

2

THE ANATOMY OF WEB APIS



Most of what the average user knows about a web application comes from what they can see and click in the graphical user interface (GUI) of their web browser. Under the hood, APIs perform much of the work. In particular, web APIs provide a way for applications to use the functionality and data of other applications over HTTP to feed a web application GUI with images, text, and videos.

This chapter covers common API terminology, types, data interchange formats, and authentication methods and then ties this information together with an example: observing the requests and responses exchanged during interactions with Twitter's API.

How Web APIs Work

Like web applications, web APIs rely on HTTP to facilitate a client/server relationship between the host of the API (the *provider*) and the system or person making an API request (the *consumer*).

An API consumer can request resources from an *API endpoint*, which is a URL for interacting with part of the API. Each of the following examples is a different API endpoint:

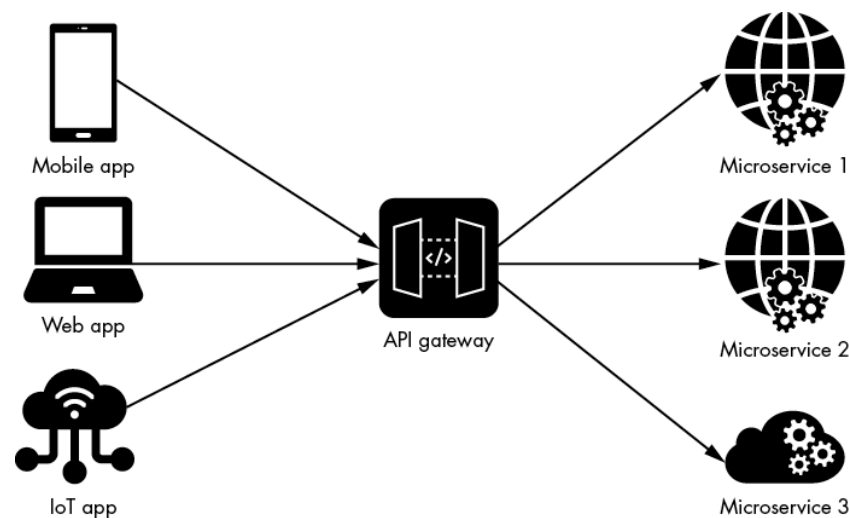
```
https://example.com/api/v3/users/  
https://example.com/api/v3/customers/  
https://example.com/api/updated_on/  
https://example.com/api/state/1/
```

Resources are the data being requested. A *singleton* resource is a unique object, such as `/api/user/{user_id}`. A *collection* is a group of resources, such as

`/api/profiles/users`. A *subcollection* refers to a collection within a particular resource. For example, `/api/user/{user_id}/settings` is the endpoint to access the *settings* subcollection of a specific (singleton) user.

When a consumer requests a resource from a provider, the request passes through an *API gateway*, which is an API management component that acts as an entry point to a web application. For example, as shown in [Figure 2-1](#), end users can access an application's services using a plethora of devices, which are all filtered through an API gateway. The API gateway then distributes the requests to whichever microservice is needed to fulfill each request.

The API gateway filters bad requests, monitors incoming traffic, and routes each request to the proper service or microservice. The API gateway can also handle security controls such as authentication, authorization, encryption in transit using SSL, rate limiting, and load balancing.



[Figure 2-1](#): A sample microservices architecture and API gateway

A *microservice* is a modular piece of a web app that handles a specific function. Microservices use APIs to transfer data and trigger actions. For example, a web application with a payment gateway may have several different features on a single web page: a billing feature, a feature that logs customer account information, and one that emails receipts upon purchase. The application's backend design could be monolithic, meaning all the services exist within a single application, or

it could have a microservice architecture, where each service functions as its own standalone application.

The API consumer does not see the backend design, only the endpoints they can interact with and the resources they can access. These are spelled out in the API *contract*, which is human-readable documentation that describes how to use the API and how you can expect it to behave. API documentation differs from one organization to another but often includes a description of authentication requirements, user permission levels, API endpoints, and the required request parameters. It might also include usage examples. From an API hacker's perspective, the documentation can reveal which endpoints to call for customer data, which API keys you need in order to become an administrator, and even business logic flaws.

In the following box, the GitHub API documentation for the `/applications/{client_id}/grants/{access_token}` endpoint, taken from <https://docs.github.com/en/rest/reference/apps>, is an example of quality documentation.

REVOKE A GRANT FOR AN APPLICATION

OAuth application owners can revoke a grant for their OAuth application and a specific user.

DELETE `/applications/{ client_id }/grants/{ access_token }`

PARAMETERS

Name	Type	In	Description
accept	string	header	Setting to <code>application/vnd.github.v3+json</code> is recommended.
<code>client_id</code>	string	path	The client ID of your GitHub app.
<code>access_token</code>	string	body	Required. The OAuth access token used to authenticate to the GitHub API.

The documentation for this endpoint includes the description of the purpose of the API request, the HTTP request method to use when interacting with the API endpoint, and the endpoint itself, */applications*, followed by variables.

The acronym *CRUD*, which stands for *Create, Read, Update, Delete*, describes the primary actions and methods used to interact with APIs. *Create* is the process of making new records, accomplished through a POST request. *Read* is data retrieval, done through a GET request. *Update* is how currently existing records are modified without being overwritten and is accomplished with POST or PUT requests. *Delete* is the process of erasing records, which can be done with POST or DELETE, as shown in this example. Note that CRUD is a best practice only, and developers may implement their APIs in other ways. Therefore, when you learn to hack APIs later on, we'll test beyond the CRUD methods.

By convention, curly brackets mean that a given variable is necessary within the path parameters. The *{client_id}* variable must be replaced with an actual client's ID, and the *{access_token}* variable must be replaced with your own access token. Tokens are what API providers use to identify and authorize requests to approved API consumers. Other API documentation might use a colon or square brackets to signify a variable (for example, */api/v2:customers/* or */api:collection:client_id*).

