Chapter 3. The Structure of a Modern Web Application

Before you can effectively evaluate a web application for recon purposes, it is best to gain an understanding of the common technologies that many web applications share as dependencies. These dependencies span from JavaScript helper libraries and predefined CSS modules, all the way to web servers and even operating systems. By understanding the role of these dependencies and their common implementations in an applications stack, it becomes much easier to quickly identify them and look for misconfigurations.

Modern Versus Legacy Web Applications

Today's web applications are often built on top of technology that didn't exist 10 years ago. The tools available for building web applications have advanced so much in that time frame that sometimes it seems like an entirely different specialization today.

A decade ago, most web applications were built using server-side frameworks that rendered an HTML/JS/CSS page that would then be sent to the client. Upon needing an update, the client would simply request another page from the server to be rendered and piped over HTTP. Shortly after that, web applications began making use of HTTP more frequently with the rise of AJAX (asynchronous JavaScript and XML), allowing network requests to be made from within a page session via JavaScript.

Today, many applications actually are more properly represented as two or more applications communicating via a network protocol versus a sin-

gle monolithic application. This is one major architectural difference between the web applications of today and the web applications of a decade ago.

Oftentimes, today's web applications are composed of several applications connected with a Representational State Transfer (REST) API. These APIs are stateless and only exist to fulfill requests from one application to another. This means they don't actually store any information about the requester.

Many of today's client (UI) applications run in the browser in ways more akin to a traditional desktop application. These client applications manage their own life cycle loops, request their own data, and do not require a page reload after the initial bootstrap is complete.

It is not uncommon for a standalone application deployed to a web browser to communicate with a multitude of servers. Consider an image hosting application that allows user login—it likely will have a specialized hosting/distribution server located at one URL and a separate URL for managing the database and logins.

It's safe to say that today's applications are often actually a combination of many separate but symbiotic applications working together in unison. This can be attributed to the development of more cleanly defined network protocols and API architecture patterns.

The average modern-day web application probably makes use of several of the following technologies:

- REST API
- JSON or XML
- JavaScript
- SPA framework (ReactJS, VueJS, EmberJS, AngularJS)
- An authentication and authorization system
- One or more web servers (typically on a Linux server)
- One or more web server software packages (ExpressJS, Apache, NGINX)
- One or more databases (MySQL, MongoDB, etc.)

- A local data store on the client (cookies, web storage, IndexedDB)

NOTE

This is not an exhaustive list, and considering there are now billions of individual websites on the internet, it is not feasible to cover all web application technologies in this book. You should make use of other books and coding websites, like Stack Overflow, if you need to get up to speed with a specific technology not listed in this chapter.

Some of these technologies existed a decade ago, but it wouldn't be fair to say they have not changed in that time frame. Databases have been around for decades, but NoSQL databases and client-side databases are definitely a more recent development. The development of full stack JavaScript applications was also not possible until Node.js and npm began to see rapid adoption. The landscape for web applications has been changing so rapidly in the last decade or so that many of these technologies have gone from unknown to ubiquitous.

There are even more technologies on the horizon: for example, the Cache API for storing requests locally and WebSockets as an alternative network protocol for client-to-server (or even client-to-client) communication. Eventually, browsers intend to fully support a variation of assembly code known as web assembly, which will allow non-JavaScript languages to be used for writing client-side code in the browser.

Each of these new and upcoming technologies brings with it new security holes to be found and exploited for good or for evil. It is an exciting time to be in the business of exploiting or securing web applications.

Unfortunately, I cannot explain every technology in use on the web today—that would require its own book! But the remainder of this chapter will give an introduction to the technologies listed previously. Feel free to focus on the ones you are not yet intimately familiar with.

REST APIs

REST stands for *Representational State Transfer*, which is a fancy way of defining an API that has a few unique traits:

It must be separate from the client

REST APIs are designed for building highly scalable, but simple, web applications. Separating the client from the API but following a strict API structure makes it easy for the client application to request resources from the API without being able to make calls to a database or perform server-side logic itself.

It must be stateless

By design, REST APIs only take inputs and provide outputs. The APIs must not store any state regarding the client's connection. This does not mean, however, that a REST API cannot perform authentication and authorization—instead, authorization should be tokenized and sent on every request.

It must be easily cacheable

To properly scale a web application delivered over the internet, a REST API must be able to easily mark its responses as cacheable or not. Because REST also includes very tight definitions on what data will be served from what endpoint, this is actually very easy to configure on a properly designed REST API. Ideally, the caches should be programmatically managed to not accidentally leak privileged information to another user.

Each endpoint should define a specific object or method

Typically these are defined hierarchically; for example, `/moderators/joe/logs/12_21_2018`. In doing so, REST APIs can easily make use of HTTP verbs like GET, POST, PUT, and DELETE. As a result, one endpoint with multiple HTTP verbs becomes self-documenting.

Do you want to modify the moderator account “joe”? Use `PUT /moderators/joe`. Want to delete the `12_21_2018` log? All that takes is a simple deduction: `DELETE /moderators/joe/logs/12_21_2018`.

Because REST APIs follow a well-defined architectural pattern, tools like Swagger can easily integrate into an application and document the endpoints so it is easier for other developers to pick up an endpoint’s intentions (see Figure 3-1).

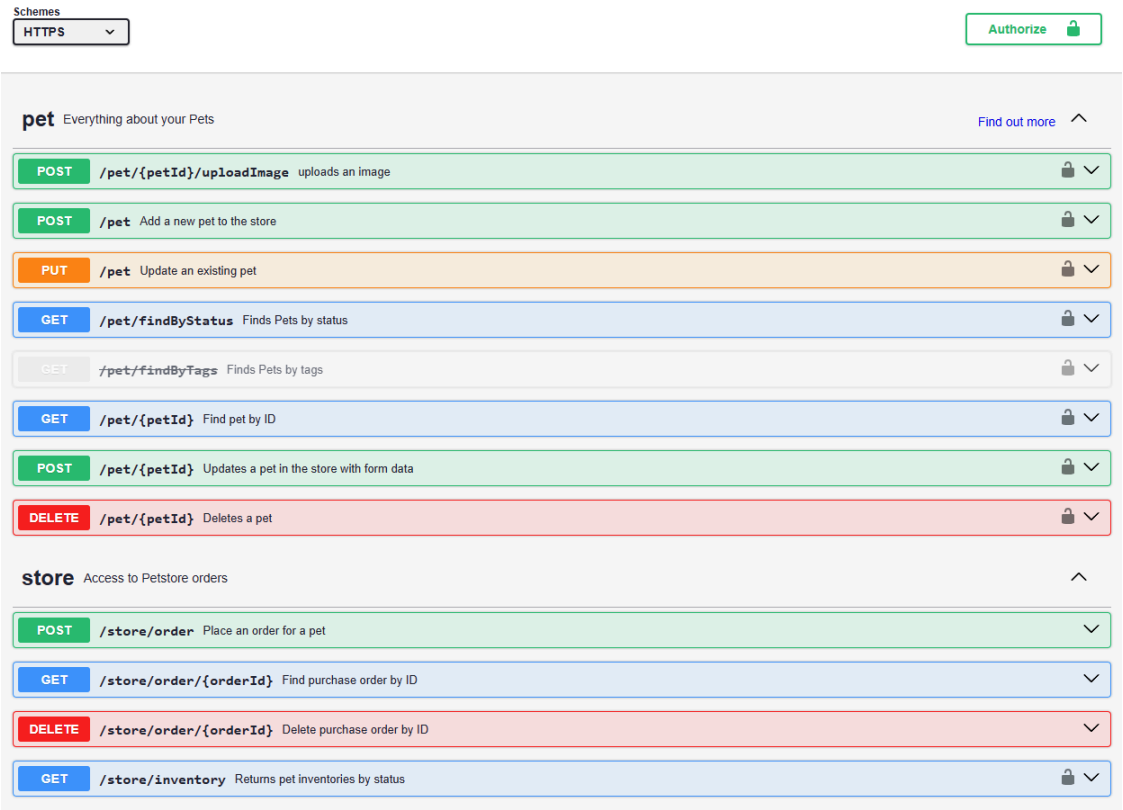

Figure 3-1. Swagger, an automatic API documentation generator designed for easy integration with REST APIs

In the past, most web applications used *Simple Object Access Protocol* (SOAP)-structured APIs. REST has several advantages over SOAP:

- Requests target data, not functions
- Easy caching of requests
- Highly scalable

Furthermore, while SOAP APIs must utilize XML as their in-transit data format, REST APIs can accept any data format, but typically JSON is used. JSON is much more lightweight (less verbose) and easier for humans to read than XML, which also gives REST an edge against the competition.

