APIs

Overview

- APIs
- REST
- Endpoints and paths
- HTTP Methods
- Idempotency

Learning Objectives

- Summarize the key features of an APIs
- Identify the types of API and what makes a good one
- Explain the key features of REST
- Identify the parts of an endpoint
- List the four most common HTTP methods
- Have a better understanding of Idempotency
- Have a clearer understanding of request Header and Body
- Identify response codes
- Practice accessing an API

Example

If you wanted to find out how long it takes to get from the IW Leeds office to the IW Manchester office, how would you do that?

1. Open google.com/maps in your web browser
2. Search for Infinity Works Manchester
3. Select the right place from the list of options
4. Select 'directions to here'
5. Select 'from' box
6. Search for Infinity Works, Leeds
7. etc...

Can we program a computer to do these things?

The process we just described was using a Graphical User Interface (GUI)

- **Graphical** - How you interact with it
- **User** - Who it's for
- **Interface** - Like a touchscreen, you can interact with it to send data to your phone and get data back on the screen.

Computers can't use GUIs. If you want computer to be able to use a system, they need their own interface. If the world had any sense of order, they would be called:

- **Programmatic** - How you interact with it
- **Application** - Who it's for
- **Interface** - As before

But alas, they're called APIs (**A**pplication **P**rogramming **I**nterface)

Instead of clicking buttons, you send some kind of request like:

```
GET https://maps.googleapis.com/maps/api/directions/json
    ?origin=M24LQ&destination=LS12EQ
```

We will break down each part of the above request so we can fully understand what it's doing shortly.

Instead of human-understandable graphically displayed results, you get computer parsable data.

Open `google-maps-data.json` and take a look.

APIs Explained

An interface which allows your application to interact with an external service using a set of commands.

You don't need to know the internals of the service, just how to interact with it (remember back to encapsulation, abstraction).

Imagine you (service A) ordering food at a restaurant. The waiter (API) takes the order to the kitchen (service B), the food is created and is passed back to you

---

Why are they important?

- Allows access to data from a service (eg. getting shopping results with a search)
- Allows for the change of data through a service (eg. updating a shopping item's information)
- Allows filtering and transformation of data from a service (eg. getting shopping results with a search and filtering by date added)

Good APIs

A well built API will:

- Be **Easily Understood** by conforming to some familiar patterns
- Provide a **Uniform Interface** so different clients can all use the same API (web, mobile)
- Provide **Separation of Concerns**

  - Server and client can be developed independently without one needing to know the internal workings of the other
- Be **Stateless**
  - Each request should stand on its own. Operations should not require multiple requests that require the server to remember things between requests.

Types of API

There is no single way to build an API. Different protocols have been developed over the years including:

- SOAP (*Simple Object Access Protocol*)
- RPC (*Remote Procedure Call*)
- REST (*Representational state transfer*)
- GraphQL

REST is by *far* the most common right now. So that's what we're going to focus on.

REST

Representational **S**tate **T**ransfer.

REST is built on top of HTTP.

API calls take the form of a single **request** made by the client and a single **response** that the server sends back.

REST was originally envisioned as a way of synchronising state between a client and a server.

REST request

A REST *request* has 4 key components:

- The **endpoint** - The URL *eg*:
  - https://api.github.com/users/torvalds/repos?page=0
- The **method** - A verb indicating the kind of action *eg*:
  - GET
  - POST
  - PUT
  - DELETE
- The **headers** - Metadata about the request *eg*:
  - `content-type=application/json`
- The **body** - Data you are sending to the server (sometimes)

The Endpoint

`https://api.github.com/users/torvalds/repos?page=0`

We can break the endpoint into bits:

- The **protocol** (`https://`) - the underlying transport system for the REST request. This is `http` or `https`
- The **domain** (`api.github.com`) - the unique identifier for the server that we are sending our request to
- The **path** (`/user/torvalds/repos`) - tells the server what 'resource' we want to access
- The **query parameters** (`?page=0`) - optional extra data about how we'd like to access the resource

---

Paths

Paths can refer to a *document* or a *collection*.

