Client-Server Overview

Created by 이 윤 준 (yoonjoon.lee@gmail.com)

May, 2019

We explain in detail what happens when a server receives a "dynamic request" from a browser.

As most website server-side code handles requests and responses in similar ways, this will help you understand what you need to do when writing most of your own code.

Web servers and HTTP (a primer)

When users click a link on a web page, submit a form, or run a search, the browser sends an HTTP Request to the server.

This request includes:

- A URL identifying the target server and resource.
- A method that defines the required action.
  ```
    - GET/POST/HEAD/PUT/DELETE/TRACE/OPTIONS/CONNECT/PATCH```
- Additional information can be encoded with the request. Information can be encoded as:
  ```
    - URL parameters/POST data/Client-side cookie```

Web servers wait for client request messages, process them when they arrive, and reply to the web browser with an HTTP Response message.

The response contains an HTTP Response status code indicating whether or not the request succeeded (e.g. "200 OK" for success, "404 Not Found" if the resource cannot be found, "403 Forbidden" if the user isn't authorised to see the resource, etc).

The body of a successful response to a GET request would contain the requested resource.

GET request/response example

Users can make a simple GET request by clicking on a link or searching on a site.

The request

Each line of the request contains information about it. The first part is called the **header**, and contains useful information about the request:

```
GET https://developer.mozilla.org/en-US/search?q=client+se
Host: developer.mozilla.org
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebK
Accept: text/html,application/xhtml+xml,application/xml;q=
Referer: https://developer.mozilla.org/en-US/
Accept-Encoding: gzip, deflate, sdch, br
Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7
Accept-Language: en-US,en;q=0.8,es;q=0.6
Cookie: sessionid=6ynxs23n521lu21b1t136rhbv7ezngie; csrfto
```

The first and second lines contain most of the information about above:

- The type of request (`GET`).
- The target resource URL (`/en-US/search`).
- The URL parameters (`q=client+server+overview&topic=apps&topic=html&topic=css&topic=js&topic=api&topic=webdev`).
- The target/host website (`developer.mozilla.org`).
- The end of the first line also includes a short string identifying the specific protocol version (`HTTP/1.1`).

The final line contains information about the client-side cookies. The cookie includes an id for managing sessions (

```
Cookie: sessionid=6ynxs23n521lu21b1t136rhbv7ezngie;
...
```

).

The remaining lines contain information about the browser used and the sort of responses it can handle.

- My browser (`User-Agent`) is Mozilla Firefox (`Mozilla/5.0`).
- It can accept gzip compressed information (`Accept-Encoding: gzip`).
- It can accept the specified set of characters (`Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7`) and languages (`Accept-Language: de,en;q=0.7,en-us;q=0.3`).

- The Referer line indicates the address of the web page that contained the link to this resource (i.e. the origin of the request, `https://developer.mozilla.org/en-US/`).

HTTP requests can also have a body, but it is empty in this case.

The response

```
HTTP/1.1 200 OK
Server: Apache
X-Backend-Server: developer1.webapp.scl3.mozilla.com
Vary: Accept,Cookie, Accept-Encoding
Content-Type: text/html; charset=utf-8
Date: Wed, 07 Sep 2016 00:11:31 GMT
Keep-Alive: timeout=5, max=999
Connection: Keep-Alive
X-Frame-Options: DENY
Allow: GET
X-Cache-Info: caching
Content-Length: 41823
```

```
<!DOCTYPE html>
<html lang="en-US" dir="ltr" class="redesign no-js"  data-
<head prefix="og: http://ogp.me/ns#">
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=Edge">
  <script>(function(d) { d.className = d.className.replace
  ...
```

The first part of the response for this request consists of the header contains information:

- The first line includes the response code `200 OK`, which tells us that the request succeeded.
- We can see that the response is `text/html` formatted (`Content-Type`).
- We can also see that it uses the `UTF-8` character set (`Content-Type: text/html; charset=utf-8`).
- The head also tells us how big it is (`Content-Length: 41823`).

At the end of the message we see the body content.

POST request/response example

An HTTP POST is made when you submit a form containing information to be saved on the server.