The "Parameters" section lays out the authentication and authorization requirements to perform the described actions, including the name of each parameter value, the type of data to provide, where to include the data, and a description of the parameter value.

Standard Web API Types

APIs come in standard types, each of which varies in its rules, functions, and purpose. Typically, a given API will use only one type, but you may encounter endpoints that don't match the format and structure of the others or don't match a standard type at all. Being able to recognize typical and atypical APIs will help you know what to expect and test for as an API hacker. Remember, most public APIs are designed to be self-service, so a given API provider will often let you know the type of API you'll be interacting with.

This section describes the two primary API types we'll focus on throughout this book: RESTful APIs and GraphQL. Later parts of the book, as well as the book's labs, cover attacks against RESTful APIs and GraphQL only.

RESTful APIs

Representational State Transfer (REST) is a set of architectural constraints for applications that communicate using HTTP methods. APIs that use REST constraints are called *RESTful* (or just REST) APIs.

REST was designed to improve upon many of the inefficiencies of other older APIs, such as Simple Object Access Protocol (SOAP). For example, it relies entirely on the use of HTTP, which makes it much more approachable to end users. REST APIs primarily use the HTTP methods GET, POST, PUT, and DELETE to accomplish CRUD (as described in the section “How Web APIs Work”).

RESTful design depends on six constraints. These constraints are “shoulds” instead of “musts,” reflecting the fact that REST is essentially a set of guidelines for an HTTP resource-based architecture:

1. **Uniform interface:** REST APIs should have a uniform interface. In other words, the requesting client device should not matter; a mobile device, an IoT (internet of things) device, and a laptop must all be able to access a server in the same way.
2. **Client/server:** REST APIs should have a client/server architecture. Clients are the consumers requesting information, and servers are the providers of that information.
3. **Stateless:** REST APIs should not require stateful communications. REST APIs do not maintain state during communication; it is as though each request is the first one received by the server. The consumer will therefore need to supply everything the provider will need in order to act upon the request. This has the benefit of saving the provider from having to remember the consumer from one request to another. Consumers often provide tokens to create a state-like experience.
4. **Cacheable:** The response from the REST API provider should indicate whether the response is cacheable. *Caching* is a method of increasing request throughput by storing commonly requested data on the client side or in a server cache. When a request is made, the client will first check its local storage for the requested information. If it doesn't find the information, it passes the request to the server, which checks its local storage for the requested information. If the data is not there either, the request could be passed to other servers, such as database servers, where the data can be retrieved.
As you might imagine, if the data is stored on the client, the client can immediately retrieve the requested data at little to no processing cost to the server.

This also applies if the server has cached a request. The further down the chain a request has to go to retrieve data, the higher the resource cost and the longer it takes. Making REST APIs cacheable by default is a way to improve overall REST performance and scalability by decreasing response times and server processing power. APIs usually manage caching with the use of headers that explain when the requested information will expire from the cache.

5. **Layered system:** The client should be able to request data from an endpoint without knowing about the underlying server architecture.
6. **Code on demand (optional):** Allows for code to be sent to the client for execution.

REST is a style rather than a protocol, so each RESTful API may be different. It may have methods enabled beyond CRUD, its own sets of authentication requirements, subdomains instead of paths for endpoints, different rate-limit requirements, and so on. Furthermore, developers or an organization may call their API “RESTful” without adhering to the standard, which means you can’t expect every API you come across to meet all the REST constraints.

[Listing 2-1](#) shows a fairly typical REST API GET request used to find out how many pillows are in a store’s inventory. [Listing 2-2](#) shows the provider’s response.

```
GET /api/v3/inventory/item/pillow HTTP/1.1
HOST: rest-shop.com
User-Agent: Mozilla/5.0
Accept: application/json
```

[Listing 2-1](#): A sample RESTful API request

```
HTTP/1.1 200 OK
Server: RESTfulServer/0.1
Cache-Control: no-store
Content-Type: application/json

{
  "item": {
    "id": "00101",
    "name": "pillow",
    "count": 25
    "price": {
      "currency": "USD",
      "value": "19.99"
```

```
}  
  },  
}
```

***Listing 2-2:** A sample RESTful API response*

This REST API request is just an HTTP GET request to the specified URL. In this case, the request queries the store’s inventory for pillows. The provider responds with JSON indicating the item’s ID, name, and quantity of items in stock. If there was an error in the request, the provider would respond with an HTTP error code in the 400 range indicating what went wrong.

One thing to note: the *rest-shop.com* store provided all the information it had about the resource “pillow” in its response. If the consumer’s application only needed the name and value of the pillow, the consumer would need to filter out the additional information. The amount of information sent back to a consumer completely depends on how the API provider has programmed its API.

REST APIs have some common headers you should become familiar with. These are identical to HTTP headers but are more commonly seen in REST API requests than in other API types, so they can help you identify REST APIs. (Headers, naming conventions, and the data interchange format used are normally the best indicators of an API’s type.) The following subsections detail some of the common REST API headers you will come across.

Authorization

Authorization headers are used to pass a token or credentials to the API provider. The format of these headers is `Authorization: <type> <token/credentials>`. For example, take a look at the following authorization header:

```
Authorization: Bearer Ab4dtok3n
```

There are different authorization types. `Basic` uses base64-encoded credentials. `Bearer` uses an API token. Finally, `AWS-HMAC-SHA256` is an AWS authorization type that uses an access key and a secret key.

Content Type

`Content-Type` headers are used to indicate the type of media being transferred. These headers differ from `Accept` headers, which state the media type you want to receive; `Content-Type` headers describe the media you're sending.

Here are some common `Content-Type` headers for REST APIs:

`application/json` Used to specify JavaScript Object Notation (JSON) as a media type. JSON is the most common media type for REST APIs.

