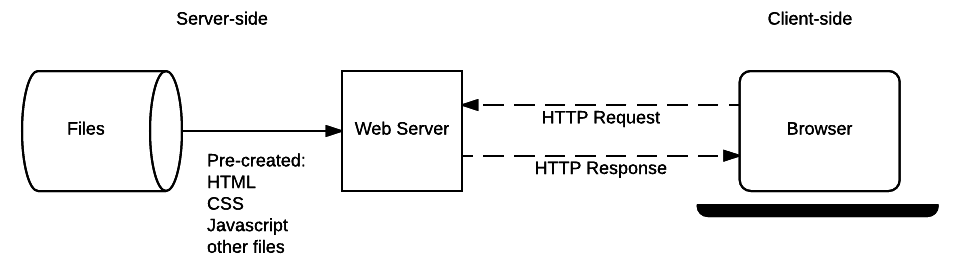
**Exact process when you type a web address in your browser:**

1. The browser goes to the DNS server, and finds the real address of the server that the website lives on
2. The browser sends an HTTP request message to the server, asking it to send a copy of the website to the client. This message, and all other data sent between the client and the server, is sent across your internet connection using TCP/IP.
3. If the server approves the client's request, the server sends the client a "200 OK" message, which means "Of course you can look at that website! Here it is", and then starts sending the website's files to the browser as a series of small chunks called data packets.
4. The browser assembles the small chunks into a complete website and displays it to you.

**Static sites:**

The diagram below shows a basic web server architecture for a static site (a static site is one that returns the same hard-coded content from the server whenever a particular resource is requested). When a user wants to navigate to a page, the browser sends an HTTP "GET" request specifying its URL.

The server retrieves the requested document from its file system and returns an HTTP response containing the document and a [success status](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#Successful_responses) (usually 200 OK). If the file cannot be retrieved for some reason, an error status is returned (see [client error responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#Client_error_responses) and [server error responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#Server_error_responses)).



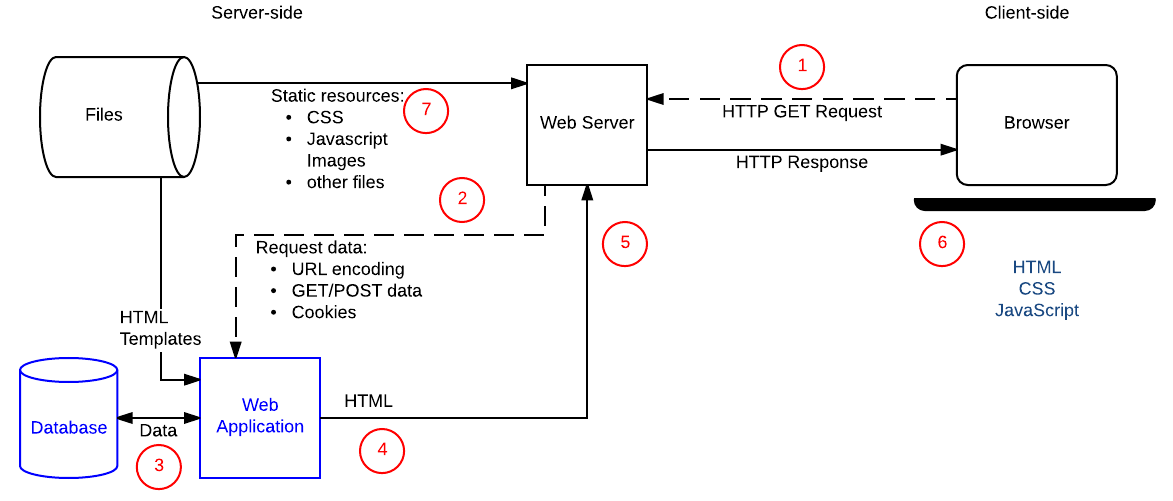
**Dynamic sites:**

A dynamic site can return different data for a URL based on information provided by the user or stored preferences and can perform other operations as part of returning a response (e.g. sending notifications).

Most of the code to support a dynamic website must run on the server. Creating this code is known as "**server-side programming**" (or sometimes "**back-end scripting**").

The diagram below shows a simple architecture for a dynamic website. As in the previous diagram, browsers send HTTP requests to the server, then the server processes the requests and returns appropriate HTTP responses.

Requests for static resources are handled in the same way as for static sites (static resources are any files that don't change —typically: CSS, JavaScript, Images, pre-created PDF files etc).



Requests for dynamic resources are instead forwarded (2) to server-side code (shown in the diagram as a Web Application). For "dynamic requests" the server interprets the request, reads required information from the database (3), combines the retrieved data with HTML templates (4), and sends back a response containing the generated HTML (5,6).

Cross-Site Request Forgery (CSRF)

CSRF attacks allow a malicious user to execute actions using the credentials of another user without that user’s knowledge or consent.

This type of attack is best explained by example. John is a malicious user who knows that a particular site allows logged-in users to send money to a specified account using an HTTP POST request that includes the account name and an amount of money. John constructs a form that includes his bank details and an amount of money as hidden fields, and emails it to other site users (with the *Submit* button disguised as a link to a "get rich quick" site).

If a user clicks the submit button, an HTTP POST request will be sent to the server containing the transaction details and any client-side cookies that the browser associated with the site (adding associated site cookies to requests is normal browser behavior). The server will check the cookies, and use them to determine whether or not the user is logged in and has permission to make the transaction.

The result is that any user who clicks the *Submit* button while they are logged in to the trading site will make the transaction. John gets rich.

CSRF attacks allow a malicious user to execute actions using the credentials of another user without that user’s knowledge or consent.

This type of attack is best explained by example. John is a malicious user who knows that a particular site allows logged-in users to send money to a specified account using an HTTP POST request that includes the account name and an amount of money. John constructs a form that includes his bank details and an amount of money as hidden fields, and emails it to other site users (with the *Submit* button disguised as a link to a "get rich quick" site).

If a user clicks the submit button, an HTTP POST request will be sent to the server containing the transaction details and any client-side cookies that the browser associated with the site (adding associated site cookies to requests is normal browser behavior). The server will check the cookies, and use them to determine whether or not the user is logged in and has permission to make the transaction.

The result is that any user who clicks the *Submit* button while they are logged in to the trading site will make the transaction. John gets rich.

**Note**: The trick here is that John doesn't need to have access to the user's cookies (or access credentials). The browser of the user stores this information and automatically includes it in all requests to the associated server.

One way to prevent this type of attack is for the server to require that POST requests include a user-specific site-generated secret. The secret would be supplied by the server when sending the web form used to make transfers. This approach prevents John from creating his own form, because he would have to know the secret that the server is providing for the user. Even if he found out the secret and created a form for a particular user, he would no longer be able to use that same form to attack every user.

Web frameworks often include such CSRF prevention mechanisms.

One way to prevent this type of attack is for the server to require that POST requests include a user-specific site-generated secret. The secret would be supplied by the server when sending the web form used to make transfers. This approach prevents John from creating his own form, because he would have to know the secret that the server is providing for the user. Even if he found out the secret and created a form for a particular user, he would no longer be able to use that same form to attack every user.

Web frameworks often include such CSRF prevention mechanisms.