Here is an example payload written in XML:

```
<user>
 <username>joe</username>
 <password>correcthorsebatterystaple</password>
 <email>joe@website.com</email>
 <joined>12/21/2005</joined>
 <client-data>
  <timezone>UTF</timezone>
  <operating-system>Windows 10</operating-system>
  <licenses>
   <videoEditor>abc123-2005</videoEditor>
   <imageEditor>123-456-789</imageEditor>
  </licenses>
 </client-data>
</user>
```

And similarly, the same payload written in JSON:

```
{
 "username": "joe",
 "password": "correcthorsebatterystaple",
 "email": "joe@website.com",
 "joined": "12/21/2005",
 "client_data": {
  "timezone": "UTF",
  "operating_system": "Windows 10",
  "licenses": {
   "videoEditor": "abc123-2005",
   "imageEditor": "123-456-789"
  }
 }
}
```

Most modern web applications you will run into either make use of RESTful APIs or a REST-like API that serves JSON. It is becoming increasingly rare to encounter SOAP APIs and XML outside of specific enterprise apps that maintain such rigid design for legacy compatibility.

Understanding the structure of REST APIs is important as you attempt to reverse engineer a web application's API layer. Mastering the basic fundamentals of REST APIs will give you an advantage, as you will find that many APIs you wish to investigate follow REST architecture—but additionally, many tools you may wish to use or integrate your workflow with will be exposed via REST APIs.

JavaScript Object Notation

REST is an architecture specification that defines how HTTP verbs should map to resources (API endpoints and functionality) on a server. Most REST APIs today use JSON as their in-transit data format.

Consider this: an application's API server must communicate with its client (usually some code in a browser or mobile app). Without a client/server relationship, we cannot have stored state across devices and persist that state between accounts. All states would have to be stored locally.

Because modern web applications require a lot of client/server communication (for the downstream exchange of data and upstream requests in the form of HTTP verbs), it is not feasible to send data in ad hoc formats. The in-transit format of the data must be standardized.

JSON is one potential solution to this problem. JSON is an open standard (not proprietary) file format that meets a number of interesting requirements:

- It is very lightweight (reduces network bandwidth).
- It requires very little parsing (reduces server/client hardware load).
- It is easily human readable.
- It is hierarchical (can represent complex relationships between data).
- JSON objects are represented very similarly to JavaScript objects, making consumption of JSON and building new JSON objects quite easy in the browser.

All major browsers today support the parsing of JSON natively (and fast!), which, in addition to the preceding bullet points, makes JSON a great format for transmitting data between a stateless server and a web browser.

The following JSON:

```
{
"first": "Sam",
"last": "Adams",
"email": "sam.adams@company.com",
"role": "Engineering Manager",
"company": "TechCo.",
"location": {
  "country": "USA",
  "state": "california",
  "address": "123 main st.",
  "zip": 98404
  }
}
```

can be parsed easily into a JavaScript object in the browser:

```
const jsonString = `{
 "first": "Sam",
 "last": "Adams",
 "email" "sam.adams@company.com",
 "role": "Engineering Manager",
 "company": "TechCo.",
 "location": {
  "country": "USA",
  "state": "california",
  "address": "123 main st.",
  "zip": 98404
 }
}`;

// convert the string sent by the server to an object
const jsonObject = JSON.parse(jsonString);
```

JSON is flexible, lightweight, and easy to use. It is not without its drawbacks, as any lightweight format has trade-offs compared to heavyweight alternatives. These will be discussed later in the book when we evaluate specific security differences between JSON and its competitors, but for now it's important to just grasp that today, a significant number of network requests between browsers and servers are sent as JSON.

Get familiar with reading through JSON strings, and consider installing a plug-in in your browser or code editor to format JSON strings. Being able to rapidly parse these and find specific keys will be very valuable when penetration testing a wide variety of APIs in a short time frame.

JavaScript

Throughout this book we will continually discuss client and server applications. A *server* is a computer (typically a powerful one) that resides in a data center (sometimes called *the cloud*) and is responsible for handling requests to a website. Sometimes these servers will actually be a cluster of many servers; other times it might just be a single lightweight server used for development or logging.

A *client*, on the other hand, is any device a user has access to that they manipulate to use a web application. A client could be a mobile phone, a mall kiosk, or a touch screen in an electric car—but for our purposes it will usually just be a web browser.

Servers can be configured to run almost any software you could imagine, in any language you could imagine. Web servers today run on Python, Java, JavaScript (JS), C++, etc. Clients (in particular, the browser) do not have that luxury. JavaScript is a dynamic programming language that was originally designed for use in internet browsers. JavaScript is not only a programming language but also the sole programming language for client-side scripting in web browsers. JavaScript is now used in many applications, from mobile to the internet of things, or IoT.

Many code examples throughout this book are written in JavaScript (see Figure 3-2). When possible, the backend code examples are written using

a JavaScript syntax as well so that no time is wasted in context switching. I'll try to keep the JavaScript as clean and simple as possible, but I may use some constructs that JavaScript supports that are not as popular (or well known) in other languages.

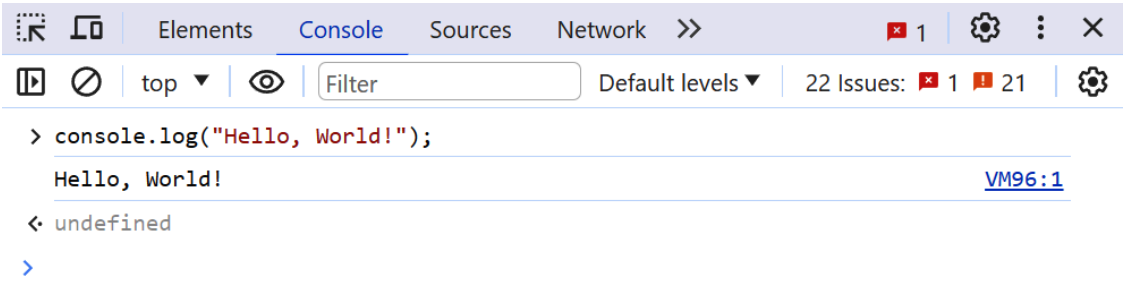

Figure 3-2. JavaScript example

JavaScript is a unique language as development is tied to the growth of the browser and its partner, the Document Object Model (DOM). Because of this, there are some quirks you might want to be aware of before moving forward.

Variables and Scope

In ES6 JavaScript (a recent version), there are four ways to define a variable:

```
// global definition
age = 25;

// function scoped
var age = 25;

// block scoped
let age = 25;

// block scoped, without reassignment
const age = 25;
```

These may all appear similar, but they are functionally very different.

Without including a keyword like `var`, `let`, or `const`, any variable you define will get hoisted into global scope. This means that any other object

defined as a child of the global scope will be able to access that variable. Generally speaking, this is considered a bad practice and we should stay away from it. (It could also be the cause of significant security vulnerabilities or functional bugs.)

Note that all variables lacking `var`, `let`, or `const` will also have a pointer added to the `window` object in the browser:

```
// define global integer
age = 25;

// direct call (returns 25)
console.log(age);

// call via pointer on window (returns 25)
console.log(window.age);
```

This, of course, can cause namespacing conflicts on `window` (an object the browser DOM relies on to maintain window state), which is another good reason to avoid it:

`var age = 25`

Any variable defined with the identifier `var` is scoped to the nearest function or globally if there is no outer function block defined (in the global case, it appears on `window` similarly to an identifierless variable, as shown previously).

