Introductory Web Application Architecture

John Verwolf

We will cover...

Purpose: To gain a conceptual framework for learning about web-technologies. Understanding the context makes it much easier to learn the specifics.

- ➔ The role of HTML, CSS, Javascript
- ➔ Client/Server model
- ➔ What happens when you plug a URL into your web browser
- ➔ HTTP(s)
- ➔ Data Exchange formats
- ➔ URLS, APIs, URIs, REST
- ➔ Statelessness
- ➔ Cookies
- ➔ CORS

The Roles

Web browsers only understand 3 things :

HTML: The content

CSS: The styling

Javascript: The logic.

When you hear about frameworks written in all sorts of languages and using all sorts of technologies, these are actually all *transpiled* into HTML, CSS, and Javascript. Web browsers do not understand anything else.

Clients, Servers, and APIs

Restaurant Analogy

When you go to a restaurant, a “server” will bring you the menu items that you ask for. In this scenario, you are the “client”, because you are the person making requests. The “server” responds by bringing you the things that you ordered. The items that you are able to order must be on the menu (the API).

Clients, Servers, and APIs

Client: Initiates communication, asks for data

Server: Accepts requests and responds with data

API: The things the server knows about and is able to serve.

(The menu.)

Clients and servers are just a concept. They mean different things depending on the context.

Clients, Servers, and APIs

The word “**server**” is overloaded - it means multiple things!

- ➔ A program that delivers data upon request
- ➔ A physical computer that is used for running server programs

The word “**client**” also means multiple things!

- ➔ Most commonly used to refer to a web-browser, or the web-page/code running inside a web-browser.
- ➔ Can also refer to a portion of code that is used to make requests to a server. You can have “clients” inside of servers! Usually this is a class in the code that knows how to make requests to another server. IOT devices can also be clients.

What happens when I enter a URL into my browser?

http://www.example.com/

First, your browser needs to get the IP address from the hostname.

Hostname: www.example.com

- ➔ Hostname is a human-readable alias for a server's IP address
- ➔ IP address gives the computer's location in the internet
- ➔ Your browser will ask a DNS server for the IP address corresponding to the hostname

What happens when I enter a URL into my browser?

Next, your browser will make an HTTP GET request to the **URI** location.

What is a URI:

- ➔ **U**niform **R**esource **I**dentifier
- ➔ Specifies location of a resource on a server
- ➔ The combined total of a server's resource locations are called an "API"
- ➔ A URI can be thought of as a file-path (although no correspondence to server's actual file-structure)

Here, the URI location is simply "/", which is the "root" location in the API

What happens when I enter a URL into my browser?

For http://www.example.com/

1. Browser makes an HTTP “GET /” request to the server
2. Server receives the request.
3. Server checks what resource it has at “/” in it’s API. In this case, it’s “index.html”
4. Server sends back an HTTP response containing “index.html”
5. Browser receives response, opens it, and starts rendering “index.html”

Rendering “Index.html” may require subsequent requests for resources. I.e style sheets, javascripts, images, etc

HTTP

Hyper**T**ext **T**ransfer **P**rotocol

- Foundation for internet data-communication. Most important for web-dev!
- HTTP messages are how data is exchanged between a server and a client
- Messages have 3 parts: start line, headers, and body
- Body contains the data being transferred
- Headers contain meta-information about the body and nature of the message
- Start line identifies the HTTP version, verb, and URI

See: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

HTTP Methods

HTTP methods are used for defining *actions* to take on a resource:

- ➔ Get: Gets a resource
- ➔ Post: Create a new resource
- ➔ Put: Update an existing resource
- ➔ Delete: Delete a resource

There are other methods, but these are the most common/important.

HTTP Status Codes

Status codes are returned in the HTTP response.
The Server sends a status code back to the client to indicate its success/failure in processing the request.

- ➔ **1xx:** Informational response (aka “I’m doing a thing over here, FYI”)
- ➔ **2xx:** Success (aka “I did it!”)
- ➔ **3xx:** Redirect (aka “You need to go somewhere else to do this”)
- ➔ **4xx:** Client error (aka “you can’t do that”)
- ➔ **5xx:** Server error (aka “I’m so broken inside”)

HTTPS - What Even Is Security?

