Last Updated on 2023/02/19

Application Programming Interfaces (APIs)

Software Architecture

February 20, 2023

Brae Webb

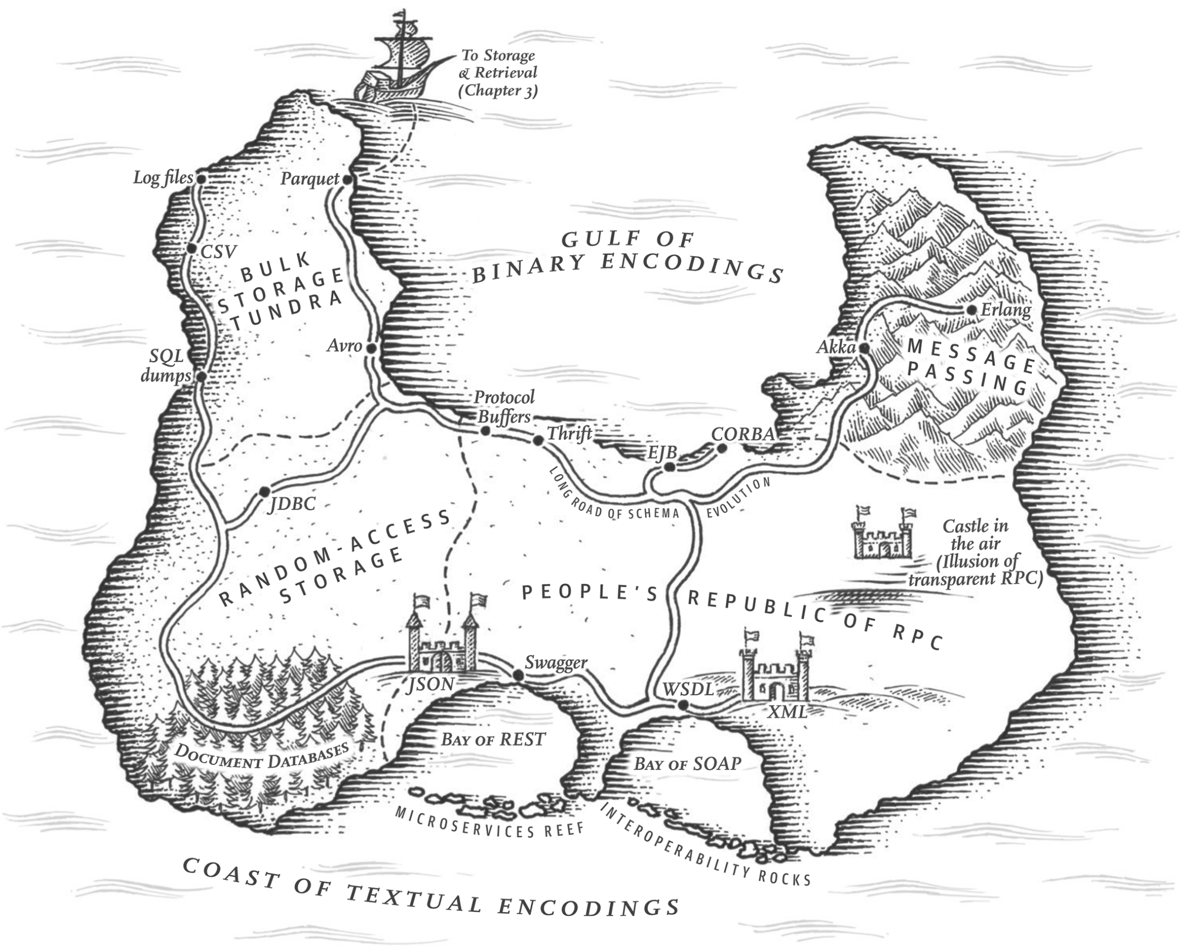

Figure 1: A map of communication techniques from Designing Data-Intensive Applications [1].

1 This Week

This week our goal is to:

- explore techniques used by developers use to communicate between components in distributed systems;
- initialize 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 cloud and team 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 application. You should aim to keep up with the practicals or you will not be able to complete the projects.

3 Communication

The world relies (heavily) on distributed systems, single machines are no longer sufficient to handle the demands of modern applications. There is no machine in the world powerful enough to process the requests Google receives every second[1]. But you don't need to be the size of Google to require distributed systems, even relatively small applications require multiple machines to handle the load. This inter-machine teamwork requires each machine to know how to talk to the others.

In this practical, we'll be exploring the different techniques used to communicate between components in distributed systems. We will then build a simple HTTP API using the Flask framework which allows other applications to interact with our application.

