should delete a task from the todo list. If the task does not exist, a 200 is returned with an empty response.

```
DELETE /api/v1/todos/1 HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": 1,
    "title": "Join the Richard Thomas fan club",
```

```
    "description": "Watch the CSSE6400 lecture on ECHO360 for week 1",
    "completed": false,
    "deadline_at": "2023-02-27T00:00:00",
    "created_at": "2023-02-20T00:00:00",
    "updated_at": "2023-02-20T00:00:00"
}
```

5.2 Implementation with Flask

5.2.1 Setting up your environment

The practicals will be using Python as our implementation language but the course is not restricted to Python. Feel free to implement the practicals and your assignments in any language you feel comfortable with but be aware that help with other languages or frameworks may be limited.

> **Warning**
>
> If you are on Windows, we recommend using a Unix based environment, such as WSL2 or a virtual machine running Ubuntu.

Since Python 3.12 now requires virtual environments to be used we will start by installing pipx to help manage the python environment, followed by poetry to manage our application dependencies.

For Mac and Linux users you can install pipx via your package manager:

```
>> brew install pipx
>> pipx ensurepath
```

or

```
>> sudo apt update // apt or dnf depending on distro
>> sudo apt install pipx // apt or dnf depending on distro
>> pipx ensurepath
```

Once pipx is installed we can install poetry safely without affecting our system python environment.

```
>> pipx install poetry
```

Navigate to your cloned practical repository in a terminal and start your project by creating the Python environment using the following:

```
>> poetry init
```

This will give you a few prompts which for our purposes you can just stick with the default values. After this process is completed you will have a `pyproject.toml` file in your repository. The `pyproject.toml` is where we will specify the libraries we need for our project.

```
 » cat pyproject.toml
[tool.poetry]
name = "practical01"
version = "0.1.0"
description = ""
authors = ["Evan Hughes <uqehugh3@uq.edu.au>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.8"


[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Info

Your Python version may be different, this is fine as long as it is Python 3. For our automated tests we would like you to change the version number from the `pyproject` so that it is `python = "3̂.8"`. If you are running a version of python lower than this then please consider upgrading as it is end-of-life.

Next we are going to add Flask as a dependency to our project. This library is a small web server wrapper that will allow us to quickly build our todo application. To add a dependency to our project, we can run the following command:

```
>> poetry add flask
```

You will see it has made some changes to our `pyproject` and a `poetry.lock` is created.

5.2.2 Initialising with Flask

Create a folder called `todo` in the root of your project and create a file called `__init__.py` and add the following code to it:

```
from flask import Flask

def create_app():
    app = Flask(__name__)
    return app
```

Your repository should now look like this:

```
.
|-- README.md
|-- pyproject.toml
```

```
|-- poetry.lock
|-- todo
    |-- __init__.py
```

We have created a basic Flask app but how do we run it? When using poetry we need to run it in the following way:

```
>> poetry run flask --app todo run
```

This command runs the `flask -app todo run` command inside the poetry environment with our dependencies installed.

This web server is a bit boring though, let's add an endpoint so we can see that it works. In the todo folder, create a folder called views and a file `routes.py`. Add the following code to `routes.py`:

```
from flask import Blueprint

api = Blueprint('api', __name__, url_prefix='/api/v1')

@api.route('/health')
def health():
    return "ok"
```

The above has made an endpoint within our API under the `/api/v1` prefix and we have created a `/health` route below this. Now we need to register this with our flask app. In the `__init__.py` file, change the contents to the following:

```
from flask import Flask

def create_app():
    app = Flask(__name__)

    from .views.routes import api
    app.register_blueprint(api)

    return app
```

Now when we run the app we should see the following:

```
>> poetry run flask --app todo run -p 6400

* Serving Flask app 'todo'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
* Running on http://127.0.0.1:6400
Press CTRL+C to quit
```

Note that the `-p 6400` flag is used to specify the port we want to run the server on, this allows us to run multiple servers at the same time.

Open your browser and go to http://localhost:6400/api/v1/health and you should see a blank page with "ok" written on it.

If you run into any issues, make sure your files are in the following structure and that you are running these commands in the root folder.