`application/xml` Used to specify XML as a media type.

`application/x-www-form-urlencoded` A format in which the values being sent are encoded and separated by an ampersand (&), and an equal sign (=) is used between key/value pairs.

Middleware (X) Headers

`X-<anything>` headers are known as *middleware headers* and can serve all sorts of purposes. They are fairly common outside of API requests as well. `X-Response-Time` can be used as an API response to indicate how long a response took to process. `X-API-Key` can be used as an authorization header for API keys. `X-Powered-By` can be used to provide additional information about backend services. `X-Rate-Limit` can be used to tell the consumer how many requests they can make within a given time frame. `X-RateLimit-Remaining` can tell a consumer how many requests remain before they violate rate-limit enforcement. (There are many more, but you get the idea.) `X-<anything>` middleware headers can provide a lot of useful information to API consumers and hackers alike.

ENCODING DATA

As we touched upon in Chapter 1, HTTP requests use encoding as a method to ensure that communications are handled properly. Various characters that can be problematic for the technologies used by the server are known as *bad characters*. One way of handling bad characters is to use an encoding scheme that formats the message in such a way as to remove them. Common encoding schemes include Unicode encoding, HTML encoding, URL encoding, and base64 encoding. XML typically uses one of two forms of Unicode encoding: UTF-8 or UTF-16.

When the string “hAPI hacker” is encoded in UTF-8, it becomes the following:

```
|x68|x41|x50|x49|x20|x68|x61|x63|x6b|x65|x72
```

Here is the UTF-16 version of the string:

```
|u{68}|u{41}|u{50}|u{49}|u{20}|u{68}|u{61}|u{63}|u{6b}|u{65}|u{72}
```

Finally, here is the base64-encoded version:

```
aEFQSSBoYWNrZXI=
```

Recognizing these encoding schemes will be useful as you begin examining requests and responses and encounter encoded data.

GraphQL

Short for *Graph Query Language*, *GraphQL* is a specification for APIs that allow clients to define the structure of the data they want to request from the server. GraphQL is RESTful, as it follows the six constraints of REST APIs. However, GraphQL also takes the approach of being *query-centric*, because it is structured to function similarly to a database query language like Structured Query Language (SQL).

As you might gather from the specification’s name, GraphQL stores the resources in a graph data structure. To access a GraphQL API, you’ll typically access the URL where it is hosted and submit an authorized request that contains query parameters as the body of a POST request, similar to the following:

```
query {  
  users {  
    username  
    id  
    email  
  }  
}
```

In the right context, this query would provide you with the usernames, IDs, and emails of the requested resources. A GraphQL response to this query would look like the following:

```
{  
  "data": {  
    "users": {  
      "username": "hapi_hacker",  
      "id": 1111,  
      "email": "hapihacker@email.com"  
    }  
  }  
}
```

GraphQL improves on typical REST APIs in several ways. Since REST APIs are resource based, there will likely be instances when a consumer needs to make several requests in order to get all the data they need. On the other hand, if a consumer only needs a specific value from the API provider, the consumer will need to filter out the excess data. With GraphQL, a consumer can use a single request to get the exact data they want. That's because, unlike REST APIs, where clients receive whatever data the server is programmed to return from an endpoint, including the data they don't need, GraphQL APIs let clients request specific fields from a resource.

GraphQL also uses HTTP, but it typically depends on a single entry point (URL) using the POST method. In a GraphQL request, the body of the POST request is what the provider processes. For example, take a look at the GraphQL request in [Listing 2-3](#) and the response in [Listing 2-4](#), depicting a request to check a store's inventory for graphics cards.

```
POST /graphql HTTP/1.1  
HOST: graphql-shop.com
```

```
Authorization: Bearer ab4dt0k3n

{query❶ {
  inventory❷ (item:"Graphics Card", id: 00101) {
    name
    fields❸{
      price
      quantity} } }
}
```

Listing 2-3: An example GraphQL request

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

{
  "data": {
    "inventory": { "name": "Graphics Card",
    "fields":❹[
      {
        "price": "999.99"
        "quantity": 25 } ] } }
}
```

Listing 2-4: An example GraphQL response

As you can see, a query payload in the body specifies the information needed. The GraphQL request body begins with the query operation ❶, which is the equivalent of a GET request and used to obtain information from the API. The GraphQL node we are querying for, "inventory" ❷, is also known as the root query type. Nodes, similar to objects, are made up of fields ❸, similar to key/value pairs in REST. The main difference here is that we can specify the exact fields we are looking for. In this example, we are looking for the "price" and "quantity" fields. Finally, you can see that the GraphQL response only provided the requested fields for the specified graphics card ❹. Instead of getting the item ID, item name, and other superfluous information, the query resolved with only the fields that were needed.

If this had been a REST API, it might have been necessary to send requests to different endpoints to get the quantity and then the brand of the graphics card, but with GraphQL you can build out a query for the specific information you are looking for from a single endpoint.

GraphQL still functions using CRUD, which may sound confusing at first since it relies on POST requests. However, GraphQL uses three operations within the POST request to interact with GraphQL APIs: query, mutation, and subscription. *Query* is an operation to retrieve data (read). *Mutation* is an operation used to submit and write data (create, update, and delete). *Subscription* is an operation used to send data (read) when an event occurs. Subscription is a way for GraphQL clients to listen to live updates from the server.

GraphQL uses *schemas*, which are collections of the data that can be queried with the given service. Having access to the GraphQL schema is similar to having access to a REST API collection. A GraphQL schema will provide you with the information you'll need in order to query the API.

You can interact with GraphQL using a browser if there is a GraphQL IDE, like GraphiQL, in place (see [Figure 2-2](#)).

Otherwise, you'll need a GraphQL client such as Postman, Apollo-Client, GraphQL-Request, GraphQL-CLI, or GraphQL-Compose. In later chapters, we'll use Postman as our GraphQL client.

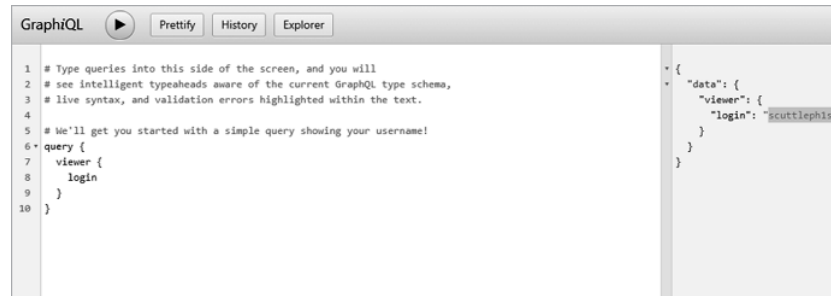


Figure 2-2: The GraphQL interface for GitHub

SOAP: AN ACTION-ORIENTED API FORMAT

Simple Object Access Protocol (SOAP) is a type of action-oriented API that relies on XML. SOAP is one of the older web APIs, originally released as XML-RPC back in the late 1990s, so we won't cover it in this book.

Although SOAP works over HTTP, SMTP, TCP, and UDP, it was primarily designed for use over HTTP. When SOAP is used over HTTP, the requests are all made using HTTP POST. For example, take a look at the following sample SOAP request:

```
POST /Inventory HTTP/1.1
Host: www.soap-shop.com
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

❶<soap:Envelope
❷xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
  soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

❸<soap:Body xmlns:m="http://www.soap-shop.com/inventory">
  <m:GetInventoryPrice>
    <m:InventoryName>ThebestSOAP</m:InventoryName>
  </m:GetInventoryPrice>
</soap:Body>

</soap:Envelope>
```

The corresponding SOAP response looks like this:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
```

```
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
  soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

  <soap:Body xmlns:m="http://www.soap-shop.com/inventory">
    ❶<soap:Fault>
      <faultcode>soap:VersionMismatch</faultcode>
      <faultstring, xml:lang='en">
        Name does not match Inventory record
      </faultstring>
    </soap:Fault>
  </soap:Body>

</soap:Envelope>
```

SOAP API messages are made up of four parts: the envelope ❶ and header ❷, which are necessary, and the body ❸ and fault ❹, which are optional. The *envelope* is an XML tag at the beginning of a message that signifies that the message is a SOAP message. The *header* can be used to process a message; in this example, the `Content-Type` request header lets the SOAP provider know the type of content being sent in the POST request (`application/soap+xml`). Since APIs facilitate machine-to-machine communication, headers essentially form an agreement between the consumer and the provider concerning the expectations within the request. Headers are a way to ensure that the consumer and provider understand one another and are speaking the same language. The *body* is the primary payload of the XML message, meaning it contains the data sent to the application. The *fault* is an optional part of a SOAP response that can be used to provide error messaging.

REST API Specifications

The variety of REST APIs has left room for other tools and standardizations to fill in some of the gaps. *API specifications*, or description languages, are frameworks that help organizations design their APIs, automatically create consistent human-readable documentation, and therefore help developers and users know what to expect regarding the API's functionality and results. Without specifications, there would be little to no consistency between APIs. Consumers would have to learn

how each API's documentation was formatted and adjust their application to interact with each API.

Instead, a consumer can program their application to ingest different specifications and then easily interact with any API using that given specification. In other words, you can think of specifications as the home electric sockets of APIs. Instead of having a unique electric socket for every home appliance, the use of a single consistent format throughout a home allows you to buy a toaster and plug it into a socket on any wall without any hassle.

OpenAPI Specification 3.0 (OAS), previously known as Swagger, is one of the leading specifications for RESTful APIs. OAS helps organize and manage APIs by allowing developers to describe endpoints, resources, operations, and authentication and authorization requirements. They can then create human- and machine-readable API documentation, formatted as JSON or YAML. Consistent API documentation is good for developers and users.

The *RESTful API Modeling Language (RAML)* is another way to consistently generate API documentation. RAML is an open specification that works exclusively with YAML for document formatting. Similar to OAS, RAML was designed to document, design, build, and test REST APIs. For more information about RAML, check out the raml-spec GitHub repo (<https://github.com/raml-org/raml-spec>).

In later chapters, we will use an API client called Postman to import specifications and get instant access to the capabilities of an organization's APIs.

API Data Interchange Formats

APIs use several formats to facilitate the exchange of data. Additionally, specifications use these formats to document APIs. Some APIs, like SOAP, require a specific format, whereas others allow the client to specify the format to use in the request and response body. This section introduces three common formats: JSON, XML, and YAML. Familiarity with data interchange formats will help you recognize API types, what the APIs are doing, and how they handle data.

JSON

JavaScript Object Notation (JSON) is the primary data interchange format we'll use throughout this book, as it is widely used for APIs. It organizes data in a way that

is both human-readable and easily parsable by applications; many programming languages can turn JSON into data types they can use.

JSON represents objects as key/value pairs separated by commas, within a pair of curly brackets, as follows:

```
{
  "firstName": "James",
  "lastName": "Lovell",
  "tripsToTheMoon": 2,
  "isAstronaut": true,
  "walkedOnMoon": false,
  "comment" : "This is a comment",
  "spacecrafts": ["Gemini 7", "Gemini 12", "Apollo 8", "Apollo 13"],
  "book": [
    {
      "title": "Lost Moon",
      "genre": "Non-fiction"
    }
  ]
}
```

Everything between the first curly bracket and the last is considered an object. Within the object are several key/value pairs, such as `"firstName": "James"`, `"lastName": "Lovell"`, and `"tripsToTheMoon": 2`. The first entry of the key/value pair (on the left) is the *key*, a string that describes the value pair, and the second is the *value* (on the right), which is some sort of data represented by one of the acceptable data types (strings, numbers, Boolean values, null, an array, or another object). For example, notice the Boolean value `false` for `"walkedOnMoon"` or the `"spacecrafts"` array surrounded by square brackets. Finally, the nested object `"book"` contains its own set of key/value pairs. [Table 2-1](#) describes JSON types in more detail.