This type of variable is a bit confusing, which is probably part of the reason `let` was eventually introduced.

```
const func = function() {
 if (true) {
  // define age inside of if block
  var age = 25;
 }

 /*
  * logging age will return 25
  *
```

```
   * this happens because the var identifier binds to the nearest
   * function, rather than the nearest block.
   */
  console.log(age);
};
```

In the preceding example, a variable is defined using the `var` identifier with a value of `25`. In most other modern programming languages, age would be undefined when trying to log it.
Unfortunately, `var` doesn't follow these general rules and scopes itself to functions rather than blocks. This can lead new JavaScript developers down a road of debugging confusion.

`let age = 25`

ECMAScript 6 (a specification for JavaScript) introduced `let` and `const`—two ways of instantiating an object that act more similarly to those in other modern languages.

As you would expect, `let` is block scoped. That means:

```
const func = function() {
  if (true) {
    // define age inside of if block
    let age = 25;
  }

  /*
   * This time, console.log(age) will return `undefined`.
   *
   * This is because `let`, unlike `var`, binds to the nearest block.
   * Binding scope to the nearest block rather than the nearest function
   * is generally considered to be better for readability, and
   * results in a reduction of scope-related bugs.
   */
  console.log(age);
};
```

`const age = 25`

`const`, much like `let`, is also block scoped, but also cannot be reassigned. This makes it similar to a `final` variable in a language

like Java:

```
const func = function() {
  const age = 25;

  /*
   * This will result in: TypeError: invalid assignment to const `age`
   *
   * Much like `let`, `const` is block scoped.
   * The major difference is that `const` variables do not support
   * reassignment after they are instantiated.
   *
   * If an object is declared as a const, its properties can still be
   * changed. As a result, `const` ensures the pointer to `age` in memory
   * is not changed, but does not care if the value of `age` or a property
   * on `age` changes.
   */
  age = 25;
};
```

In general, you should always strive to use `let` and `const` in your code to avoid bugs and improve readability.

Functions

In JavaScript, functions are objects. That means they can be assigned and reassigned using the variables and identifiers from the last section.

These are all functions:

```
// anonymous function
function() {};

// globally declared named function
a = function() {};

// function scoped named function
var a = function() { };

// block scoped named function
```

```
let a = function() {};

// block scoped named function without reassignment
const a = function() {};

// anonymous function inheriting parent context
() => {};

// immediately invoked function expression (IIFE)
(function() { })();
```

The first function is an `anonymous` function—that means it can't be referenced after it is created. The next four are simply functions with scope specified based on the identifier provided. This is very similar to how we previously created variables for age. The sixth function is a shorthand function—it shares `context` with its parent (more on that soon).

The final function is a special type of function you will probably only find in JavaScript, known as an IIFE—immediately invoked function expression. This is a function that fires immediately when loaded and runs inside of its own namespace. These are used by more advanced JavaScript developers to encapsulate blocks of code from being accessible elsewhere.

Context

If you can write code in any other non-JavaScript language, there are five things you will need to learn to become a good JavaScript developer: scope, context, prototypal inheritance, asynchrony, and the browser DOM.

Every function in JavaScript has its own set of properties and data attached to it. We call these the function's *context*. Context is not set in stone and can be modified during runtime. Objects stored in a function's context can be referenced using the keyword `this`:

```
const func = function() {
  this.age = 25;
```

```
  // will return 25
  console.log(this.age);
};

// will return undefined
console.log(this.age);
```

As you can imagine, many annoying programming bugs are a result of context being hard to debug—especially when some object's context has to be passed to another function. JavaScript introduced a few solutions to this problem to aid developers in sharing context between functions:

```
// create a new getAge() function clone with the context from ageData
// then call it with the param 'joe'
const getBoundAge = getAge.bind(ageData)('joe');

// call getAge() with ageData context and param joe
const boundAge = getAge.call(ageData, 'joe');

// call getAge() with ageData context and param joe
const boundAge = getAge.apply(ageData, ['joe']);
```

These three functions, `bind`, `call`, and `apply`, allow developers to move context from one function to another. The only difference between `call` and `apply` is that `call` takes a list of arguments, and `apply` takes an array of arguments.

The two can be interchanged easily:

```
// destructure array into list
const boundAge = getAge.call(ageData, ...['joe']);
```

Another new addition to aid programmers in managing context is the arrow func- tion, also called the shorthand function. This function inherits context from its parent, allowing context to be shared from a parent function to the child without requiring explicit calling/applying or binding:

```
// global context
this.garlic = false;

// soup recipe
const soup = { garlic: true };

// standard function attached to soup object
soup.hasGarlic1 = function() { console.log(this.garlic); } // true

// arrow function attached to global context
soup.hasGarlic2 = () => { console.log(this.garlic); } // false
```

Mastering these methods of managing context will make reconnaissance through a JavaScript-based server or client much easier and faster. You might even find some language-specific vulnerabilities that arise from these complexities.

Prototypal Inheritance

Unlike many traditional server-side languages that suggest using a class-based inheritance model, JavaScript has been designed with a highly flexible prototypal inheritance system. Unfortunately, because few languages make use of this type of inheritance system, it is often disregarded by developers, many of whom try to convert it to a class-based system.

In a class-based system, `classes` operate like blueprints defining objects. In such systems, `classes` can `inherit` from other classes and create hierarchical relationships in this manner. In a language like Java, subclasses are generated with the `extends` keyword or instanced with the `new` keyword.

JavaScript does not truly support these types of classes, but because of how flexible prototypal inheritance is, it is possible to mimic the exact functionality of classes with some abstraction on top of JavaScript's prototype system. In a prototypal inheritance system, like in JavaScript, any object created has a property attached to it called `prototype`. The `prototype` property comes with a `constructor` property attached that points back to the function that owns the `prototype`. This means that

any object can be used to instantiate new objects since the constructor points to the object that contains the prototype containing the constructor.

This may be confusing, but here is an example:

```
/*
 * A vehicle pseudoclass written in JavaScript.
 *
 * This is simple on purpose, in order to more clearly demonstrate
 * prototypal inheritance fundamentals.
 */
const Vehicle = function(make, model) {
 this.make = make;
 this.model = model;

 this.print = function() {
  return `${this.make}: ${this.model}`;
 };
};

const prius = new Vehicle('Toyota', 'Prius');
console.log(prius.print());
```

When a new object is created in JavaScript, a separate object called `__proto__` is also created. This object points to the prototype whose constructor was invoked during the creation of that object. This allows for comparison between objects, for example:

```
const prius = new Vehicle('Toyota', 'Prius');
const charger = new Vehicle('Dodge', 'Charger');

/*
 * As we can see, the "Prius" and "Charger" objects were both
 * created based off of "Vehicle".
 */
prius.__proto__ === charger.__proto__;
```

Oftentimes, the `prototype` on an object will be modified by developers, leading to confusing changes in web application functionality. Most no-

tably, because all objects in JavaScript are mutable by default, a change to `prototype` properties can happen at any time during runtime.

Interestingly, this means that unlike in more rigidly designed inheritance models, JS inheritance trees can change at runtime. Objects can morph at runtime as a result:

```
const prius = new Vehicle('Toyota', 'Prius');
const charger = new Vehicle('Dodge', 'Charger');

/*
 * This will fail because the Vehicle object
 * does not have a "getMaxSpeed" function.
 *
 * Hence, objects inheriting from Vehicle do not have such a function
 * either.
 */
 console.log(prius.getMaxSpeed()); // Error: getMaxSpeed is not a function

 /*
  * Now we will assign a getMaxSpeed() function to the prototype of Vehicle,
  * all objects inheriting from Vehicle will be updated in real time as
  * prototypes propagate from the Vehicle object to its children.
  */
  Vehicle.prototype.getMaxSpeed = function() {
    return 100; // mph
  };

  /*
   * Because the Vehicle's prototype has been updated, the
   * getMaxSpeed function will now function on all child objects.
   */
  prius.getMaxSpeed(); // 100
  charger.getMaxSpeed(); // 100
```

Prototypes take a while to get used to, but eventually their power and flexibility outweigh any difficulties present in the learning curve. Prototypes are especially important to understand when delving into JavaScript security because few developers fully understand them.

Additionally, because prototypes propagate to children when modified, a special type of attack is found in JavaScript-based systems called *Prototype Pollution*. This attack involves modification to a parent JavaScript object, which unintentionally changes the functionality of child objects.

