Node.js/Express “Cheat Sheet”

This reference summarizes the most useful methods/properties used in CSE 154 for Node.js/Express. It is not an exhaustive reference for everything in Node.js/Express (for example, there exist many more `fs` methods/properties than are shown below), but provide most functions/properties you will be using in this class. *Note that this Cheat Sheet is more comprehensive than the one provided in CSE154 exams.*

Basic Node.js Project Structure

```
example-node-project/
  .gitignore
  APIDOC.md
  app.js
  node_modules/
    ...
  package.json
  public/
    img/
      ...
    index.html
    index.js
    styles.css
```

Example Express app Template

```
"use strict";

/* File comment */

const express = require("express");
// other modules you use
// program constants

const app = express();
// if serving front-end files in public/
app.use(express.static(“public”));

// if handling different POST formats
app.use(express.urlencoded({ extended: true }));
app.use(express.json());
app.use(multer().none());

// app.get/app.post endpoints

// helper functions

const PORT = process.env.PORT || 8000;
app.listen(PORT);
```

npm Commands

Command	Description
`npm init`	Initializes a node project. Creates packages.json to track required modules/packages, as well as the node_modules folder to track imported packages.
`npm install`	Installs any requirements for the local node package based on the contents of package.json.
`npm install <package-name>`	Installs a package from NPM’s own repository as well as any requirements specified in that package’s package.json file.

Glossary

Term	Definition
API	Application Programming Interface.
Web Service	A type of API that supports HTTP Requests, returning data such as JSON or plain text.
Express	A module for simplifying the http-server core module in Node to implement APIs
npm	The Node Package Manager. Used to initialize package.json files and install Node project dependencies.
Module	A standalone package which can be used to extend the functionality of a Node project.
Package	Any project with a package.json file.
API Documentation	A file detailing the endpoints, usage, and functionality of various endpoints of an API.
Server	A publicly accessible machine which exchanges information with one or more clients at a time.
Client	A private or public machine which requests information from a server.

Useful Core Modules

Module	Description
`fs`	The "file system" module with various functions to process data in the file system.
`path`	Provides functions to process path strings.
`util`	Provides various "utility" functions, such as util.promisify.

Other Useful Modules

The following modules must be installed for each new project using **`npm install <module-name>`**.

Module	Description
`express`	A module for simplifying the http-server core module in Node to implement APIs.
`glob`	Allows for quick traversal and filtering of files in a complex directory.
`multer`	Used to support FormData POST requests on the server-side so we can access the req.body parameters.
`mysql`	Provides functionality for interacting with a database and tables.
`promise-mysql`	Promisified wrapper over mysql module - each function in the mysql module returns a promise instead of taking a callback as the last argument (recommended).
`cookie-parser`	A module to access cookies with req/res objects in Express.

Express Route Functions

Function	Description

`app.get(“path”, middlewareFn(s));`
 	
`app.get(“/”, (req, res) => {`
`  ...`
`});`
 	
`app.get(“/:city”, (req, res) => {`
`  let city = req.params.city;`
`  ...`
`});`
 	
`app.get(“/cityData”, (req, res) => {`
`  let city = req.query.city;`
`  ...`
`});`
 	
`// Example with multiple middleware functions`
`app.get(“/”, validateInput, (req, res) => {`
`  ...`
`}, handleErr);`
	Defines a server endpoint which accepts a valid **`GET`** request. Request and response objects are passed as **`req`** and **`res`** respectively. Path parameters can be specified in **`path`** with “:varname” and accessed via **`req.params`**. Query parameters can be accessed via **`req.query`**.

`app.post(“path”, middlewareFn(s));`
 	
`app.post(“/addItem”, (req, res) => {`
`  let itemName = req.body.name;`
`  ...`
`}`
	Defines a server endpoint which accepts a valid **`POST`** request. Request and response objects are passed as **`req`** and **`res`** respectively. POST body is accessible via **`req.body`**. Requires POST middleware and multer module for FormData POST requests.

GOMY CODE

Request Object Properties/Functions

Property/Function	**Description**
`req.params`	Captures a dictionary of desired path parameters. Keys are placeholder names and values are the URL itself.
`req.query`	Captures a dictionary of query parameters, specified in a *?key1=value1&key2=value2&* … pattern.
`req.body`	Holds a dictionary of POST parameters as key/value pairs. Requires multer module for multipart form requests (e.g. FormData) and using middleware functions (see Express template) for other POST request types.
`req.cookies`	Retrieves all cookies sent in the request. Requires cookie-parser module.

Response Object Properties/Functions

Property/Function	**Description**

`res.set(headerName, value);`
 	
`res.set("Content-Type", "text/plain");`
`res.set("Content-Type", "application/json");`
	Used to set different response headers - commonly the "Content-type" (though there are others we don't cover).