The request

The format of the request is almost the same as the GET request example shown previously, though the first line identifies this request as a POST.

```
POST https://developer.mozilla.org/en-US/profiles/hamishwi
Host: developer.mozilla.org
Connection: keep-alive
Content-Length: 432
```

```
Pragma: no-cache
Cache-Control: no-cache
Origin: https://developer.mozilla.org
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebK
Content-Type: application/x-www-form-urlencoded
Accept: text/html,application/xhtml+xml,application/xml;q=
Referer: https://developer.mozilla.org/en-US/profiles/hami
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.8,es;q=0.6
Cookie: sessionid=6ynxs23n521lu21b1t136rhbv7ezngie; _gat=1
csrfmiddlewaretoken=zIPUJsAZv6pcgCBJSCj1zU6pQZbfMUAT
&user-username=hamishwillee
&user-fullname=Hamish+Willee
&user-title=&user-organization=&user-location=Australia&us
```

The main difference is that the URL doesn't have any parameters.

The response

The status code of "`302 Found`" tells the browser that the post succeeded, and that it must issue a second HTTP request to load the page specified in the `Location` field. The information is otherwise similar to that for the response to a `GET` request.

```
HTTP/1.1 302 FOUND
Server: Apache
X-Backend-Server: developer3.webapp.scl3.mozilla.com
Vary: Cookie
Vary: Accept-Encoding
Content-Type: text/html; charset=utf-8
Date: Wed, 07 Sep 2016 00:38:13 GMT
Location: https://developer.mozilla.org/en-US/profiles/ham
Keep-Alive: timeout=5, max=1000
Connection: Keep-Alive
X-Frame-Options: DENY
X-Cache-Info: not cacheable; request wasn't a GET or HEAD
Content-Length: 0
```

Static sites

A *static site* is one that returns the same hard coded content from the server whenever a particular resource is requested.

The server for a static site will only ever need to process GET requests, because the server doesn't store any modifiable data. It also doesn't change its responses based on HTTP Request data (e.g. URL parameters or cookies).

Dynamic sites

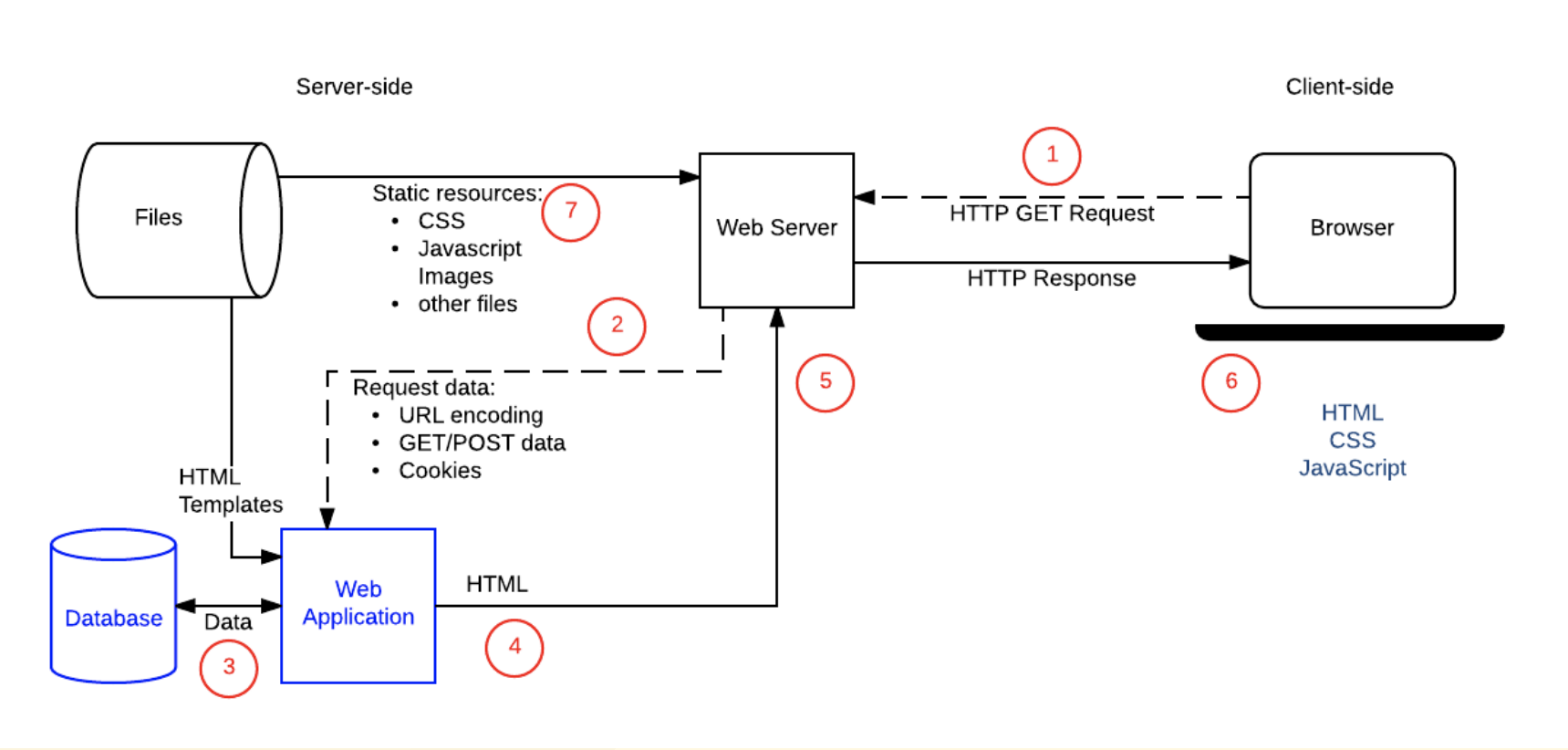
A *dynamic site* is one that can generate and return content based on the specific request URL and data.