Asynchrony

Asynchrony is one of those "hard to figure out, easy to remember" concepts that seem to come along frequently in network programming. Because browsers must communicate with servers on a regular basis, and the time between request and response is nonstandard (factoring in payload size, latency, and server processing time), asynchrony is used often on the web to handle such variation.

In a synchronous programming model, operations are performed in the order they occur. For example:

```
console.log('a');
console.log('b');
console.log('c');
// a
// b
// c
```

In the preceding case, the operations occur in order, reliably spelling out "abc" every time these three functions are called in the same order.

In an asynchronous programming model, the three functions may be read in the same order by the interpreter each time but might not resolve in the same order. Consider this example, which relies on an asynchronous logging function:

```
// --- Attempt #1 ---
async.log('a');
async.log('b');
async.log('c');
// a
```

```
// b
// c

// --- Attempt #2 ---
async.log('a');
async.log('b');
async.log('c');
// a
// c
// b

// --- Attempt #3 ---
async.log('a');
async.log('b');
async.log('c');
// a
// b
// c
```

The second time the logging functions were called, they didn't resolve in order. Why? With network programming, requests often take variable amounts of time, time out, and operate unpredictably. In JavaScript-based web applications, this is often handled via asynchronous programming models rather than simply waiting for a request to complete before initiating another. The benefit is a massive performance improvement that can be dozens of times faster than the synchronous alternative. Instead of forcing requests to complete one after another, we initiate them all at the same time and then program what they should do upon resolution—prior to resolution occurring.

In older versions of JavaScript, this was usually done with a system called `callbacks`:

```
const config = {
  privacy: public,
  acceptRequests: true
};

/*
 * First request a user object from the server.
```

```
 * Once that has completed, request a user profile from the server.
 * Once that has completed, set the user profile config.
 * Once that has completed, console.log "success!"
 */
getUser(function(user) {
  getUserProfile(user, function(profile) {
    setUserProfileConfig(profile, config, function(result) {
      console.log('success!');
    });
  });
});
```

While callbacks are extremely fast and efficient, compared to a synchronous model, they are very difficult to read and debug. A later programming philosophy suggested creating a reusable object that would call the next function once a given function completed. These are called `promises`, and they are used in many programming languages today:

```
const config = {
  privacy: public,
  acceptRequests: true
};

/*
 * First request a user object from the server.
 * Once that has completed, request a user profile from the server.
 * Once that has completed, set the user profile config.
 * Once that has completed, console.log "success!"
 */
const promise = new Promise((resolve, reject) => {
  getUser(function(user) {
    if (user) { return resolve(user); }
    return reject();
  });
}).then((user) => {
  getUserProfile(user, function(profile) {
   if (profile) { return resolve(profile); }
   return reject();
  });
}).then((profile) => {
  setUserProfile(profile, config, function(result) {
   if (result) { return resolve(result); }
```

```
    return reject();
  });
}).catch((err) => {
  console.log('an error occurred!');
});
```

Both of the preceding examples accomplish the same exact application logic. The difference is in readability and organization. The promise-based approach can be broken up further, growing vertically instead of horizontally and making error handling much easier. Promises and callbacks are interoperable and can be used together, depending on programmer preference.

The latest method of dealing with asynchrony is the `async` function. Unlike normal `function` objects, these functions are designed to make dealing with asynchrony a cakewalk.

Consider the following `async` function:

```
const config = {
  privacy: public,
  acceptRequests: true
};

/*
 * First request a user object from the server.
 * Once that has completed, request a user profile from the server.
 * Once that has completed, set the user profile config.
 * Once that has completed, console.log "success!"
 */
const setUserProfile = async function() {
  let user = await getUser();
  let userProfile = await getUserProfile(user);
  let setProfile = await setUserProfile(userProfile, config);
};

setUserProfile();
```

You may notice this is so much easier to read. Great, that's the point!

`async` functions turn functions into promises. Any method call inside of a promise with `await` before it will halt further execution within that function until the method call resolves. Code outside of the `async` function can still operate normally.

Essentially, the `async` function turns a normal function into a `promise`. You will see these more and more in client-side code and JavaScript-based server-side code as time goes on.

Browser DOM

You should now have sufficient understanding of asynchronous programming—the model that is dominant on the web and in client/server applications. With that information in your mind, the final JavaScript-related concept you should be aware of is the browser DOM.

The DOM is the hierarchical representation data used to manage state in modern web browsers. Figure 3-3 shows the `window` object, one of the topmost standard objects defined by the DOM specification.

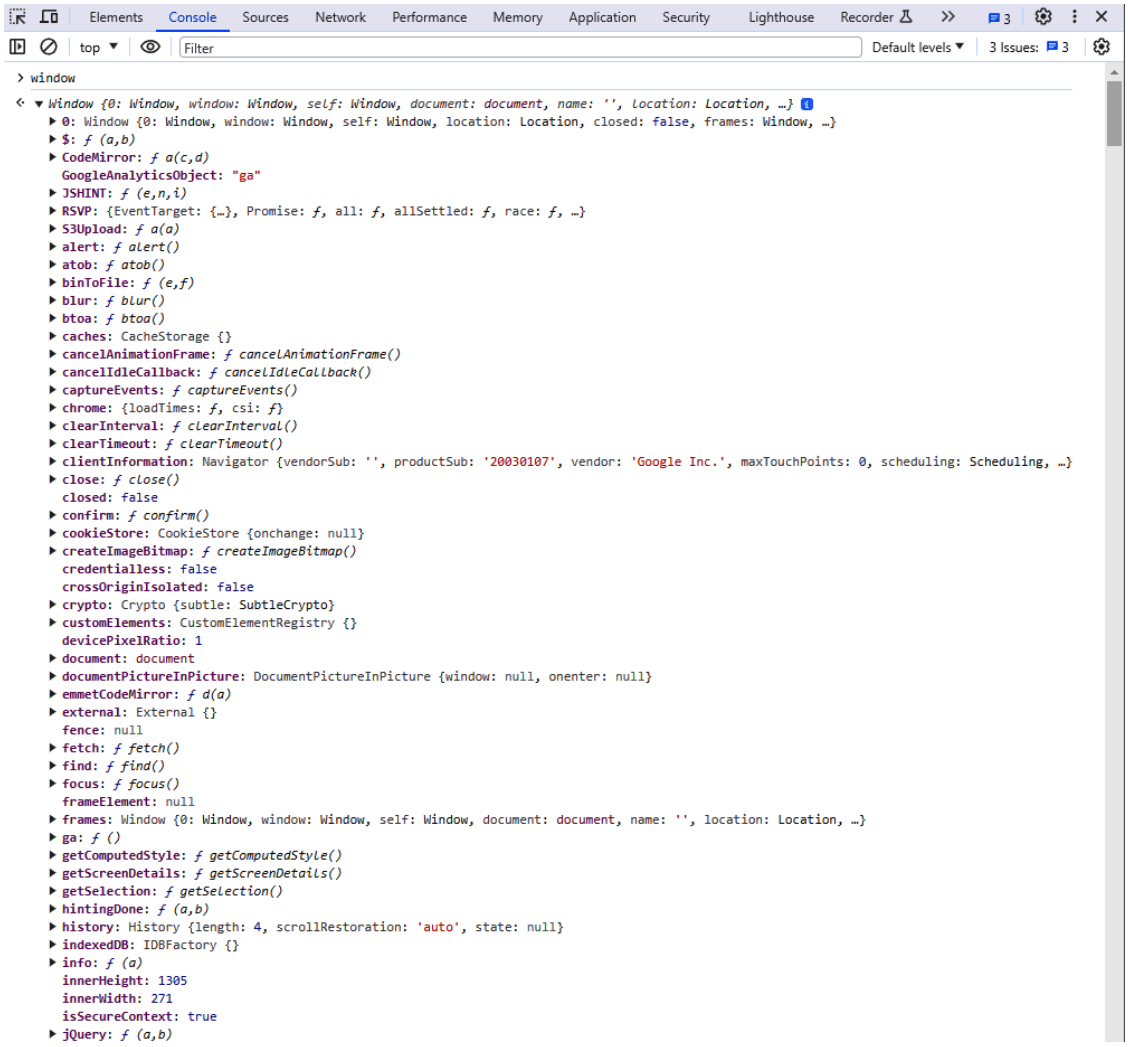

Figure 3-3. The DOM `window` object

JavaScript is a programming language, and like any good programming language, it relies on a powerful standard library. This library, unlike standard libraries in other languages, is known as the DOM. The DOM provides routine functionality that is well tested and performant, and is implemented across all major browsers, so your code *should* function identically or nearly identically regardless of the browser it is run on.

