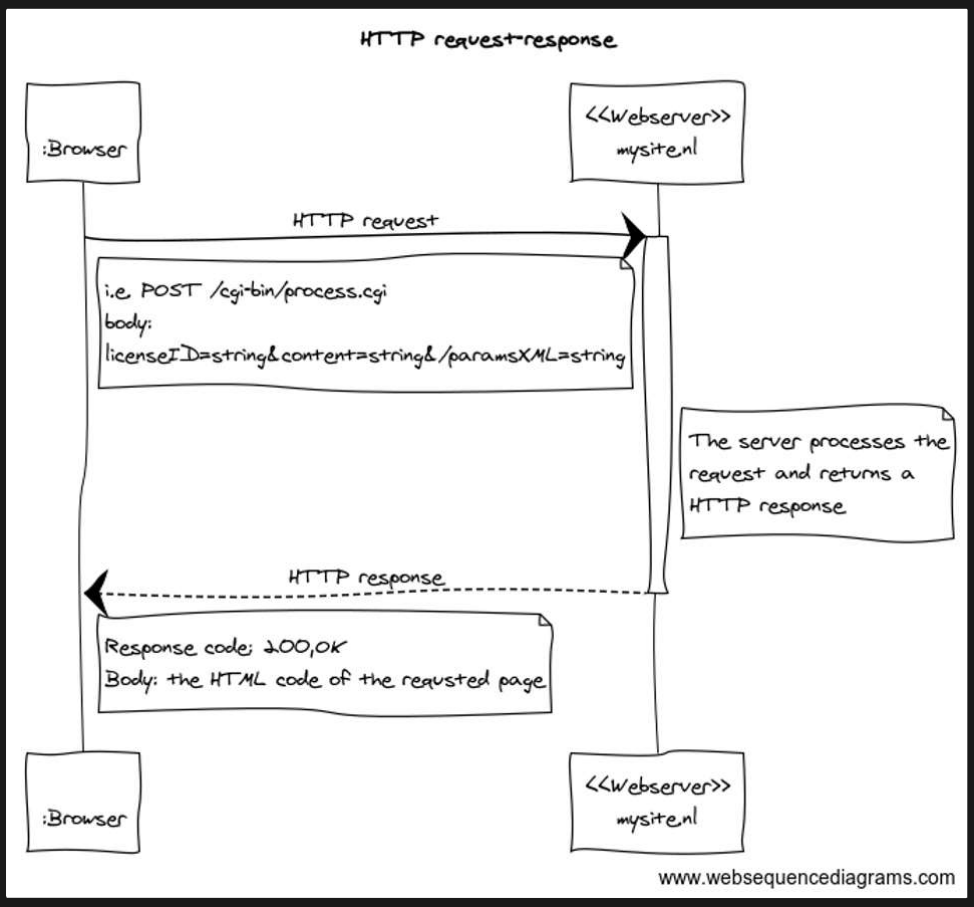
HTTP basics

- The Essentials
  - Anatomy of an HTTP Request
  - Anatomy af an HTTP Response
  - Error-like status codes
  - Redirects
- Digging deeper
  - Visualising HTTP in your browser
  - Cookies
    - Cookie essentials
    - Cookie details
  - Sessions
    - How sessions work
    - Using sessions in Laravel
- Further Reading
  - More HTTP details
  - More response code

The Essentials

HTTP makes the internet work. It forms the basis of how in most web applications, including mobile apps and lots of IoT devices exchange data between servers and clients. Therefore it is important to know a bit about the basics of this protocol.

HTTP is a client-server protocol. A client, ususally a browser on a computer or phone, sends a message which is called a *request* to a webserver (server applications that understand HTTP requests are called webservers). The request is processed by the webserver, and it returns a *response*. An example is given in the following sequence diagram:



One other aspect of HTTP is that it is **stateless** by design. A stateless protocol does not require the server to retain information or status about each user for the duration of multiple requests. This means that a webserver must process each request independent of previous requests that are being recieved.

