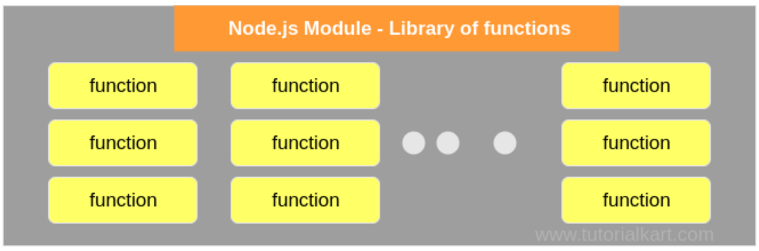
Node.js Modules

In the Node.js module system, each file is treated as a separate module. A Node.js Module is a library of functions that could be used in a Node.js file.



There are three types of modules in Node.js based on their location to access. They are:

1. **Built-in Modules/Core Modules:** These are the modules that come along with Node.js installation.
2. **User defined Modules:** These are the modules written by user or a third party.
3. **Third party Modules:** There are many modules available online which could be used in Node.js. Node Package Manager (NPM) helps to install those modules, extend them if necessary and publish them to repositories like Github for access to distributed machines.

- Install a Node.js module using NPM
- Extend a Node.js module
- Publish a Node.js module to Github using NPM

____

1. Built-in Modules/Core Modules:

The core modules include bare minimum functionalities of Node.js. These core modules are compiled into its binary distribution and load automatically when Node.js process starts. However, you need to import the core module first in order to use it in your application.

The following table lists some of the important core modules in Node.js.

Core Module	Description
http	http module includes classes, methods and events to create Node.js http server.
url	url module includes methods for URL resolution and parsing.
querystring	querystring module includes methods to deal with query string.
path	path module includes methods to deal with file paths.
fs	fs module includes classes, methods, and events to work with file I/O.
util	util module includes utility functions useful for programmers.

Loading Core Modules

In order to use Node.js core or NPM modules, you first need to import it ***using require() function*** as shown below.

```
var module = require('module_name');
```

As per above syntax, specify the module name in the require() function. The require() function will return an object, function, property or any other JavaScript type, depending on what the specified module returns.

The ***<u>following example demonstrates how to use Node.js http module</u>*** to create a web server.

```
var http = require('http');

var server = http.createServer(function(req, res){

  console.log('http demo')
  res.end('hello world')

});

server.listen(5000);
```

In the above example, require() function returns an object because http module returns its functionality as an object, you can then use its properties and methods using dot notation e.g. http.createServer().

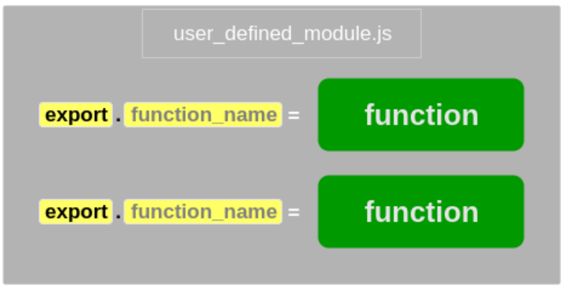
____

_

2. User defined Modules

- How to Create a Node.js Module

Most of the necessary functions are included in the Built-in Modules. Sometimes it is required that, when you are implementing a Node.js application for a use case, you might want to keep your business logic separately. In such cases you create a Node.js module with all the required functions in it.


user_defined_module.js
export . function_name = function
export . function_name = function

Create a Node.js Module

A Node.js Module is a .js file with one or more functions.
Following is the syntax to define a function in Node.js module :

```
exports.<function_name> = function (argument_1, argument_2, .. argument_N) {  /** function body */ };
```

exports – It is a keyword which tells Node.js that the function is available outside the module.