Unlike other standard libraries, the DOM exists not to plug functionality holes in the language or provide common functionality (that is a secondary function of the DOM) but mainly to provide a common interface from which to define a hierarchical tree of nodes that represents a web page. You may have accidentally called a DOM function and assumed it was a JS function. Examples of this are `document.querySelector()` and `document.implementation`.

The main objects that make up the DOM are `window` and `document`. Each is carefully defined in a specification maintained by an organization called WHATWG.

Regardless of if you are a JavaScript developer, web application pen tester, or security engineer, developing a deep understanding of the browser DOM and its role in a web application is crucial to spotting vulnerabilities that become evident at the presentation layer in an application. Consider the DOM to be the framework from which JavaScript-based applications are deployed to end users. Keep in mind that not all script-related security holes will be the result of improper JavaScript; they can sometimes result from improper browser DOM implementation.

SPA Frameworks

Older websites were usually built on a combination of ad hoc scripts to manipulate the DOM and a lot of reused HTML template code. This was not a scalable model, and while it worked for delivering static content to an end user, it did not work for delivering complex, logic-rich applications.

Desktop application software at the time was robust in functionality, allowing for users to store and maintain application state. Websites in the old days did not provide this type of functionality, although many companies would have preferred to deliver their complex applications via the web as it provided many benefits from ease of use to piracy prevention.

Single-page application (SPA) frameworks were designed to bridge the functionality gap between websites and desktop applications. SPA frameworks allow for the development of complex JavaScript-based applications that store their own internal state and are composed of reusable UI components, each of which has its own self-maintained life cycle, from rendering to logic execution.

SPA frameworks are rampant on the web today, backing the largest and most complex applications (such as Facebook and YouTube) where functionality is key and near-desktop-like application experiences are deliv-

ered. Some of the largest open source SPA frameworks today are ReactJS, EmberJS, VueJS, and AngularJS (Figure 3-4). These are all built on top of JavaScript and the DOM but bring with them added complexity from both security and functionality perspectives.

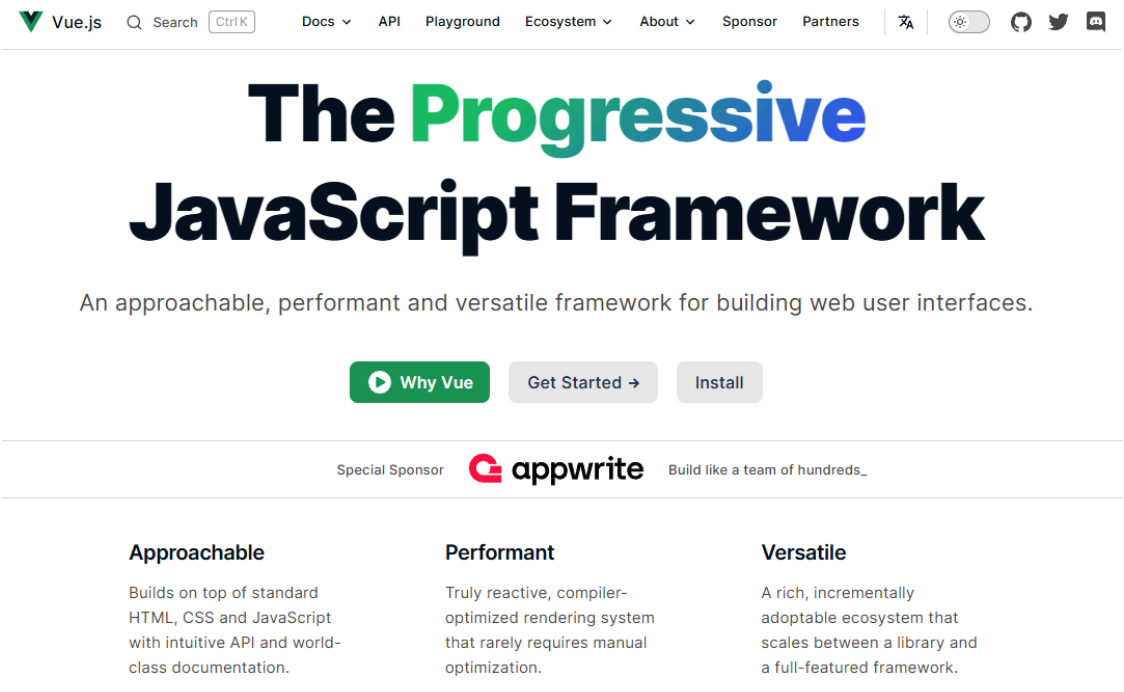

Figure 3-4. VueJS, a popular SPA framework that builds on top of web components

Authentication and Authorization Systems

In a world where most applications consist of both clients (browsers/phones) and servers, and servers persist data originally sent from a client, systems must be in place to ensure that future access of persisted data comes from the correct user. We use the term *authentication* to describe a flow that allows a system to *identify* a user. In other words, authentication systems tell us that "joe123" is actually "joe123" and not "susan1988."

The term *authorization* is used to describe a flow inside a system for determining what resources "joe123" has access to, as opposed to "susan1988." For example, "joe123" should be able to access his own uploaded private photos, and "susan1988" should be able to access hers, but they should not be able to access each other's photos.

Both processes are critical to the functionality of a web application, and both are functions in a web application where proper security controls are critical.

Authentication

Early authentication systems were simple in nature. For example, HTTP basic authentication performs authentication by attaching an Authorization header on each request. The header consists of a string containing `Basic: <base64-encoded username:password>`. The server receives the username:password combination and, on each request, checks it against the database. Obviously, this type of authentication scheme has several flaws—for example, it is very easy for the credentials to be leaked in a number of ways, from compromised WiFi over HTTP to simple XSS attacks.

Later authentication developments include digest authentication, which employs cryptographic hashes instead of base64 encoding. After digest authentication, a multitude of new techniques and architectures popped up for authentication, including those that do not involve passwords and those that require external devices.

Today, most web applications choose from a suite of authentication architectures, depending on the nature of the business. For example, the OAuth protocol is great for websites that want to integrate with larger websites. OAuth allows for a major website (such as Facebook, Google, etc.) to provide a token verifying a user's identity to a partner website. OAuth can be useful to a user because the user's data only needs to be updated on one site rather than on multiple sites—but OAuth can be dangerous because one compromised website can result in multiple compromised profiles.

HTTP basic authentication and digest authentication are still used widely today, with digest being more popular as it has more defenses against interception and replay attacks. Often these are coupled with tools like multifactor authentication (MFA) to ensure that authentication tokens are not compromised, and that the identity of the logged-in user has not changed.

Authorization

Authorization is the next step after authentication. Authorization systems are more difficult to categorize, as authorization very much depends on the business logic inside of the web application. Generally speaking, well-designed applications have a centralized authorization class that is responsible for determining if a user has access to certain resources or functionality.

If APIs are poorly written, they will implement checks on a per-API basis, which manually reproduce authorization functionality. Oftentimes, if you can tell that an application reimplements authorization checks in each API, that application will likely have several APIs where the checks are not sufficient simply due to human error.

Some common resources that should always have authorization checks include settings/profile updates, password resets, private message reads/writes, any paid functionality, and any elevated user functionality (such as moderation functions).

Web Servers

A modern client/server web application relies on a number of technologies built on top of each other for the server-side component and client-side components to function as intended. In the case of the server, application logic runs on top of a software-based web server package so that application developers do not have to worry about handling requests and managing processes. The web server software, of course, runs on top of an operating system (usually some Linux distro like Ubuntu, CentOS, or Red Hat), which runs on top of physical hardware in a data center somewhere.

As far as web server software goes, there are a few big players in the modern web application world. Apache still serves nearly half of the websites in the world, so we can assume Apache serves the majority of web applications as well. Apache is open source, has been in development for

almost 30 years, and runs on almost every Linux distro, as well as some Windows servers (see Figure 3-5).