`res.type("text");`
`res.type("json");`
	Shorthand for setting the "Content-Type" header.

`res.send(data);`
 	
`res.send("Hello");`
`res.send({ "msg" : "Hello" });`
	Sends the data back to the client, signaling an end to the response (does not terminate your JS program).

`res.end();`	Ends the request/response cycle without any data (does not terminate your JS program).
`res.json(data);`	Shorthand for setting the content type to JSON and sending JSON.

`res.status(statusCode)`
 	
`res.status(400).send("client-side error message");`
`res.status(500).send("server-side error message");`
	Specifies the HTTP status code of the response to communicate success/failure to a client.

Useful fs Module Functions

Note: You will often see the following “promisified” for use with async/await. The promisified versions will return a Promise that resolves callback’s contents or rejects if there was an error. Remember that you should always handle potential fs function errors (try/catch if async/await, if/else if standard callback).

Example of promisifying callback-version of `fs.readFile`:

```
fs.readFile(“file.txt”, “utf8”, (err, contents) => {
  ...
});

// Promisified:
const util = require(“util”);
const readFile = util.promisify(fs.readFile);

async function example() {
  try {
    let contents = await readFile(“file.txt”);
    ...
  } catch(err) {
    ...
  }
}
```

Function	Description

`fs.readFile(filename, “utf8”, callback);`
 	
`fs.readFile(“file.txt”, “utf8”,`
`  (err, contents) => { ... }`
`);`
	Reads the contents of the file located at relative directory **`filename`**. If successful, passes the file contents to the callback as **`contents`** parameter. Otherwise, passes error info to callback as **`error`** parameter.

`fs.writeFile(filename, data, “utf8”, callback);`
 	
`fs.writeFile(“file.txt”, “new contents”, “utf8”,`
`  (err) => {  ...  }`
`);`
	Writes **`data`** string to the file specified by **`filename`**, overwriting its contents if it already exists. If an error occurs, **`error`** is passed to the callback function.

`fs.appendFile(filename, data, “utf8”, callback);`
 	
`fs.appendFile(“file.txt”, “added contents”, “utf8”,`
`  (err) => { ... }`
`);`
	Writes **`data`** to the file specified by **`filename`**, appending to its contents. Creates a new file if the filename does not exist. If an error occurs, **`error`** is passed to the callback function.

`fs.existsSync(filename);`	Returns true if the given filename exists. <u>This is the only synchronous fs function you may use in CSE154</u> (the asynchronous function is deprecated due to race conditions).

`fs.readdir(path, callback);`
 	
`fs.readdir(“dir/path”, (err, contents) => {`
`  ...`
`});`
	Retrieves all files within a directory located at **`path`**. If successful, passes the array of directory content paths (as strings) to the callback as **`contents`** parameter. Otherwise, passes error info to callback as **`error`** parameter.

Useful path Module Functions

Function	Description
`path.basename(pathStr);`	Returns the filename of the **pathStr**. Ex. “picture.png” for “img/picture.png”
`path.extname(pathStr);`	Returns the file type/file extension of the **pathStr**. Ex. “.png” for “img/picture.png”
`path.dirname(pathStr);`	Returns the directory name of the **pathStr**. Ex: “img/” for “img/picture.png”

The glob module

Function	Description

`glob(pattern, callback);`
 	
`glob(“img/*”, (err, paths) => {`
`  ...`
`});`
 	
`// promisified`
`paths = await glob(“img/*”);`
	Globs all path strings matching a provided `pattern`. The pattern will generally follow the structure of a file path, along with any wildcards. If no paths match, the result array will be empty. If successful, passes the array of directory content paths (as strings) to the callback as `contents` parameter. Otherwise, passes error info to callback as `error` parameter.
	
	Common selectors:
	* - A wildcard selector that will contextually match a single filename, suffix, or directory.
	** - A recursive selector that will search into any subdirectories for the pattern that follows it.

The promise-mysql module

Function	Description

`mysql.createConnection({`
`  host : hostname, // default localhost`
`  port : port,`
`  user : username,`
`  password : pw,`
`  database : dbname`
`});`
	Returns a connected database object using config variables. If an error occurs during connection (e.g. the SQL server is not running), does not return a defined database object.

`db.query(qryString);`
 	
`db.query(qryString, [placeholders]);`
	Executes the SQL query. If the query is a SELECT statement, returns a Promise that resolves to an array of RowDataPackets with the records matching the **qryString** passed. If the query is an INSERT statement, the Promise resolves to an OkPacket. Throws an error if something goes wrong during the query.
	
	When using variables in your query string, you should use ? placeholders in the string and populate [placeholders] with the variable names to sanitize the input against SQL injection