Example:

circle.js	foo.js

const {PI}=Math
exports.f1= function (r)
{
   return PI * r ** 2;
}
 	
exports.datetime= function()
{
   return Date();
}
 	
var x1=function()
{
   return 4 * 5
}
exports.s=x1;
	var app = require('./circle.js');
	
	console.log(`circle area of 4 is ${app.f1(4)}`);
	
	console.log("date is " +app.datetime());
	
	console.log('Multiplication '+app.s())
	
	console.log('arrow function ' +app.multiplyes6(5,3));

exports.multiplyes6 = (a,b) =>
{
    console.log('from arrow test');
    return a * b
};
console.log('from circle '+x1());	

Exports an object:

Ex:

log.js

```
var log = {
    info: function (info) {
        console.log('Info: ' + info);
    },
    warning:function (warning) {
        console.log('Warning: ' + warning);
    },
    error:function (error) {
        console.log('Error: ' + error);
    }
};

var str="hello world";

exports.str=str;

exports.sr = log
```

app.js

```
var myLogModule = require('./log');

myLogModule.sr.info('Node.js started');

console.log(myLogModule.str);
```

Output:

```
H:\node\myexamples>node app.js

Info: Node.js started

hello world
```

Here, *var* log is an object. Info, warning and error are function. The *module.exports* is a special object which is included in every JS file in the Node.js application by default. Use **module.exports** or **exports** to expose a function, object or variable as a module in Node.js.

---

3. **Third party Modules:**

- The 3rd party modules can be downloaded using NPM (Node Package Manager).
- 3rd party modules can be install inside the project folder or globally.

Load and Use Third Party Module with Example:

3rd party modules can be downloaded using NPM in following way.

```
//Syntax

npm install -g module_name // Install Globally

npm install --save module_name //Install and save in package.json



//Install express module

npm install --save express

npm install --save mongoose



//Install multiple module at once

npm install --save express mongoos
```

HTTP Module

Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

Node.js as a Web Server

The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client.

Example:

```
var http = require('http');

//create a server object:
http.createServer(function (req, res) {
  res.write('Hello World!'); //write a response to the client
  res.end(); //end the response
}).listen(8080); //the server object listens on port 8080
```

The function passed into the http.createServer() method, will be executed when someone tries to access the computer on port 8080. We are using the server.listen function to make our server application listen to client requests on port no 8080. You can specify any available port over here.

Add an HTTP Header

If the response from the HTTP server is supposed to be displayed as HTML, you should include an HTTP header with the correct content type:

Example

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('Hello World!');
  res.end();
}).listen(8080);
```

The first argument of the res.writeHead() method is the status code, 200 means that all is OK, the second argument is an object containing the response headers.

Status-Code

The Status-Code element in a server response, is a 3-digit integer where the first digit of the Status-Code defines the class of response and the last two digits do not have any categorization role. There are 5 values for the first digit:

S.N.	Code and Description

1	1xx: Informational It means the request has been received and the process is continuing.
2	2xx: Success It means the action was successfully received, understood, and accepted.
3	3xx: Redirection It means further action must be taken in order to complete the request.
4	4xx: Client Error It means the request contains incorrect syntax or cannot be fulfilled.
5	5xx: Server Error It means the server failed to fulfill an apparently valid request.

HTTP status codes are extensible and HTTP applications are not required to understand the meaning of all the registered status codes. Given below is a list of all the status codes.

Read the Query String

The function passed into the http.createServer() has a req argument that represents the request from the client, as an object (http.IncomingMessage object).

This object has a property called "url" which holds the part of the url that comes after the domain name:

demo_http_url.js

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write(req.url);
  res.end();
}).listen(8080);
```

Save the code above in a file called "demo_http_url.js" and initiate the file:

Initiate demo_http_url.js:

```
C:\Users\BNJ>node demo_http_url.js
```

If you have followed the same steps on your computer, you should see two different results when opening these two addresses:

http://localhost:8080/UVPCE

Will produce this result:

```
/UVPCE
```

http://localhost:8080/Ganpatuniversity

Will produce this result:

```
/ Ganpatuniversity
```

Split the Query String

There are built-in modules to easily split the query string into readable parts, such as the URL module.

