

XJCO3011: Web Services and Web Data



Session #3 – Alice in Webland

Instructor: Dr. Guilin Zhao
Spring 2023



Administrative Issues

- Lab section: 1st Lab: **28 Feb. @ X7304**
- TA office hour schedule:
 - ✓ Lilan Peng: **Friday: 10:00am – 11:30am @X31404**
 - ✓ Lin Chen: **Thursday: 10:00am – 11:30am @X9432**

Review of Sessions #1 and #2

Ture or False:

1. SOAP is a protocol, but REST is an architectural style. ()
2. SOAP is simple, REST is complex. ()
3. There are two types of HTTP messages: the request message and the response message. ()

Topic: Surfing the Web



Topic: Surfing the Web

- The World Wide Web became popular because ordinary people can use it to do really useful things with minimal training. But behind the scenes, the Web is also a powerful platform for distributed computing.
- The principles that make the Web usable by ordinary people also work when the “user” is an automated software agent. A piece of software designed to transfer money between bank accounts (or carry out any other real-world task) can accomplish the task using the same basic technologies a human being would use.
- Start off by telling a simple story about the World Wide Web, as a way of explaining the principles behind its design and the reasons for its success
- The story needs to be simple because although you’re certainly familiar with the Web, you might not have heard of the concepts that make it work. I want you to have a simple, concrete example to fall back on if you ever get confused about terminology like “hypermedia as the engine of application state.”



Alice's Story

I am sure you have heard this story (Alice's Adventures in Wonderland) many times before, but there are always **new lessons** to learn from a good story.

- Episode 1: The Billboard
- Episode 2: The Home Page
- Episode 3: The Link
- Episode 4: The Form and the Redirect



Episode 1: The Billboard

This is the story of Alice, a little girl who was one day walking around town when she saw this billboard



Alice was intrigued by this web address, so she pulled out her mobile phone and put <http://www.youtypeitwepostit.com/> in her browser's address bar.

But, what's at the other end of that URL?



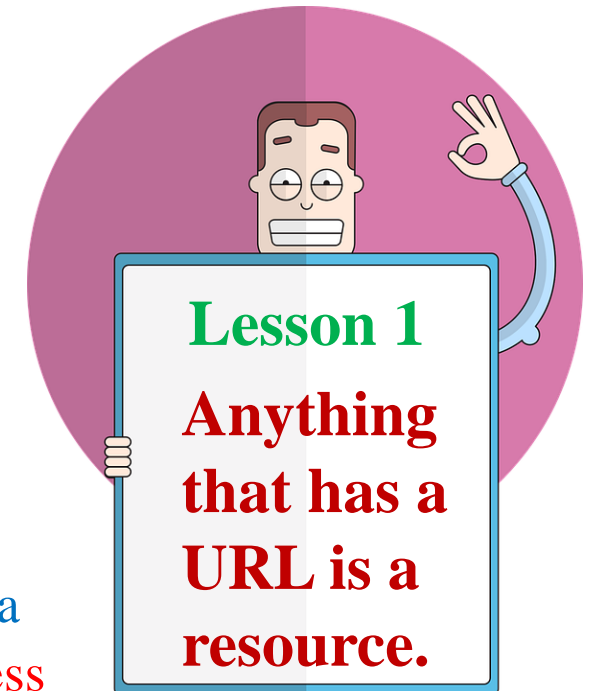
Note: Try the above link, it should work for you as it did work for Alice.

Alice's web browser sent an HTTP request to a web server—specifically, to the URL <http://www.youtypeitwepostit.com/>.

Well, you probably know that the technical term for the thing named by a URL is a **resource**.

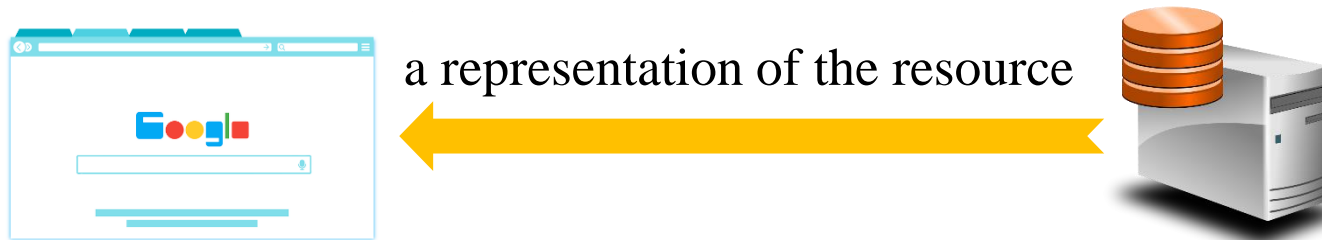


- ❖ In the context of HTTP, the term "host" typically refers to the name or IP address of the server that is hosting a particular website or web application. When a client makes a request to a server using HTTP, it typically includes a "Host" header in the request. This header specifies the hostname or IP address of the server that the client is trying to connect to.