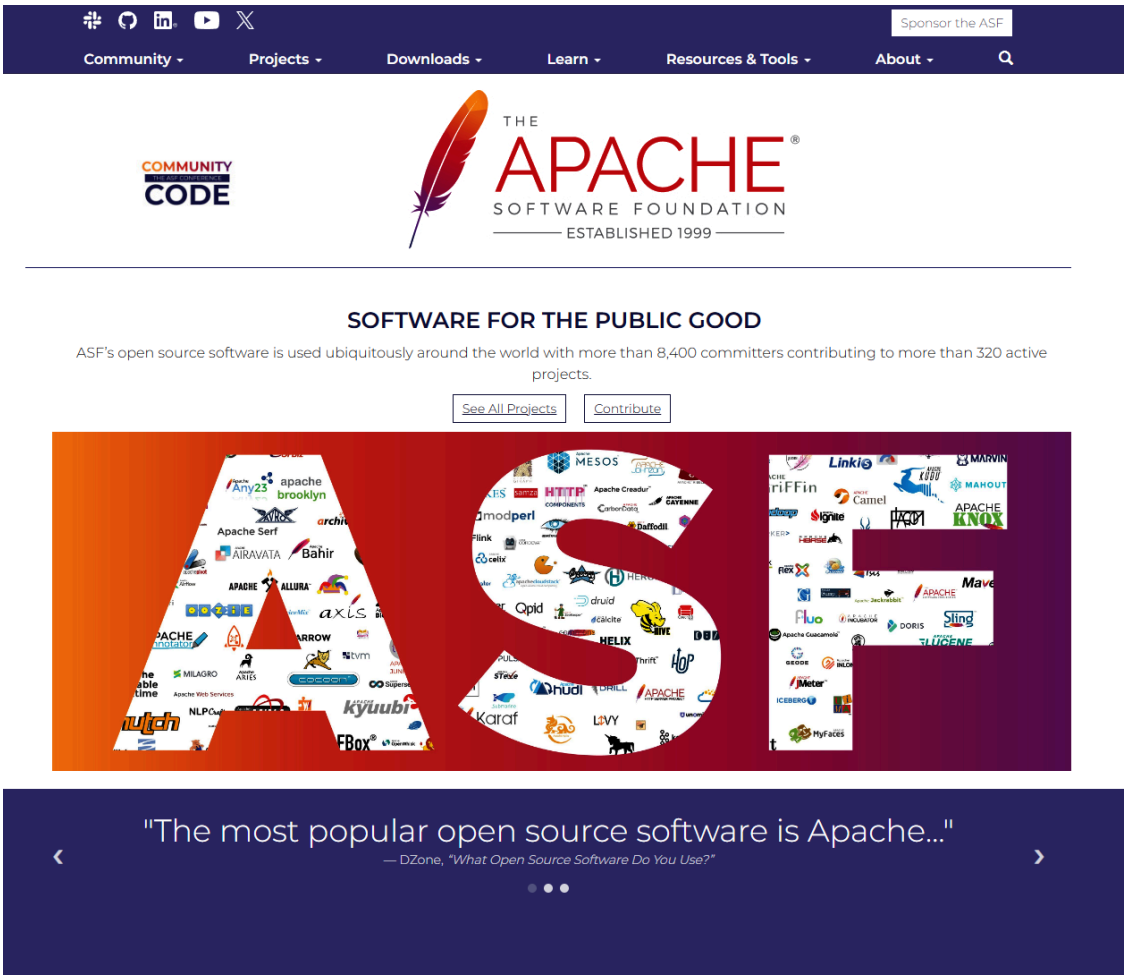

Figure 3-5. Apache, one of the largest and most frequently implemented web server software packages, has been in development since 1995

Apache is great not only due to its large community of contributors and open source nature, but also because of how easily configurable and pluggable it has become. It's a flexible web server that you will likely see for a long time.

Apache's biggest competitor is NGINX (pronounced "Engine X"). NGINX runs around 30% of web servers and is growing rapidly. Although NGINX can be used for free, its parent company (currently F5 Networks) uses a paid+ model where support and additional functionality come at a cost.

NGINX is used for high-volume applications with a large number of unique connections, as opposed to those with few connections requiring a lot of data. Web applications that are serving many users simultaneously may see large performance improvements when switching from Apache

to NGINX, as the NGINX architecture has much less overhead per connection.

Behind NGINX is Microsoft IIS, although the popularity of Windows-based servers has diminished due to expensive licenses and lack of compatibility with Unix-based open source software (OSS) packages. IIS is the correct choice of web server when dealing with many Microsoft-specific technologies, but it may be a burden to companies trying to build on top of open source.

There are many smaller web servers out there, and each has its own security benefits and downsides. Becoming familiar with the big three will be useful as you move on throughout this book and learn how to find vulnerabilities that stem from improper configuration rather than just vulnerabilities present in application logic.

Server-Side Databases

Once a client sends data to be processed to a server, the server must often persist this data so that it can be retrieved in a future session. Storing data in memory is not reliable in the long term, as restarts and crashes could cause data loss. Additionally, random-access memory is quite expensive when compared to disk.

When storing data on disk, proper precautions need to be taken to ensure that the data can be reliably and quickly retrieved, stored, and queried. Almost all of today's web applications store their user-submitted data in some type of database—often varying the database used depending on the particular business logic and use case.

SQL databases are still the most popular general-purpose database on the market. SQL query language is strict but reliably fast and easy to learn. SQL can be used for anything from storage of user credentials to managing JSON objects or small image blobs. The largest of these are PostgreSQL, Microsoft SQL Server, MySQL, and SQLite.

When more flexible storage is needed, schema-less NoSQL databases can be employed. Databases like MongoDB, DocumentDB, and CouchDB store information as loosely structured "documents" that are flexible and can be modified at any time, but they are not as easy or efficient at querying or aggregating.

In today's web application landscape, more advanced and particular databases also exist. Search engines often employ their own highly specialized databases that must be synchronized with the main database on a regular basis. An example of this is the widely popular Elasticsearch.

Each type of database carries unique challenges and risks. SQL injection is a well-known vulnerability archetype effective against major SQL databases when queries are not properly formed. However, injection-style attacks can occur against almost any database if a hacker is willing to learn the database's query model.

It is wise to consider that many modern web applications can employ multiple databases at the same time, and often do. Applications with sufficiently secure SQL query generation may not have sufficiently secure MongoDB or Elasticsearch queries and permissions.

Client-Side Data Stores

Traditionally, minimal data is stored on the client because of technical limitations and cross-browser compatibility issues. This is rapidly changing. Many applications now store significant application state on the client, often in the form of configuration data or large scripts that would cause network congestion if they had to be downloaded on each visit.

In most cases, a browser-managed storage container called local storage is used for storing and accessing key/value data from the client. Local storage follows browser-enforced SOP, which prevents other domains (websites) from accessing each other's locally stored data. Web applications can maintain state even when the browser or tab is closed (see Figure 3-6).

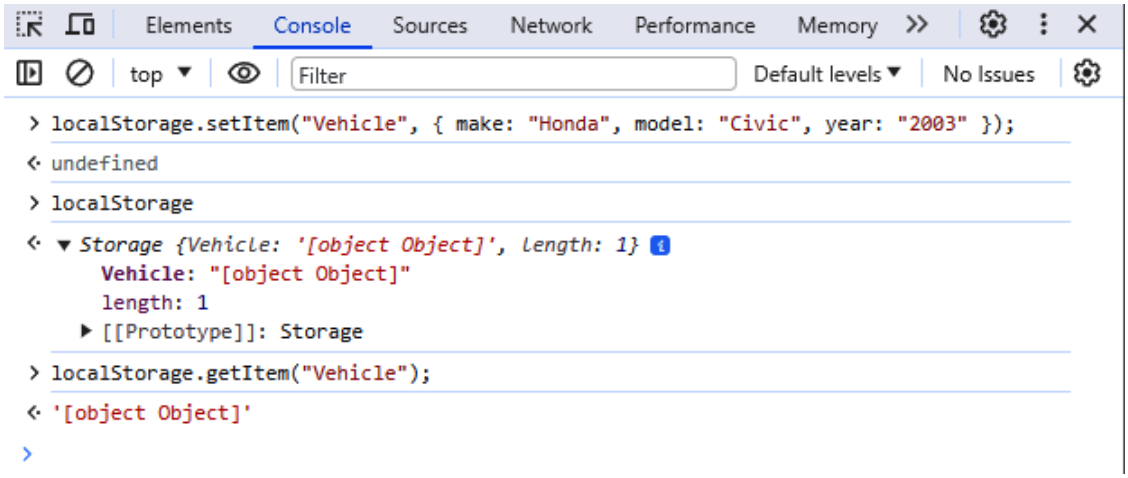

Figure 3-6. Local storage is a powerful and persistent key/value store supported by all modern browsers

A subset of `local storage` called `session storage` operates identically but persists data only until the tab is closed. This type of storage can be used when data is more critical and should not be persisted if another user uses the same machine.