Example

Split the query string into readable parts:

```
var http = require('http');
var url = require('url');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  var q = url.parse(req.url, true).query;
  var txt = q.year + " " + q.month;
  res.end(txt);
}).listen(8080);
```

Save the code above in a file called "demo_querystring.js" and initiate the file:

```
C:\Users\Your Name>node demo_querystring.js
```

The address:

http://localhost:8080/?year=2020&month=July

Will produce this result:

```
2020 July
```

Note:

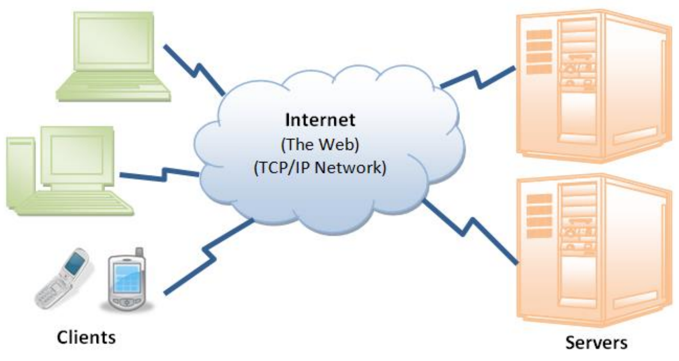
*The **url.parse()** method takes a URL string, parses it, and it will return a URL object with each part of the address as properties.*

Syntax:

```
url.parse( urlString, parseQueryString, slashesDenoteHost)
```

***Parameters:** This method accepts three parameters a smentioned above and described below:*

HTTP (HyperText Transfer Protocol)



As mentioned, whenever you enter a URL in the address box of the browser, the browser translates the URL into a request message according to the specified protocol; and sends the request message to the server.

For example, the browser translated the URL `http://www.nowhere123.com/doc/index.html` into the following request message:

```
GET /docs/index.html HTTP/1.1
Host: www.nowhere123.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
(blank line)
```

When this request message reaches the server, the server can take either one of these actions:

1. The server interprets the request received, maps the request into a *file* under the server's document directory, and returns the file requested to the client.
2. The server interprets the request received, maps the request into a *program* kept in the server, executes the program, and returns the output of the program to the client.
3. The request cannot be satisfied, the server returns an error message.

An example of the HTTP response message is as shown:

```
HTTP/1.1 200 OK
Date: Sun, 18 Oct 2009 08:56:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Sat, 20 Nov 2004 07:16:26 GMT
ETag: "10000000565a5-2c-3e94b66c2e680"
Accept-Ranges: bytes
Content-Length: 44
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug

<html><body><h1>It works!</h1></body></html>
```

The browser receives the response message, interprets the message and displays the contents of the message on the browser's window according to the media type of the response (as in the Content-Type response header). Common media type include "`text/plain`", "`text/html`", "`image/gif`", "`image/jpeg`", "`audio/mpeg`", "`video/mpeg`", "`application/msword`", and "`application/pdf`".

In its idling state, an HTTP server does nothing but listening to the IP address(es) and port(s) specified in the configuration for incoming request. When a request arrives, the server analyzes the message header, applies rules specified in the configuration, and takes the appropriate action. The webmaster's main control over the action of web server is via the configuration, which will be dealt with in greater details in the later sections.

____

File System: To handle file operations like creating, reading, deleting, etc., Node.js provides an inbuilt module called FS (File System). Node.js gives the functionality of file I/O by providing wrappers around the standard POSIX(IEEE STANDARD) functions. All file system operations can have synchronous and asynchronous forms depending upon user requirements.

The `process` object in Node.js is a global object that can be accessed inside any module without requiring it. There are very few global objects or properties provided in Node.js and `process` is one of them. It is an essential component in the Node.js ecosystem as it provides various information sets about the runtime of a program.

To explore we will use one of its properties which is called `process.versions`. This property tells us the information about Node.js version we have installed. It has to be used with `-p` flag.

```
$ node  -p "process.versions"

# output
{ http_parser: '2.8.0',
  node: '8.11.2',
  v8: '6.2.414.54',
  uv: '1.19.1',
  zlib: '1.2.11',
  ares: '1.10.1-DEV',
  modules: '57',
  nghttp2: '1.29.0',
  napi: '3',
  openssl: '1.0.2o',
  icu: '60.1',
  unicode: '10.0',
  cldr: '32.0',
  tz: '2017c' }
```

Another property you can check is `process.release` that is the same as the command `$ node --version` which we used when we installed Node.js. But the output this time is going to be more detailed.

```
node -p "process.release"

# output
{ name: 'node',
  lts: 'Carbon',
  sourceUrl: 'https://nodejs.org/download/release/v8.11.2/node-v8.11.2.tar.gz',
  headersUrl: 'https://nodejs.org/download/release/v8.11.2/node-v8.11.2-
headers.tar.gz' }
```

These are some of the different commands that we can use in a command line to access information that otherwise no module can provide.