Anatomy of an HTTP Request

See also: http://www.tutorialspoint.com/http/http_requests.htm

The HTTP request in the example might look like:

```
POST /cgi-bin/process.cgi HTTP/1.1 User-Agent: Mozilla/4.0 (compatible;
MSIE5.01; Windows NT) Host: www.tutorialspoint.com Content-Type:
application/x-www-form-urlencoded Content-Length: length Accept-Language:
en-us Accept-Encoding: gzip, deflate Connection: Keep-Alive
licenseID=string&content=string&/paramsXML=string
```

The first line is the *Request-Line*. It always begins with a *method* token, followed by the *Request-URI* and the protocol version, and ending with CRLF. The elements are separated by space SP characters.

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

The request contains a wide variety of data. Parts of it are important for us and others more for the webserver. Important parts are:

- **POST**: *Request method*. The request **method** indicates the method to be performed on the resource identified by the given **Request-URI.** Examples of other methods are GET, PUT and DELETE.
- **/cgi-bin/process.cgi**: *Request URI*: The Request-URI is a Uniform Resource Identifier and identifies the resource upon which to apply the request.

Next comes a section of **Request Header Fields**. The request-header fields allow the client to pass additional information about the request, and about the client itself, to the server. Most of them aren't important for basic webdevelopment for now.

Finally there is the optional **Request Body** which might hold the data that the client wants to send to the server. In the example it's:

```
licenseID=string&content=string&/paramsXML=string
```

Anatomy af an HTTP Response

See also: http://www.tutorialspoint.com/http/http_responses.htm

After receiving and interpreting a request message, a server responds with an HTTP response message:

```
HTTP/1.1 200 OK Date: Mon, 27 Jul 2009 12:28:53 GMT Server: Apache/2.2.14 (Win32) Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT Content-Length: 88 Content-Type: text/html Connection: Closed <html><head><title>My page</title></head><body><h1>Hello, World!</h1></body></html>
```

The first line is called the Status-Line. A Status-Line consists of the protocol version followed by a numeric status code and its associated textual phrase. The elements are separated by space SP characters:

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

The status-code of the example is **200 OK**. The Status-Code element is a 3-digit integer where first digit of the Status-Code defines the class of response. A status code 200 for instance lets the receiver know that everything went well and a correct answer is given. Other examples of status codes you might encounter are: 302, 404, 419 and 500

Next is, again, a section of headers: the response-header fields. These allow the server to pass additional information about the response which cannot be placed in the Status- Line. These header fields give information about the server and about further access to the resource identified by the Request-URI.

The response finishes with the response body which contains the content of the requested resource or any other message. In the example it's the `<html> … </html>` part. The client browser uses the content to render the page for the user.

Error-like status codes

Sometimes the web server cannot provide a proper response. The cause of this can be anything. The web server then responds with a response with a status code other than 200. For example

```
HTTP/1.1 404 Not Found Date: Sun, 18 Oct 2012 10:36:20 GMT Server: Apache/2.2.14 (Win32) Content-Length: 230 Connection: Closed Content-Type: text/html; charset=iso-8859-1 <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN"> <html> <head> <title>404 Not Found</title> </head> <body> … </body> </html>
```

The webserver responds in this case with a `404 Not Found`. The body might contain html code with more detailed info to be presented to the user. A 404 means that the server was not able to find the requested resource. There are other error-like responses (actually any code from 400 and up are errors). Here is a list of the most common ones:

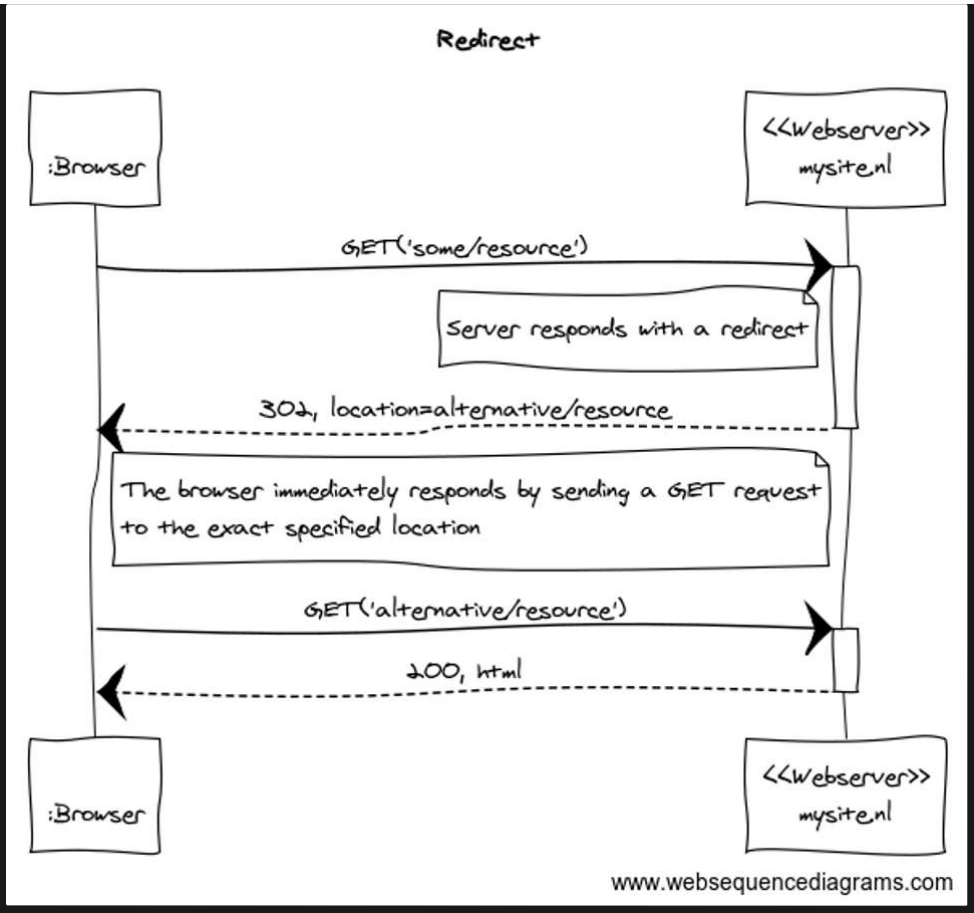
- **400 Bad request** - There is something wrong with the recieved request.
- **404 Not Found** - As described above
- **418 I'm a teapot** - https://evertpot.com/http/418-im-a-teapot 😉
- **419 Page Expired** - This error is created by the Laravel framework to indicate that the CSRF validation failed. When you recieve this error, you probably forgot to add a `@scrf` directive to your form
- **500 Internal Server Error** - There is something wrong at the server. Most likely an error in the code

Redirects

An HTTP redirect is a way for a web server to inform a client that a requested resource has been moved or no longer exists. The server responds to the client's request with a status code indicating the redirect, along with the new location where the requested resource can be found. The client then automatically sends a new request to the new location specified by the server.

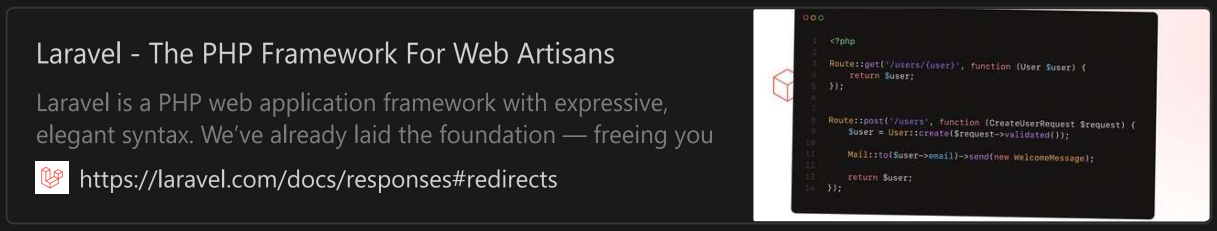
Redirects are mainly useful when collecting user data with Forms, which we will describe later.

The following sequence diagram shows the basic flow of a redirect:



The server redirects by sending a **302 Found** status code (there are more possible status code, basically any code that starts wit a 3 is a redirect), together with a location header. This header holds a URI to the location where the browser must redirect to.

