**INTRO TO SERVERS**

* A server is a hardware or software that pushes and pulls data across a network.
* A server's purpose is to send data to a client.
* A client can request data from a server, and the server processes that request and responds with the desired data.

**Role of a Server in a Full-Stack Application**

* The front-end client requests web pages or data from the back-end server. The back-end server sends the web pages or data to the front-end client which is responsible for displaying web pages to a user.

**Back-end servers provide many functionalities:**

* Send the requested web pages
* Handle email messages
* Send files
* Store and send data in a database
* Process data
* Handle web traffic

Sometimes, there are multiple servers in a full-stack application to spread out the different roles and responsibilities.

**EXAMPLE:**

There could be a server for sending web pages to the front-end client. That SERVER could request data from a SERVER storing data in a database.

**HTTP BASICS**

**HT-: HyperText**

* Hypertext is simply "content with references to other content". The ability to link these pages is what makes the kind of interactivity you're learning to build possible, and it was a revolutionary concept when it was introduced

**-TP: Transfer Protocol**

* A set of guidelines surrounding the transmission of data. Protocols define the process of exchanging data, but don't define exactly what that data must be. Think of it like a multi-course meal: you expect the appetizer, then the entree, then the dessert, but you could have any type of food for each of those courses! As long as the plates arrive in the particular order you expect, protocol is being followed.
* HTTP acts as a transfer protocol. It defines the expectations for both ends of the transfer, and it defines some ways the transfer might fail. More specifically, HTTP is defined as a request/response protocol.

**PROTOCOL FOR EXCHANGING DATA BETWEEN A CLIENT AND SERVER**

* Client - data consumer
* Server - data provider

HTTP defines the process of exchanging hypertext between systems.

Client sends HTTP request to the server which replies with an HTTP response

**PROPERTIES OF HTTP**

**1. Reliable Connections -** messages passed between a client & server sacrifice a little speed for the sake of trust, and you can rest assured that each message will be confirmed.

* HTTP doesn't work well if messages aren't received in the correct order, so it's critical that the connection your hypertext is crossing is reliable!

**2. Stateless Transfer -** HTTP is considered a stateless protocol, meaning it doesn't store any information. Each request you send across an HTTP connection should contain all its own context.

**3. Intermediaries -** your request will pass through a series of intermediaries: other servers or devices that pass your request along

**TYPES OF INTERMEDIARIES**

**1. proxies -** modifies your request so it appears to come from a different source. It hides your computer details behind a shared IP address.

**2. tunnels -** which simply pass your request along.

**3. gateways -** which pretend to be the resource server you requested. It hides the server’s details behind a shared IP address

**CLIENT =>** **PROXY** (your router) **=>** **TUNNE**L (internet provider) **=>** **GATEWAY** (server’s router) **=>** **SERVER**

**HTTP REQUEST COMPONENTS**

* A server expects to receive requests in a specific format with specific kinds of information. The client needs to know how to make those requests to initiate an HTTP exchange with the server.

**RETRIEVING HYPERTEXT**

* The Web follows that old tried-and-true pattern. You tell your browser which website you would like to access, and your browser hands that request off to a server that can get you what you've asked for.
* Your browser's part in this transaction is called the request. Since the browser is acting on your behalf, the browser is sometimes referred to as the USER-AGENT (you being the user). You might also hear this referred to more generically as the CLIENT in the exchange.

**COMPONENTS OF AN HTTP REQUEST**

* Your browser is designed to be compliant with the HTTP specification, so it knows how to translate your instructions into a well-formatted HTTP request. An important part of the HTTP spec is that it's simple to read.

Here's what the HTTP request looks like for visiting appacademy.io:

POST / HTTP/1.0

Host: appacademy.io

Content-Length: 31

Content-Type: application/x-www-form-urlencoded

Host: appacademy.io

Connection: keep-alive

Upgrade-Insecure-Requests: 1

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_14\_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/76.0.3809.132 Safari/537.36

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8,application/signed-exchange;v=b3

Accept-Encoding: gzip, deflate

Accept-Language: en-US,en;q=0.9

username=azure&password=hunter2

**BREAKING DOWN THE REQUEST**

**REQUEST LINE**

* The first line of an HTTP request is called the request-line, and it sets the stage for everything to come. **POST / HTTP/1.0**

**1. Method -** indicated by an HTTP verb

**2. URI (Uniform Resource Indicator) -** identifies what you've requested

**3. HTTP version** you expect to use (usually HTTP/1.1 or HTTP/2).

* The URI identifies the requested resource. (e.g. users, posts, and likes)
* The resource in the URI identifies the object that the request wants to perform the operation on.
* Root resource or the root of the application which is used when the URI of the request looks like /

**In the appacademy.io example,**

* POST is the HTTP verb used for this request.
* The URI is /, or the root resource of the target.
* The version matches the most common HTTP version (1.1)

**HEADERS**

* The request-line sets the table, but it's the headers that describe the menu! Headers are key/value pairs that come after the request-line.

The header key or name is case-insensitive,

* ACCEPT-ENCODING or AcCePt-EnCoDiNg are all processed the same by the server.
* In the appacademy.io example, every line after the first line is a key-value pair separated by a colon.
* These headers define how the server might process the request.

**BODY**

* When sending data that doesn't fit in a header and is too complex for the URI, the data can be placed in the body of our HTTP request. Examples include form data or a file. The body comes right after the headers and can be formatted a few different ways.

The most common way form data is formatted is URL encoding. This is the default for data from web forms and looks a little like this:

name=claire&age=29&iceCream=vanilla

To tell the server how to interpret your body, it's important to set the Content-Type header. The Content-Type header value for URL encoding is application/x-www-form-urlencoded

**5 COMMON HTTP VERBS**

GET, POST, PUT, PATCH, and DELETE.

**1. GET -** is used for retrieving resources from the server, they only require that the server return a resource.

* + When you go to a link in the browser, the browser makes a GET request to the server.
  + These types of requests will never have a body. Any data you need to send in a GET request must be shared via the URI.

**2. POST –** is typically used for creating new resources on the server.

* + When you submit a form a POST request is generated.
  + These types of requests can have a body containing any data the server might need to complete your request, like your username & password or the contents of your shopping cart.

**3. PUT -** are used to update a resource on the server. These will contain the whole resource you'd like to update.

* They can have a body containing the data needed to update the resource.
* For example: when updating your name on a website, a PUT request will be generated containing not just your new name but also your user ID, email, etc.

**4. PATCH -** are also used to update a resource on the server. They do not require the whole resource to perform the update.

* They can have a body containing the data needed to update the resource.
* Keeping with our example of updating your name: a PATCH request would only require your new name, not the rest of your account details, to succeed.

**5. DELETE -** destroy resources on the server.

* + Do not recommended to include a body.
  + These might be saved database records, like removing a product that's sold out, or more ephemeral resources, like logging a user out of their current session.

**CONTENT TYPE HEADER**

* Any header beginning with Content- are headers that define details about the body of the request. Content headers will only show up on requests that support content in the body, so GET requests should never have any content headers!
* Content-Type - lets the server know the format of the request body data and how to process it.
* The values for the Content-Type header follow a standard and are called MIME types or media types. They define how the receiver of the data should format and process the data.

**EXAMPLES OF MIME TYPE**

**1. application/x-www-form-urlencoded -** info submitted directly from an HTML web form

**2. application/json -** JSON data format similar to JavaScript objects

**3. multipart/form-data -** info submitted from an HTML web form with multiple media types

**OTHER COMMON HEADERS**

* + Don't need to define yourself because they are set by the browser.

**1. HOST :** The root path for the URI. This is typically the domain we'd like to request our resource from.

**2. User-Agent :** This header displays information about which browser the request originated from (e.g. Chrome/76)

**3. Referer:** This defines the URL you're coming from(e.g. https://appacademy.io/)

**4. Accept:** headers indicate what the client can receive

* + When we go to most websites, our Accept header will be long to ensure we get all the various types of content that site might include.
  + However, we can modify this header in our requests to only get back certain types of data.
  + One common use is setting Accept: application/json to get a response in JSON format instead of HTML.

**EXAMPLES OF HTTP REQUEST**

**1. View the home page of Google**

method: GET

URI: /

headers:

Host: google.com

body: none

* To make a request to view the home page, you will need to use a GET request. The URI for the home page is / and the host is google.com. Since this is a GET request, there will be no body component.

**2. Click a link to an Instagram post that your friend thought was funny.**

method: GET

URI: /p/funny-post-name

headers:

Host: instagram.com

body: none

* Use a GET request to view a specific post. The URI will include an identifier for the specific post (such as the name or id). There is no body for the GET request.

**3. Upload your sick mixtape to SoundCloud!**

method: POST

URI: /audio

headers:

Host: soundcloud.com

Content-Type: audio/mpeg

body: the mixtape

* When you upload content, you need to make a POST request. The Content-Type header specifies the type of file that is being uploaded.

**4. Submit an Amazon review for a product through an HTML form**

method: POST

URI: /product-name/review

headers:

Host: amazon.com

Content-Type: x-www-form-urlencoded

body: the product review, the rating of the product

* When you make a POST request like this, the URI generally includes an identifier for the specific product followed by /review. The host will never include identifiers for products or reviews, and will always just include the domain information.

**HTTP RESPONSE COMPONENTS**

**HYPERTEXT DELIVERED**

* An HTTP response contains either the content requested or an explanation of why that content couldn't be delivered.
* It's important to note to the client that there's a problem with their request, but it's equally important to provide reliable, helpful details in the response.

**COMPONENTS OF AN HTTP RESPONSE**

* There's a status-line (instead of a request-line), headers that provide helpful metadata about the response, and the response body: a representation of the requested resource.

HTTP/1.1 200 OK

Content-Type: text/html; charset=utf-8

Transfer-Encoding: chunked

Connection: close

X-Frame-Options: SAMEORIGIN

X-Xss-Protection: 1; mode=block

X-Content-Type-Options: nosniff

Cache-Control: max-age=0, private, must-revalidate

Set-Cookie: \_rails-class-site\_session=BAh7CEkiD3Nlc3Npb25faWQGOgZFVEkiJTM5NWM5YTVlNTEyZDFmNTNlN; path=/; secure; HttpOnly

X-Request-Id: cf5f30dd-99d0-46d7-86d7-6fe57753b20d

X-Runtime: 0.006894

Strict-Transport-Security: max-age=31536000

Vary: Origin

Via: 1.1 vegur

Expect-CT: max-age=604800, report-uri="https://report-uri.cloudflare.com/cdn-cgi/beacon/expect-ct"

Server: cloudflare

CF-RAY: 51d641d1ca7d2d45-TXL

<!DOCTYPE html>

<html>

...

...

</html>

**STATUS**

* high-level overview of the server's intention.

HTTP/1.1 200 OK

**1. The HTTP version the server is responding with**

**2. Status-Code**

**3. Reason-Phrase**

These give a quick way of understanding if the request was successful or not.

**STATUS CODES 100-199: INFORMATIONAL**

* Let the client know that a request was received, and provide extra info from the server. Uncommon codes

**STATUS CODES 200-299: SUCCESSFUL**

* 200 OK: Request received and fulfilled. These usually come with a body that contains the resource you requested.
* 201 Created: Your request was received and a new record was created as a result. You'll often see this response to POST requests.

**STATUS CODES 300-399: REDIRECTION**

* Let the client know that there has been a change in the URL path and should redirect the user there.
* 301 Moved Permanently: The resource you requested is in a totally new location. This might be used if a webpage has changed domains, or if resources were reorganized on the server. Most clients will automatically process this redirect and send you to the new location, so you may not notice this response at all.
* 302 Found: This indicates that a resource has moved. However, this code is used to indicate a temporary move. It's not often that you see temporary moves online, but this code may be used to indicate a permanent move where the old domain should still be valid too. Clients will usually follow this redirect automatically as well, but you shouldn't necessarily update your links until the server returns a 301.

**When should you use a 302 Found?**

**Transition from HTTP to HTTPS.**

* + HTTPS is secure HTTP messaging, where requests & responses are encrypted so they can't be read by prying eyes while en route to their destinations.
* There are folks still trying to access our content from the older http:// approach!
* In this case, you'll return a 302 Found response to the client, letting them know that it's okay to keep navigating to http://our-website.com, but you're going to redirect them to https://our-website.com for their protection.
* These status codes are paired with a Location header that specifies the URL path that the client should redirect to.

**STATUS CODES 400-499: CLIENT ERROR**

* There was a typo, or maybe the resource requested is no longer available.
* 400 Bad Request: Whoops! The server received your request, but couldn't understand it. This is often referred to as malformed requests.
* 401 Unauthorized: The resource you requested may exist, but you're not allowed to see it without authentication. Either you didn't log in yet, or you tried to log in but your credentials aren't being accepted.
* 403 Forbidden: The resource you requested may exist, but you're not allowed to see it at all. This response code means this resource isn't accessible to you, even if you're logged in. You just don't have the correct permission to see it.
* 404 Not Found: The resource you requested doesn't exist. You may see this response if you have a typo in your request (for example: going to appaccccademy.io), or if you're looking for something that has been removed.
* 429 Too many Request: The specific user (client) has made too many requests in a certain amount of time.

**RISK ON 403**

* 403 Forbidden requests let the client know that a valid resource was requested. This can be a security risk!
* For example: if I guess that you have passwords.html on your website because you just want to be hacked, a 403 Forbidden response tells me I'm correct. For this reason, some sites will return a 404 Not Found for resources that exist but aren't accessible.
* A well-known example is GitHub. If you try to open a repository you don't have permission to access, GitHub will return a 404 Not Found even if your URL is correct! This protects you from random users guessing the names of your projects.

**STATUS CODES 500-599: SERVER ERROR**

* Your request was formatted correctly but the server couldn't do what you asked due to an internal problem.
* 500 Internal Server Error: Your request was received, and the server tried to process it, but something went awry!
* 503 Service Unavailable: Alert a client that the service is temporarily unavailable due to overloaded by traffic
* 504 Gateway Timeout: Your request was received but the server didn't respond in a reasonable amount of time.

**Timeout errors can be tricky:** your first instinct may be that your own connection is bad, but this code means the problem is likely on the server's side. You'll often see these when a server is no longer reachable (maybe due to an unexpected outage or power failure).

**HEADERS ON HTTP RESPONSE**

* CONTENT TYPE - lets the client know the format of the response body data and how to process it

**MIME TYPES OF CONTENT TYPE OF HTTP RESPONSE**

**1. text/html -** HTML document

**2. text/css -** CSS styles document

**3. text/javascript -** JavaScript script

**4. text/plain -** plain text

**5. image/png -** PNG Image

**6. application/json -** JSON data format similar to JavaScript object

**OTHER COMMON HEADERS**

**1. Location:** Used by the client for redirection responses. This contains the URL the client should redirect to.

**2. Expires:** When the response should be considered stale, or no longer valid. The Expires header lets your client cache responses (that is: save them locally to prevent having to repeatedly re-download them). The client may ignore requests to that same resource until after the date set in the Expires header.

**3. Content-Disposition:** This header lets the client know how to display the response, and is specifically devoted to whether the response should be visible to the client or delivered as a download.

Think about your own experience online: sometimes you click a button and get an immediate download, while in other cases you click a button and get to "preview" the content before you download it. This is controlled by the Content-Disposition header.

**4. Set-Cookie:** This header sends data back to the client to set on the cookie, a set of key/value pairs associated with the server's domain. Remember how HTTP is stateless? Cookies are one way to get around that! Set-Cookie may send back information like a unique ID for the user you've logged in as or details about other resources you've requested on this domain.

**BODY**

* the HTML of the page you're accessing
* The format of the body is dictated by the Content-Type header. MUST BE text/html
* Headers may change how the browser handles the body, but they won't modify the body's content.

**EXAMPLE OF HTTP RESPONSE**

**1. View the home page of Google**

status code: 200

headers:

Content-Type: text/html

body: home page

* After making a GET request to view the home page, you should receive a 200 (success!) status code. The body of the response will be the html of the home page that you requested.

**2. Edit a Facebook comment that isn't your own**

status code: 403

body: message saying you don't have the authority to edit this comment

* You would not be authorized to edit someone else's comment, so you would receive a 403 status code (forbidden), with an error message in the body explaining that you are not authorized to complete the action.

**3. Attempt to submit an Amazon review for a product through an HTML form, but you aren't logged in yet**

status code: 401

headers:

Content-Type: text/html

body: message saying that you are not logged in

* The 401 status code represents that the resource exists, but you are not authorized to access it.

**4. Try to sell one of your stocks on Robinhood, but Robinhood is having issues with their server because so many people are trying to sell that stock. The browser is displaying what seems like an infinite loading screen.**

status code: 504 or 500

* A 500 means an internal server error. A 504 means a timeout, generally caused by an issue on the server's side.

**5. Browse GIPHY for a cute puppy gif...but there are no cute puppy gifs!**

status code: 404

* The 404 means that the resource cannot be found (does not exist).

**RESTful ROUTES CONVENTION**

* REpresentational State Transfer

**ROUTES VS ENDPOINTS**

* Route - is the URL path for a request.
* Endpoint - is a pattern for a request that has a specific route and HTTP verb combination to define how the server should process the request and what the response is expected to look like.

**Endpoint: GET /users Route: /users**

**Endpoint: POST /users Route: /users**

**Endpoint: POST /session Route: /session**

**Endpoint: DELETE /login Route: /login**

**REST**

* + is a convention for defining endpoints that other developers can easily understand how the server may process requests with those endpoints and what they should expect from their responses.

**ROUTE PARAMETERS**

* + is a named segment of the URL path that acts as a placeholder for a changeable part of the path.
  + used to generalize routes to a certain pattern.
  + indicated by colon ( : ) followed by the name of the variable

**EXAMPLE:**

**Route: /tweets/17 Parameter: /tweets/:tweetId**

These are often used in documentation to group and generalize route paths with a variable segment.

**RULES OF REST**

* ReST is an architecture style for designing networked applications
* ReST is not an official standard. Instead, it's a set of rules/constraints.
* ReST defines six architectural constraint

**THREE OF THE CONSTRAINT**

**1. Decoupled client-server:** The client and the server should be decoupled so that they can evolve separately without any dependence on one another.

**2. Stateless:** This means that there is no necessary session between the client and the server. Data received from the server can be used by the client independently. This allows you to have short discrete operations. Luckily, this is a natural fit for HTTP operations in which requests are intended to be independent and short-lived.

**3.Uniform interface:** RESTful endpoints are meant to be self-describing and uniform in their definition. Each operation is intended to be separated by a separate endpoint or URL.

* In practical real world terms, most RESTful endpoints implement the classic CRUD (Create, Read, Update, Delete) operations against a resource that could just happen to be in your data model. This uniformity allows developers to easily learn the usage pattern of each endpoint.

**TWO KINDS OF URL: COLLECTION vs SINGULAR**

* Resource - is the data entity or group of data in the server that you want to perform a CRUD action on (read or manipulate the data).

**COLLECTION OF RESOURCES**

* A path that ends in a plural noun represents a COLLECTION OF RESOURCES

**EXAMPLES:**

**/invoices -** represent a collection of invoices that you're allowed to see

**/people -** represent the people in the application that you're allowed to see

**/houses -** represent a collection of houses

**SINGLE RESOURCE**

* A **RECORD** is a single set of data under a resource(e.g. /recordID)
* A path that combines a plural noun and a record identifier represents a **SINGLE RESOURCE** in your application.

**EXAMPLES:**

* **/invoices/PK-11 -** represent the single invoice that has the invoice number PK-11 (record id)
* **/people/10103 -** represent the single person with id 10103 (record id)
* **/houses/bdfa5ef9 -** represent the house with the specific globally unique record identifier bdfa5ef9

**TWITTER APP EXAMPLE:**

* **/my/tweets** would point to a collection of tweets made by you.
* **/my/tweets/17** would point to a tweet made by you with the id of 17.

**HOW TO CREATE RESTful ENDPOINTS**

* The endpoints that return HTML can follow a RESTful concept.
* HTML-based views can only generate GET and POST requests.
* Hyperlinks, or just links perform GET requests.
* Submission of an HTML form can only perform POST requests.

The following tables show the paths and HTTP verbs used to interact with HTML-based versions of a RESTful application:

* **GET /resource-name =** Index page: Get an HTML-based list of the resource
* **GET /resource-name/new =** Create form page: Show a form to create a new record for the resource
* **POST /resource-name =** Submit create form: Create a new record for the resource
* **GET /resource-name/:record-id =** Detail page: See the details of the specified record
* **GET /resource-name/:record-id/edit =** Edit form page: Show the edit form for the specified record
* **POST /resource-name/:record-id =** Submit edit form: Update the specified record
* **POST /resource-name/:record-id/delete =** Submit delete form: Delete the specified record

**TWITTER APP EXAMPLE:**

* **GET /my/tweets =** Index page: Get an HTML-based list of your tweets
* **GET /my/tweets/new =** Create form page: Show a form to create a new tweet
* **POST /my/tweets =** Submit create form: Create a new tweet
* **GET /my/tweets/17 =** Detail page: See the details of your tweet with the id of 17
* **GET /my/tweets/17/edit =** Edit form page: Show the edit form for your tweet with the id of 17
* **POST /my/tweets/17 =** Submit edit form: Update the tweet with the submitted details
* **POST /my/tweets/17/delete =** Submit delete form: Delete your tweet with the id of 17

**USAGE:**

* GET requests get HTML view pages as responses for the browser to render.
* POST requests usually end in a redirect to another HTML view page that makes sense. (e.g. After editing a resource, redirect to its detail page)

**NESTING RESOURCES**

* You can add resources to routes to create nested resources
* The URL path can consist of multiple collections and singular resources.

The following tables show the paths and HTTP verbs used to interact with nested resources for HTML-based versions of a RESTful application:

* **GET /resource-name/:record-id/nested-resource =** Index page: Get an HTML-based list of the nested resource related to the specified record
* **GET /resource-name/:record-id/nested-resource/new =** Create form page: Show a form to create a new record for the nested resource related to the specified record

**TWITTER APP EXAMPLE:**

* To create a comment resource for a specific tweet resource, the route should have the information about the specific tweet resource and that a comment should be created for it.

**POST /tweets/:tweetId/comments.**

* The endpoint for seeing all the comments for a tweet can look like

**GET /tweets/:tweetId/comments.**

**RESTful vs other conventions**

* While RESTful routes are certainly one of the most popular styles for designing a web application, it is not the only way.
* As you browse other websites, you may see variations of RESTful routes or endpoints that look nothing like REST.

**SUMMARY:**

* REST is a convention for routes and is not an official standard.

**1. Resource:** is an identifiable object or entity that can be manipulated, such as a "User" in a social media application accessed via `/users/{userId}`.

**2. Route:** defines the URL pattern associated with a specific functionality, like `/tasks/{taskId}` in a task management application.

**3. Endpoint:** is a specific URL that combines the route and HTTP method to perform a particular action, such as `GET /tasks/{taskId}` for retrieving task details.

**EXAMPLES:**

**1. to access the edit page of a post's comment.**

GET /comments/:commentId/edit

**2. to submit a creation of a post's comment.**

POST /posts/:postId/comments

**HTTP REQUEST AND RESPONSE CYCLE**

* On the left is the client side, or the browser.
* The internet, in the middle, is a series of these client requests and server responses.
* On the right is the server side, with a database where data is stored.

**The browser's role in the request-response cycle**

**1.** Parses HTML, CSS, and JS

**2.** Renders that information to the user by constructing and rendering a DOM tree

**HTML FORM SUBMISSION REQUEST/ RESPONSE**

* HTML attributes that influence the components of the request made when the form is submitted.
* **method -** method of the request, can only be set to "POST"
* **action -** URL path of the request

**EXAMPLE:**

* HTML form to submit a POST /dog request

<form method="post" action="/dog">

<input type="text" name="name" />

<select name="color">

<option value="black">Black</option>

<option value="brown">Brown</option>

<option value="yellow">Yellow</option>

<option value="white">White</option>

</select>

<input type="number" name="age" />

<textarea name="description"></textarea>

<button type="submit">Create Dog</button>

</form>

**SENDING REQUEST:**

**1.** When the form is submitted the request body will contain key-value pairs for the form inputs, name, color, age, and description.

**2.** The components of the request will be:

**method -** defined by the method HTML form attribute

**URL path -** defined by the action HTML form attribute

**Content-Type header -** application/x-www-form-urlencoded

**body -** form input names and values

**SERVER RESPONSE:**

**1.** The server should parse the body of the request, do some CRUD action with the data, then redirect the user to another page.

**2.** The components of the response typically looks like:

**status code -** 302 (a redirection response code)

**Location header -** path to redirect the user to

**body -** none

**PROCESS OF FORM SUBMISSION:**

**1.** Form is submitted

**2.** Browser makes request to the server

**3.** Server parses the request body and does some CRUD action with the data

**4.** Server sends a redirection response

**5.** Browser receives response

**6.** Browser redirects user to the path specified in the Location header of the response

**HTTP NODE.JS PACKAGE**

**CREATING A SERVER**

**1.** import built in http package

**2.** call the createServer method on the import

**3.** pass in a function as a single argument

const http = require('http')

const server = http.createServer((req, res) => {

// ...

})

* Nothing will really happen yet because the server isn't connected to your network to listen for incoming requests and send outgoing responses.

**LISTENING FOR REQUEST ON A PORT**

**4.** define the port number and assign it to a variable `port`

* Port numbers can range from 0 to 65535, some ports are reserved
* Common development port numbers to use are 3000, 5000, and 8000.

**5.** use the listen method on the newly created server

**6.** pass in 2 arguments (port, callback <= will run once the server connection to the port is successful)

**7.** log a message to know the server is connected

**EXAMPLE:**

// server.js

const http = require('http');

const server = http.createServer((req, res) => {

// ...

});

const port = 5000;

server.listen(port, () => console.log('Server is listening on port', port));

**node server.js**

**REQUEST OBJECT**

* http will create a Request object that holds methods and properties to interact with the components of the request

To examine the contents of Request object:

**1.** Log the req parameter in the createServer function

**2.** Make any request to http://localhost:5000 using your browser or Postman

const server = http.createServer((req, res) => {

console.log(req);

});

**Properties and methods that you will be using:**

**1. method -** is a string of the method of the request

**2. url -** is a string of the URL path of the request /hello-world.

**3. headers -** is an object with the key-value pairs as the header names and values

**4. on -** Method that listens for an event on the request and triggers a callback function to run when that event is triggered

* The method, URL, and headers properties will be used to identify the route of the request and formulate a response based on that route.
* The on method will be used to parse the body of the request.

**RESPONSE OBJECT**

* http will create a Response object with every request made to the server.
* To examine the contents of the Response object, you will need to do similar steps as examining the Request object

**Properties and methods on res that you will be using:**

**1. statusCode -** is the status code of the response

**2. setHeader -** Method that sets a header name to a value

**3. write -** Method that allows you to add to the body of the request

**4. end -** Method that allows you to add to the body of the request AND send the response

**FORMULATE AND SEND A RESPONSE IN HTTP**

**SET STATUS CODE**

* to set the status code for every request that comes into the server:

const server = http.createServer((req, res) => {

res.statusCode = 500;

});

**SET HEADER**

* header name as the first argument and header value as the second.

const server = http.createServer((req, res) => {

res.setHeader("Content-Type", "text/css");

});

**WRITE THE RESPONSE BODY**

**FIRST WAY:**

* write method doesn't set the body of the response, it adds to the body of the response. Can call this method multiple times on the same res object to continuously add to the body.

const server = http.createServer((req, res) => {

res.write('Hello');

res.write(' ');

res.write('World');

res.write('!');

});

* you can either pass in the full string into write or separate it by line

**SECOND WAY:**

* end method also doesn't set the body of the response, but adds to the body.
* ends the creation of the response and send it.
* you cannot call the end method multiple times for a single request.

const server = http.createServer((req, res) => {

res.write('Hello World');

res.end('!');

});

**OR choose not to pass in any arguments into the end**

const server = http.createServer((req, res) => {

res.status = 200;

res.end();

});

* Start the server and use Postman to make a request to the server.
* Examine the response sent back to Postman to confirm the status code, headers, and body of the response formulated.

**HANGING SERVER**

* Your server is not returning a response for any requests, your server will hang or not be able to accept any other requests after the initial request.
* Make sure you are always sending a response for all your requests to your http server even if you have to send an error response!

**PARSE THE REQUEST BODY IN HTTP**

* how to parse the body of the request that have a Content-Type header of application/x-www-form-urlencoded.

**READING THE BODY OF THE REQUEST**

* The entire body of the request will be separated into data packets that you have to put together to get the entire request body as a single string.
* To read a data packet and add to the body of the request in http, you have to listen for the data event on the req object.
* The data event will be triggered whenever a data packet is received. Then, you need to add the contents of the data packet to the content compiled from data packets that were already received.

**1.** Call the req.on method to listen to the data event.

**2.** Concatenate the data received to a string representing the body of the request getting put together.

**3.** The end event on the request object will be triggered once the entire server finishes receiving the request body.

const server = http.createServer((req, res) => {

let reqBody = '';

req.on('data', (data) => {

reqBody += data;

});

req.on('end', () => {

console.log(reqBody);

});

});

* You should have the entire body of the request and be able to read it. However,
* When the Content-Type of the request is application/x-www-form-urlencoded,

the request body will be a string with the names and values of HTML form inputs are separated by &, with a = between the key and the value.

**EXAMPLE:**

name=Fido&color=black&age=1&description=Hello+World%21

* To convert these key-value pairs from this encoded string into an object, you need to parse the string into a JavaScript object.

**1.** First, separate the key-value pairs in the string from each other by separating the string by the &.

**[name=Fido, color=black, age=1, description=Hello+World%21]**

**2.** Next separate the key from the value by separating the key-value pair by the =.

**[[name, Fido], [color, black], [age, 1], [description, Hello+World%21]]**

**3.** Then, replace the plus symbols, +, in the values with a space.

**[[name, Fido], [color, black], [age, 1], [description, Hello World%21]]**

**4.** Also, decode the values from Percent Encoding.

**(Use Node.js built-in decodeURIComponent(encodedString) function)**

**[[name, Fido], [color, black], [age, 1], [description, Hello World!]]**

**5.** Finally turn the key-value pairs into an object!

**{**

**name: "Fido",**

**color: "black",**

**age: "1",**

**description: "Hello World!"**

**}**

**CREATE ROUTE HANDLERS IN HTTP**

* To get the server to send responses based on specific routes, you need to create different route handlers.

**DEFINING A ROUTE HANDLER**

* A set of code that will be executing for a particular route or request method and URL path combination.
* A response formulated and sent by one route handler will be different from the another route handler.

**CREATING ROUTE HANDLER BY USING CONDITIONALS**

const server = http.createServer((req, res) => {

if (req.method === 'GET' && req.url === '/') {

res.statusCode = 200;

res.setHeader('Content-Type', 'text/plain');

return res.end('Splash Page');

}

if (req.method === 'POST' && req.url === '/cat') {

res.statusCode = 201;

res.setHeader('Content-Type', 'text/plain');

return res.end('Created a Cat!');

}

});

* The response cannot be changed after you call res.end, but it's best practice to exit out of the request/response cycle after you send the response by using `return`.

**STATIC ASSETS**

* A server is used for sending all kinds of data.
* A static asset is some data or resource that doesn't change no matter how many times you ask the server for it. It's usually just a file that the server holds in memory and sends to a client when asked for.

**EXAMPLE:**

CSS and image files

**SERVING STATIC ASSETS**

* The URL path for requesting a static asset usually includes the static asset's file extension at the end of the URL path.

**EXAMPLE:**

Request for a file called dog.jpg would have a URL path of something like /images/dog.jpg

**SERVE STATIC ASSETS IN HTTP**

**FINDING AND READING FILES**

* Use the built-in fs Node.js library method that takes in a file path and returns the contents of the file at that path.
* If you pass in a string of utf-8 as a second argument, then the file contents can be read as a string.

const fs = require('fs');

const fileContents = fs.readFileSync('./file-name.txt', 'utf-8');

* If the file is not found, the method will throw an error.

**SENDING FILES**

* To send a file as a static asset in your http server as a response, you can write the contents of a file that you read as the body of the response. Then, set the appropriate status code. Finally set the Content-Type header specific for the file.

const server = http.createServer((req, res) => {

const catImage = fs.readFileSync('./images/cat.png');

res.statusCode = 200;

res.setHeader('Content-Type', 'image/png');

res.end(catImage);

});

* Best practice to set the Content-Type headers specific to the extension of the file you are sending.

**NOTE:**

* setHeader can only be called once
* If your method is POST, you need to set the setHeader('Location', where), this is where the response will redirect since POST method has a status code of 302
* res.end() only accepts single argument as a response

**HTML TEMPLATING**

**The Problem**

* What if you want to insert data into the HTML page?
* For example, there are about 200 billion tweets on Twitter created a year. To be able to display all those tweet pages to users using static HTML files, you would have to create an HTML page for every single tweet. That's over 200 billion files!

**The Solution**

* What if you could insert the data of a single tweet into a template?
* Each tweet page is the exact same except the data about the tweet displayed.

**HTML templating is when you insert specific elements of data into an HTML file.**

**TEMPLATE ENGINES**

* A template engine is usually a package or library that processes HTML template files and inserts data into it.
* They attempt to make HTML templating easier for developers.

**POPULAR TEMPLATE ENGINES**

* **Node -** Pug and Handlebars
* **Python -** Genshi and Jinja

**BASIC HTML TEMPLATING IN HTTP**

* In JavaScript string interpolation**, ${ }** (dollar sign with curly braces) is wrapped around the variable
* In the HTML template, **#{ }** (hash tag sign with curly braces) will be used to indicate the start and end of the variable name.

**EXAMPLE:**

* An HTML file `profile-page.html`,
* user's username should be inserted in #{username}
* user's biography should be inserted in #{biography}.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>#{username}'s Profile Page</title>

</head>

<body>

<h1>Welcome to #{username}'s profile page!</h1>

<p>#{biography}</p>

<h2>Comment Box</h2>

<form method="post" action="/comment">

<p>To send a comment to #{username}, fill out this form</p>

<textarea name="commentBody"></textarea>

<button type="submit">Comment</button>

</form>

</body>

</html>

**USING AN HTML TEMPLATE IN NODE.JS**

* read the file contents to a string, then replace the variables inside of the string with the appropriate user information.

**The result of reading an HTML file using fs.readFileSync turns the file contents into a JavaScript string.**

* Use the String.replace method to replace variables inside of the template string with values.

**const fs = require('fs');**

// Get the file contents of the profile-page.html as a string

**const htmlTemplate = fs.readFileSync('./profile-page.html', 'utf-8');**

**const htmlPage = htmlTemplate**

// replace all instances of #{username} with 'DemoUser'

**.replace(/#{username}/g, 'DemoUser')**

// replace all instances of #{biography} with 'Hello World!'

**.replace(/#{biography}/g, 'Hello World!');**

**OUTPUT:**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>DemoUser's Profile Page</title>

</head>

<body>

<h1>Welcome to DemoUser's profile page!</h1>

<p>Hello World!</p>

<h2>Comment Box</h2>

<form method="post" action="/comment">

<p>To send a comment to DemoUser, fill out this form</p>

<textarea name="commentBody"></textarea>

<button type="submit">Comment</button>

</form>

</body>

</html>

**NOTES:**

* If you’re using GET as a method you can use res.end()
* If you’re using POST as a method you will use res.write()