Learning Objectives

- **Explain why having raw data in a structured format is beneficial**
- **Define REST API**
- **Explain how `GET`, `POST`, `PUT`, and `DELETE` work with the Post pattern**
- **Explain how `GET`, `POST`, `PUT`, and `DELETE` work with the Post ID pattern**

What is a REST API?

Intro to REST APIs

What is a REST API?

When we want to read and understand some information online, we'll do so using our own intelligence. Heading, labels, colors and other visual cues on the page will help to guide us to figure out what certain elements on the page mean.

Take Blango as an example. We can tell who the author is, even if they have a name we don't recognize, because it says *By **Author Name***. We can tell when it was published because the byline says *on **Date***, and the date is in a format that is familiar to us. We recognize the title, since it's in larger text. Page comments, recent posts and the author's bio are distinguishable because they have headings. And the most important part, the content, takes up most of the page.

Even if the layout of the page were different, you could still read and comprehend it. Indeed, not many blogs you see have the same layout, and yet thanks to your intelligence and intuition you're able to read and understand them.

Think about how other types of sites work, and the clues they give. Shopping sites will highlight the price, as well as show the currency symbol ($, £, €, etc), rather than just a number. News sites have larger headlines with smaller summary and content text. If you're reading a recipe, you can discern the ingredients because it will have a heading and some quantities (in grams, ounces, cups, etc).

While computers are getting smarter (at least, in terms of "artificial" intelligence), having it automatically read a web page and extract the content in a meaningful way is not an easy thing to set up. Web scrapers can extract elements from a web page but need to be manually configured to tell the scraper what type of content each element contains. Furthermore, they don't allow you to save data back to the web server.

Why do we want a computer to be able to "read" a web page? It's not actually about needing a computer to interpret/understand the page, we just want the raw data in a structured format. Most commonly, this is to be

able to interact with it using a different interface, usually a mobile application, but also different web applications that you, or others, may build.

Take for example an app like Facebook on your phone. When you open it to see your timeline, the app isn't logging into the Facebook website, scraping the HTML, then displaying just the data. It's actually making a special request to a specific URL on the Facebook web server(s), fetching the timeline data in a structured format, then rendering it. Similarly if you post a status update, your phone isn't finding the status update form, pasting the content in and clicking *Submit*. It will send a request to a different URL with the status data. This illustrates how we'd interact with web data using a non-browser interface (mobile phone app).

We might also want to access structured data that another website provides, in our own website, or vice versa. For example, we can create a Django view to list all posts by a given author. A website that aggregates blog posts from multiple other sites could use this list to help build its content.

Now that we know the reason for providing computer-readable structured data, let's see how to do it.

Post Pattern

Post Pattern

REST stands for **RE**presentation **S**tate **T**ransfer, and describes a way of architecting web services to transfer data, to facilitate the operations like what we've just described. It emphasizes a stateless protocol, the use of standard HTTP verbs (*GET*, *POST*, *PUT* and *DELETE*), and consistent (and optionally hierarchical) URLs. The responses should be some kind of structured text, like JSON or XML. Any system designed in such a way can be described as **RESTful**.

Let's look at this in practice, and see how we would design a REST API for `Post` objects in Blango. We'll look at the two URLs patterns we could create, and how they behave for different HTTP verbs that could be used to access them.

The first URL is `/posts/`.

- `GET`: return a list of `Post` objects in the system. This list could be filtered by query arguments in the URL. For example, `/posts/?published=true` to return only published posts. This might not contain all the attributes of the `Post` objects, but probably just the ones necessary for a summary view. The response body could look like:

```
[
    {
        "author_id": 2,
        "created_at": "2021-07-20T04:23:21.693Z",
        "modified_at": "2021-07-20T04:23:21.693Z",
        "published_at": "2021-07-21T04:21:52Z",
        "title": "Leo: A Personal Profile",
        "slug": "leo-a-personal-profile",
        "summary": "Who is Leo? We investigate in this post."
    },
    {
        "author_id": 1,
        "created_at": "2021-07-20T04:21:42.688Z",
        "modified_at": "2021-07-20T04:21:42.688Z",
        "published_at": "2021-07-21T04:08:45Z",
        "title": "Django vs Python",
        "slug": "django-vs-python",
        "summary": "Which is better, Django or Python? We
        investigate."
        ...
    }
]
```

- `POST`: create a new `Post` object. The body of the HTTP request would be the `Post` data, serialized into the right format for the API – in our case we're going to be using JSON.