TIP

In poorly architected web applications, client-side data stores may also reveal sensitive information such as authentication tokens or other secrets.

Finally, for more complex applications, browser support for IndexedDB is found in all major web browsers today. IndexedDB is a JavaScript-based NoSQL database capable of storing and querying asynchronously in the background of a web application.

Because IndexedDB is queryable, it offers a much more powerful developer interface than `local storage` is capable of. IndexedDB finds use in web-based games and web-based interactive applications (like image editors). You can check if your browser supports IndexedDB by typing the following in the browser developer console: `if (window.indexedDB) { console.log('true'); }`.

GraphQL

GraphQL is a powerful new method of querying API endpoints that has rapidly gained popularity since its initial release in September of 2015. Typically, GraphQL is implemented by wrapping existing API endpoints on a server and allowing the endpoints to be queried via GraphQL query language—a scripting language that allows requests to be bundled, resulting in significant performance savings at the network level (see Figure 3-7).

GraphQL's query language allows for more advanced server requests than traditional REST by itself. Some of the additional features provided by a GraphQL-wrapped REST API are as follows:

- Requests for specific fields
- Complex request arguments
- Field aliases
- Field fragments
- Operations
- Variables
- Directives
- Mutations

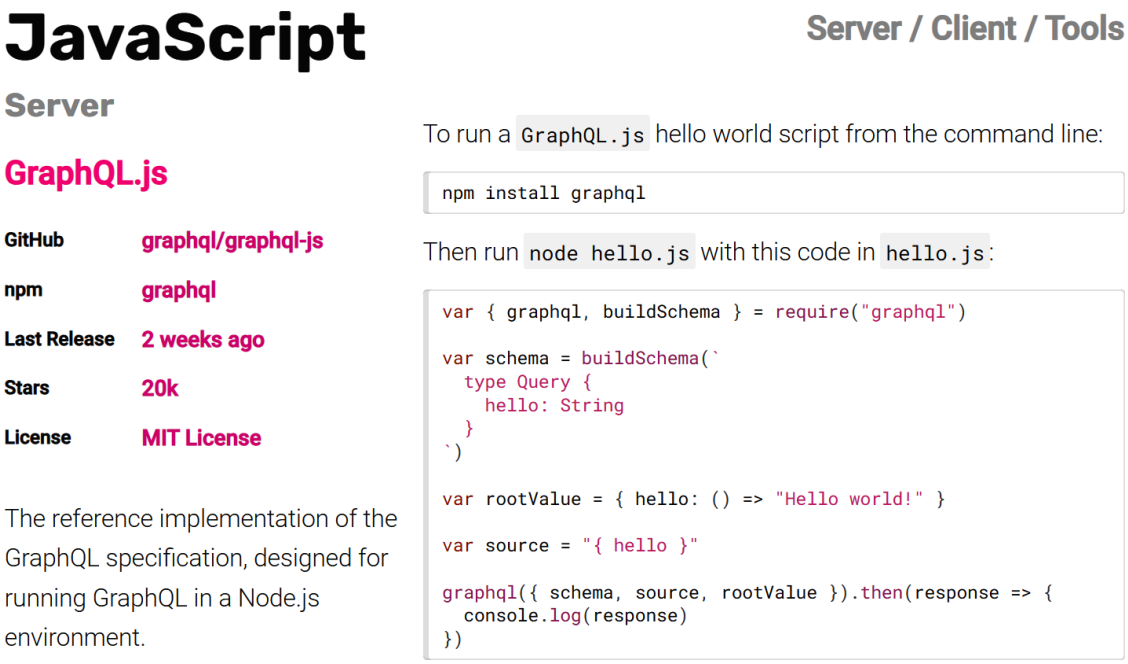

Figure 3-7. A JavaScript implementation of GraphQL

By using the structures provided by the GraphQL query language, you can put together queries like the following:

```
query GetLeadActorFromMovie($movie: Movie)
 actor(movie: $movie) {
   name, age, gender,
   otherMovies {
     name
  }
}
```

Provide the correct variables to the query:

```
{
  "movie": "Raiders of the Lost Ark"
}
```

and the server will return the following:

```
{
  "data": {
    "actor": {
      "name": "Harrison Ford",
      "age": 81,
      "gender": "male",
      otherMovies: [
        "Blade Runner",
        "Patriot Games",
        "Clear and Present Danger"
      ]
    }
  }
}
```

As you can see, GraphQL allows for significantly more complex queries than a traditional REST API by itself. With just a REST API, we would first need to query `getLeadActorByMovie/:movieName` and then query `getMoviesByLeadActor/:actorName`. But with GraphQL these queries

have been combined into one—allowing for one network hop rather than two.

The security implications of utilizing GraphQL will be evaluated in full later on in this book. For now, be aware that GraphQL allows clients to provide complex scripting to be run against existing REST APIs on a server in order to reduce network latency.

Version Control Systems

Almost all modern web applications are built using some type of version control system (VCS). These systems allow developers of modern web applications to walk back through their changes to identify and revert changes, create "branches" or alternate implementations of a feature, and handle a multitude of common deployment tasks.

The most commonly used VCS is Git. Git was originally developed by Linus Torvalds and the Linux foundation, but has since seen widespread adoption and become the most popular VCS in use. Git provides tools like branches, stashes, and forks, which allow developers to branch out and store multiple variations of a codebase with just a few commands.

Later, Git allows these alternate copies to be merged together using a variety of algorithms, each of which produces history that can be revisited later on should an issue need to be tracked down. The most common implementations of hosted Git version control are GitHub (now owned by Microsoft, see Figure 3-8) and GitLab. Both of these provide simple and easy-to-use APIs and integrations on top of the Git version control protocol.

A common Git workflow may look as follows:

```
git init
git remote add origin <github-repository-url>
touch .gitignore
echo "node_modules" > .gitignore
git add -A
```

```
git commit -m "initial commit"
git push --set-upstream origin main
```

This workflow uses the Git command-line tools to initialize a Git repository at the current working directory. Then, a remote cloud-hosted GitHub repository is linked to the local Git repository. Next, a *.gitignore* file is created to tell Git what files *not* to track in version control. Then the command `git add -A` tells Git to add all changed files to staging to prepare a commit. The `git commit` command lets Git know that the staged files are ready to be recorded, and the `-m` flag provides a message detailing what is in the changeset. Finally, `git push` tells Git to push the staged files to version control history and push them remotely to the GitHub repository.