Using the example of a product site, the server would store product "data" in a database rather than individual HTML files. When receiving an HTTP GET Request for a product, the server determines the product ID, fetches the data from the database, and then constructs the HTML page for the response by inserting the data into an HTML template.

Anatomy of a dynamic request

The diagram shows the main elements of the "team coach" website.

The parts of the site that make it dynamic are the *Web Application*, the *Database*, which contains information about players, teams, coaches and their relationships, and the *HTML Templates*.



After the coach submits the form with the team name and number of player,

1. The web browser creates an HTTP `GET` request to the server using the base URL for the resource (`/best`) and encoding the team and player number either as URL parameters (e.g. `/best?team=my_team_name&show=11`) or as part of the URL pattern (e.g. `/best/my_team_name/11/`). A `GET` request is used because the request is only fetching data (not modifying data).
2. The *Web Server* detects that the request is "dynamic" and forwards it to the *Web Application* for processing

3. The *Web Application* identifies that the intention of the request is to get the "best team list" based on the URL (`/best/`) and finds out the required team name and number of players from the URL. The *Web Application* then gets the required information from the database.

4. The *Web Application* dynamically creates an HTML page by putting the data into placeholders inside an HTML template.

5. The *Web Application* returns the generated HTML to the web browser, along with an HTTP status code of 200 ("success"). If anything prevents the HTML from being returned then the Web Application will return another code.
6. The Web Browser will then start to process the returned HTML, sending separate requests to get any other CSS or JavaScript files that it references.
7. The Web Server loads static files from the file system and returns them to the browser directly.

Doing other work

A *Web Application*'s job is to receive HTTP requests and return HTTP responses.

A good example of an additional task that a *Web Application* might perform would be sending an email to users to confirm their registration with the site. The site might also perform logging or other operations.

Returning something other than HTML

Server-side website code does not have to return HTML snippets/files in the response. It can instead dynamically create and return other types of files.

The idea of returning data to a web browser so that it can dynamically update its own content (AJAX) has been around for quite a while.

More recently "Single-page apps" have become popular, where the whole website is written with a single HTML file that is dynamically updated when needed.

Websites created using this style of application push a lot of computational cost from the server to the web browser, and can result in websites that appear to behave a lot more like native apps.

Web frameworks simplify server-side web programming

One of the most important operations is providing simple mechanisms to map URLs for different resources/pages to specific handler functions. This makes it easier to keep the code associated with each type of resource separate. It also has benefits in terms of maintenance, because you can change the URL used to deliver a particular feature in one place, without having to change the handler function.

```
# file: best/urls.py
#

from django.conf.urls import url

from . import views

urlpatterns = [
    # example: /best/
    url(r'^$', views.index),
    # example: /best/junior/
    url(r'^junior/$', views.junior),
]
```

```python
#best/views.py

from django.shortcuts import render

from .models import Team


def junior(request):
    list_teams = Team.objects.filter(team_type__exact="jun
    context = {'list': list_teams}
    return render(request, 'best/index.html', context)
```