JSON does not allow inline comments, so any sort of comment-like communications must take place as a key/value pair like `"comment" : "This is a comment"`. Alternatively, you can find comments in the API documentation or HTTP response.

Table 2-1: JSON Types

Type	Description	Example
Strings	Any combination of characters within double quotes.	<pre>{ "Motto": "Hack the planet", "Drink": "Jolt", "User": "Razor" }</pre>
Numbers	Basic integers, fractions, negative numbers, and exponents. Notice that the multiple items are comma-separated.	<pre>{ "number_1" : 101, "number_2" : -102, "number_3" : 1.03, "number_4" : 1.0E+4 }</pre>
Boolean values	Either true or false.	<pre>{ "admin" : false, "privesc" : true }</pre>
Null	No value.	<pre>{ "value" : null }</pre>
Arrays	An ordered collection of values. Collections of values are surrounded by brackets ([]) and the values are comma-separated.	<pre>{ "uid" : ["1", "2", "3"] }</pre>
Objects	An unordered set of value pairs inserted between curly brackets ({ }). An object can contain multiple key/value pairs.	<pre>{ "admin" : false, "key" : "value", "privesc" : true, "uid" : 101, "vulnerabilities" :</pre>

Type	Description	Example
		<pre>"galore" }</pre>

To illustrate these types, take a look at the following key/value pairs in the JSON data found in a Twitter API response:

```
{
  "id":1278533978970976256, ❶
  "id_str":"1278533978970976256", ❷
  "full_text":"1984: William Gibson published his debut novel, Neuromancer. It's a cyberpunk tale about Henry Case, a washed u
  "truncated":false ❸
}
```

In this example, you should be able to identify the number 1278533978970976256 ❶, strings like those for the keys "id_str" and "full_text" ❷, and the Boolean value ❸ for "truncated".

XML

The *Extensible Markup Language (XML)* format has been around for a while, and you'll probably recognize it. XML is characterized by the descriptive tags it uses to wrap data. Although REST APIs can use XML, it is most commonly associated with SOAP APIs. SOAP APIs can only use XML as the data interchange.

The Twitter JSON you just saw would look like the following if converted to XML:

```
<?xml version="1.0" encoding="UTF-8" ?> ❶
<root> ❷
  <id>1278533978970976300</id>
  <id_str>1278533978970976256</id_str>
  <full_text>1984: William Gibson published his debut novel, Neuromancer. It's a cyberpunk tale about Henry Case, a was
  <truncated>>false</truncated>
</root>
```

XML always begins with a *prolog*, which contains information about the XML version and encoding used ❶.

Next, *elements* are the most basic parts of XML. An element is any XML tag or information surrounded by tags. In the previous example,

`<id>1278533978970976300</id>`, `<id_str>1278533978</id_str>`, `<full_text>`, `</full_text>`, and `<truncated>>false</truncated>` are all elements. XML must have a root element and can contain child elements. In the example, the root element is `<root>` ❷. The child elements are XML attributes. An example of a child element is the `<BookGenre>` element within the following example:

```
<LibraryBooks>
  <BookGenre>SciFi</BookGenre>
</LibraryBooks>
```

Comments in XML are surrounded by two dashes, like this: `<!--XML comment example-->`.

The key differences between XML and JSON are JSON's descriptive tags, character encoding, and length: the XML takes much longer to convey the same information, a whopping 565 bytes.

YAML

Another lightweight form of data exchange used in APIs, *YAML* is a recursive acronym that stands for *YAML Ain't Markup Language*. It was created as a more human- and computer-readable format for data exchange.

Like JSON, YAML documents contain key/value pairs. The value may be any of the YAML data types, which include numbers, strings, Booleans, null values, and sequences. For example, take a look at the following YAML data:

```
---
id: 1278533978970976300
id_str: 1278533978970976256
#Comment about Neuromancer
full_text: "1984: William Gibson published his debut novel, Neuromancer. It's a cyberpunk tale about Henry Case, a washed up
truncated: false
...
```

You'll notice that YAML is much more readable than JSON. YAML documents begin with

```
---
```

and end with

```
...
```

instead of with curly brackets. Also, quotes around strings are optional. Additionally, URLs don't need to be encoded with backslashes. Finally, YAML uses indentation instead of curly brackets to represent nesting and allows for comments beginning with `#`.

API specifications will often be formatted as JSON or YAML, because these formats are easy for humans to digest. With only a few basic concepts in mind, we can look at either of these formats and understand what is going on; likewise, machines can easily parse the information.

If you'd like to see more YAML in action, visit <https://yaml.org>. The entire website is presented in YAML format. YAML is recursive all the way down.

API Authentication

APIs may allow public access to consumers without authentication, but when an API allows access to proprietary or sensitive data, it will use some form of authentication and authorization. An API's authentication process should validate that users are who they claim to be, and the authorization process should grant them the ability to access the data they are allowed to access. This section covers a variety of API authentication and authorization methods. These methods vary in complexity and security, but they all operate on a common principle: the consumer must send some kind of information to the provider when making a request, and the provider must link that information to a user before granting or denying access to a resource.

Before jumping into API authentication, it is important to understand what authentication is. Authentication is the process of proving and verifying an identity. In a web application, authentication is the way you prove to the web server that you are a valid user of said web app. Typically, this is done through the use of credentials, which consist of a unique ID (such as a username or email) and password. After a client sends credentials, the web server compares what was sent to

the credentials it has stored. If the credentials provided match the credentials stored, the web server will create a user session and issue a cookie to the client.

When the session ends between the web app and user, the web server will destroy the session and remove the associated client cookies.

As described earlier in this chapter, REST and GraphQL APIs are stateless, so when a consumer authenticates to these APIs, no session is created between the client and server. Instead, the API consumer must prove their identity within every request sent to the API provider's web server.

Basic Authentication

The simplest form of API authentication is *HTTP basic authentication*, in which the consumer includes their username and password in a header or the body of a request. The API could either pass the username and password to the provider in plaintext, like `username:password`, or it could encode the credentials using something like base64 to save space (for example, as `dXN1cm5hbWU6cGFzc3dvcmQK`).

Encoding is not encryption, and if base64-encoded data is captured, it can easily be decoded. For example, you can use the Linux command line to base64-encode `username:password` and then decode the encoded result:

```
$ echo "username:password" | base64
dXN1cm5hbWU6cGFzc3dvcmQK
$ echo "dXN1cm5hbWU6cGFzc3dvcmQK" | base64 -d
username:password
```

As you can see, basic authentication has no inherent security and completely depends on other security controls. An attacker can compromise basic authentication by capturing HTTP traffic, performing a man-in-the-middle attack, tricking the user into providing their credentials through social engineering tactics, or performing a brute-force attack in which they attempt various usernames and passwords until they find some that work.

Since APIs are often stateless, those using only basic authentication require the consumer to provide credentials in every request. It is common for an API provider to instead use basic authentication once, for the first request, and then issue an API key or some other token for all other requests.

API Keys

API keys are unique strings that API providers generate and grant to authorize access for approved consumers. Once an API consumer has a key, they can include it in requests whenever specified by the provider. The provider will typically require that the consumer pass the key in query string parameters, request headers, body data, or as a cookie when they make a request.

API keys typically look like semi-random or random strings of numbers and letters. For example, take a look at the API key included in the query string of the following URL:

```
/api/v1/users?apikey=ju574n3x4mp134p1k3y
```

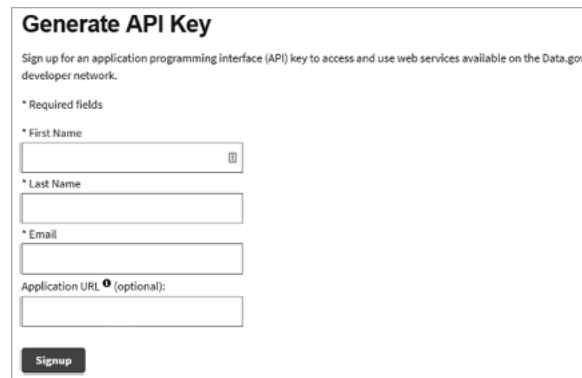
The following is an API key included as a header:

```
"API-Secret": "17813fg8-46a7-5006-e235-45be7e9f2345"
```

Finally, here is an API key passed in as a cookie:

```
Cookie: API-Key= 4n07h3r4p1k3y
```

The process of acquiring an API key depends on the provider. The NASA API, for example, requires the consumer to register for the API with a name, email address, and optional application URL (if the user is programming an application to use the API), as shown in [Figure 2-3](#).



Generate API Key

Sign up for an application programming interface (API) key to access and use web services available on the Data.gov developer network.

* Required fields

* First Name

* Last Name

* Email

Application URL (optional):

Figure 2-3: NASA's form to generate an API key

The resulting key will look something like this:

```
roS6SmRjLdxZzrNSAkxjCdb6WodSda2G9zc2Q7sK
```

It must be passed as a URL parameter in each API request, as follows:

```
api.nasa.gov/planetary/apod?  
api_key=roS6SmRjLdxZzrNSAkxjCdb6WodSda2G9zc2Q7sK
```

API keys can be more secure than basic authentication for several reasons. When keys are sufficiently long, complex, and randomly generated, they can be exceedingly difficult for an attacker to guess or brute-force. Additionally, providers can set expiration dates to limit the length of time for which the keys are valid.

However, API keys have several associated risks that we will take advantage of later in this book. Since each API provider may have their own system for generating API keys, you'll find instances in which the API key is generated based on user data. In these cases, API hackers may guess or forge API keys by learning about the API consumers. API keys may also be exposed to the internet in online repositories, left in code comments, intercepted when transferred over unencrypted connections, or stolen through phishing.

JSON Web Tokens

A *JSON Web Token (JWT)* is a type of token commonly used in API token-based authentication. It's used like this: The API consumer authenticates to the API provider with a username and password. The provider generates a JWT and

sends it back to the consumer. The consumer adds the provided JWT to the `Authorization` header in all API requests.

JWTs consist of three parts, all of which are base64-encoded and separated by periods: the header, the payload, and the signature. The *header* includes information about the algorithm used to sign the payload. The *payload* is the data included within the token, such as a username, timestamp, and issuer. The *signature* is the encoded and encrypted message used to validate the token.

[*Table 2-2*](#) shows an example of these parts, unencoded for readability, as well as the final token.

NOTE

The signature field is not a literal encoding of `HMACSHA512 . . .`; rather, the signature is created by calling the encryption function `HMACSHA512()`, specified by `"alg": "HS512"`, on the encoded header and payload, and then encoding the result.

Table 2-2: JWT Components

Component	Content
Header	<pre>{ "alg": "HS512", "typ": "JWT" }</pre>
Payload	<pre>{ "sub": "1234567890", "name": "hAPI Hacker", "iat": 1516239022 }</pre>
Signature	<pre>HMACSHA512(base64UrlEncode(header) + "." + base64UrlEncode(payload), SuperSecretPassword)</pre>
JWT	<pre>eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6ImhBUeKgSGFja2VyIiwiaWF0IjoxNTE2MjM5MDIyYQ.zsUjGDbBjqI-bJbaUmvUdKaGSEvROKfNjy9K6TckK5sd97AMdPDLxUZwsneff401ZWQikhgPm7HHlXyn4jm0Q</pre>

JWTs are generally secure but can be implemented in ways that will compromise that security. API providers can implement JWTs that do not use encryption, which means you would be one base64 decode away from being able to see what is inside the token. An API hacker could decode such a token, tamper with the contents, and send it back to the provider to gain access, as you will see in Chapter 10. The JWT secret key may also be stolen or guessed by brute force.

