- ***Basic Node and Express***
    - -> Node.js is a JavaScript runtime
    - -> for writing backend (server-side) programs in js
    - -> Node.js
        - -> this contains modules
        - -> independent programs
        - -> these modules include
            - <u>-> HTTP <- this acts like a server and file system</u>
            - <u>-> a module to read / modify files</u>
        - <u>-> it's like a js module which contains different functions and packages you can use</u>
            - <u>-> for managing files on the backend side of projects et al</u>
- ***The last section of the course***
    - -> this was on installing / managing packages from npm
    - -> collections of smaller modules
    - -> packages can be combined to build larger / more complex applications
- ***Express***
    - <u>-> this is a lightweight js web applications framework</u>
    - <u>-> this is an npm package</u>
    - -> this is used to create a server and handle routing for the application
    - -> for directing people to the correct page
- ***This section of the course***
    - -> this is on Node and Express
    - -> how to create a server
    - -> how to serve different files
    - -> how to handle different requests from the browser
- ***The Node console***
    - -> clone the GitHub repo
    - -> you can run the code in Gitpod
    - -> you can also use a site builder
    - ***-> Node***
        - <u>-> this is a js environment</u>
        - <u>-> you can use the console in an IDE to display debug information</u>
        - -> Gitpod is an IDE (VSCode) in Chrome
        - -> the terminal can be used to show you the errors in the code
    - ***-> task***
        - -> the task is to motify the myApp.js file to log "Hello World"
- ***Start working a Express server***
    - -> the first two lines of the myApp.js file create an Express app object
    - ***-> there are several methods in this object***
        - ***-> app.listen(port) <- this is one common method***
            - -> the server listens to a port
            - -> this puts the port in its running state
            - -> the server.js file contains this method -> to ensure that the app is running in the background
        - <u>***-> app.METHOD(PATH, HANDLER) <- this is the syntax for a route***</u>
            - ***-> METHOD***
                - -> this is an http method in lowercase
            - ***-> PATH***
                - -> this is a relative path on the server
                - -> this can be a string or a regular expression
            - ***-> HANDLER***
                - -> this is a function which Express calls when the route is matched

            - -> the syntax for a HANDLER is function(req, res) {...}
                - -> req <- the request object
                - -> res <- response object
            - -> Example handler
                - function(req, res) {
                -   res.send('Response String');
                - }
    - ***-> task***
        - -> using the app.get() method
        - -> to serve the string "Hello Express"
        - -> to GET requests matching the / (root) path
        - -> looking at the logs to see if the code works
        - app.get('/', (req, res) => {
        -     res.send("Hello Express");
        -   });
        - -> .get is the METHOD <- this is an http method
        - -> '/' is the PATH
        - -> the HANDLER is (req, res) => {
        -     res.send("Hello Express");
        -   }
            - -> this is the function that Express calls when the route is matched
            - -> the arguments this takes are the request and response objects
            - -> this is the same handler as the example -> just changed to include "Hello Express" rather than "Response string"
        - -> checking that this works by using the console
- ***Serve an HTML file***
    - ***Serving an HTML file using the res.sendFile(path) method***
        - <u>-> responding to requests using the res.sendFile(path) method</u>
        - -> this is put inside the app.get('/',...) route handler
        - -> this method sets the headers to tell the browser how to handle the file you want to send
            - -> when a file is sent
            - -> there is request from the user on the other side
            - -> and then the server sends it across
            - -> the users don't see the servers on the backend
        - -> this can be done according to the type of the file
        - -> it then reads and send this file
        - -> this method needs an absolute file name path
            - -> this is done with the __dirname Node global variable
            - -> the path should look like this: absolutePath = __dirname + '/relativePath/file.ext'
        - -> express evaluates routes from top to bottom
    - ***Task***
        - -> send the /views/index.html file
        - -> we are doing this as a response to GET requests to the / path
            - -> the user makes get requests to this path
            - -> in response we are sending this index.html file across
            - -> we are setting it to send this file to the user
        - -> it needs an absolute path
- ***Serve static assets***
    - ***-> HTML servers have multiple directories which the user can access***
        - -> these can store static assets
        - -> static assets being stylesheets, scripts, images

    - ***-> this can be used in express with the js -> <u>express.static(path)</u>***
        - -> <u>the static assets are the things which stay 'static' / the same, independent of the user interactions -> they are the assets for the page which are stored on the HTML server</u>
        - -> the path is the path to the directory with the assets
        - -> absolute file path
    - ***-> middleware***
        - -> these are functions which intercept route handlers
        - -> these add information to them
        - -> these path need to be mounted -> app.use(path, middlewareFunction)
        - -> the first argument is optional -> if you don't give it a path, then the middleware will be executed for all requests
    - ***-> Task***
        - -> mount the express.static() middleware
        - -> we are using the /public path, and app.use()
        - -> the absolute path we are using is __dirname + /public
        - -> once this is done, the app can serve a css stylesheet
        - -> the /public/style.css file is referenced in the /views/index.html in the project boilerplate
- ***Serve JSON on a Specific Route***
    - ***-> APIs & json files***
        - -> HTML servers serve HTML
        - <u>-> APIs serve data <- Application Programming Interface</u>
        - -> REST APIs <- REpresentational State Transfer
            - -> this allows the data to be served without the clients having to know information about the server
            - -> the client only needs to know where the URL <- where the resource is
                - -> and the action is wants to perform on what resource
                - <u>-> GET is used when you are fetching information and not modifying anything in the file you are calling from</u>
                - <u>-> json files represent js objects as strings -> for transmitting this information</u>
    - ***-> When you create an API***
        - -> creating an API
        - -> creating a route which responds with json
        - -> at the path /json
        - -> you can use the app.get() method and use the res.json() method inside the route handler
        - -> using the res.json() method and passing an object inside the argument
            - -> this closes the request response loop
        - -> it's converting a js object into a string, then setting headers on the browser to tell it we are sending json data back
            - -> {key: data} <- object syntax
            - -> data <- this can be a number, string, nested object or an array
            - -> this can also be the result of a function call or a variable
    - ***-> Task***
        - -> serve the object {"message": "Hello json"}
        - -> this is as a response to GET requests to the /json route
        - -> in JSON format
        - -> then pointing the browser to your-app-url/json
        - -> you should see the message on the screen
        - -> we are serving the message "Hello json"
- ***Use the .env File***
    - <u>-> the .env file <- this is a hidden file, to pass environment variables to the application</u>
    - <u>-> this file is secret <- you can use it to keep data you want hidden (e.g API keys)</u>

    - -> you can also use this to store configuration options
        - -> this can be used to change the behaviour of the application without having to rewrite code
    - ***-> process.env.VAR_NAME***
        - -> this is how you access the environment variables
        - -> process.env <- this is a global Node object
        - -> variables are passed as strings
        - -> their names are all uppercase and their words are separated by an underscore
        - -> .env is a shell file
            - -> you don't need to wrap names or values in quotes
            - -> there cannot be space around the equals sign when you name a variable
            - -> it has to be LIKE_THIS=value
    - ***-> Task***
        - -> adding an environment variable as a configuration option
        - -> create a .env file in the root of the project directory
        - -> store the variable MESSAGE_STYLE=uppercase in it
        - -> in the json route handler from the previous question
            - -> the response object should be uppercase
            - -> depending on the message style value
            - -> you have to read the env file inside the route handler due to the way their tests run
            - -> depending on the message style value, the string it returns should be upper or lowercase
        - -> locally
            - -> you will need the dotenv package
            - -> this has already been installed and is in the project package.json file
- ***Implement a Root-Level Request Logger Middleware***
    - -> the express.static() middleware function <- earlier
    - ***-> middleware functions***
        - ***-> arguments they take***
            - -> the request object
            - -> the response object
            - -> the next function in the application's request-response cycle
        - -> these functions add information to the request or response objects
        - ***-> they can end the cycle***
            - -> you send a response when some condition is met
            - -> when they are finished, they start the execution of the next function in the stack
        - ***-> example***
            - function(req, res, next) {
            -   console.log("I'm a middleware...");
            -   next();
            - }
            - -> this triggers calling the third argument
            - -> if this function was mounted on a route, when a request matches the route, it returns the string
            - -> this is done via the execution of some function in the stack
        - ***-> the app.use(<mware-function>) method***
            - -> this is building root-level middleware
            - -> this is done using the app.use(<mware-function>) method
            - -> the function is executed for all the requests
            - -> more specific consitions can be set
                - -> app.post(<mware-function>) <- this only executes the function for POST

requests
            - -> there are other methods for HTTP requests -> GET, DELETE, PUT
        - ***-> order of operations***
            - -> Express evaluates functions in the order they appear in the code <- as does CSS (with styles)
                - -> this is also true for middleware
                - -> if you want the function to work for all the routes, it should be mounted before them
    - ***-> Task***
        - -> building a logger
        - -> for each request, we log a string in a certain format -> method path - ip
            - -> example -> GET /json - ::ffff:127.0.0.1
            - -> there is a space between method and path
            - -> the method is GET and the path is /json
            - -> the server is getting the file stored at that directory
        - -> you can use different methods to get data
            - -> req.method
            - -> req.path
            - -> req.ip
        - -> call next() when you are done <- this stops the server getting stuck
        - -> the logs should be opened
        - -> you test the code once it's been written to see if it works
- ***Chain Middleware to Create a Time Server***
- ***-> app.METHOD(path, middlewareFunction)***
    - -> this is used to mount middleware on a specific route
    - -> this can also be chained within a route definition
    - ***-> example***
        - app.get('/user', function(req, res, next) {
        -   req.user = getTheUserSync(); // Hypothetical synchronous operation
        -   next();
        - }, function(req, res) {
        -   res.send(req.user);
        - });
        - -> this can be used to split the server operations into smaller units
        - -> this improves the app structure and the code reusability
        - -> this can also be used to perform data validation
        - -> at each point of the middleware stack you can block the execution of the current chain
            - -> you can also pass control to functions which handle errors
            - -> you can also pass this onto the next matching route, or to handle special cases
            - -> this is done in the advanced Express section
    - ***-> Task***
        - -> in the route app.get('/now', ...), chain a middleware function and the final handler
            - -> we are chaining a middleware function
        - -> the middleware function should have a current time to request the object in the req.time key
            - -> we can use new Date().toString() for this
        - -> the handler should respond with a JSON object
            - -> the structure of this should be {time: req.time}
        - -> you need to chain the middleware or the test won't pass
            - -> you need to mount the function inside it for their unit tests to pass this
- ***Get Route Parameter Input from the Client***
    - ***Getting a route parameter input from the client***

            - -> when building an API <- a server which returns data
            - -> we want users to input what they want from our service
            - -> if the client is requesting information in our databse
            - -> we can use reout parameters for this
            - -> these are named segments of the URL -> delimited by dashes /
            - -> each segment captures the value of the URL which matches its position
            - -> captured values can be found in the req.params object
                - route_path: '/user/:userId/book/:bookId'
                - actual_request_URL: '/user/546/book/6754'
                - req.params: {userId: '546', bookId: '6754'}
    - ***Task***
        - -> build an echo server
        - -> this should be mounted at the route GET /:word/echo
        - -> respond with a json object
        - -> this should take the structure {echo: word}
        - -> the word to be repeated is found at req.params.word
        - -> you can test the route from the browser's address bar
            - -> you do this by visiting matching routes -> e.g your-app-rootpath/freecodecamp/echo
- ***Get Query Parameter Input from the Client***
    - ***Getting the query parameter input from the client***
        - -> another way to get input from the client
        - -> encoding the data after the route path, using a query string
            - -> ? and an includes field=value couple
            - -> each couple is separated by an &
            - -> Express parses the data from the query string and populates the object req.query
            - <u>-> % cannot be in a URL and has to be encoded in a different format before it can be sent</u>
                - -> if you are using the API from js, you can use specific methods to encode and decode these characters
                - -> example
                    - route_path: '/library'
                    - actual_request_URL: '/library?userId=546&bookId=6754'
                    - req.query: {userId: '546', bookId: '6754'}
                    - -> the top line is the route path on the server
                    - -> then we have the url which is being requested encoded in a specific form
                    - -> and finally the query
    - ***Task***
        - -> build an API endpoint
        - -> mounted at Get /name
        - -> respond with a JSON document
        - -> this should take the structure { name: 'firstname lastname'}
        - -> the first and last name characters should be encoded in a query string
            - -> this should be in the syntax ?first=firstname&last=lastname
        - -> note
            - -> in this exercise we are receiving data from a POST request
            - -> this is from the same /name route path
            - -> you can use the method app.route(path).get(handler).post(handler) for this
            - -> this allows you to chain different verb handlers on the same path route
- ***Use body-parser to Parse POST Requests***
    - -> GET is a HTTP verb

    - -> another HTTP verb is POST
    - -> this is the default method to send client data with HTML forma
    - ***-> this is used to send data to create new items in the database***
        - -> e.g a new user / blog post
        - -> instances of a class, for example
        - -> handling post requests
        - <u>-> making a new post on a website, and that sending requests to the server</u>
        - <u>-> how servers handle those requests</u>
    - ***-> post requests***
        - -> the data doesn't appear in the URL, it's hidden in the request body
        - -> the body is a part of the HTTP request -> this is also called the payload
        - -> data which is not visible in the URL is still not private
        - -> this is an example HTTP POST request
            - POST /path/subpath HTTP/1.0
            - From: john@example.com
            - User-Agent: someBrowser/1.0
            - Content-Type: application/x-www-form-urlencoded
            - Content-Length: 20
            -
            - name=John+Doe&age=25
            - -> the body is encoded like a query string
            - -> this is the deault for HTML forms
            - <u>-> Ajax can also be used to handle data having a more complex structure</u>
        - ***-> multipart/form-data***
            - -> this is another way of encoding data
            - -> this is used to upload binary files
            - <u>-> to parse the data from POST requests, you use the body-parser package</u>
                - -> this allows you to use a series of middleware
                - -> you can also decode data into different formats using this
    - ***-> Task***
        - -> body-parser is already installed in the project's package.json file
        - -> require it at the top of the myApp.js file
        - -> store it in a variable named bodyParser
        - -> bodyParser.urlencoded({extended: false}) <- this is the middleware to handle URL encoded data
        - -> pass the function returned by the previous method call to app.use()
        - -> the middleware function(s) must be mounted before all the routes that depend on them
        - -> extended is a configuration option that tell body-parser which parsing needs to be used
        - ***-> when extended=false, then it uses the encoding querystring library***
            - -> in this case, values can only be strings or arrays
            - -> otherwise, it uses the qs library for parsing
        - ***-> the object this returns does not typically inherit from the default js `Object`***
            - -> this means other functions are not available for use on this (for example toString)
            - -> the extended version allows more data flexibility, but is outmatched by JSON
- ***Get Data from POST Requests***
    - -> mount a POST handler
    - -> this should be at the path /name
    - -> we have prepared a form in the html fontpage
    - -> submitting the same data from exercise 10 <- a query string
    - -> if the body-parser is configured correctly, you should find the parameters in the object

req.body
    - -> this is the usual libary example
        - route: POST '/library'
        - urlencoded_body: userId=546&bookId=6754
        - req.body: {userId: '546', bookId: '6754'}
    - -> responding with the same JSON object as before -> {name: 'firstname lastname'}
    - -> then testing if the endpoint works using the html form we provided in the app frontpage
- -> there are several other http methods other than GET and POST
    - -> depending on the operation you want to execute on the server
    - -> POST
        - -> this is sometimes also called PUT
        - -> to create a new resource using the information sent with the request
    - -> GET
        - -> to read an existing resource without modifying it
    - -> PUT, MATCH or POST
        - -> to update a resource using the data sent
    - -> DELETE
        - -> to delete a resource
    - -> there are other methods for connecting with the server
        - -> all of the methods above apart from GET can serve a payload
        - -> <u>the GET method is just used for data retrieval</u>
        - -> a payload is data put into the request body (or other)
        - -> the body-parser middleware also works with these methods