4 Data Formats

Communication requires an exchange of information. On single computer systems information is often stored at runtime in memory as primitive data which your programming language can interpret; bytes, integers, strings, etc. In object-oriented languages, primitive data is wrapped up into useful packages: objects. If we want this information to escape the confines of our programming language runtime, we need to package it up in a language-independent format. To be language-independent we need many languages to implement an encoding and decoding mechanism for the format. We have several language-independent formats available but a few defacto standards.

4.1 XML

Extensible Markup Language (XML) is one of the most widely used language-independent formats. The use cases of XML are extensive, it is the foundation for many popular utilities[2], such as SVG file formats, SAML authentication, RSS feeds, and ePub books.

```
<root>
  <item>
    <key>Course Code</key>
    <value>CSSE6400</value>
  </item>
  <item>
    <key>Course Title</key>
    <value>Software Architecture</value>
  </item>
</root>
```

[1]Current estimates are that Google requires over a million servers

[2]https://en.wikipedia.org/wiki/List_of_XML_markup_languages

XML is designed as a markup language, similar to HTML, it is not designed as a data exchange format. Developers have come to point out that the verbosity and complexity of XML, compared to alternatives such as JSON, are deal breakers. While XML can be used as a data exchange format it is not designed for it, and as a result APIs built around XML as a data format are becoming less common.

4.2 JSON

JavaScript Object Notation (JSON) is quickly replacing XML as the data format used in APIs. As you will note, it is more succinct and communicates the important points to a human reader better. 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 is that it lacks a schema. However, schemas are possible in JSON[3], 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"
}
```

4.3 MessagePack

It should not be a surprise that the JSON and XML formats are not resource efficient. Nowadays, we are less concerned with squeezing data into a tiny amount of data on the hard drive as storage is cheap. However, we are often concerned with how much data is being transmitted via network communication as bandwidth can be expensive.

In the example JSON snippet above, we use 78 bytes to encode the message. MessagePack[4] is a standard for encoding and decoding JSON. When encoding our original JSON snippet with MessagePack we shrink to just 57 bytes. At our scale, a negligible difference, but at the scale of terrabytes or petabytes, a significant factor.

```
» cat csse6400.msgpack

82 ab 43 6f 75 72 73 65 20 43 6f 64 65 a8 43 53 53 45 36 34 30 30 ac 43 6f 75 72 73
   65 20 54 69 74 6c 65 b5 53 6f 66 74 77 61 72 65 20 41 72 63 68 69 74 65 63 74 75
   72 65
```

Info

For those interested, 0x82 specifies a map type (0x80) with two fields (0x02). Followed by a string type (0xa0) of size eleven (0x0b). The rest is left as an exercise: `https://github.com/msgpack/msgpack/blob/master/spec.md`.

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

[4] https://msgpack.org/

4.4 Protobuf

Protocol Buffers (protobuf) is another type of binary encoding. However, unlike MessagePack, the format was designed from scratch, allowing a more compact and better designed format. Protobufs require all data to be defined by a schema. For example:

```
» cat csse6400.proto

message Course {
    required string code = 1;
    required string name = 2;
}
```

Protobufs differ from XML, JSON, and MessagePack via their method of integration. In the previous examples, your language would have a library to encode and decode the data format into and out of your language's type system. With protobuf, an external tool, *protoc*, takes the schema and generates a model of the schema in your target language. This gives every language a native interface for interacting with the data format, often allowing developers to not be aware of the underlying encoding.