HMAC

A *hash-based message authentication code (HMAC)* is the primary API authentication method used by Amazon Web Services (AWS). When using HMAC, the provider creates a secret key and shares it with consumer. When a consumer interacts with the API, an HMAC hash function is applied to the consumer's API re-

quest data and secret key. The resulting hash (also called a *message digest*) is added to the request and sent to the provider. The provider calculates the HMAC, just as the consumer did, by running the message and key through the hash function, and then compares the output hash value to the value provided by the client. If the provider’s hash value matches the consumer’s hash value, the consumer is authorized to make the request. If the values do not match, either the client’s secret key is incorrect or the message has been tampered with.

The security of the message digest depends on the cryptographic strength of the hash function and secret key. Stronger hash mechanisms typically produce longer hashes. [Table 2-3](#) shows the same message and key hashed by different HMAC algorithms.

Table 2-3: HMAC Algorithms

Algorithm	Hash output
HMAC-MD5	f37438341e3d22aa11b4b2e838120dcf
HMAC-SHA1	4c2de361ba8958558de3d049ed1fb5c115656e65
HMAC-SHA256	be8e73ffbd9a953f2ec892f06f9a5e91e6551023d1942ec7994fa1a78a5ae6bc
HMAC-SHA512	6434a354a730f888865bc5755d9f498126d8f67d73f32ccd2b775c47c91ce26b66dfa59c25aed7f4a6bcb4786d3a3c6130f63ae08367822af3f967d3a7469e1b

You may have some red flags regarding the use of SHA1 or MD5. As of the writing of this book, there are currently no known vulnerabilities affecting HMAC-SHA1 and HMAC-MD5, but these functions are cryptographically weaker than SHA-256 and SHA-512. However, the more secure functions are also slower. The choice of which hash function to use comes down to prioritizing either performance or security.

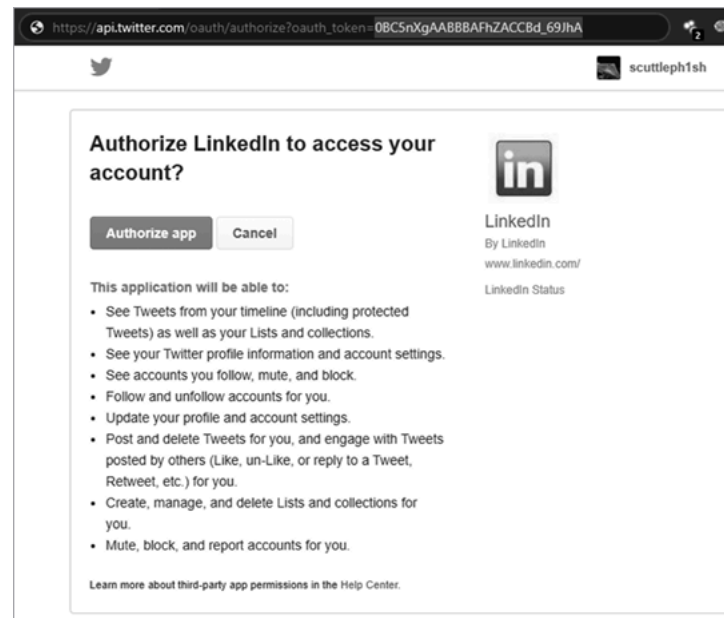
As with the previous authentication methods covered, the security of HMAC depends on the consumer and provider keeping the secret key private. If a secret

key is compromised, an attacker could impersonate the victim and gain unauthorized access to the API.

OAuth 2.0

OAuth 2.0, or just *OAuth*, is an authorization standard that allows different services to access each other's data, often using APIs to facilitate the service-to-service communications.

Let's say you want to automatically share your Twitter tweets on LinkedIn. In OAuth's model, we would consider Twitter to be the service provider and LinkedIn to be the application or client. In order to post your tweets, LinkedIn will need authorization to access your Twitter information. Since both Twitter and LinkedIn have implemented OAuth, instead of providing your credentials to the service provider and consumer every time you want to share this information across platforms, you can simply go into your LinkedIn settings and authorize Twitter. Doing so will send you to *api.twitter.com* to authorize LinkedIn to access your Twitter account (see [Figure 2-4](#)).



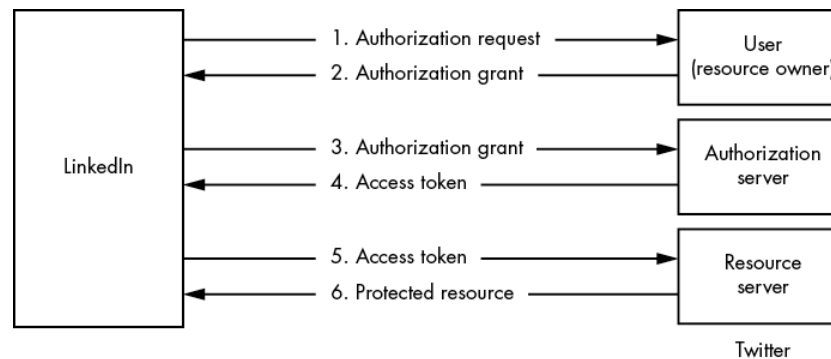
[Figure 2-4](#): LinkedIn-Twitter OAuth authorization request

When you authorize LinkedIn to access your Twitter posts, Twitter generates a limited, time-based access token for LinkedIn. LinkedIn then provides that token

to Twitter to post on your behalf, and you don't have to give LinkedIn your Twitter credentials.