```
.
|-- README.md
|-- pyproject.toml
|-- poetry.lock
|-- todo
    |-- __init__.py
    |-- views
        |-- routes.py
```

5.2.3 Returning JSON with Flask

We have a web server running now but we are communicating with text instead of a structured format. JSON is a common format to communicate data between services and is human readable. To start using JSON we are going to make a small change to our health endpoint. In the `routes.py` file, add a new import for jsonify and change the health endpoint to the following:

```
1  from flask import Blueprint, jsonify
```

```
6  @api.route('/health')
7  def health():
8      return jsonify({"status": "ok"})
```

Now let's go back to our browser and refresh, we should see the following:

```
{
  "status": "ok"
}
```

> **Info**
>
> If you are using Firefox the JSON will be parsed and presented in a structured form. To get this for Chrome you can install extensions from the Chrome web store.

5.2.4 Calling your API locally

We have many choices when it comes to calling our API locally. We could use curl, Postman, VS Code, or our browser. We are going to focus on using curl and the REST Visual Studio Code extension.

Info

For GET requests we can use our browser to call the API but for POST, PUT, and DELETE requests we will need to use a tool like curl or Postman.

cURL

Install curl if it is available for your operating system. If you are on a Mac, you can install it with homebrew otherwise it is available for most Linux distributions.

Now that we have the tool installed let's have our API running in a terminal window and open up a new terminal so we can make requests to our API. Enter in the following into your terminal to call your API.

```
$ curl -X GET http://localhost:6400/api/v1/health
```

You should see the following response:

```
{
  "status": "ok"
}
```

VS Code

If you are using Visual Studio Code for your text editor you can install the "Rest Client" by Huachao Mao. This extension allows you to have files with requests that you can then run from within the editor. The benefit of this method is that we can also check this into our repository.

Find the extension in the VS Code marketplace and install it. Once installed, create a new file in the root of your project called `endpoints.http` and add the following to it:

```
@baseUrl = http://localhost:6400

### Health
GET {{baseUrl}}/api/v1/health
```

A subtle "Send Request" should be visible between the comment and the GET. Click on this and a new tab should open up with the response from the API. The response should be similar to below:

```
HTTP/1.1 200 OK
Server: Werkzeug/2.2.3 Python/3.10.9
Date: Sun, 19 Feb 2023 04:13:00 GMT
Content-Type: application/json
Content-Length: 16
Connection: close

{
  "status": "ok"
}
```

We can also see in the terminal that our webserver has logged our request by the output of:

```
127.0.0.1 - - [19/Feb/2023 14:13:00] "GET /api/health HTTP/1.1" 200 -
```

5.2.5 Creating more endpoints

Let's expand our endpoints.http file to include the other endpoints that we need to create for our todo application. Expand the file to include the following GET, POST, PUT, and DELETE endpoints:

```
@baseUrl = http://localhost:6400

### Health
GET {{baseUrl}}/api/v1/health

### List All Todos
GET {{baseUrl}}/api/v1/todos

### Get a specific Todo
GET {{baseUrl}}/api/v1/todos/1

### Create a Todo
POST {{baseUrl}}/api/v1/todos
Content-Type: application/json

{
    "title": "An example Todo",
    "description": "This is an example todo",
}

### Update a Todo
PUT {{baseUrl}}/api/v1/todos/1
Content-Type: application/json

{
    "title": "updated title",
}

### Delete a Todo
DELETE {{baseUrl}}/api/v1/todos/1
```

Let's run the GET request and see what happens. We should see the following:

```
HTTP/1.1 404 NOT FOUND
Server: Werkzeug/2.2.3 Python/3.10.9
Date: Sun, 19 Feb 2023 04:27:42 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 207
Connection: close
```

```
<!doctype html>
<html lang=en>
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server. If you entered the URL manually
   please check your spelling and try again.</p>
```

Now we can start to create the endpoints that we need for our todo application. In the `routes.py` file, add the following code to the bottom of the file:

```
@api.route('/todos', methods=['GET'])
def get_todos():
    return jsonify([])
```

Now the server should reload, if it does not, you can manually reload it by stopping the process and restarting it. Now if we run the GET request again we should see the following:

```
HTTP/1.1 200 OK
Server: Werkzeug/2.2.3 Python/3.10.9
Date: Sun, 19 Feb 2023 04:38:44 GMT
Content-Type: application/json
Content-Length: 3
Connection: close

[]
```

This endpoint is for listing all the todos that the user has. For now we are going to return a hard coded todo item so we can get used to having the API return data. In the `routes.py` file, modify the `get_todos` function to return a hard coded todo item:

```
@api.route('/todos', methods=['GET'])
def get_todos():
    return jsonify([{
      "id": 1,
      "title": "Watch CSSE6400 Lecture",
      "description": "Watch the CSSE6400 lecture on ECHO360 for week 1",
      "completed": True,
      "deadline_at": "2023-02-27T00:00:00",
      "created_at": "2023-02-20T00:00:00",
      "updated_at": "2023-02-20T00:00:00"
    }])
```

Now let's run the GET request again and we should see the following:

```
HTTP/1.1 200 OK
Server: Werkzeug/2.2.3 Python/3.10.9
Date: Sun, 19 Feb 2023 04:44:00 GMT
Content-Type: application/json
Content-Length: 200
Connection: close

[
  {
    "id": 1,
    "title": "Watch CSSE6400 Lecture",
    "description": "Watch the CSSE6400 lecture on ECHO360 for week 1",
    "completed": true,
    "deadline_at": "2023-02-27T00:00:00",
    "created_at": "2023-02-20T00:00:00",
    "updated_at": "2023-02-20T00:00:00"
  }
]
```

Next is our endpoint to get an individual todo by its id. In the `routes.py` file, add the following code to the bottom of the file:

```python
@api.route('/todos/<int:id>', methods=['GET'])
def get_todo(id):
    return jsonify({
      "id": id,
      "title": "Watch CSSE6400 Lecture",
      "description": "Watch the CSSE6400 lecture on ECHO360 for week 1",
      "completed": True,
      "deadline_at": "2023-02-27T00:00:00",
      "created_at": "2023-02-20T00:00:00",
      "updated_at": "2023-02-20T00:00:00"
    })
```

You will notice in this function we have a single parameter which is the ID fetched from the URL. You can see this in the `<int:id>` part of the route annotation. This is a Flask feature that allows you to fetch parameters from the URL.

These were our read only methods, now let's get to the mutating methods. First is our endpoint to create a new todo. In the `routes.py` file, add the following code to the bottom of the file:

```python
@api.route('/todos', methods=['POST'])
def create_todo():
    return jsonify({
      "id": 1,
      "title": "Watch CSSE6400 Lecture",
      "description": "Watch the CSSE6400 lecture on ECHO360 for week 1",
      "completed": True,
```

```
      "deadline_at": "2023-02-27T00:00:00",
      "created_at": "2023-02-20T00:00:00",
      "updated_at": "2023-02-20T00:00:00"
    }), 201
```

You will notice that currently this function is the same as the GET request but in future weeks we will build out the functionality to actually create a todo. Next is our endpoint to update a todo. In the `routes.py` file, add the following code to the bottom of the file:

```
@api.route('/todos/<int:id>', methods=['PUT'])
def update_todo(id):
    return jsonify({
      "id": id,
      "title": "Watch CSSE6400 Lecture",
      "description": "Watch the CSSE6400 lecture on ECHO360 for week 1",
      "completed": True,
      "deadline_at": "2023-02-27T00:00:00",
      "created_at": "2023-02-20T00:00:00",
      "updated_at": "2023-02-20T00:00:00"
    })
```

Likewise with the POST request, this function is the same as the GET request but in future weeks we will build out the functionality to actually update a todo. Finally is our endpoint to delete a todo. In the `routes.py` file, add the following code to the bottom of the file:

```
@api.route('/todos/<int:id>', methods=['DELETE'])
def delete_todo(id):
    return jsonify({
      "id": id,
      "title": "Watch CSSE6400 Lecture",
      "description": "Watch the CSSE6400 lecture on ECHO360 for week 1",
      "completed": True,
      "deadline_at": "2023-02-27T00:00:00",
      "created_at": "2023-02-20T00:00:00",
      "updated_at": "2023-02-20T00:00:00"
    })
```

Likewise with the POST request, this function is the same as the GET request but in future weeks we will build out the functionality to actually delete a todo.

With the endpoints you may have noticed us defining the methods, this is because we want to define which HTTP methods are allowed for each endpoint. For example, we do not want a user to be able to delete a todo by sending a GET request, we want them to send a DELETE request.

This concludes this week's practical. Next week we will add storage to our API so that we can actually create, update, and delete todos. We will also add some tests to our API to ensure that it is working as expected and meets our specification. Remember to ask for assistance on the Ed Discussion board if you get stuck or have any questions.