- ➔ HTTPS is the same as HTTP, but over an encrypted connection.
- ➔ Client and server start by performing a “handshake” to set up encrypted “tunnel” (encrypted connection). Once the encrypted connection is established, they send regular old HTTP messages back and forth.
- ➔ Lots of people can, and do, sniff HTTP traffic (ISPs, Govs, NSA, network owners, malicious actors, VPN providers, etc)
- ➔ NEVER send a password over regular HTTP
- ➔ Always use HTTPS in real life
- ➔ Sadly, there are laws limiting encryption strength in many countries. It may be illegal to use strong encryption in some countries...😞
- ➔ Note: Also nevers store passwords in cleartext in your webapp! Instead, hash and salt, store salted hash.

Server Frameworks (Programs)

- ➔ A **server framework** is a program that is meant to be extended into a server application.
- ➔ Contains logic for interfacing with the operating system to connect to the network hardware.
- ➔ Gives you a lot of things for free: concurrent connections, parsing HTTP messages, constructing HTTP responses, etc.
- ➔ Different frameworks have different levels of complexity. Some have more “batteries included” than others. Most are extendable with additional libraries for doing specific jobs (ie authentication, logging, encryption, etc)
- ➔ Server frameworks can be written in any language since they live on the back end. Unlike client code, they are not constrained to use javascript.

What Goes in the HTTP Body?

Data Exchange Formats:

- ➔ **JSON (Javascript Object Notation)** is the most popular and will suit most of your needs. It's fairly human-readable. Learn it first. (https://json.org/example.html, https://www.w3schools.com/js/js_json_xml.asp)

- ➔ **XML:** looks like HTML. Old-skool. It's harder to read and less efficient, since it requires more characters for the same amount of data. Many systems still use it. Learn it eventually.

REST: Stay Sane

Eventually your API will grow. As it grows, it will get more complex. If you are not intentional in your design for managing complexity, the complexity will doom you.

- RESTful design is a way of organizing your API/URIs
- REST is an architectural principle, not a specific technology
- REST has good core ideas
- Everybody says they do REST, but it's *rarely* fully implemented.
- You should be able to talk about it in interviews. Everyone ***loves*** REST!

REST describes how to name/identify resources, how to discover resources, and how to manage state. https://restfulapi.net/ http://www.restapitutorial.com/

Simple Model: 3-Tier Architecture

1. **Client**
    - Displays information
    - Should not know details about underlying structure of API implementation (encapsulation)
2. **Server**
    - Processes Information
    - Should be transparent to client (Stateless, encapsulation)
3. **Database**
    - Stores all the information used by the server
    - Not just a single machine, can be a multi-node cluster with caching
    - Complexity of database should be transparent to server

Statelessness

- ➔ State is held in two places: the client, and the database.
- ➔ The server should just act as a “pipe” (pure function).
- ➔ The server should never store state. This means it should not need to past information about previous requests in order to process current requests. All the information it needs to produce a response should either be in the request, or in the database.
- ➔ You need to be intentional about state if you want to scale. In real life you can have many duplicate server instances. They need to be able to operate independently and transparently.

Cookies

A cookie is a way of storing state in the client.

It is a tiny lookup table that stores keys and values. Your browser saves cookies for the duration of their specified TTL (time to live).

➔ **Used for authentication** (log-in state):
When you log-in, website will give you a cookie containing a session-token. You use this with every request you make to show that you are authorized. (Remember, server doesn't store state, so it doesn't remember you. You use the token for every request. This is how it knows who you are, and that you are allowed to be there.)

➔ **Used for identification** (tracking)

➔ **Domain can only access it's own cookie**

CORS

Cross Origin Resource Sharing:

- ➔ You browser will not let your application talk to other domains unless you specifically allow it to. This is a security feature.
- ➔ This can also appear like a frustrating bug if you don’t know about it.
- ➔ Need to include an http header like this:
  - ◆ “Access-Control-Allow-Origin: http://www.example.com”

See https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS For more details

Questions?

See ya l8r!