[Figure 2-5](#) shows the general OAuth process. The user (*resource owner*) grants an application (the *client*) access to a service (the *authorization server*), the service creates a token, and then the application uses the token to exchange data with the service (also the *resource server*).

In the LinkedIn–Twitter example, you are the resource owner, LinkedIn is the application/client, and Twitter is the authorization server and resource server.



[Figure 2-5](#): An illustration of the OAuth process

OAuth is one of the most trusted forms of API authorization. However, while it adds security to the authorization process, it also expands the potential attack surface—although flaws often have more to do with how the API provider implements OAuth than with OAuth itself. API providers that poorly implement OAuth can expose themselves to a variety of attacks such as token injection, authorization code reuse, cross-site request forgery, invalid redirection, and phishing.

No Authentication

As in web applications generally, there are plenty of instances where it is valid for an API to have no authentication at all. If an API does not handle sensitive data and only provides public information, the provider could make the case that no authentication is necessary.

APIs in Action: Exploring Twitter's API

After reading this and the previous chapter, you should understand the various components running beneath the GUI of a web application. Let's now make these concepts more concrete by taking a close look at Twitter's API. If you open a web browser and visit the URL <https://twitter.com>, the initial request triggers a series of communications between the client and the server. Your browser automatically orchestrates these data transfers, but by using a web proxy like Burp Suite, which we'll set up in Chapter 4, you can see all the requests and responses in action.

The communications begin with the typical kind of HTTP traffic described in Chapter 1:

1. Once you've entered a URL into your browser, the browser automatically submits an HTTP GET request to the web server at *twitter.com*:

```
GET / HTTP/1.1
Host: twitter.com
User-Agent: Mozilla/5.0
Accept: text/html
--snip--
Cookie: [...]
```

2. The Twitter web application server receives the request and responds to the GET request by issuing a successful 200 OK response:

```
HTTP/1.1 200 OK
cache-control: no-cache, no-store, must-revalidate
connection: close
content-security-policy: content-src 'self'
content-type: text/html; charset=utf-8
server: tsa_a
--snip--
x-powered-by: Express
x-response-time: 56

<!DOCTYPE html>
<html dir="ltr" lang="en">
--snip--
```

This response header contains the status of the HTTP connection, client instructions, middleware information, and cookie-related information. *Client instructions* tell the browser how to handle the requested information, such as caching data, the content security policy, and instructions about the type of content that was sent. The actual payload begins just below `x-response-time`; it provides the browser with the HTML needed to render the web page. Now imagine that the user looks up “hacking” using Twitter’s search bar. This kicks off a POST request to Twitter’s API, as shown next. Twitter is able to leverage APIs to distribute requests and seamlessly provide requested resources to many users.

```
POST /1.1/jot/client_event.json?q=hacking HTTP/1.1
Host: api.twitter.com
User-Agent: Mozilla/5.0
--snip--
Authorization: Bearer AAAAAAAAAAAAAAAAAA...
--snip--
```

This POST request is an example of the Twitter API querying the web service at *api.twitter.com* for the search term “hacking.” The Twitter API responds with JSON containing the search results, which includes tweets and information about each tweet such as user mentions, hashtags, and post times:

```
"created_at": [...]
"id":1278533978970976256
"id_str": "1278533978970976256"
"full-text": "1984: William Gibson published his debut novel..."
"truncated":false,
--snip--
```

The fact that the Twitter API seems to adhere to CRUD, API naming conventions, tokens for authorization, *application/x-www-form-urlencoded*, and JSON as a data interchange makes it pretty clear that this API is a RESTful API. Although the response body is formatted in a legible way, it’s meant to be processed by the browser to be displayed as a human-readable web page. The browser renders the search results using the string from the API request. The provider’s response then populates the page with search results, images, and social media–related information such as likes, retweets, comments (see [Figure 2-6](#)).

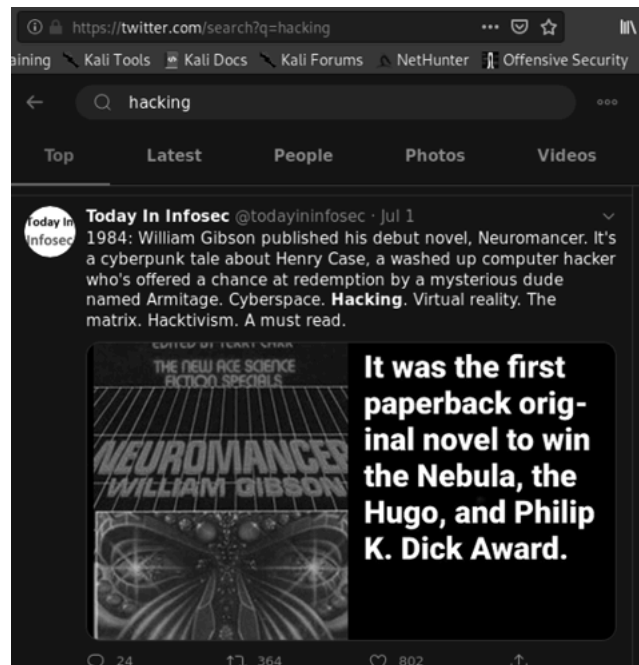


Figure 2-6: The rendered result from the Twitter API search request

From the end user's perspective, the whole interaction appears seamless: you click the search bar, type in a query, and receive the results.

Summary

In this chapter, we covered the terminology, parts, types, and supporting architecture of APIs. You learned that APIs are interfaces for interacting with web applications. Different types of APIs have different rules, functions, and purposes, but they all use some kind of format for exchanging data between applications. They often use authentication and authorization schemes to make sure consumers can access only the resources they're supposed to.

Understanding these concepts will prepare you to confidently strike at the components that make up APIs. As you continue to read, refer to this chapter if you encounter API concepts that confuse you.