the server sent a document in response (maybe an HTML document, maybe a binary image or maybe something else).

Whatever document the server sent is called a **representation** of the resource.



Lesson 2

When a client makes an HTTP request to a URL, it gets a representation of the underlying resource.

The client never sees a resource directly.

GOT IT?



Lesson 3

- A URL identifies **one and only** one resource.
- If an API provides two different functionality, the API should treat them as two resources with different URLs.
- The principle of addressability says **that every resource should have its own URL**.
- If something is important to your application, it should have a unique name, a URL, so that you and your users can refer to it unambiguously.

Got it?

Alice's Story

- ✓ Episode 1: The Billboard
- Episode 2: The Home Page
- Episode 3: The Link
- Episode 4: The Form and the Redirect



Episode 2: The Home Page

So, when the web server handled this request, it sent this response:

```
HTTP/1.1 200 OK
```

```
Content-type: text/html
```

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
<title>Home</title>
```

```
</head>
```

```
<body>
```

```
<div>
```

```
<h1>You type it, we post it!</h1>
```

```
<p>Exciting! Amazing!</p>
```

```
<p class="links">
```

```
<a href="/messages">Get started</a>
```

```
<a href="/about">About this site</a>
```

```
</p>
```

```
</div>
```

```
</body>
```

```
</html>
```

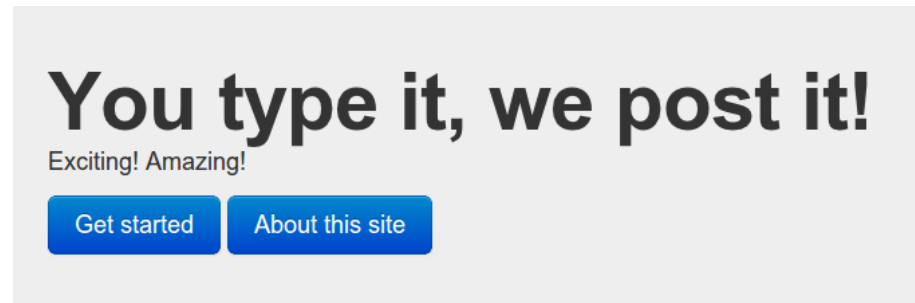
- ❖ In HTTP, the "a" tag (also known as the **hyperlink tag**) is one of the HTML tags used to create hyperlinks. The "a" tag is used to link a text or an image from one webpage to another webpage or to another part of the same webpage. The "a" tag typically contains an **"href" attribute**, which specifies the URL or file path of the target of the link.

- ❖ For example, the following is an example of using the "a" tag to create a hyperlink:

```
< a href=" https://www.example.com ">Click here to visit Example website</ a>
```

- ❖ This tag will display a link text "Click here to visit Example website" on the webpage. When the user clicks the link, it will take the user to a website named "https://www.example.com".

Alice's web browser decoded the response as an HTML document and displayed it graphically



Now Alice could read the web page and understand what the billboard was talking about. It was advertising a microblogging site.

She was feeling so sleepy, so she put her phone down and fell asleep. The next morning she woke up and clicked her phone to resume what she was doing the night before.

Lesson 4

Short Sessions

- **HTTP sessions last for one request.** The client sends a request, and the server responds.
- This means Alice could turn her phone off overnight, and the web server will not sit up all night worrying about Alice. When she's not making an HTTP request, the server doesn't know Alice exists.
- This principle is called statelessness. The server doesn't care what state the client is in.

Got it?

The markup for the home page contains two links: one to the relative URL /about and one to /messages.

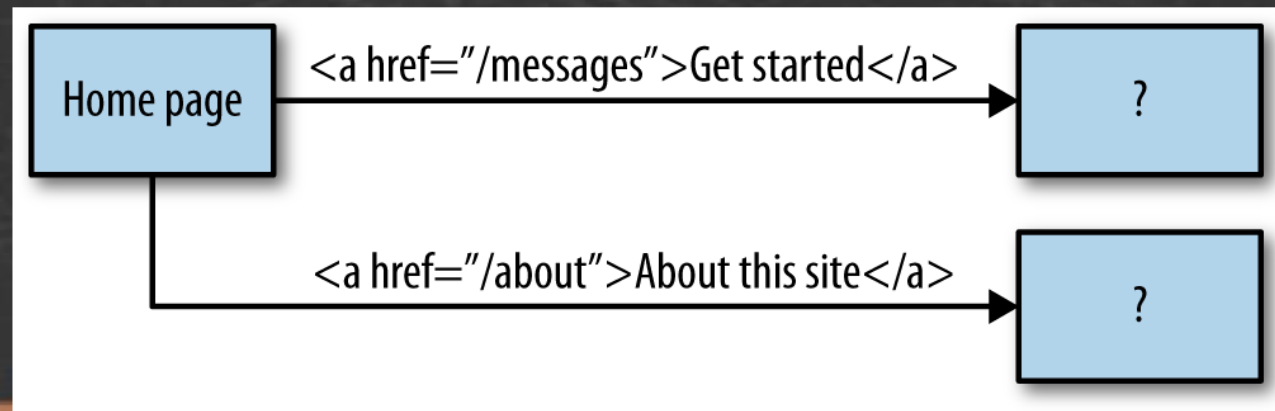
```
HTTP/1.1 200 OK
Content-type: text/html
<!DOCTYPE html>
<html>
  <head>
    <title>Home</title>
  </head>
  <body>
    <div>
      <h1>You type it, we post it!</h1>
      <p>Exciting! Amazing!</p>
      <p class="links">
        <a href="/messages">Get started</a>
        <a href="/about">About this site</a>
      </p>
    </div>
  </body>
</html>
```

At first Alice only knew one URL—the URL to the home page—but now she knows three. The server is slowly revealing its structure to her.

Lesson 5

Self-Descriptive Messages

When a client requests a resource, the representation (e.g. HTML document) it receives doesn't just give immediate information about this resource. **The representation can also help the client decide the next step** to make (what state it must go to next).



Alice's Story

- ✓ Episode 1: The Billboard
- ✓ Episode 2: The Home Page
- Episode 3: The Link
- Episode 4: The Form and the Redirect



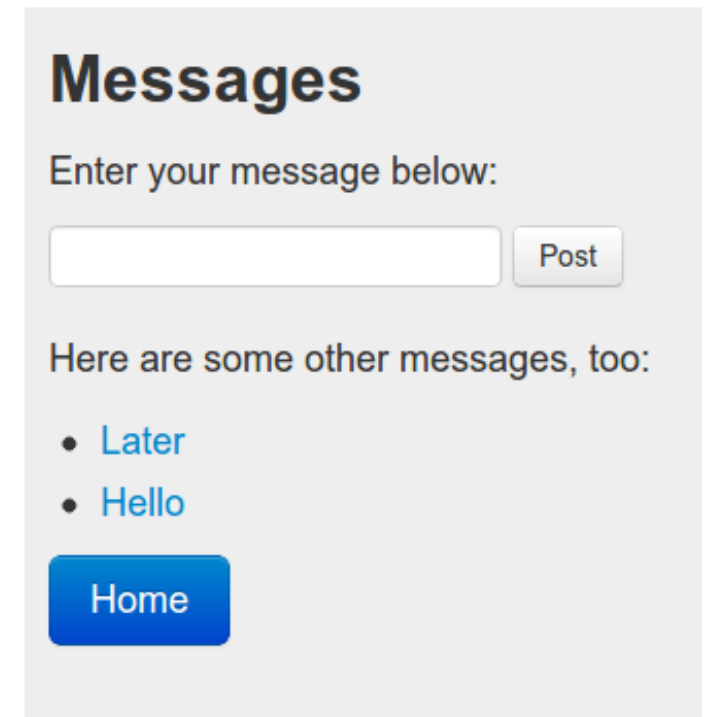
Episode 3: The Link

After reading the home page, Alice decided to give this site a try, so she clicked the link that said “Get started.” and her browser made an HTTP request:

```
GET /messages HTTP/1.1  
Host: www.youtypeitwepostit.com
```

The server handled this particular GET request and sent a representation of /messages,

and Alice’s browser rendered the HTML graphically



Messages

Enter your message below:

Here are some other messages, too:

- [Later](#)
- [Hello](#)

Alice's Story

- ✓ Episode 1: The Billboard
- ✓ Episode 2: The Home Page
- ✓ Episode 3: The Link
- Episode 4: The Form and the Redirect



Episode 4: The Form and the Redirect

Alice was tempted by the form, so she typed in “Test” and clicked the Post button.:
Again, Alice’s browser made an HTTP request:

```
POST /messages HTTP/1.1  
Host: www.youtypeitwepostit.com  
Content-type: application/x-www-form-urlencoded  
  
message=Test&submit=Post
```



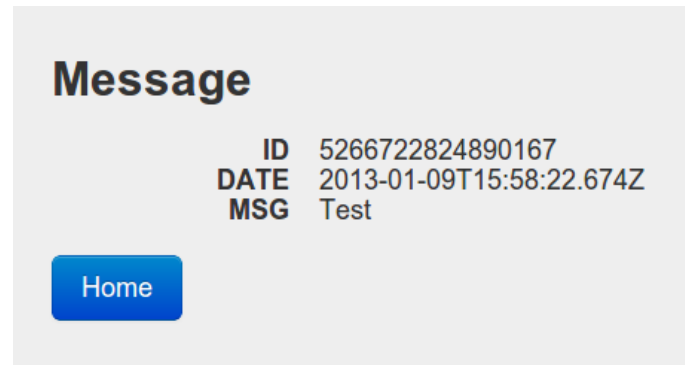
And the server responded with the following:

```
HTTP/1.1 303 See Other  
Content-type: text/html  
Location:  
http://www.youtypeitwepostit.com/messages/5266722824890167
```

Status code 303 told Alice’s browser to automatically make a fourth HTTP request, to the URL given in the Location header. Without asking Alice’s permission, her browser did just that:

```
GET /messages/5266722824890167 HTTP/1.1
```

This time, the server responded with 200 (“OK”) and an HTML document. Alice’s browser displayed this document graphically



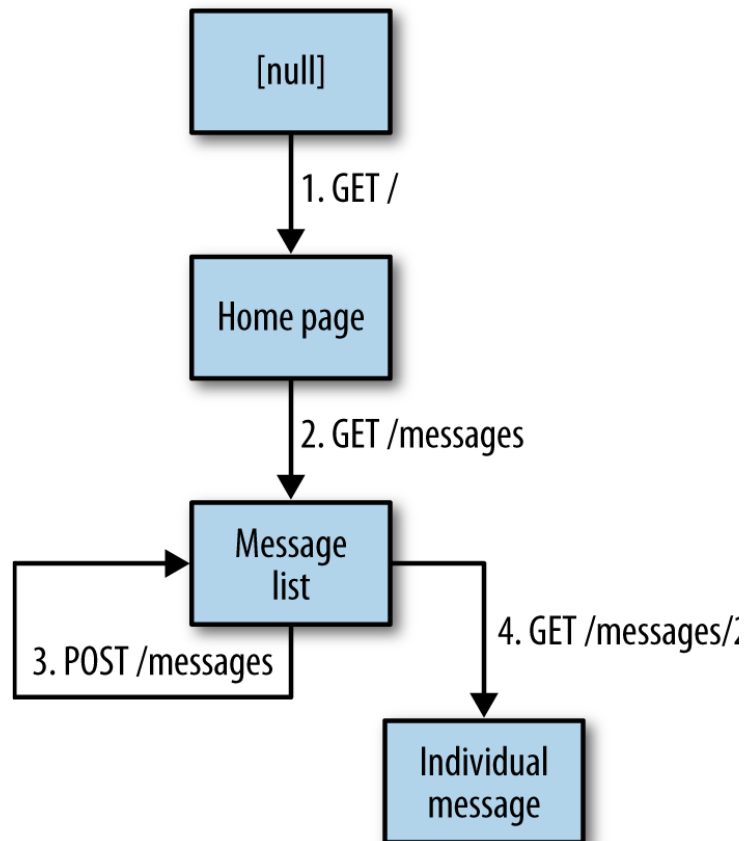
By looking at the graphical rendering, Alice saw that her message (“Test”) is now a fully fledged post on YouTypeItWePostIt.com. Our story ends here—Alice has accomplished her goal of trying out the microblogging site.

Lesson 6

Standardized Methods

- The HTTP standard (RFC 2616) defines eight methods a client can apply to a resource.
- The most frequently used are **GET, HEAD, POST, PUT, DELETE, and OPTION.**
- We distinguish between resources not by defining new methods, but by assigning different URLs to these resources.
- A single URL can exhibit different behaviour depending on the method sent to it.

To Sum up, here is the state diagram that shows Alice's entire adventure from the perspective of her web browser.



- ❖ When Alice started up the browser on her phone, it didn't have any particular page loaded. It was an empty slate.
 - Then Alice typed in a URL and a GET request took the browser to the site's home page.
 - Alice clicked a link, and a second GET request took the browser to the list of messages.
 - She submitted a form, which caused a third request (a POST request).
 - The response to that was an HTTP redirect, which Alice's browser made automatically. Alice's browser ended up at a web page describing the message Alice had just created.
- ❖ Every state in this diagram corresponds to a particular page (or to no page at all) being open in Alice's browser window.
- ❖ In REST terms, we call this bit of information—which page are you on?—**the application state**.
- ❖ When you surf the Web, every transition from one application state to another corresponds to a link you decided to follow or a form you decided to fill out.
- ❖ Not all transitions are available from all states. Alice can't make her POST request directly from the home page, because the home page doesn't

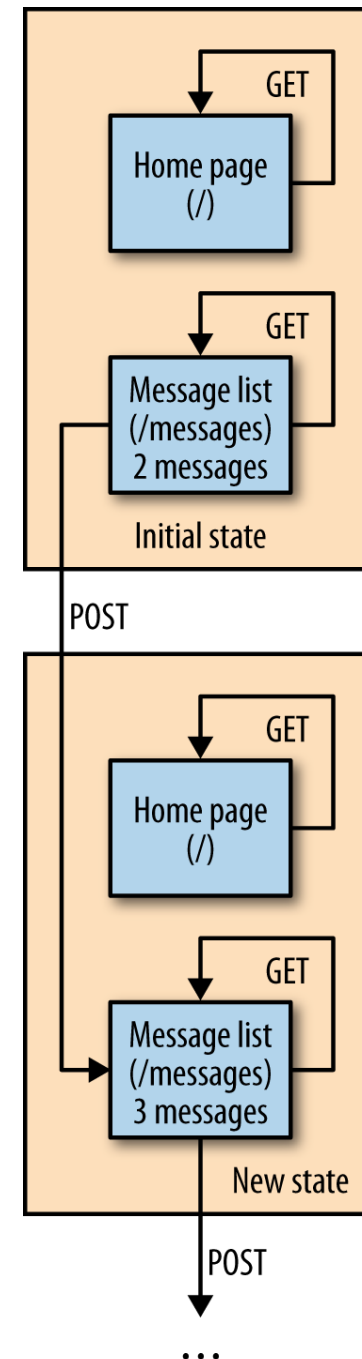
Lesson 7

Application State

- Clients can change state when they receive a new representation of a resource.
- Because HTTP sessions are so short, the server doesn't know anything about a client's application state.
- Application state is kept on the client, but the server can manipulate it by sending representations that describe the possible state transitions.

And, here is a state diagram showing Alice's adventure from the perspective of the web server.

- ❖ The server manages two resources: the home page (served from /) and the message list (served from /messages). The state of these resources is called *resource state*.