This `process` object is an instance of the EventEmitter class. It does it contain its own pre-defined events such as `exit` which

can be used to know when a program in Node.js has completed its execution.
Run the below program and you can observe that the result comes up with status code `0`. In Node.js this status code means that a program has run successfully.

```
process.on('exit', code => {
        setTimeout(() => {
                console.log('Will not get displayed');
        }, 0);

        console.log('Exited with status code:', code);
});
console.log('Execution Completed');
```

Output of the above program:

```
Execution Completed
Exited with status code: 0
```

`Process` also provides various properties to interact with. Some of them can be used in a Node application to provide a gateway to communicate between the Node application and any command line interface. This is very useful if you are building a command line application or utility using Node.js

- process.stdin: a readable stream
- process.stdout: a writable stream
- process.stderr: a wriatable stream to recognize errors

Using `argv` you can always access arguments that are passed in a command line. `argv` is an array which has Node itself as the first element and the absolute path of the file as the second element. From the third element onwards it can have as many arguments as you want.
Try the below program to get more insight into how you can use these various properties and functions.

```
process.stdout.write('Hello World!' + '\n');

process.argv.forEach(function(val, index, array) {
        console.log(index + ':' + val);
});
```

If you run the above code with the following command you will get the output and the first two elements are of `argv` are printed.

```
$ node test.js


# output
Hello World!
0: /usr/local/bin/node
1: /Users/amanhimself/Desktop/articles/nodejs-text-tuts/test.js
```

Buffer

`Buffer` objects are used to represent a fixed-length sequence of bytes. Many Node.js APIs support `Buffer`s.

The `Buffer` class is a subclass of JavaScript's `Uint8Array` class and extends it with methods that cover additional use cases. Node.js APIs accept plain `Uint8Array`s wherever `Buffer`s are supported as well.

While the `Buffer` class is available within the global scope, it is still recommended to explicitly reference it via an import or require statement.

What Are Buffers?

The `Buffer` class in Node.js is designed to handle raw binary data. Each buffer corresponds to some raw memory allocated outside V8. Buffers act somewhat like arrays of integers, but aren't resizable and have a whole bunch of methods specifically for binary data. The integers in a buffer each represent a byte and so are limited to values from 0 to 255 inclusive. When using `console.log()` to print the `Buffer` instance, you'll get a chain of values in hexadecimal values.

Where You See Buffers:

In the wild, buffers are usually seen in the context of binary data coming from streams, such as `fs.createReadStream`.

Usage:

Creating Buffers:

There are a few ways to create new buffers:

```
const buffer = Buffer.alloc(8);
// This will print out 8 bx`ytes of zero:
// <Buffer 00 00 00 00 00 00 00 00>
```

This buffer is initialized and contains 8 bytes of zero.

```
const buffer = Buffer.from([8, 6, 7, 5, 3, 0, 9]);
// This will print out 8 bytes of certain values:
// <Buffer 08 06 07 05 03 00 09>
```

This initializes the buffer to the contents of this array. Keep in mind that the contents of the array are integers representing bytes.

```
const buffer = Buffer.from("I'm a string!", 'utf-8');
// This will print out a chain of values in utf-8:
// <Buffer 49 27 6d 20 61 20 73 74 72 69 6e 67 21>
```

This initializes the buffer to a binary encoding of the first string as specified by the second argument (in this case, `'utf-8'`). `'utf-8'` is by far the most common encoding used with Node.js, but `Buffer` also supports others. See Supported Encodings for more details.

Writing to Buffers

Given that there is already a buffer created:

```
> var buffer = Buffer.alloc(16)
```

we can start writing strings to it:

```
> buffer.write("Hello", "utf-8")
5
```

The first argument to `buffer.write` is the string to write to the buffer, and the second argument is the string encoding. It happens to default to utf-8 so this argument is extraneous.

`buffer.write` returned 5. This means that we wrote to five bytes of the buffer. The fact that the string "Hello" is also 5 characters long is coincidental, since each character *just happened* to be 8 bits apiece. This is useful if you want to complete the message:

```
> buffer.write(" world!", 5, "utf-8")
7
```

When `buffer.write` has 3 arguments, the second argument indicates an offset, or the index of the buffer to start writing at.

Reading from Buffers:

toString:

Probably the most common way to read buffers is to use the `toString` method, since many buffers contain text:

```
> buffer.toString('utf-8')
'Hello world!\u0000�k\t'
```

Again, the first argument is the encoding. In this case, it can be seen that not the entire buffer was used! Luckily, because we know how many bytes we've written to the buffer, we can simply add more arguments to "stringify" the slice that's actually interesting:

```
> buffer.toString("utf-8", 0, 12)
'Hello world!'
```

`setImmediate()`

When you want to execute some piece of code asynchronously, but as soon as possible, one option is to use the `setImmediate()` function provided by Node.js:

```
setImmediate(() => {

  // run something

});
```

Any function passed as the setImmediate() argument is a callback that's executed in the next iteration of the event loop.

How is `setImmediate()` different from `setTimeout(() => {}, 0)` (passing a 0ms timeout), and from `process.nextTick()` and `Promise.then()`?

A function passed to `process.nextTick()` is going to be executed on the current iteration of the event loop, after the current operation ends. This means it will always execute before `setTimeout` and `setImmediate`.

A `setTimeout()` callback with a 0ms delay is very similar to `setImmediate()`. The execution order will depend on various factors, but they will be both run in the next iteration of the event loop.

A `process.nextTick` callback is added to `process.nextTick queue`.
A `Promise.then()` callback is added to `promises microtask queue`.
A `setTimeout`, `setImmediate` callback is added to `macrotask queue`.

Event loop executes tasks in `process.nextTick queue` first, and then executes `promises microtask queue`, and then executes `macrotask queue`.

Here is an example to show the order between `setImmediate()`, `process.nextTick()` and `Promise.then()`:

```
const baz = () => console.log('baz');
const foo = () => console.log('foo');
const zoo = () => console.log('zoo');
const start = () => {
  console.log('start');
  setImmediate(baz);
  new Promise((resolve, reject) => {
    resolve('bar');
  }).then((resolve) => {
    console.log(resolve);
    process.nextTick(zoo);
  });
  process.nextTick(foo);
};
start();
```

Output

```
Foo
Bar
Zoo
Bas
```

`setTimeout()`

When writing JavaScript code, you might want to delay the execution of a function.

This is the job of `setTimeout`. You specify a callback function to execute later, and a value expressing how later you want it to run, in milliseconds:

```
setTimeout(() => {
  // runs after 2 seconds
}, 2000);

setTimeout(() => {
  // runs after 50 milliseconds
}, 50);
```

This syntax defines a new function. You can call whatever other function you want in there, or you can pass an existing function name, and a set of parameters:

`setTimeout` returns the timer id. This is generally not used, but you can store this id, and clear it if you want to delete this scheduled function execution:

```
const id = setTimeout(() => {
  // should run after 2 seconds
}, 2000);

// I changed my mind
clearTimeout(id);
```

Zero delay

If you specify the timeout delay to `0`, the callback function will be executed as soon as possible, but after the current function execution:

```
setTimeout(() => {
  console.log('after ');
}, 0);

console.log(' before ');
```

This code will print

```
before
after
```

This is especially useful to avoid blocking the CPU on intensive tasks and let other functions be executed while performing a heavy calculation, by queuing functions in the scheduler.