Collection:

- `/users`
- *you would expect this to return a list (array) of users*

Document:

- `/users/john` (or sometimes `/user/john`)
- *you would expect this to return an object describing a single user*

Paths

Documents can have sub-document or sub-collections

- `/users/john/devices` - sub-collection
- `/users/john/devices/laptop` - sub-document
- `/users/john/laptop` - sub-document

Paths

Sometimes paths reference a *controller resource*. These represent actions rather than objects and are described with verbs. They do a thing rather than getting or setting a thing.

- `users/john/laptop/reset`
- `users/john/playlists/study-music/play`

The Method

There are many HTTP methods available. REST APIs typically make use of these 4:

- `GET` - for fetching a resource from a server
- `POST` - for sending a resource to a server
- `PUT` - creates or overwrites a resource
- `DELETE` - deletes a resource

The first two are the most common, and some APIs will only use these.

---

method	send data	receive data	idempotent
GET	No	Yes	Yes
POST	Yes	Yes	No
PUT	Yes	Yes	Yes
DELETE	No	Yes	Yes

Idempotency

No matter how many times you call an operation, the result will be the same.

Idempotent: Requesting the same image from photo website

Not Idempotent: Sending a payment of £100 to a friend

Examples

GET **is idempotent**, as multiple calls to the GET resource will always return the same response.

PUT **is idempotent**, as calling the PUT method multiple times will update the same resource and not change the outcome.

POST **is NOT idempotent**, and calling the POST method multiple times can have different results and will result in creating new resources.

DELETE **is idempotent** because once the resource is deleted, it is gone and calling the method multiple times will not change the outcome.

Quiz Time! 🤓

You are writing an API for a to-do list application. You wish to create a new todo-list item. Which method is most appropriate for this API endpoint?

1. GET
2. POST
3. PUT
4. DELETE

Answer: POST (Non-idempotent creation of a resource)

You want to be able to list the current to-do items still left to do. Which method is most appropriate?

1. GET
2. POST
3. PUT
4. DELETE

Answer: `GET` (You are fetching data)

You want to mark a specific to-do item as done. Which method is most appropriate?

1. `GET`
2. `POST`
3. `PUT`
4. `DELETE`

Answer: `PUT` (Idempotent update of a resource)

You would like to be able to remove to-do items created accidentally? Which method is most appropriate?

1. `GET`
2. `POST`
3. `PUT`
4. `DELETE`

Answer: `DELETE` (Idempotent removal of a resource)

Combining Methods and Paths

One path could do different things depending on the method used, for example:

- `GET /orders/90345/items` - get items from the order
- `POST /orders/90345/items` - add a new item to the order
- `DELETE /orders/90345/items` - remove all items from the order

Quiz Time! 🤓

What is the most RESTful way to express the following?

Getting the top 10 recommended items for you

1. `GET products/recommended?limit=10`
2. `GET products/recommended/10`
3. `POST products/recommend?limit=10`
4. `POST products/recommend/10`

Answer: `1` - We are getting data. The number of items is not part of the resource as such, so it goes in the query string.

What is the most RESTful way to express the following?

Attaching a supporting file to a job application

1. `GET /applications/002/files`

2. `POST /applications/002/files`
3. `POST /applications/002/files/upload`
4. `PUT /applications/002/files?name=cover-letter.docx`

Answer: `2` - We are adding a resource to the collection of files.

What is the most RESTful way to express the following?

Setting your profile picture on a social media site

1. `POST /user-profile/picture`
2. `PUT /user-profile/picture`
3. `POST /user-profile/picture/update`
4. `PUT /user-profile?field=picture`

Answer: `2` - We are updating an existing resource.

What is the most RESTful way to express the following?

Remove a repository from your online git account

1. `POST /repositories/CIA-Hack/remove`
2. `PUT /repositories/CIA-Hack/remove`
3. `DELETE /repositories/CIA-Hack/`

Answer: `3` - Prefer the http verb over verb endpoints.

What is the most RESTful way to express the following?