- ❖ When the story begins, there are two messages in the message list: “Hello” and “Later.” Sending a GET to the home page doesn't change resource state, since the home page is a static document that never changes. Sending a GET to the message list won't change the state either.
- ❖ But when Alice sends a POST to the message list, it puts the server in a new state. Now the message list contains three messages: “Hello,” “Later,” and “Test.” There's no way back to the old state, but this new state is very similar. As before, sending a GET to the home page or message list won't change anything. But sending another POST to the message list will add a fourth message to the list.



Lesson 8

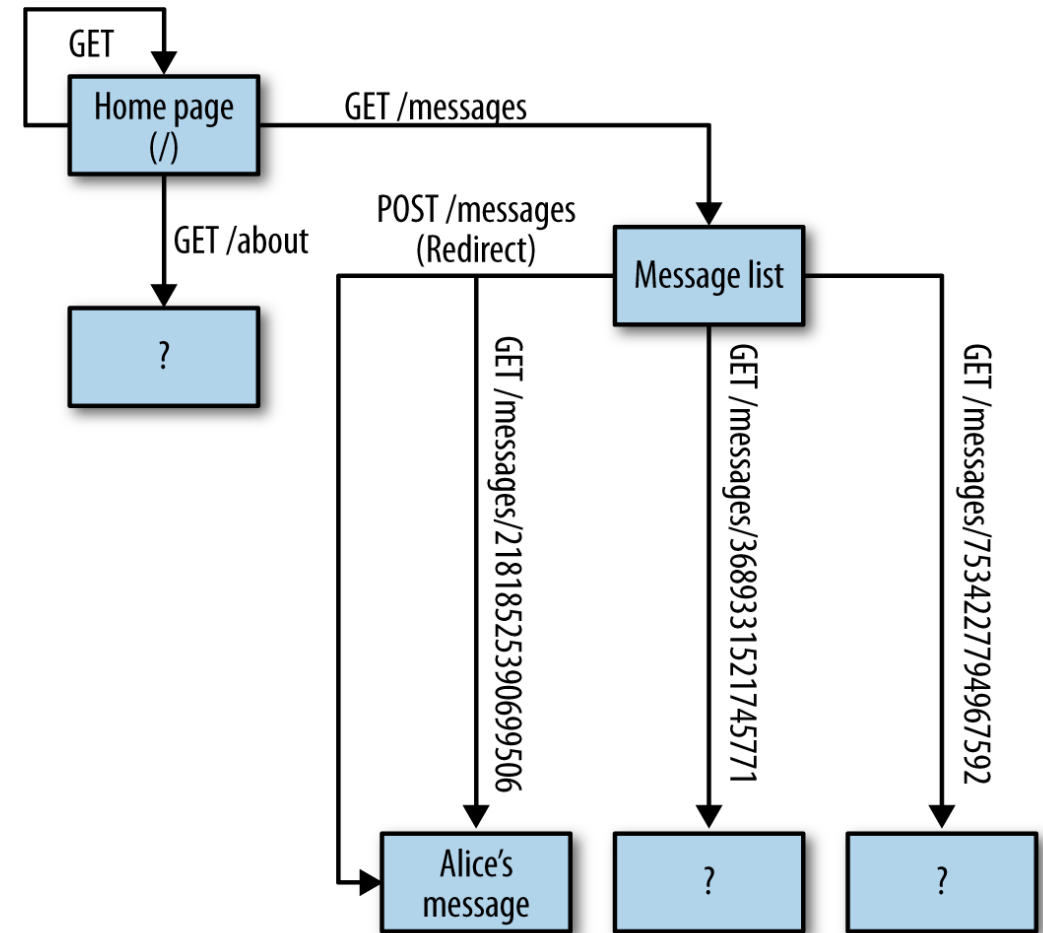
Resource State

- The client has no direct control over resource state—all that stuff is kept on the server.
- Resource state is kept on the server, but the client can manipulate it by sending the server a representation describing the desired new state.
- REST web APIs work through **representational state transfer**.

Finally, here is a partial map of the website from the client's perspective.

This is a web of HTML pages.

The **strands** of the web are the HTML `<a>` tags and `<form>` tags, each describing a GET or POST HTTP request.



Lesson 9

The principle of Connectedness

Each web page tells you how to get to the adjoining pages.

The Web as a whole works on the principle of connectedness, which is better known as “hypermedia as the engine of application state,” sometimes abbreviated as HATEOAS

The Semantic Challenge

- The story of Alice's trip through a website went as smoothly as it did thanks to a very slow and expensive piece of hardware: Alice herself.
- Every time her browser rendered a web page, Alice, a human being, had to look at the rendered page and decide what to do next.
- The Web works because human beings make all the decisions about which links to click and which forms to fill out.
- The whole point of web APIs is to get things done without making a human sit in front of a web browser all day.
- How can we program a computer to make the decisions about which links to click?
- A computer can parse the HTML markup `Get started`, but it can't understand the phrase "Get started."
- Why bother to design APIs that serve self-descriptive messages if those messages won't be understood by their software consumers?
- This is the biggest challenge in web API design: **bridging the semantic gap between understanding a document's structure and understanding what it means**

That's it Folks



Further Reading

Chapter 1 from RESTful Web APIs by Leonard Richardson and Mike Amundsen