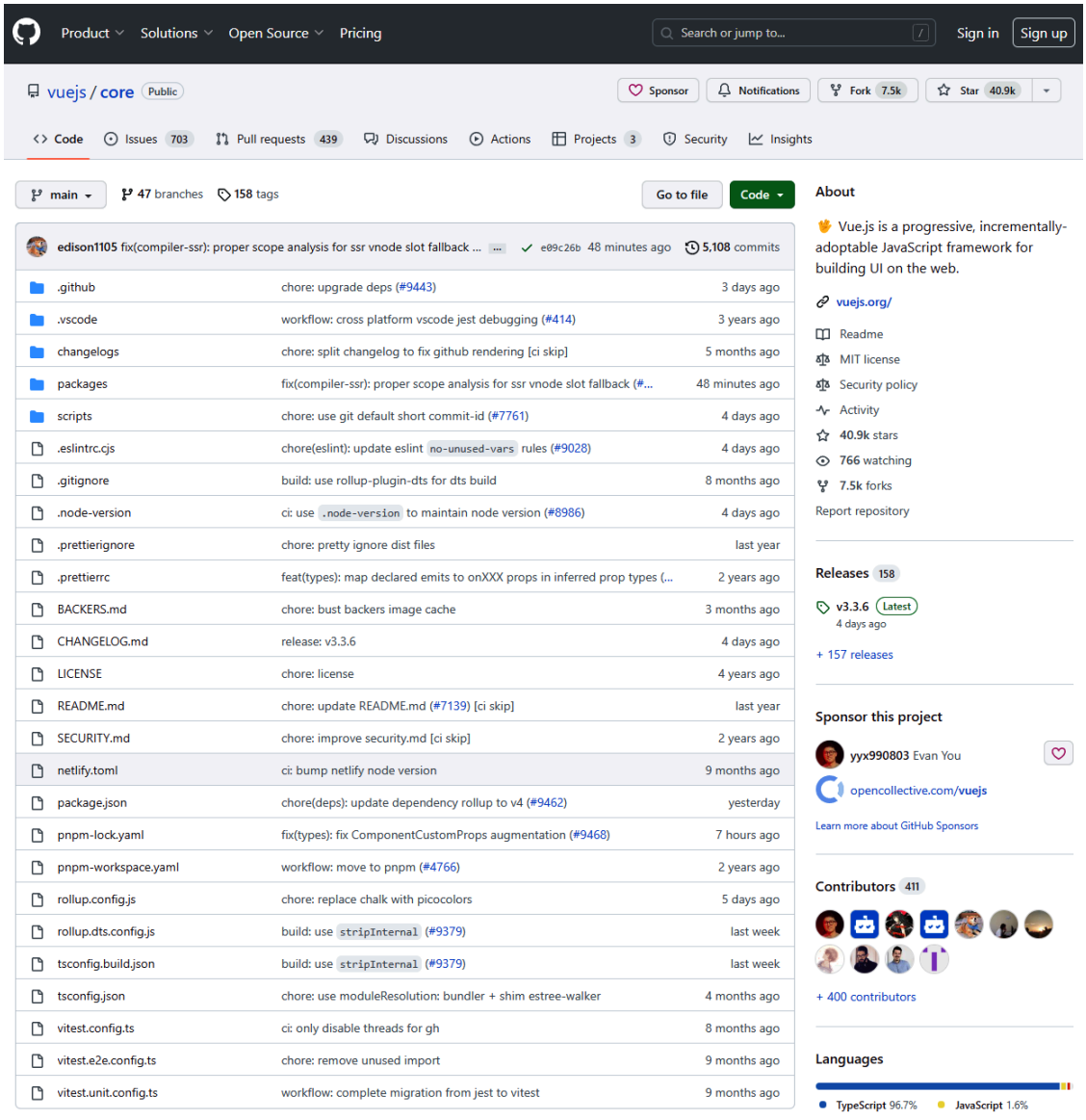

Figure 3-8. GitHub, a popular code sharing and collaboration website, is built on the Git version control system

This is just a simple workflow on Git, however, and most modern web applications make use of much more advanced features. A prime example of this is *continuous integration* and *continuous delivery* aka CI/CD. CI/CD tools, while not natively part of the Git protocol, are often integrated alongside Git for ease of use.

Often, modern web applications are configured to automatically run tests and then deploy a build when new code is pushed upstream to cloud-hosted Git repositories. This process, known as *continuous deployment*, is one of the most common CI/CD integrations. It is often triggered by a *Git hook* or code snippet that runs on a particular Git command.

For these reasons, the security of a web application is no longer tied to only the application's network stack and application code, but also to its VCS and CI/CD pipeline. Compromises at either of these layers can be catastrophic, and as such, a holistic picture of the entire application and its VCS and delivery mechanisms should be considered.

CDN/Cache

Legacy web applications often served content directly to the client (browser), with smart estimates being used to calculate hardware requirements for peak request hours. Many modern web applications take a different approach by offloading static or infrequently updated content to content delivery networks (CDN) and caching at various parts of the application stack, including client-side caches.

A CDN can scale by offering servers worldwide that replicate static content provided by your web server over an API. The CDN handles scalability while the web server communicates directly with the CDN, operating as the single source of truth (see Figure 3-9).

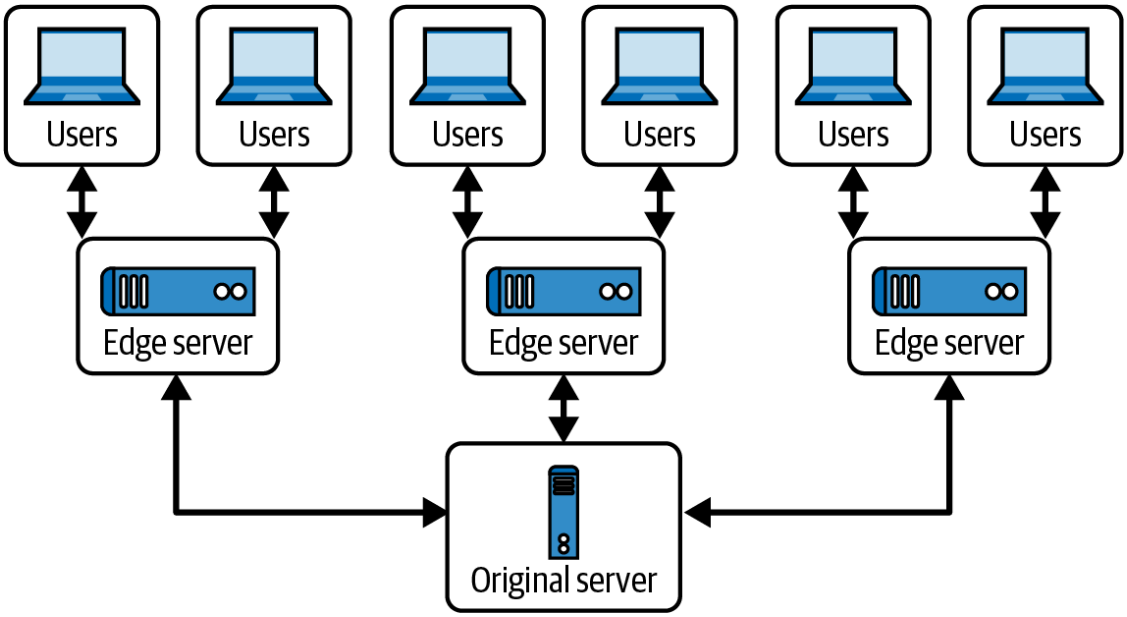

Figure 3-9. CDNs offload content from your primary web server and distribute that content geographically to reduce server load and provide lower latency with higher uptime

This approach has the positive effect of allowing developers to scale an application to worldwide very rapidly, but it introduces a number of new security risks. One of the major risks is stale caching, which might not have reliable or accurate information, and could lead to privilege escalation or information disclosure.

While CDNs operate as a global cache, local and network caches are used more frequently. One common form of cache that was not utilized in legacy web applications is the client-side (browser) cache. Browsers offer default mechanisms for caching, which can be configured via headers like `etag` or `cache-control`, but many applications also make use of `local storage`, `session storage`, and `indexedDB` for further customization of client-side caches.

Due to the expansion of client-side and third-party CDN/cache adoption, engineers need to evaluate data security in a number of places not often considered a decade ago. These new systems lead to additional complexity, as these caches can become corrupted or out of sync, which leads to new security challenges. Many of these challenges will be addressed in the following chapters.

Summary

Modern web applications are built on a number of new technologies not found in older applications. Because of this increased surface area due to expanded functionality, many more forms of attack can target today's applications compared to the websites of the past.

To be a security expert in today's application ecosystem, you need not only security expertise, but some level of software development skill as well. The top hackers and security experts of this decade bring with them deep engineering knowledge in addition to their security skills. They understand the relationship and architecture between the client and the server of an application. They can analyze an application's behavior from the perspective of a server, a client, or the network in between.

The best of the best understand the technologies that power these three layers of a modern web application as well. As a result, they understand the weaknesses inherent in different databases, client-side technologies, and network protocols.

While you do not need to be an expert software engineer to become a skilled hacker or security engineer, these skills will aid you and you will find them very valuable. They will expedite your research and allow you to see deep and difficult vulnerabilities that you would not otherwise be able to find.