Searching for a picture by key words on a stock imagery database

1. `GET /photos?tags=hackers,code`
2. `GET /photos/search?tags=hackers,code`
3. `POST /photos/search` (with tags in the body)

Answer: Probably `2`? But this is a contentious one!

The Headers

Headers contain metadata that are generally sent with every request you make. Many are set automatically by your browser. For example:

- `User-Agent` - the type of browser you are using
- `Cookies` - the cookies saved in your browser for this website
- `Referrer` - what page you made this request from
- `Accept` - what type of responses your browser can handle (xml, json, text)
- `Accept-Encoding` - what forms of compression your browser can handle to reduce bandwidth use

---

The most common ones will be:

- `Authorization` - some token that proves who you are and what access you have to this API
- `Content-Type` - what form the data in your body takes (sometimes this is set for you)

You can create your own custom headers, but you rarely need to do this. Most request parameters belong in either the path or the query string.

The Body

`POST` and `PUT` requests are for sending data to the server. That data can take many forms:

- JSON - the most common for structured data
- Form data - what you get by default when you submit a form
- Binary - when uploading files
- XML
- Plaintext

REST response

- The **response code** - A number indicating the status of the response *eg. `200` (success)*
- The **headers** - Metadata about the response *eg: content-type=application/json*
- The **body** - Data you receive from the server

Response Code

Response codes are three digit numbers (between 100 and 599). Their first digit tells you what kind of status it represents:

- **1xx** - intermediate status. You won't encounter these
- **2xx** - everything was ok
- **3xx** - redirect (your request seems fine, but you're in the wrong place)
- **4xx** - client error (you messed up)
- **5xx** - server error (we messed up)

Common examples:

- **200** - success (what you hope to receive every time)
- **400** - bad request
- **401** - unauthorized
- **403** - forbidden
- **404** - not found
- **418** - I'm a teapot (yes, seriously!)
- **500** - internal server error
- **503** - service unavailable

The body of the response should include more information about why you got that code and what to do about it.

Response Headers

There are many. Here are some examples:

- `Content-Type` - what type of data the body contains
- `Cache-Control` - tells the client how long it is acceptable to cache the response for
- `Cookie` - sets a cookie in the user's browser

The Body

Unlike in requests, it doesn't matter what method the request was made with. You can always send a body with the response.

What that response represents is up to you and may depend on the nature of the request and the response:

- `GET /users` - the body should be a list of users
- `POST /users` - the body might contain the new user you just created, or just its ID
- `Error 400` - the body might tell you what you did wrong and how to correct it
- `Error 500` - the body might tell you what went wrong on the server

---

How are APIs made?

You can make an API in any major programming language. It is normal to use a module to do the hard work for you. Once you've chosen a module all you need to do is configure it to meet your requirements.

There are loads of python API modules you can use, two of the more common ones are:

Flask

- Simple and lightweight
- Its quick and easy to set up
- Has good online support

Django

- Lots of functionality
- High versatility

Exercise

Instructor will distribute `exercise.pdf` and `rest-ful-cafe.html`

More API Exercises

If you want to try out more APIs, check these out:

- An incredibly simple API visualiser
- An API that gets you interacting with Pokemon data
- An API that returns Kanye West quotes and docs

Learning Objectives Revisited

- Summarize the key features of an APIs
- Identify the types of API and what makes a good one
- Explain the key features of REST
- Identify the parts of an endpoint
- List the four most common HTTP methods
- Have a better understanding of Idempotency
- Have a clearer understanding of request Header and Body
- Identify response codes
- Practice accessing an API

Terms and Definitions Recap

API: An **A**pplication **P**rogramming **I**nterface is a computing interface that defines interactions between multiple software intermediaries.

REST: **RE**presentational **S**tate **T**ransfer is a software architectural style that defines a set of constraints to be used for creating Web services.

Further Reading

- HTTP codes explained
- An amazing guide to REST by the National Bank of Belgium
- An essay on REST

For fun:

- HTTP response codes for cats
- HTTP response codes for dogs