The request body would therefore look like:

```
{
    "slug": "some-world-news",
    "title": "Some World News",
    "content": "<p>Here's the latest breaking world new, from
        around the world.</p>"
    "author_id": 4
}
```

A response to a valid request could be a serialized version of the object that was created, with missing fields interpolated. For example, we don't `POST` the `id`, `created_at` or `modified_at` fields, but they would be in the response.

Or, the API could respond with a redirect to the URL of the new `Post` (the *detail URL*). We'll look at the detail URL soon. This allows the client to choose whether or not to should fetch the newly created object, and the server saves some overhead by not having to fetch and serialize the object.

- `PUT`: listing URLs usually don't respond to `PUT` requests. Generally in REST term, `POST`ing to a URL means "accept this data and create it as a sub-object of the URL to which it was posted". Whereas `PUT` means "update the object at this URL to match the data I am `PUT`ing". So for listing URLs it doesn't make sense to update the data in this way.
- `DELETE`: usually does nothing to a listing URL either. In theory you could implement this to delete *all* the objects with one request, but the convention is to not do this!

Post ID Pattern

Post ID Pattern

The other URL pattern is for `Post` details, and is in the format `/posts/<post_id>/`.

- `GET`: return a `Post` object in JSON format, including all attributes (or, at least the publicly visible ones). The response body would look like this:

```
{
    "author_id": 2,
    "created_at": "2021-07-20T04:23:21.693Z",
    "modified_at": "2021-07-20T04:23:21.693Z",
    "published_at": "2021-07-21T04:21:52Z",
    "title": "Leo: A Personal Profile",
    "slug": "leo-a-personal-profile",
    "summary": "Who is Leo? We investigate in this post.",
    "content": "<p>Leo Lucio loves to write about himself in the
        third person. His hobbies include blogging, coding,
        walking on the beach, sleeping, and watching educational
        films.</p>"
}
```

`POST`: will do nothing, as we don't want to create any sub-objects of a `Post`.

`PUT`: update the specified `Post` with the data in the request body. The request body would look similar to the `POST` request to the `Post` list view.

`DELETE`: delete the specified `Post` at the URL. There is no body to the request.

We can also specify sub-resources on detail URLs. For example, `/posts/<post_id>/comments/`, allowing us to `GET` the list of comments for a `Post`, or `POST` a new comment.

Progressing deeper by building detail URLs for sub-objects doesn't always make sense, and can come down to personal preference. For example, let's say the `Comment` with ID `1` exists on `Post` with ID `1`, and similarly with `Comment` and `Posts` with ID `2`.

If we are going to get a `Comment` by its ID, then we can just get it directly from a `/comments/<comment_id>/` detail URL. There's no point in trying to request `/posts/1/comments/1/`, since it requires us to needlessly pass in the `Post` ID. Furthermore, what would happen if we tried to get `/posts/2/comments/1/`? Should we allow it, or return a `404 Not Found` since

that `Comment` doesn't belong to that `Post`? With these inconsistencies you can see why we would want to forgo implementing such nested detail URLs.

`GET` and `PUT` requests should be idempotent, that is, the state of the system after one request should be the same after the second and subsequent requests. With `GET` this is obvious: it is just fetching data not updating it, so this makes sense. If you repeat a `GET`, you should get the same information back again.

Similarly `PUT` requests update an object to match the data you've sent. If you send that same data again, the object shouldn't change as it already matches it.

`DELETE` requests are also idempotent by design. Once an object is deleted it can't be deleted again, and so repeating a `DELETE` request won't cause further system changes.

Compare these to `POST` requests. If you repeat a `POST`, the system creates *another* new object, so it's not idempotent.

Now that we've covered quite a bit of REST background, let's see how you could create a basic REST API with JSON support, using only Django's built-in functionality.