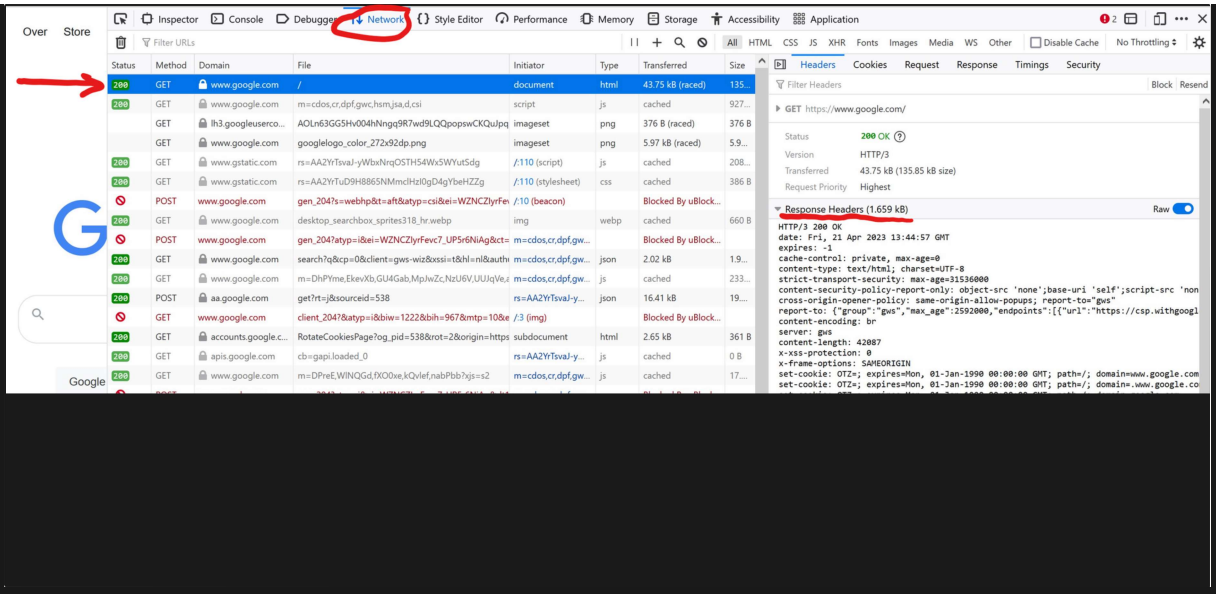
Laravel provides features for creating redirect responses, see:



Digging deeper

Visualising HTTP in your browser

You can easily visualize HTTP info in your browser by using the debug bar. Most modern browsers are equipped with such a bar (rightclick anywhere in the browser window and select 'inspect' or something lika that). You should see something like:



The debug-bar consists of tabs. The **Network** tab shows details of all the network traffic - the HTTP requests and responses - needed to fetch the displayed page. The first request (pointed by the red arrow) is the one that is currently selected. In the right window you see details about this request, like its request and response headers and the status code.

