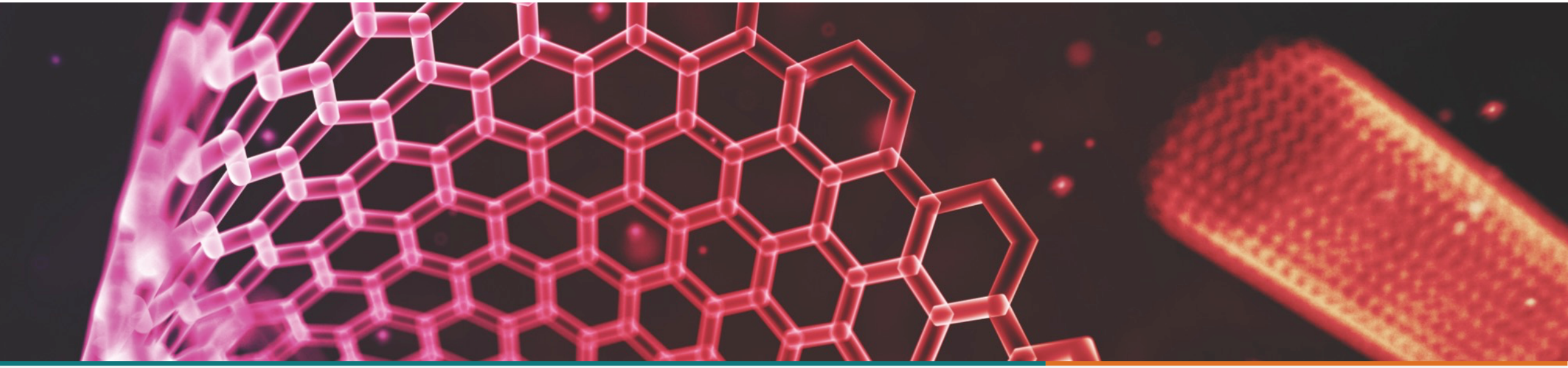
STEVENS
INSTITUTE of TECHNOLOGY
Schaefer School of Engineering & Science

# CS 546 – Web Programming I

# Fundamentals of Web Development




STEVENS

INSTITUTE of TECHNOLOGY

Schaefer School of Engineering & Science

**stevens.edu**

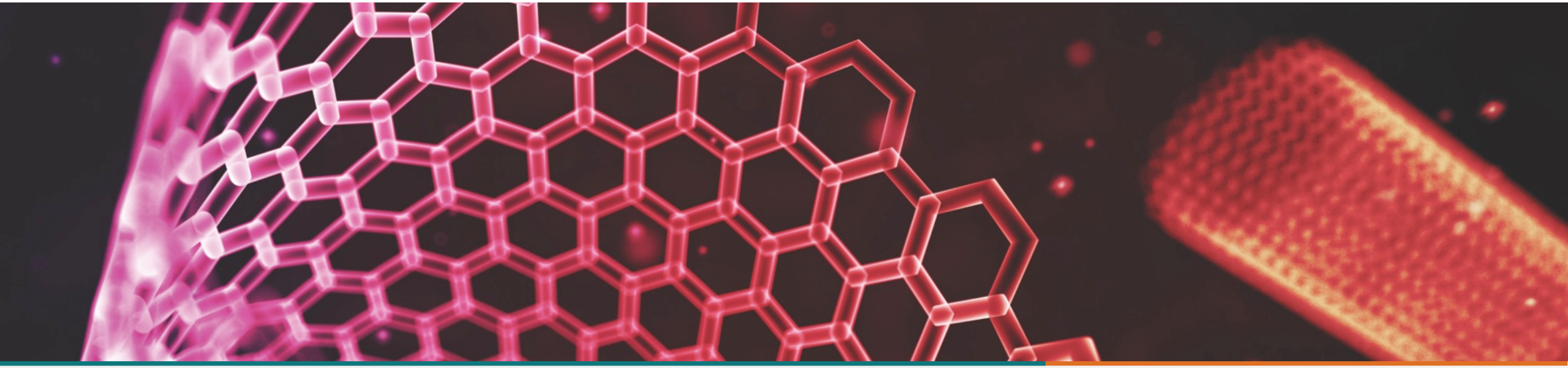
Patrick Hill

Adjunct Professor

Computer Science Department

Patrick.Hill@stevens.edu

# What is the Web and How Does it Work?



# The Core Process of the Web: HTTP

One of the most commonly used services on the Internet is the World Wide Web (WWW). The application protocol that makes the web work is **Hypertext Transfer Protocol** or **HTTP**.

Do not confuse this with the Hypertext Markup Language (HTML). HTML is the language used to write web pages. HTTP is the protocol that clients and web servers use to communicate with each other over the Internet. It is an application level protocol because it sits on top of the TCP layer in the protocol stack and is used by specific applications to talk to one another. In this case the applications are web browsers and web servers.

# The Core Process of the Web: HTTP

HTTP is a connectionless text-based protocol. Clients (usually web browsers, but not limited to just web browsers) send requests to web servers for web elements such as web pages, assets and data.

After the request is serviced by a server, the connection between client and server across the Internet is disconnected. A new connection must be made for each request. Most protocols are connection oriented. This means that the two computers communicating with each other keep the connection open over the Internet. HTTP does not however. Before an HTTP request can be made by a client, a new connection must be made to the server.

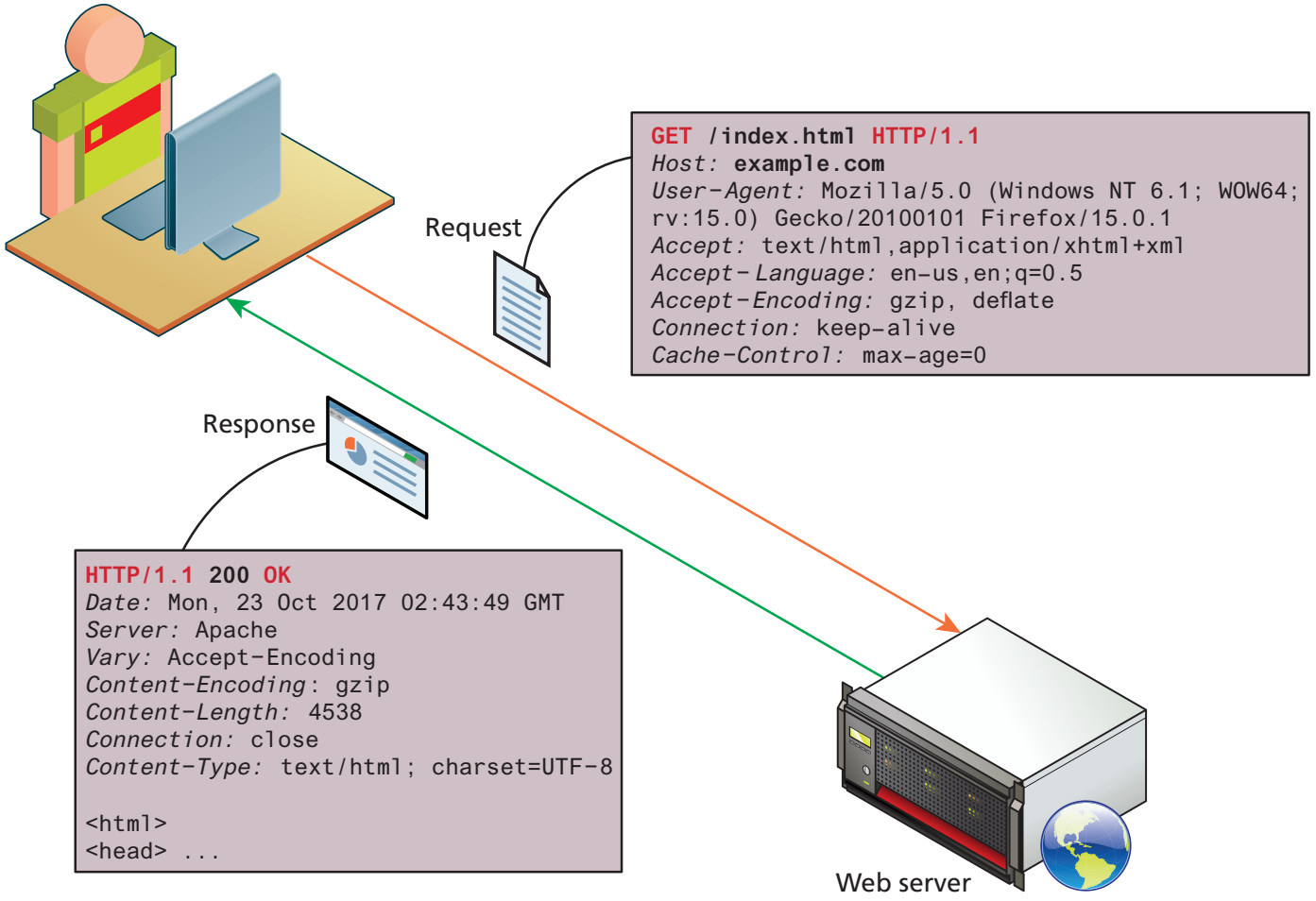
# The Core Process of the Web

At the end of the day, the web is all about the communication of ideas. Everything in the web can be seen as a request and a response

- When you go to a news website, you're requesting news and receiving news in response.
- When you go to a shopping website, you're requesting product information and receiving relevant information.
- When your server receives input, it determines what to do with that input and outputs the proper response.

Your duty as a web developer is to make that communication possible. Your programs will get a request and give a response and allow that communication to occur as smoothly as possible.

# The Core Process of the Web



# The Request

Every time you navigate to a website, your browser sends a request on your behalf to the server.

- There is a lot to an HTTP request! https://developer.mozilla.org/en-US/docs/Web/HTTP
- Each request follows a standardized process! https://www.w3.org/wiki/How_does_the_Internet_work

Every request is formatted in a specific way, with the same data provided on each request.  A request to http://google.com/ would look like:

```
GET /?gws_rd=ssl HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:40.0) Gecko/20100101 Firefox/40.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Cookie: PREF=ID=1111111111111111:FF=0:TM=1441084937:LM=1441084937:V=1:S=DKD8wPI-NAGQztx5; NID=71=0zCx
Connection: keep-alive
Cache-Control: max-age=0
```

# Parts of That Request

The HTTP request has a lot of data that is sent including:

- HTTP verb signifying what action you are trying to (**GET**, **POST PUT**, **DELETE**).
- The protocol.
- The server you want to connect to (the HOST).
- The User-Agent: Information about the user's browser, platform etc..
- The location of the resource you want to access on that server (the location).
- Headers: headers are metadata about your request, these include cookies!.
- Request body (if there is one).

# Parts of That Request

**GET** Request:

```
GET /?gws_rd=ssl HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:40.0) Gecko/20100101 Firefox/40.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Cookie: PREF=ID=1111111111111111:FF=0:TM=1441084937:LM=1441084937:V=1:S=DKD8wPI-NAGQztx5; NID=71=0zCx
Connection: keep-alive
Cache-Control: max-age=0
```

**POST** Request:

```
POST / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh;… )… Firefox/51.0
Accept: text/html,application/xhtml+xml,…,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Content-Type: multipart/form-data; boundary=-12656974
Content-Length: 345

-12656974
(more data)
```

Request headers

General headers

Entity headers

# What Does the Server Do with That Information?


The server reads that request and determines what needs to be done in order to generate a response that makes sense for the data that the server has been given.

The server uses data sent in the request in order to generate a response.  Some common types of data in a request that servers use are:

- Querystring parameters
- Headers
- Cookies
- Request body

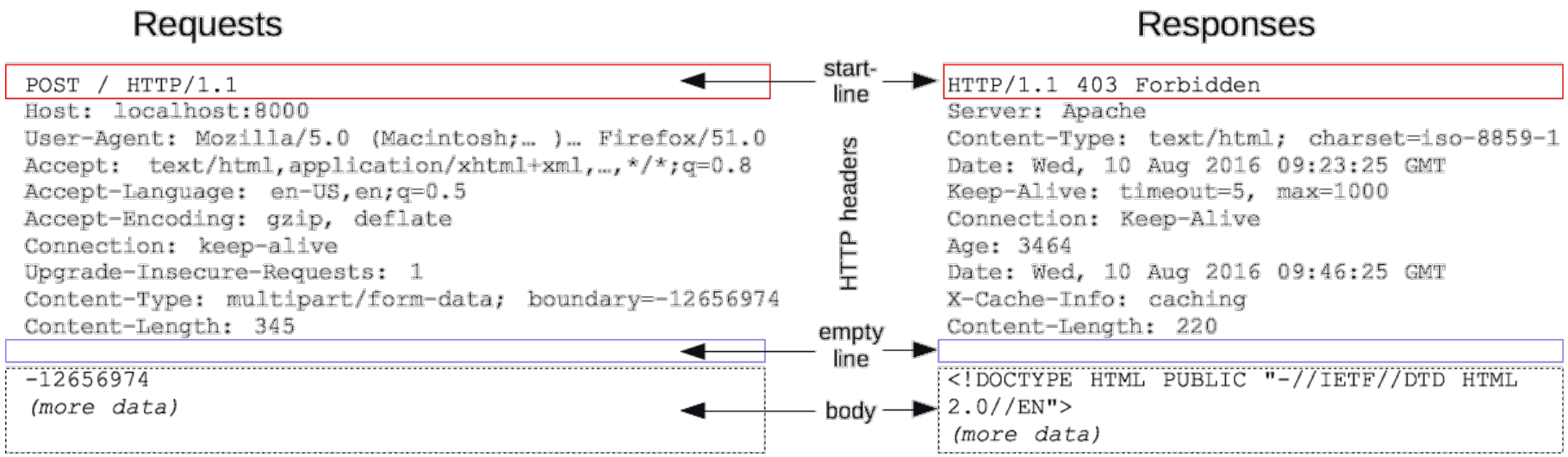
# The Response

The server sends back a response that is similar to the  request. It contains:

- A status code
  - Indicates whether or not the operation succeeded
- A set of headers
  - Cookies, data about the response such as content type
  - This is often “meta-data” about the response.
- Some form of response body:
  - An HTML Document
  - A JSON response
  - A File Stream
  - Plain Text
  - Etc.

```
HTTP/2.0 200 OK
Cache-Control: private, max-age=0
Content-Encoding: gzip
Content-Type: text/html; charset=UTF-8
Date: Tue, 01 Sep 2015 05:31:36 GMT
Expires: -1
Server: gws
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
X-Firefox-Spdy: h2
```

# Request/Response Example



# Status Codes

Each response must return a status code indicating whether the request was successful, and a description is often included with each of the status code

Status codes in the...

- 200-299 range indicate a successful operation
- 300-399 range indicate some sort of redirection must occur
- 400-499 range indicate an error was made by the client during the request
- 500-599 range indicate some sort of error occurred on the server

You will use different status codes to describe different errors in this course  Some status codes:

https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

# The Browser

There are many different browsers, but they each allow for the same fundamental actions to occur:

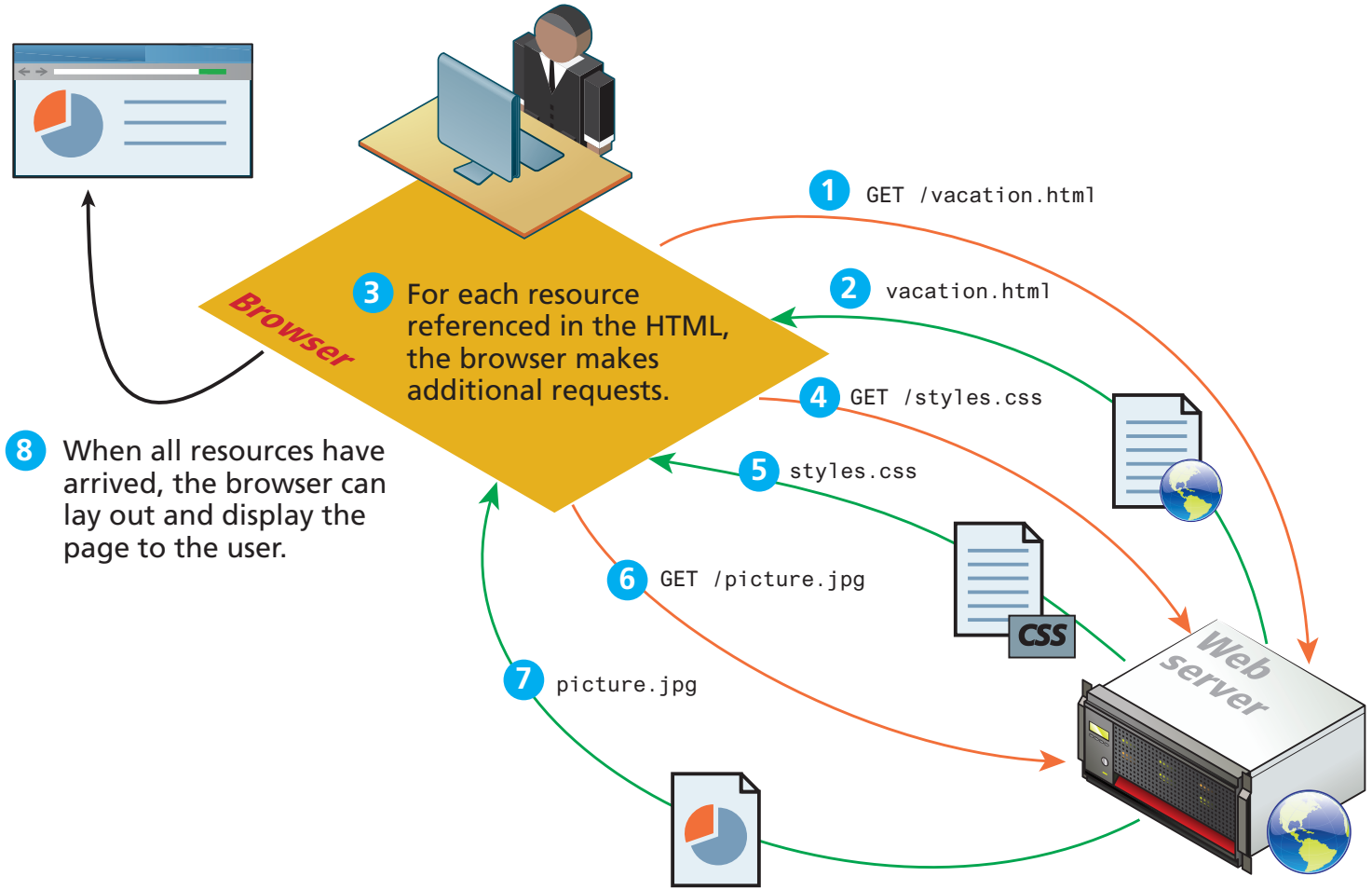
- They allow a user to navigate to a URL
- They then submit a request on the user's behalf to the server at that URL
- They receive a response back from the server
- They render these responses
- They execute any code that they may have received in these responses, for the lifetime of the user being on the web page.

# The Browser

When you type a URL into a web browser, this is what happens:

1. If the URL contains a domain name, the browser first connects to a domain name server and retrieves the corresponding IP address for the web server.
2. The web browser connects to the web server and sends an HTTP request (via the protocol stack) for the desired web page.
3. The web server receives the request and checks for the desired page. If the page exists, the web server sends it. If the server cannot find the requested page, it will send an HTTP 404 error message. (404 means 'Page Not Found' as anyone who has surfed the web probably knows.)
4. The web browser receives the page back and the connection is closed.
5. The browser then parses through the page and looks for other page elements it needs to complete the web page. These usually include images, stylesheets, scripts, etc.
6. For each element needed, the browser makes additional connections and HTTP requests to the server for each element.
7. When the browser has finished loading all images, stylesheets, scripts, etc. the page will be completely loaded in the browser window.

# The Browser


1 GET /vacation.html
2 vacation.html
3 For each resource referenced in the HTML, the browser makes additional requests.
Browser
4 GET /styles.css
5 styles.css
6 GET /picture.jpg
7 picture.jpg
8 When all resources have arrived, the browser can lay out and display the page to the user.
CSS
Web server

# More Than Just Web Pages

There is a fundamental idea you must hold onto from now on:

- Your server is not a tool to transmit a web page to a user
- Your server is a tool to transmit information in response to a request

Web pages are a very small part of the internet. As a web developer:

- You will transmit text that represents data in JSON format
- You will transmit text that represents an HTML document
- You will transmit text that represents an XML document
- You will transmit binary data
- You will transmit media files

**None of these require a browser!**

You will often write programs that make requests and rely on responses for data!

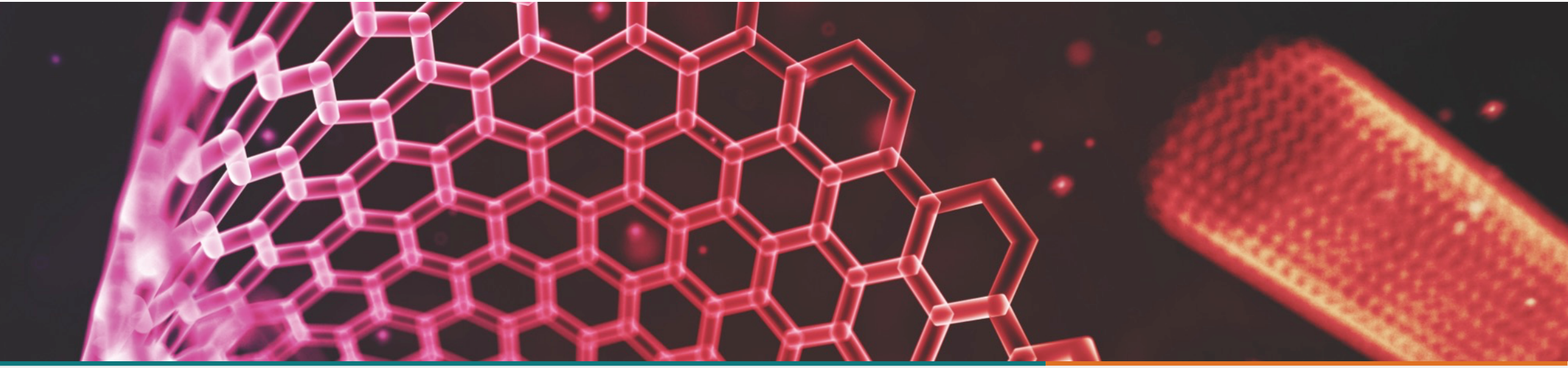
# What is JSON?

JSON is JavaScript Object Notation and is probably the most popular way for representing data on the internet.

Many technologies in the modern era can easily communicate with each other by using JSON as a common way of representing data between them.

```
{
    "firstName": "Patrick",
    "lastName": "Hill",
    "favoriteDrink": "Coffee",
    "thingsIEnjoy": ["Playing music", "Spending time with my family", "Playing with my dogs"]
}
```

# Running a Node.js Web Server



# Express

Express is a very popular node package that is distributed on NPM which allows you to configure and run an entire web server.

- http://expressjs.com/

Express allows you to configure different routes and how they should compose a response.

Essentially, by using the Express Node module, you will use code to configure a server that will listen to requests and send out responses.

- This is all a web server is! It's not magic, it's just something that takes in requests and sends back responses!

# What is a Route?

Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

- http://expressjs.com/en/starter/basic-routing.html

For your web applications, you will configure many routes that will each perform some action.

By configuring routes, you can have the same URL perform different tasks based on having different request methods (HTTP Verbs)

# Request Methods

There are many different types of request methods.  For this course, you will be using 4:

- **GET**
  - The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.
- **POST**
  - The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.
- **PUT**
  - The PUT method replaces all current representations of the target resource with the request payload.
- **DELETE**
  - The DELETE method deletes the specified resource.
- Other Methods
  - https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

# GET Requests

Let us use a blog as an example.

GET requests are made when the client is requesting the representation of a resource

| Location | Client is requesting… | Server is responding with… |
|---|---|---|
| **http://myblog.com** | The homepage of the blog | An HTML document with the most recent blog posts perhaps |
| **http://myblog.com/post/2** | The blog post with an ID of 2 | An HTML document with the contents of the blog post |
| **http://myblog.com/editor** | A page with an editor to write new posts | An HTML document with a form so the user can compose a new blog post |
| **http://myblog.com/post/2.json** | A blog post with and ID of 2, but only the raw JSON data of the content | A JSON document representing the content of the blog post |

# POST Requests

POST requests are made when the client is requesting to create some form of resource.

POST data comes with a message body that describes the content (which usually comes from an HTML form the user filled out, but not always)

| Location | Client is requesting… | Server is responding with… |
|---|---|---|
| **http://myblog.com/post** | The data from the form that was filled out in the editor is submitted to this route for processing | An HTML document with the newly created post |

# Creating Route Modules

If you open routes/posts.js from this week's lecture code, you will see a router being created, with three routes setup; one to get a list of all posts, one to create a new post, and one to get a specific post.

A router is a set of rules that dictate how to respond to requests with particular paths and HTTP verbs.

```
router.get("/", async (req, res) => {
  try {
    const postList = await postData.getAllPosts
    res.json(postList);
  } catch (e) {
    res.status(500).send();
  }
});
```

```
router.get("/:id", async (req, res) => {
  try {
    const post = await postData.getPostById(req.params.id);
    res.json(post);
  } catch (e) {
    res.status(404).json({ message: "Post not found" });
  }
});
```

# General Organization

When running a server, running ***`npm start`*** should start your server and print a message with its address (including port).

You will have:

- A file that creates, configures, and runs your server (***app.js***)
- A folder with all your route modules (**./routes**)
  - An index file in the route folder that returns a function that attaches all your routes to your app (**./routes/index.js**)
  - Route modules (**./routes/posts.js**)
- A folder for your data access layer modules (***./data***)
  - Your connection (**./data/mongoConnection.js**)
  - Your collection file (**./data/mongoCollection.js**)
  - Your data modules (**./data/posts.js**)
  - Potentially, an index file that exports all other data access modules

# Express

**app.js**:

```
const express = require("express");
const app = express();
const configRoutes = require("./routes");

configRoutes(app);

app.listen(3000, () => {
console.log("We've now got a server!");
console.log("Your routes will be running on http://localhost:3000");
});
```

# Requiring a Folder

You may require a folder by placing a file called ***index.js*** inside the folder.

This is useful for organizing things, such as defining all your routes in a routes folder and all your data modules in a data folder.

`const configRoutes = require("./routes");` Here we are requiring the entire routes folder. If we require a whole folder, we need an index.js inside to "glue" the other files together.

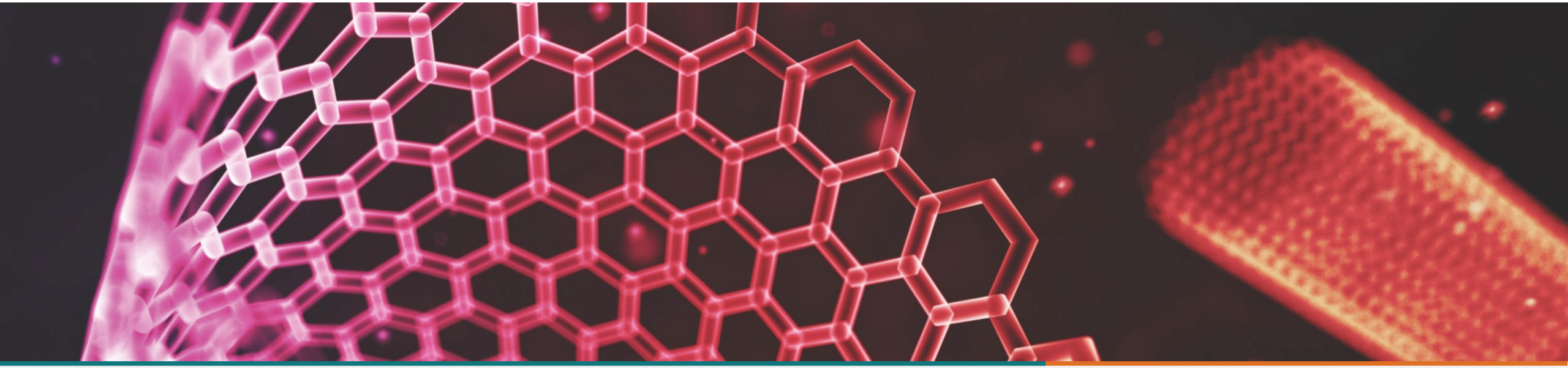

```
const postRoutes = require("./posts");
const userRoutes = require("./users");

const constructorMethod = app => {
  app.use("/posts", postRoutes);
  app.use("/users", userRoutes);

  app.use("*", (req, res) => {
    res.status(404).json({ error: "Not found" });
  });
};

module.exports = constructorMethod;
```

# Node as an API (GET)



# The Purpose

The purpose of our first servers will be simple: to create an API (Application Program Interface)

- An API is a way to interact with a program.

We will begin with treating our servers as an access point to run our applications.  Our first server will allow us to read some blog posts that will be stored in MongoDB.

# Seeding Our Database


"Seeding" a database means adding initial data.

By running `npm run seed` we can run a task we've defined in our `package.json` file, which will run a script that will seed our database with a single user and a single post.

# Requesting Data

There is a great deal of data available in a request! For now, we will be focusing on requests and  only be making simple responses.

In our blog, we will be using request parameters

- Request parameters are dynamic parameters we define in our route.

You should read up on the API for request data

- http://expressjs.com/en/api.html#req

# Sending a Response

For now, we will only be using two response methods:

- `res.json(someData);` will send a JSON object as the response with a status code of 200 (unless otherwise specified)
- `res.status(statusCode).send();` will issue a response with the provided status code

# Questions?