`setInterval()`

`setInterval` is a function similar to `setTimeout`, with a difference: instead of running the callback function once, it will run it forever, at the specific time interval you specify (in milliseconds):

```
setInterval(() => {
  // runs every 2 seconds
}, 2000);
```

The function above runs every 2 seconds unless you tell it to stop, using `clearInterval`, passing it the interval id that `setInterval` returned:

```
const id = setInterval(() => {
  // runs every 2 seconds
}, 2000);

clearInterval(id);
```

It's common to call `clearInterval` inside the setInterval callback function, to let it auto-determine if it should run again or stop. For example this code runs something unless App.somethingIWait has the value `arrived`:

```
const interval = setInterval(() => {
  if (App.somethingIWait === 'arrived') {
    clearInterval(interval);
  }
  // otherwise do things
}, 100);
```

FileSystem

To use this File System module, use the require() method:

```
var fs = require('fs');
```

Common use for File System module:

Read Files
Write Files
Append Files
Close Files
Delete Files
Rename file

What is **Synchronous** and **Asynchronous** approach?

Synchronous approach: They are called blocking functions as it waits for each operation to complete, only after that, it executes the next operation, hence blocking the next command from execution i.e. a command will not be executed until & unless the query has finished executing to get all the result from previous commands.

Asynchronous approach: They are called non-blocking functions as it never waits for each operation to complete, rather it executes all operations in the first go itself. The result of each operation will be handled once the result is available i.e. each command will be executed soon after the execution of the previous command. While the previous command runs in the background and loads the result once it is finished processing the data.

Use cases:

If your operations **are not doing very heavy lifting like querying huge data from DB** then go ahead with **Synchronous way** otherwise asynchronous way.

In an Asynchronous way, you can show some progress indicator to the user while in the background you can continue with your heavyweight works. This is an ideal scenario for GUI based apps.

Reading File

Use fs.readFile() method to read the physical file asynchronously.

Syntax:

fs.readFile (fileName ,[options], callback)

Parameter Description:

- filename: Full path and name of the file as a string.
- options: The options parameter can be an object or string which can include encoding and flag. The default encoding is utf8 and default flag is "r". This field is optional.
- callback: A function with two parameters err and data. This will get called when readFile operation completes.

Example:

```
var fs = require("fs");

// Asynchronous read
fs.readFile('input.txt', function (err, data) {
   if (err) {
      return console.error(err);
   }
   console.log("Asynchronous read: " + data.toString());
});
```

Writing File

Use fs.writeFile() method to write data to a file. If file already exists then it overwrites the existing content otherwise it creates a new file and writes data into it.

Syntax:

fs.writeFile(filename, data[, options], callback)

Parameter Description:

- filename: Full path and name of the file as a string.

- Data: The content to be written in a file.
- options: The options parameter can be an object or string which can include encoding, mode and flag. The default encoding is utf8 and default flag is "r".
- callback: A function with two parameters err and fd. This will get called when write operation completes.

The following example creates a new file called test.txt and writes "Hello World" into it asynchronously.

```
var fs = require('fs');

fs.writeFile('test.txt', 'Hello World!', function (err) {
                        if (err)
        console.log(err);
                        else
        console.log('Write operation complete.');
});
```

In the same way, use fs.appendFile() method to append the content to an existing file.

Example: Append File Content

```
var fs = require('fs');

fs.appendFile('test.txt', 'Hello World!', function (err) {
      if (err)
        console.log(err);
      else
        console.log('Append operation complete.');
});
```

Delete File

Use fs.unlink() method to delete an existing file.
Syntax :
fs.unlink(path, callback);

The following example deletes an existing file.

Example: Delete file

```
var fs = require('fs');

fs.unlink('test.txt', function () {

  console.log('Deleted successfully..');

});
```

Renaming a File

A file can be renamed using fs.rename(). Here, it takes old file name, new file name, and a callback function as arguments.

```
fs.rename('hello1.txt', 'hello2.txt', function(err) {
  if (err)
    console.log(err);

  console.log('Rename complete!');
});
```

For career objective

What Is LeetCode?

It's a website where people–mostly software engineers–practice their coding skills. There are 800+ questions (and growing), each with multiple solutions. Questions are ranked by level of difficulty: easy, medium, and hard.

Similar websites include **HackerRank**, **Topcoder**, **InterviewBit**, among others. There's also a popular book, "Cracking the Coding Interview," which some call the Bible for engineers. The Blind community uses a mix of these resources, but based on mentions, LeetCode seems to be the most popular. Our active users cite the following reasons for preferring LeetCode: more questions, better quality, plus a strong user base.