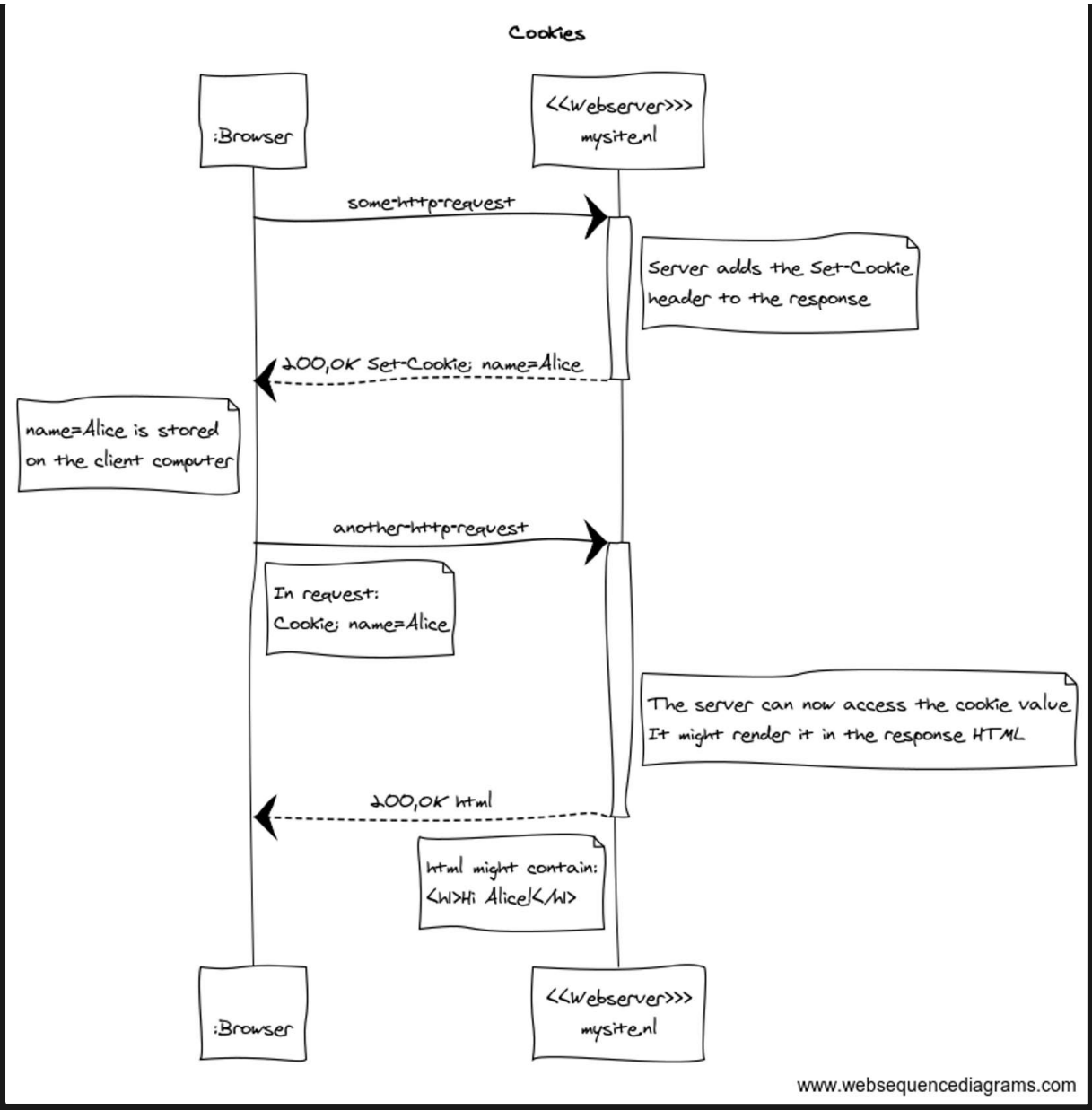
Cookies

It is impossible nowadays to not know that cookies exist on the internet. HTTP cookies are small pieces of data that a web server sends to a client's web browser when the client visits a website. The browser stores this data on the client's computer and sends it back to the server with every subsequent request.

Cookies are commonly used in web development to maintain session state, remember user preferences, and personalize content. Laravel also uses these cookies for technical reasons but also enables developers to read and set cookies with ease.

Cookie essentials

Webservers use the `Set-Cookie` header, added to a HTTP response, to tell which cookie to store. The basic flow is as follows:



You can see that, although HTTP is stateless, the name Alice, set by the first request, is reused in the second request. Cookies are in a way a mechanism to remember some state between multiple requests in a user session.

Cookie details

In most cases it's enough to know how to set and read a cookie. However, it is possible to define more detailed rules and restrictions like:

- Define the lifetime, the moment when the cookie must expire of a cookie
- Restrict access, to make sure cookies are sent securely amd aren't accessed by unintended parties
- Define where cookies are sent. Which URLs a cookie should be sent to

