2. Building web server (http & Express)

2.1. Introduction: revision on core Node modules

- We have discussed the most common core Node modules in our previous class. It is better to briefly revise what these core Node modules are and what they do in this class before jumping into explaining how we can use them.
- **The most used core Node modules**: These are built-in modules and come automatically when we install Node.js. Here are the most common code Node modules:
  - **fs**: It is used to handle file and directory/folder system. **Examples**: creating a folder, accessing and opening a file, editing a file, copying a file, removing a file or a directory.
  - **os**: It provides information about your computer's operating system. **Examples**: Getting the name of the host computer and getting the right network information of the computer.
  - **path:** It includes methods to deal with file paths. **Examples**: Identify the path of a specific file or folder, identify the path to the root directory, and identify the extension of a file.
  - **http**: It creates an HTTP server in Node.js. **Examples**: Ability to receive and handle HTTP requests and manage connections.
  - **Events:** It is used to own and trigger events. **Examples**: Creating, firing, and listening for your own events.

2.2. Most common core Node modules (fs, os, path, http)

- **How to use these core Node modules**: Please remember to always import (require) any core Node module into your local module before you use it.
  - **Getting detailed location info about a file:** Use the "path" Module
    - **Documentation**: https://nodejs.org/api/path.html
    - **Path parsing**: The method we are going to use often with path module is the parse() method. The parse method takes

some "path" as an argument and returns an object. The returned object will have these five properties:

- dir <string>, root <string>, base <string>, name <string>, ext <string>

- **Documentation for path parse()**:
  - https://nodejs.org/api/path.html#path_path_parse_path
- **Example**: To get the name of your current file /module

```
// myFirst.js
const path = require("path"); // importing path module
const parsedFileName = path.parse(__filename); // use the parse method in path module to get significant elements of the path
console.log(parsedFileName.name); // will print name of your file, i.e., myFirst.js
```

- **Getting information from our operating system**: Use the os module for that
  - **Documentation**: https://nodejs.org/api/os.html
  - **Example**: Let us get the free memory of the working computer.

```
// myFirst.js
const os = require("os");
const freeRAMinByte = os.freemem();
console.log(freeRAMinByte);// prints free memory in byte
```

- **Dealing with files**: Use the fs module for that
  - **Documentation**: https://nodejs.org/api/fs.html
  - The fs module is one of the most used core Node modules.
  - We will learn about the synchronous and asynchronous ways of using modules next week. Let us focus on the main methods for now.

- **Common methods:**
    - **readFileSync**: This method lets us read a file that is on the server. It takes two arguments: the file to read and the encoding method used.
    - **writeFileSync**: It takes two arguments: the file to write on and the content to write
- **Example**: Let's read and print in the console what is written under one of our files in our computer, called myList.txt

```
// myFirst.js
var fs = require("fs");
const dataWritten = fs.readFileSync(" myList.txt", "utf8");
console.log(dataWritten); // prints the text in myList.txt
```

- **Emitting Events**: It is used to own and trigger events
    - **Documentation**: https://nodejs.org/api/events.html
    - **The eventsEmitter class**: This is the one class that you use the most in this module. The EventEmitter class provides on() and emit() methods for binding and triggering events. There are two methods inside of this class: the on() and the emit() methods:
        - **The on() method**: It is used to listen for a specific event happening on Node
        - **The emit() method**: This is what signals the event.
        - **Example**: Execute a function when an event emits.

```
// myFirst.js
const EventEmitter = require("events");
const myEmitter = new EventEmitter();
function evangadi() {
console.log("My first event here!");
}
```

```
// When "myEvent" event is emitted, the above callback function will be executed
myEmitter.on("myEvent", evangadi);
myEmitter.emit("myEvent");
```

2.3. Building HTTP web server using HTTP module: defining a web server and listener

- **Web server**: A web server can refer to 2 things:
  - **Web server as a hardware/computer**: It can refer to the hardware (the computer) that helps deliver/handle HTTP requests and responses. When you view a webpage in your browser, you are making a request to another computer on the internet, which then provides you the webpage as a response. That computer you are talking to via the internet is a web server.
  - **Web server as a software**: It can refer to the software (computer application) that that helps deliver/handle HTTP requests. The software that handles the HTTP request is called HTTP server.
  - **Web server**: A Web server is a computer that has an HTTP server software installed on it.
  - **File server**: The computer that stores the files so that other computers on the same network can access them.
  - **Application server vs web server**:
    - **Web server**: It is a common subset of an application server. It delivers static web content. **Example:** HTML pages, files, images, video in response to HTTP requests from a web browser.
    - **Application server**: It can deliver web content too, just like any web server. However, its primary job is to enable interaction between browsers and server-side application code to generate and deliver dynamic content.

        - In current practice, however, both the web server and application server do the same thing. Most web servers support plug-ins for scripting languages (e.g., PHP) that enable the web server to generate dynamic content, and an increasing number of application servers incorporate web server capabilities and use HTTP as their primary protocol for interfacing with web servers.
    - **Building an HTTP server using Node.js**: We have said earlier that a web server receives HTTP requests from a client, like your browser, and provides an HTTP response, like an HTML page or JSON from a remote server computer. We have also seen previously that there are front-end and back-end codes involved when a server returns a webpage for a request from a browser. Front-end code is concerned with how the content is presented, such as the color of a navigation bar and the text styling. Back-end code/logic handles requests from your browser by instructing the server to bring data from the database or by storing data to the database. Node.js allows developers to use JavaScript to write the back-end code, even though traditionally it was used in the browser to write the front-end code. In this section, we will learn how to build web servers using the HTTP module that is built-in in Node.js. As an example, we will build web servers that can return HTML web pages.

2.4. Building HTTP web server using NodeJS: serving a simple message using HTTP module

- **Steps involved in building an HTTP server using Node.js that listens on port 1234 of your local server**: Refer Node.js' documentation here for detailed information.
    - **Setting up the coding environment**:
        - **Creating folder/directory for your project**: mkdir http-server-project.
        - **Enter the folder to house your code**

- **Create a package.json to hold important metadata about your project**: npm init
    - Here is where you will create the file (**Example**: app.js) that will be loaded when your module is required by another application
- **Open your "app.js" and start coding the following**:

- **Start by importing the HTTP core module**:
    - **Syntax**: const http = require("http");
    - **Why import HTTP module from Node.js?** This is because the HTTP module contains the function to create the server, which we will see later
- **Create a new server object using the HTTP module's createServer() function**: Just calling this method creates a server on your computer by creating an HTTP Server object. What handles the request behind the scene is, the code that is written inside of the HTTP module. **Note**: The "HTTP Server object" is what makes your computer behave as an HTTP server. **Example**: To create a server using the HTTP module you just imported above, the syntax is as follows:
    - **Syntax**: http.createServer()
    - **createServer() accepts a request listener function as a parameter**: When the server we just created receives any HTTP request from browser, it passes them on to a request listener function, a function that receives requests from the user, and the response back to the user. Let us talk about what a request listener function is below
- **Write a request listener/handler function**: We need a listener that listens to HTTP requests sent to the server and return an HTTP response to the client. This is a function that is executed each time the server gets a request. The listener function is an object created by the HTTP module that is triggered by an HTTP request through a

specific port. That trigger initiates the process of handling the request

- **Request listener function accepts two parameters**: A request object and a response object.
    - **req (the request object)**: It captures all the data of the HTTP request that is sent by the browser
    - **res (the response object)**: This object contains all the HTTP response that needs to be sent back to the browser
    - **Example**: Let us make our server return the "My first server!" message when someone accesses it

```
function (req, res) {
 res.writeHead(200); // sets the HTTP status code for response
  res.end("My first server!"); // writes the HTTP response back to  the client who requested it
 };
```

    - **Note**: Even though requestListener() does not use the req object, it must still be the first argument of the function.
    - **res.writeHead():** property is an inbuilt property of the 'http' module to write the header of the response that the application will serve to the client. 200 is a status code to mean that all is OK, the resource has been fetched and is transmitted in the message body. The method has a 2nd optional argument where you can put a human-readable status message.
    - **res.end()**: This method both sends the content of the response to the client browser (meaning, it will display the message on browser's localhost:1234) and signals

to the server that the response (header and content) has been sent completely.

- **Pass HTTP requests to your request listener function**: It is best if you put the server you created using http.createServer( ), on a variable so that we use this variable as opposed to the entire server creation code anywhere.

```
var myServer = http.createServer(
function (req, res) {
res.writeHead(200);
res.end("My first server!");
});
```

- **Binding the server you created with the port number of the server using myServer.listen() method**: In our previous classes, we have said that web servers are remote computers that accept requests from browsers and provide responses back to the browser. To interact with any remote computer/server, a domain name is needed. This domain name will be translated to an IP address by a DNS server. An IP address is a number that identifies any machine/computer connected to the internet. Port is a number is a "door" to the IP address of the server computer. HTTP server object (HTTP module) has built-in methods: the listen() method, the close() method, and the setTimeout() method. Please find the explanation for each below:
    - **The listen() method:** It is a built-in method in HTTP module used to create a listener on the specified port or path. This is the method you use to tell the specific port the server needs to listen
        - **Syntax**: server.listen(port, hostname, backlog, callback);

        - All the above arguments are optional, but it is a good idea to explicitly state which port and host we want a web server to use.
        - The callback function logs a message to our console so we can know when the server begins listening to connections.
    - **The close() method**: It is an inbuilt under the Server class within the HTTP module which is used to stop the server from accepting new connections. In short, it closes the server connection, and the server will stop listening to requests.
        - **Syntax**: myServer.close([callback])
    - **The setTimeout() method**: It is an inbuilt under the Server class within the HTTP module which is used to set the time out value for the server.
        - **Syntax**: myServer.setTimeout(msecs ,callback)
    - **Now, bind your server (called myServer) with "1234" port number**:

```
myServer.listen(1234, function () {
 console.log("Server is listening to HTTP requests at port 1234");
});
```

    - Congratulations 😀 you just created an HTTP server that serves a simple text on your browser. Now, see the "My first server!" Message on your browser following the next instruction.
- **Reach your server by visiting http://localhost: 2020 in a localserver browser**:
    - Yay 😀!!! You can now see the "My first server!" On your localserver browser

- **Why do we use a local host?** The value localhost is a special private address that computers use to refer to themselves. Localhost is equivalent to the IP address 127.0.0.1 and is only available to the local computer.

- **Here is what your entire code looks like when organized**:

```
const http = require("http");
const myServer = http.createServer(function (req, res) {
res.writeHead(200);
res.end("My first server!")
});
myServer.listen(1234, function () {
 console.log(" Server is listening to HTTP requests at port 1234");
});
```

- Please watch our class demo on how to serve a simple text using HTTP module for further explanation.

2.5. Building HTTP web server using HTTP module: serving static files with HTTP

- **First, you create a server**: By now, you know how to create an HTTP server by following the steps under section 4.4 above.
- **Importing fs module**: Once you have created your server, start thinking about what you want the web server needs to do to serve a static file to the browser. For a file to be displayed on the browser, your web server needs to read the file to serve the file
  - **Importing fs module**: To read a file, you need to first import one of Node's code module, the fs module
  - **readFile method () from fs module**: You will then need to use the readFile method found under fs module

- **Importing the URL module**: You will need to identify what URL path was used on the browser to request the static file.
  - **Importing the URL module:** We use this module when we want to split up/parse a web/URL address into readable parts. If you console.log(url), an object will be returned with various methods as properties of the object, parse() being one of them. If you parse a URL string, a URL object that contains properties for each of these components is returned
    - **url.parse() method**: We use this method on the url module we imported in our project. Doing so will return the requested URL in an object form with various properties/information about the URL. Using the parsed URL, we can just take out the pathname or port number part of the URL
- **Parsing the URL address/ url.parse() method**: It takes a URL string, parses it, and returns a URL object with each part of the address as a property. If you log the parsed URL address, you will see a URL object returned with different information/properties like pathname, href, hostname of the requested URL
  - **Syntax**: url.parse(requestedURL, parseQueryString, slashesDenoteHost)
    - **RequestedURL** = The requesed URL string to parse
    - **parseQueryString** = It is a Booloan value, whereby setting it to true, the query on the returned URL object will be parsed because it will be returned by the module's parse() method. If set to false, the query property on the returned URL object will be an unparsed string. Because we want the parsed value of the URL, we have to set it true. **Example**:
      - var parsedURLAdd = url.parse(req.url, true);
  - **path/pathname properties**: We said above that when we use the parse() method on the URL an object with all parts of the requested

URL address will be returned. However, if we do not need all parts of the URL address other than the initial "/", we can use the pathname property of the object that was returned after parsing the originally requested URL string. pathname property of the URL gets and sets the path portion of the above parsed URL. **Note**: You can also use the "path" property of the URL object. Example: If the requested URL was https://facebook.com, URL. Pathname would give us **Syntax**: parsedURLAddress.pathname.

- **Providing the absolute path of the file to serve**:
  - **Using _ _dirname**: _ _dirname will get us the folder name of the current working directory. **For instance**, if you want to serve an index.html file found in the found in your NodeProject folder, _ _dirname will return the name of the folder your app.js (or the file Node command is executed).
  - **_ _dirname + /static/: Once** we have the folder that contains the file we want to serve (index.html), we need to locate the folder that holds the index.html file. **Example**: if the index.html is found in a folder called "static", then we will concatenate the _ _dirname we found earlier to the static folder.
  - **_ _dirname + /static/+fileName**: The final step to providing the absolute path of the file to serve is providing the file name by concatenating _ _dirname + /static/ to the file name.
    - **Example**: _ _dirname + /static/+index.html:
- **Writing the logic to check the URL and provide the file based on the URL**: Now that we have the parsed URL name from the URL address entered on the browser, we then write a logic that checks this path, so that the appropriate HTML document related to that path address can be sent by the web server as a response.
- **Installing/importing mime-types module and adding content-type header field:** When serving files via HTTP, it is not enough to just send the contents of a file. You also should include the type of file being sent

by setting the Content-Type HTTP header with the proper MIME type for the file

- **Install mime-types from npm**: npm install mime-types
    - **mime-types (Multi-purpose Internet Mail Extensions)**: mime-types form a standard type of labeling file types that internet applications like web servers and browsers follow. In our case, we specify mime-types in our web server application so that the browser can know how to handle the index.html file
    - **mime-types.lookup**: If you console.log(mime-types), you will see the list of endless file type labeling. However, we want to specify the particular file type name from the list of mime-types. To look up the particular file type from the list, we use the lookup property. Once we look up that specific file type name, we write this file type name in the response header to notify the client browser that this is the kind of file type we are sending as a response to its request
- **Content-Type header field**: Content-Type field is mostly found in the headers of HTTP messages to describe the original file/media type of the resource/data a web server is responding. **Note**: When given an extension, mimetypes.lookup is used to get the matching content-type, otherwise, whatever we provide under our content-type will be used
    - **Example**: If we provide "text/html" in our web server, the browser will say “yes, this is an HTML document and I can render it by myself”, but if the web server provides "image/jpg", the browser can go "I will need a jpg Reader plugin installed on the user’s computer that will handle .pdf documents"

- **Now, let us code what we have explained above and serve our static apple website**

```
var http = require("http");
var fs = require("fs");
var url = require("url");
var mimeTypes = require("mime-types");
var mimeTypeLookUp = mimeTypes.lookup;
var myServer = http.createServer(function (req, res) {
var parsedUrl = url.parse(req.url, true);
var parsedURLsPath = parsedUrl.pathname;
if (parsedURLsPath == "/") {
parsedURLsPath = "/about.html";
}
var setMimeOfFileToServe =
mimeTypeLookUp(parsedURLsPath);
var locationOfFileToServe = __dirname + "/static/" +
parsedURLsPath;
fs.readFile(locationOfFileToServe, function (err, data) {
res.writeHead(200, { "content-type": setMimeOfFileToServe });
res.end(data);
});
});
myServer.listen(1234, function () {
console.log("Server listening at port 1234");
});
```

- Watch the class demo and try serving the “about.html” static file

2.6. Building HTTP web server using HTTP module: serving our static Apple website with HTTP

- Watch the class demo and try serving your “apple.com” replica website using HTTP module.

2.7. Building HTTP web server using Express module: serving our static Apple website with Express

- **Express framework**: It is an open-source server-side framework, written in JavaScript, that is used to build HTTP servers and set up connections. Using this connection, data sending and receiving can be done as long as connections use a hypertext transfer protocol. Express is built on top of Node. js' built-in HTTP module and is available through the npm registry. **Note**: Express a framework, not just a module. Let us explain these terms before getting deep into the subject: **Framework vs library vs package vs module**:
  - **Module**: Module is the smallest piece of software. A module is a set of methods or functions ready to be used somewhere else. Unlike the library, a module provides a single piece of functionality. This means that if you have a system with both modules and libraries, a library will typically contain multiple modules. **Note**: all modules are packages, but not all packages are meant to be used as modules
    - **Example**: HTTP or path core modules.
  - **Package**: It is a collection of modules of the same functional purpose. Collecting modules of the same functional purpose will make it easier to include all the related modules at once. **Example**: Any package from NPM. In Node’s case, for instance, a package is a file or directory that is described by a package.json file. A Node package must contain a package.json file in order to be published to the npm registry.

- **Library**: Both frameworks and libraries are code written by someone else that is used to help solve common problems. A library is a collection of packages. Remember, we said above that packages are a collection of modules of the same functional purposes. So, a library is what you include when you want to add some functionality to your code.
    - Unlike the framework below, the library does not force any coding style on the developer. When you use a library, you are in charge of the flow of the application. You are choosing when and where to call the library. **Example**: Bootstrap
- **Framework**: Both frameworks and libraries are code written by someone else that is used to help solve common problems. Framework is a set of libraries. But framework does not just offer functionalities, but it also provides an architecture for the development work. In other words, you do not include a framework, but you integrate your code into it.
    - In short words, framework forces its coding style on to the developer/user or it is in charge of the flow. It provides some places for you to plug in your code, but it calls the code you plugged in as needed. **Example**: Express package

- **Why Express package is better than the HTTP module to create web servers**: Both HTTP and Express are used to build HTTP modules. We can basically say that Express took the functionality of the HTTP module and added additional functionalities for faster and efficient web development
    - HTTP module is an in-build module that is pre-installed along with NodeJS. Express is installed explicitly using the npm command: npm install express.
    - HTTP is not a framework as a whole rather, it is just a module. Express is a framework as a whole

- HTTP does not provide any support for static page hosting. Express provide express.static function for static page hosting
- HTTP module provides various tools (functions) to do things for networking like making a server, client, etc. Express along with what HTTP does provide many more functions in order to make development easy. These functionalities include:
    - Setting individual routes for each asset we use (images, css files, js files, etc.)
    - There is the issue of CORS if we are requesting resources from a different server
    - Providing middleware, packages, plugins, and template code to add additional functionalities
    - Including automatic header information
    - Allowing third-party integrations of libraries and features to make customization easier

- **Express official website**: https://expressjs.com
- **Steps involved in building an Express server using Node.js that listens on port 5678 of your local server**:
    - **Setting up the coding environment**: Refer to section 4.7 above
    - **Import Express:** install and import Express
        - **Install Express**: npm i express –save
        - **Syntax to import express**: const express= require("express");
    - **Create a new express server application**: Calling the express() function creates a new express server application for you. If you console.log(express), you can see the createApplication function. This function is exported automatically from express.js file as a default export
        - **Syntax**: const serverApp= express();
        - If you console the server app that you created above using console.log(serverApp), there is an object created with “get”, “post”, “listen”, “request”, “response”, and other properties

- **Write a request listener function**: This is a function that listens to HTTP requests sent to our Express server and return an HTTP response to the client. A listener function is a callback function that executes each time the Express server gets a request.

```
Syntax:
function (req, res) {
  res.send(your code goes here);
}
```

  - **The "res" object to handle HTTP requests using Express**: The methods on the response object (res) can send HTTP response to the client/browser and terminate the request-response cycle. If none of these methods are called from a route handler, the client request will be left hanging. **Here are the most common response methods**: res.send(), res.end(), res.json(), res.download(), res.redirect()
    - **res.send()**: It will send a response of various types to client
    - **res.end()**: It ends the response process
  - **The "req" object to handle HTTP requests using Express**: The req object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on. In this documentation and by convention, the object is always referred to as req (and the HTTP response is res) but its actual name is determined by the parameters to the callback function in which you're working. All the request information is included inside the req object that is created by Express: **Here are the most important request methods**: req.url, req.ip, req.param, req.query, req.path, req.method, req.hostname. We use the above properties to correctly write the function that handles

the request. We will use these more when we make API calls in later classes. Visit this website for more on Express' request

- https://expressjs.com/en/5x/api.html#req

- **Create a new route**: Routing refers to determining how your Express server responds to a browser request to a particular path and to a specific HTTP request method (GET, POST, and so on). A route method is derived from one of the HTTP methods and is attached to an instance of the express class. Meaning that if you console the server app that you created above/ console.log(serverApp), you can see the Express server as an object with "get", "post", "listen", "request", "response", and other properties attached to the instance of the express class
  - **Syntax**: serverApp.HTTPMethod(path, listenerFunction)
    - **ServerApp**: It is the Express server you created above
    - **HTTPMethod:** It is an HTTP request method in lowercase. You define routing using methods of the Express app object that correspond to HTTP methods; for example, app.get() to handle GET requests and app.post to handle POST requests
    - **Path**: Route paths, in combination with request method(s), define the path at which requests can be made
    - **listenerFunction**: It is the function that gets executed when a server gets a request to the specified route/path
    - **Note**: Each route can have one or more listener functions that get executed when a server gets a request, and the route is matched

- **Pass HTTP requests to your request listener function by defining the request route and request method**:
    - **Example**: Using the express server you created above, respond with "Request processed with Express" when a GET request is made to the homepage or this path "/"

```
serverApp.get("/", function (req, res) {
 res.send("<h1>Request processed with Express</h1>");
});
```

- **Binding the Express server you created above, with the port number of the server using listen() function**: The listen() function, which is returned from the express server you created above, is used to bind and listen to the connections on the specified host and port. This method is identical to Node's http.Server.listen() method. **Note**: If the port number is omitted or is 0, the operating system will assign an arbitrary unused port, which is useful for cases like automated tasks (tests, etc.).
    - **Syntax**: serverApp.listen(port, host, backlog, callback)
        - **Port(optional)**: It specifies the port on which we want our app to listen.
        - **host (optional)**: It specifies the IP Address of the host on which we want our app to listen. **Note**: You can specify the host if and only if you have already specified the port
        - **Backlog (optional)**: It specifies the max length of the queue of pending connections. **Note**: You can specify the backlog if and only if you have already specified the port and host.
        - **Callback (optional)**: It specifies a function that will get executed once your app starts listening to the

specified port. **Note**: You can specify callback alone i.e., without specifying port, host, and backlogs

- **Example**: Make the server you created above listen to requests at port number 5678

```
serverApp.listen(5678, function (err) {
  if (err) {
   console.log(err);
   }
 console.log("My express server listening to requests at port 5678 ");
 });
```

- **Reach your Express server by visiting http://localhost: 5678in a local server browser**:
  - Congratulations😀 you just served a text message using an Express server.
- **Here is what your entire code looks like when put together**:

```
const express = require("express");
const serverApp = express();
serverApp.get("/", function (req, res) {
res.send("<h1>Request processed with Express</h1>");
});
serverApp.listen(5678, function (err) {
if (err) {
console.log(err);
}
console.log("My express server listening at port 5678 ");
});
```

- **Serving static files using Express**: below, let us try to serve our apple.com static website that is found in our project's "public" root folder.

- **Follow all the steps under section 4.7 to setup your coding environment**: Please refer to section 4.7 to setup your coding environment and create an express server, request handler function, and bind the server to the port it will listen at.
- **Use middleware function to serve static files using Express**: Middleware is how Express handles a sequence of actions in between the time it receives the request and the time it sends back the response. Visit Express's website for more on Middleware functions: https://expressjs.com/en/guide/using-middleware.html
    - **express.static middleware function**: This is a built-in middleware function in Express. To serve static files such as images, CSS files, and JavaScript files, use the express.static built-in middleware function in Express
        - **Syntax**: express.static(root, options)
        - **root**: It specifies the root directory from which to serve static assets, not a filename. **Note**: By default, this module will send ONLY "index.html" files in response to a request on the root directory. So, make sure to have an index.html file that the middleware will look for
        - **options**: The function determines the file to serve by combining req.url with the provided root directory
- **Use the serverApp.use() function**: The app.use() function is mostly used to setup middleware for your server application. In our specific case, we will use the app.use() function to mount the express.static middleware function at the path that is being specified
    - **Syntax**: serverApp.use(path, callback)
        - **path**: It is the path for which the middleware function is being called

        - **callback**: It is a middleware function or a series/array of middleware functions (in our case the express.static middleware function)
    - Here is how you can create an Express server that serves your static Apple website

```
const express = require("express");
const serverApp = express();
serverApp.use("/", express.static("public"));
serverApp.listen(5678, function (err) {
if (err) {
console.log(err);
}
console.log("Express listening at port 5678 ");
});
```

- Watch the class demo and try serving your “apple.com” replica website using the Express module.