```
$ protoc --java_out=. csse6400.proto
```

```
» cat csse6400.java

Course softarch = Course.newBuilder()
    .setCode("CSSE6400")
    .setName("Software Architecture")
    .build();
output = new FileOutputStream(args[0]);
softarch.writeTo(output);
```

5 API Protocols

It is worth being aware of a few of the common protocols and conventions used in APIs. Fortunately, we don't need to understand the low-level communication protocols, such as TCP/IP, HTTP, and HTTPS, as these are handled by the underlying libraries. We will focus on the higher-level protocols and conventions.

We outline a few but note that in this course we will primarily focus on REST APIs. This is only an indication that they are better understood by course staff, not their superiority.

5.1 XML-RPC

XML-RPC was one of the first API standards. It is a Remote Procedure Call (RPC) based API which communicates via XML. It is not a commonly used standard anymore.

5.2 SOAP

After XML-RPC came SOAP. SOAP was impactful for service-oriented (now microservices) architectures but has largely fallen out of favour in-place of REST APIs. We do not cover SOAP in any meaningful depth due to its increasing irrelevance and complexity.

5.3 REST

REST is not a standard like SOAP once was, instead 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

5.4 JSON-RPC

A JSON RPC is another RPC based API based around communication via JSON. When deciding whether to use an RPC-based API or a REST/SOAP based API, the distinction to make is that RPC APIs should be action based, i.e. "I want to do X". Whereas REST/SOAP APIs are Create Read Update Delete (CRUD) based for providing a interface to mutable data.

5.5 GraphQL

GraphQL is a query language which allows more dynamic querying of data than REST based APIs. You can create nested queries and specify the fields you want to receive in response. GraphQL APIs are well-suited for building APIs designed for developers to consume but when dealing with rigid inter-service based requests, REST APIs are generally preferable.

5.6 gRPC

Another RPC framework, gRPC has started gaining a lot of attention since its first release in 2016. gRPC is based on the protobuf format. In addition to a more type-safe system, gRPC based APIs provide authentication and streaming mechanisms.

6 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.

6.1 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.

6.2 Joining the Course Organization

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 organization.

6.3 Joining the GitHub Classroom

Once you have joined the organization, you will need to join the GitHub Classroom.

TODO: Add link to GitHub Classroom

6.4 Creating a Practical Repository

7 TODO

7.1 The API design

7.1.1 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**.

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

7.1.2 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 todays date a task should be due by.

e.g. http://localhost:6400/api/todos?completed=true&window=7

```
[
  {
    "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"
  },
  {
    ...
  },
  ...
]
```

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

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

```
{
  "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"
}
```

7.1.4 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.

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

```
{
  "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"
}
```

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

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

```
{
  "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"
}
```

7.1.6 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.

```
{
  "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"
}
```

7.2 Implementation with Flask

7.2.1 Setting up you 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 beaware 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.

Info

You will require an installation of python3 and pip to be able to complete this practical. Install these tools now before continuing.

We will be using 'pipenv' to manage the python libraries we need for this project so lets install that now.

```
$ python3 -m pip install pipenv
```

Navigate to you cloned practical respository in a terminal and start your project by creating the python environment using the following:

```
$ pipenv --python 3
```

This will create a 'Pipfile' and a 'Pipfile.lock' file. The Pipfile is where we will specify the libraries we need for our project and the Pipfile.lock is where pipenv will store the versions of the libraries we have installed. We will use the Pipfile to install the libraries we need for the API.

```
  » cat Pipfile

[[source]]
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true

[dev-packages]

[packages]

[requires]
python_version = "3.10"
```

Info

Your python version may be differnt, this is fine as long as it is python 3.

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

```
$ pipenv install flask
```

Once this is done you can see its made some changes to our Pipfile and Pipfile.lock files.

7.2.2 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:

```python
from flask import Flask

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

Your repository should now look like:

```
.
|-- README.md
|-- Pipfile
|-- Pipfile.lock
|-- todo
    |-- __init__.py
```

Now we have created a basic flask app but how do we run it? When using pipenv we need to run it in the following way:

```
$ pipenv run flask --app todo run
```

This webserver is a bit boring though, lets 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`:

```python

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:

TODO: Change the port to 6400

```python
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:

```
$ pipenv run flask --app todo run

* 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
```

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

```
.
|-- README.md
|-- Pipfile
|-- Pipfile.lock
|-- todo
    |-- __init__.py
    |-- views
        |-- routes.py
```

7.2.3 Returning JSON with Flask

We have a webserver 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:

```python
from flask import Blueprint, jsonify
```

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

Now lets go back to our browser and refresh, we should see the following:

```json
{
  "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.

7.2.4 Calling your API locally

We have many choices when it comes to calling our API locally. We could use curl, postman, vscode, or our browser. We are going to focus on using curl and the REST visual studio code extension.

curl

Install curl if its 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 lets 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:

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

vscode

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 vscode 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 suttle "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 out request by the output of:

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

7.2.5 Creating more endpoints

Lets 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
```

Lets 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:

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

Now the server should reload, if it doesn't, 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 use 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": false,
      "deadline_at": "2023-02-27T00:00:00",
      "created_at": "2023-02-20T00:00:00",
      "updated_at": "2023-02-20T00:00:00"
    }])
```

Now lets 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": false,
    "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:

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

Youll 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 lets 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:

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

Youll 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": false,
        "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": false,
        "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 dont 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 weeks practical, in the 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. The final files you should have for this practical are available at `github.com/csse6400/?`. Remember to ask for any assistance on the slack channel if you get stuck or have any questions.

TODO: Fill in the github link that is above.

References

[1] M. Kleppmann, *Designing Data-Intensive Applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media, Inc., March 2017.