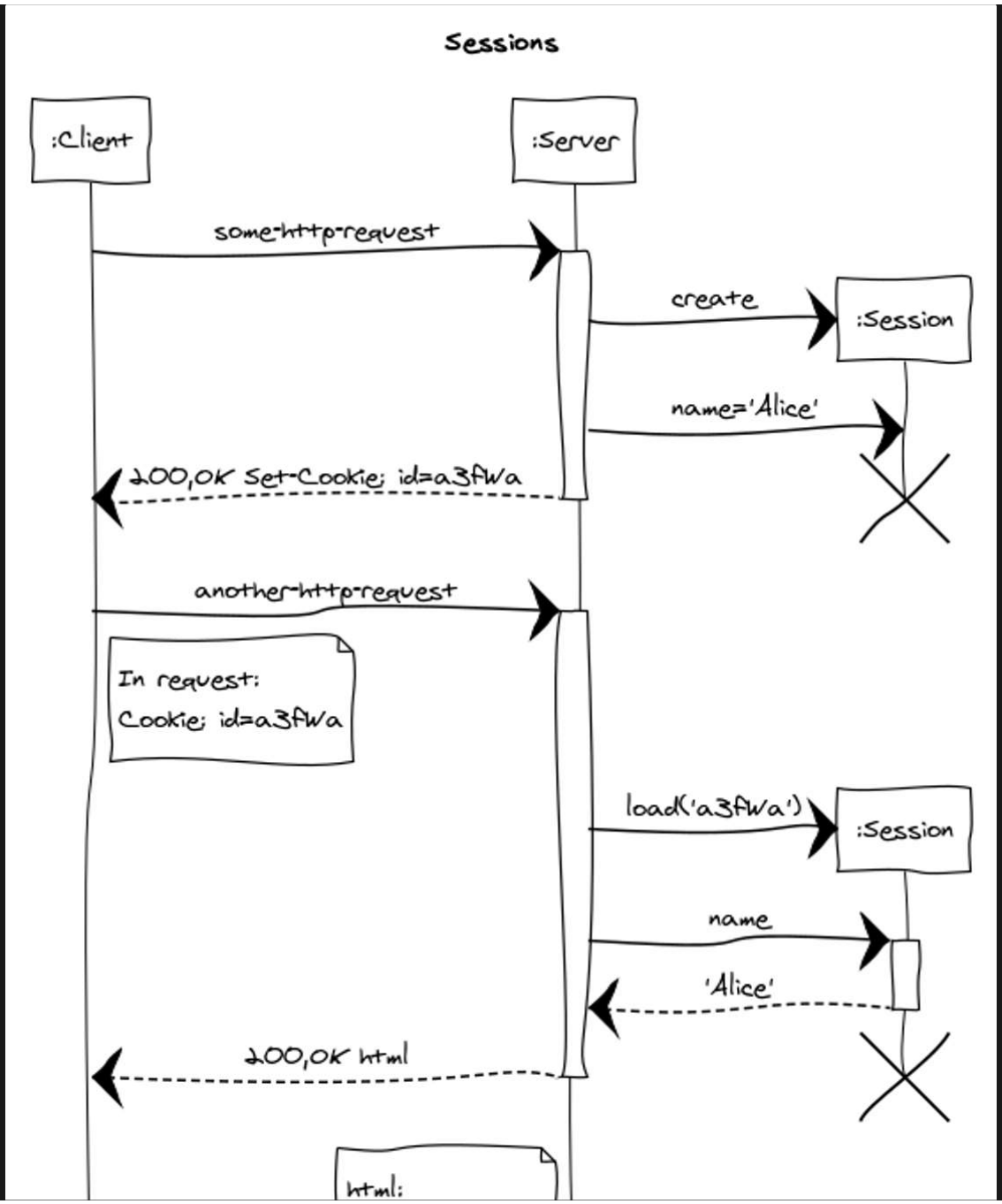
If you want to know more about this, visit:

Sessions

Cookies are only useful for small pieces of data. Also, the data is alway stored on the client computer, and can be altered if the user (i.e. a hacker) knows how to. To avoid this, sessions are used by developers in almost any case.

How sessions work

Sessions differ from cookies while they store information, the *session data*, on the server side instead of the client side. This session data may be stored in a file or in a database (most webservers use files by default). However, sessions actually need to use cookies in order to make it work. The basic flow is as follows:


Sessions
:Client
:Server
some-http-request
create
:Session
name='Alice'
200,OK Set-Cookie: id=a3fWa
another-http-request
In request:
Cookie: id=a3fWa
load('a3fWa')
:Session
name
'Alice'
200,OK html
html: