Week 11 Study Guide

Table of Contents

Regular Expressions Objectives

1. Define the effect of the * operator and use it in a regular expression

`*` **star operator** - zero or more (of what's right before it)

Example of `*` operator

`/th*e/` t, zero or more h, e

2. Define the effect of the ? operator and use it in a regular expression

`?` **optional operator** - zero or one (of what's right before it)

Example of `?` operator

`at?t` - a, zero or one t, t

3. Define the effect of the + operator and use it in a regular expression

`+` **plus operator** - one or more (of what's right before it)

Example of `+` operator

`at+` - a, one or more t

4. Define the effect of the . operator and use it in a regular expression

`.` **dot operator** - any one character

Example of `.` operator

`e .` - e, space, any char

5. Define the effect of the ^ operator and use it in a regular expression

`^` **hat operator** - start of input anchor

Example of `^` operator

`^Is` - start of input, I, s

6. Define the effect of the $ operator and use it in a regular expression

`$` **money operator** - end of input

Example of `$` operator

`at.$` - a, t, any char, end of input

7. Define the effect of the [] bracket expression and use it in a regular expression

`[]` **square brackets** - your choice

Example of `[]`

`a[tm]e` - a, t or m, e

8. Define the effect of the - inside brackets and use it in a regular expression

`-` **dash operator** - (only works inside square brackets) - range of chars

Example of `-` operator

`[a-zA-Z]at` - any lower/uppercase char, a, t

9. Define the effect of the ^ inside brackets and use it in a regular expression

`^` **hat operator inside square brackets** - none of them

Example of `^` inside `[]`

`[^a-zA-Z]` - any non letter char

Other useful informaiton about regular expressions

Grouping operator

`()` **parenthesis** - used ($1, $2, $3) - primarily used to capture groups of chars

Example of `()` operator

`at( is)?` - a, t, optional (space, i, s)

Shorthands

- `\s` - white space
- `\d` - digit
- `\w` - word char
- `\S` - not white space
- `\D` - not a digit
- `\W` - not a word char

Using Regular expressions in Javascript

```
const re = /EX$/
const str = "We're learning REGEX"

const li = [ 1, 2, di ];
const re = /regex/i;

console.log(re.test(`learning about regex`));
console.log(re.test(`LEARNING ABOUT REGEX`));

let count = 0;

const newStr = str.replace(/e/ig, match => {
    count += 1
    return count;
  });
  console.log(newStr);

const di = { name: `Mimi`, age: 2 };
const str = `My name is %name% and I am %age% years old`;
const str2 = `My name is ${name} and I am ${age} years old`;

const re = /%\w+%/g

const replaced = str.replace(re, match => {
  // match = %name%
  const key = match.replace(/%/g, ``);
  // key = name
  return di[key];
});

console.log(replaced);
```

Node HTTP Objectives

1. Identify the five parts of a URL

Given this URL `https://example.com:8042/over/there?name=ferret#nose`

Scheme	Authority	Path	Query	Fragment
https	example.com:8042	/over/theme	name=ferret	nose

2. Identify at least three protocols handled by the browser

- https - Secure HTTP
- http - HTTP
- file - Opening a file
- ws - Websocket

3. Use an `IncomingMessage` object to

An `IncomingMessage` object is usually represented by the `req` variable.

access the headers sent by a client (like a Web browser) as part of the HTTP request

```
console.log(req.headers);
// Prints something like:
//
// { 'user-agent': 'curl/7.22.0',
//   host: '127.0.0.1:8000',
//   accept: '*/*' }
```

access the HTTP method of the request

```
console.log(req.method); // prints out the method like `GET` or `POST`
```

access the path of the request

```
console.log(req.url); // access the path of the request as a string
```

access and read the stream of content for requests that have a body

`IncomingMessage` is a subclass of `stream.Readable` in node, so we can read it like a stream.

Since `stream.Readable` is an `async iterable` we can use `for await..of` with it, and we'll get each chuck of data from the stream and we can concatenate them back together.

You don't need to know all the details of how async iterators work to use them like this:

```
let body = '';
for await (let chunk of req) {
  body += chunk;
}
```

4. Use a ServerResponse object to

write the status code, message, and headers for an HTTP response

Status code

```
res.statusCode = 404;
```

message

```
res.statusMessage = 'Page not found';
```

headers

```
res.setHeader('Content-Type', 'text/html');
```

write the content of the body of the response

Assuming body is a big string of HTML…

```
res.write(body);
```

properly end the response to indicate to the client (like a Web browser) that all content has been written

You can optionally pass some more data to write in the call to end.

```
// Without a string
res.end()
// With a string (assuming rest_of_body is some more HTML)
res.end(rest_of_body)
```

Express Objectives

1. Send plain text responses for any HTTP request.

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
    res.send('Hello World!');
});

const port = 8000;
app.listen(port, () => console.log(`App listening on port ${port}...`));
```

2. Use pattern matching to match HTTP request paths to route handlers.

```
// This matches /pictures followed by anything: /pictures/summer, /pictures/prom
app.get(/^\/pictures\/.+$/i, (req, res) => {
    res.send('Here are some pictures');
});

// This route will only match URL paths that have a number in the route parameter's position
app.get('/pictures/:id(\\d+)', (req, res) => {
    res.send('Here is a single picture');
});
```

3. Use the Pug template engine to generate HTML from Pug templates to send to the browser.

```
html
  head
    title Pug Demo
  body
    h1 Pug Demo
    h2 Pug allows you to do many things
    ul
      li: a(href='google.com') This is a link to google.com
      li: span.yellow This is a span with class='yellow'
    h2 And now some colors
```

This will render into the following HTML

```
<html>
  <head>
    <title>Pug Demo</title>
  </head>
  <body>
    <h1>Pug Demo</h1>
    <h2>Pug allows you to do many things</h2>
    <ul>
      <li><a href="google.com">This is a link to google.com</a></li>
      <li><span class="yellow">This is a span with class='yellow'</span></li>
    </ul>
    <h2>And now some colors</h2>
  </body>
</html>
```

4. Pass data to Pug templates to generate dynamic content

```
// app.js
app.get('/pug', (req, res) => {
  res.render('eod', {
    title: 'EOD demo',
    header: 'Pug demo',
    colors: ['blue', 'red', 'green']});
});
```

```
html
  head
    title= title
    style
      include style.css
  body
    h1 This is the #{header} page which does string interpolation
    h2 Pug allows you to do many things
    #violet This is a div with an id='violet'
    .purple This is a div with a class='purple'

    p
      - const a = 1 + 2; // Look!, we can do inline javascript that doesn't output!
      = `The answer is ${a}` // Look this runs some inline javascript that does output!

    ul
      li: a(href='http://google.com') This is a link to google.com
      li: span.yellow This is a span with class='yellow'
    h2 And now some colors
    ul
    each color in colors
      li= color
```

This outputs this html

```
<html>
  <head>
    <title>EOD demo</title>
    <style><!-- CSS would be here from the styles.css file --></style>
  </head>
  <body>
    <h1>This is the Pug demo page which does string interpolation</h1>
    <h2>Pug allows you to do many things</h2>
    <div id="violet">This is a div with an id='violet'</div>
    <div class="purple">This is a div with a class='purple'</div>
    <p>The answer is 3
    </p>
    <ul>
      <li><a href="http://google.com">This is a link to google.com</a></li>
      <li><span class="yellow">This is a span with class='yellow'</span></li>
    </ul>
    <h2>And now some colors</h2>
    <ul></ul>
    <li>blue</li>
    <li>red</li>
    <li>green</li>
  </body>
</html>
```

5. Use the Router class to modularize the definition of routes

```
// routes.js
const express = require('express');

const eodRoutes = express.Router();

// Because, this sub-router is registered on /eod "app.use('/eod', eodRoutes);" in app.js), it will match the route /eod/questions
eodRoutes.get('/questions', (req, res) => {
  res.send('Answers');
});

module.exports = eodRoutes;
```

```
// app.js
const express = require('express');
const eodRoutes = require('./routes');

// Create the Express app.
const app = express();

app.use('/eod', eodRoutes);

// Define a port and start listening for connections.
const port = 8081;

app.listen(port, () => console.log(`Listening on port ${port}...`));
```

Pug Objectives

1. Declare HTML tags and their associated ids, classes, attributes, and content.

```
html
  head
    title
  body
    div#main
      div.blue
      div.yellow
        a(href="http://google.com") Click here
```

2. Use conditional statements to determine whether or not to render a block.

```
// app.js
res.render('layout', { isEOD: true });
```

```
if isEOD
  h2 Welcome back!
else
  h2 Keep coding!
```

3. Use interpolation to mix static text and dynamic values in content and attributes.

We use the `#{}` syntax to do interpolation in pug templates

```
res.render('layout', {
  title: 'Pug demo page',
  header: 'interpolation'
})
```

```
html
  head
    title= title
    style
      include style.css
  body
    h1 Pug does #{header}
    h2 Pug allows you to do many things
    ul
      li: a(href='http://google.com') This is a link to google.com
```

4. Use iteration to generate multiple blocks of HTML based on data provided to the template.

```
// app.js
app.get('/pug', (req, res) => {
  res.render('eod', { colors: ['blue', 'red', 'green'] });
});
```

```
ul
  each color in colors
    li= color
```

HTML Form Objectives

1. Describe the interaction between the client and server when an HTML form is loaded into the browser, the user submits it, and the server processes it

```
<form action="/users" method="post">
  <label>Username:
    <input type="text" name="user[username]">
  </label>
  <label>Email:
    <input type="email" name="user[email]">
  </label>
  <label>Age:
    <input type="number" name="user[age]">
  </label>
  <label>Password:
    <input type="password" name="user[password]">
  </label>
  <input type="submit" value="Sign Up">
</form>
```

action attribute of the form element defines the url that the request is made to

- absolute URL - https://www.wellsfargo.com/transfers
- relative URL - /users - localhost:3000/users
- no URL - form will be sent to the same page(url) the form is present on

method attribute defines how the data will be sent

if method "post" is used, form data is appended to the body of the HTTP request

the server receives a string that will be parsed in order to get the data
as a list of key/value pairs

only the `get` and `post` methods may be specified on the form. in order to do `put`, `delete`, or other methods, we must us `AJAX` requests using the `fetch` API, for example.

2. Create an HTML form using the Pug template engine

```
form(method="post" action="/users")
  input(type="hidden" name="_csrf" value=csrfToken)
  label(for="username") Username:
  input(type="username" id="username" name="username" value=username)
  label(for="email") Email:
  input(type="email" id="email" name="email" value=email)
  label(for="age") Age:
  input(type="age" id="age" name="age" value=age)
  label(for="password") Password:
  input(type="password" id="password" name="password")
  input(type="submit" value="Sign Up")
```

3. Use express to handle a form's POST request

```
// localhost:3000/about
```

```
app.get(`/`, csrfProtection, (req, res) => {
  res.render(`index`, {
    title: `User List`,
    users, errors: [],
    csrfToken: req.csrfToken(),
  });
});

const users = new Array();

app.post(`/users`, [csrfProtection, checkFields], (req, res) => {
  if (req.errors.length >= 1) {
    res.render(`index`, { errors: req.errors, users, });
    return
  }
  const { username, email, password, } = req.body;
  const user = { username, email, password, };
  users.push(user);
  res.redirect(`/`);
});
```

4. Use the built-in express.urlencoded() middleware function to parse incoming request body form data

```
app.use(express.urlencoded({
  extended: true,
}));
```

5. Explain what data validation is and why it's necessary for the server to validate incoming data

- username - limit number of chars
- password - special char requirement and/or minimum length
- email - valid email

Data validation is the process of ensuring that the incoming data is correct.

Even though you could add add validations on the client side, client-side
validations are not as secure and can be circumvented. Because client-side validations can be circumvented, it's necessary to implement server-side data validations.

Handling bad request data in our route handlers allows us to return 400 level messages with appropriate messaging to allow the user to correct their bad request. Versus allowing the bad data go all the way to the database, and the system returning the generic "500: Internal Server Error".

6. Validate user-provided data from within an Express route handler function

7. Write a custom middleware function that validates user-provided data

8 .Use the csurf middleware to embed a token value in forms to protect against Cross-Site Request Forgery exploits

```
const csrf = require("csurf");
const csrfProtection = csrf({ cookie: true })

app.use((req, res, next) => {
  req.errors = [];
})

const checkFields = (req, res, next) => {
  const { username, email, password } = req.body;
  // Alternatively:
  // const username = req.body.username
  // const email = req.body.email
  // const password = req.body.password
  if (!username || !email || !password) {
    req.errors.push(`you are missing required fields`);
  }

  next();
};

app.get(`/`, csrfProtection, (req, res) => {
  res.render(`index`, { title: `User List`, users, errors: [], csrfToken: req.csrfToken() });
});

const users = new Array();

app.post(`/users`, [csrfProtection, checkFields], (req, res) => {
  if (req.errors.length >= 1) {
    res.render(`index`, { errors: req.errors, users });
    req.errors = [];
    return
  }
  const { username, email, password, } = req.body;
  const user = { un: username, email, password, };
  users.push(user);
  res.redirect(`/`);
});
```

Data-Driven Web Sites Objectives

1. Use environment variables to specify configuration of or provide sensitive information for your code

What environment variables are and how to access them in Node.js

Environment variables are global variables that are part of your unix shell.
You can access these variables inside of Node.js by using the built-in
`process.env` object.

For instance, assume we have the following script named `test.js`

```
console.log(process.env.NODE_ENV);
```

And we run it like this, setting an environment variable `inline` on the command
prompt.

```
NODE_ENV=production node test.js
```

We would expect "production" to be printed.

We can now store sensitive information in these variables

Assuming we have an `app.js` that tries to do something with a secret password

```
// Somewhere inside app.js
const mySecretPassword = process.env.MY_SECRET_PASSWORD;
// Now we have access to the secret password
// without checking it into our code!
```

We can pass in the secret password at runtime.

```
MY_SECRET_PASSWORD=zweeble node app.js
```

This isn't super convenient though, so that's where the next learning objective
comes in….

2. Use the dotenv npm package to load environment variables defined in an .env file

dotenv is a npm package that is designed to
read values from a `.env` file and populate them as *environment variables*. In
this way we can keep the variables in a file instead of having to put them
on the command line. We can then put the `.env` file into our `.gitignore` file
so we don't check it in to git and github.

A typical `.env` file might look like this:

```
PORT=8080
DB_USERNAME=mydbuser
DB_PASSWORD=mydbuserpassword
DB_DATABASE=mydbname
DB_HOST=localhost
```

You can install the dotenv package using this command

```
npm install dotenv --save-dev
```

There are **three** ways to use `dotenv`.

Using dotenv in our JavaScript code

```
// app.js
// Just requiring dotenv will cause it to
// Load the environment variables from the .env file
require(`dotenv`).config();
```

Using dotenv as a module with `node` or `nodemon`

The `-r` command line option to node (or nodemon) makes node require
in the module BEFORE it runs our script.

```
node -r dotenv/config app.js
```

We can also use this inside package.json in the `scripts` section:

```
"scripts": {
  "start": "node -r dotenv/config app.js"
```

Using dotenv from the command line

This requires that you install the following package:

`npm install dotenv-cli --save-dev`

You can now run the dotenv command in front
of any other unix command and it'll populate the environment.

For instance to use it with the sequelize command line:

```
npx dotenv sequelize-cli db:migrate
```

3. Recall that Express cannot process unhandled Promise rejections from within route handler (or middleware) functions

If you have a route like this with an asynchronous callback function

```
app.get("/", async (req, res) => {
   // And some code here throws an error
});
```

If the code inside the callback throws an error, then you will get an "Unhandled Promise Rejection" error from express in the console and the web server will hang.

4. Use a Promise catch block or a try/catch statement with async/await to properly handle errors thrown from within an asynchronous route handler (or middleware) function

To properly handle this, you must catch the error and call "next()" passing it
then error. Then express will know it was an error and handle it gracefully.

Using async/await

```
app.get('*', async (req, res, next) => {
 try {
   // Assume some command is here that throws an error
 } catch (err) {
   // We catch it here to make express happy
   next(err);
 }
});
```

Using `.then/.catch`

```
app.get('*', async(req, res, next) => {
  someAsyncFunction().then(() => {
    // Assume some command is here that throws an error
  })
  .catch(err => {
    // We catch it here to make express happy
    next(err);
  })
});
```

5. Write a wrapper function to simplify catching errors thrown within asynchronous route handler (or middleware) functions

Here's the super shortened version:

```
const asyncHandler = (handler) => (req, res, next) => handler(req, res, next).catch(next);
```

We can expand this to better understand what it's doing, let's remove the arrow functions and put
back all the curly braces.

```
function asyncHandler(handler) {
  // This is going to be the new function that express will call in the route
  return function(req, res, next) {
    // This will call the original function we passed into `asyncHandler`
    return handler(req, rew, next).catch(function (err) {
      // When there's an error it gets caught and we call next with it
      // to make express happy
      next(err);
    });
  }
}
```

We can now use this helper function in routes like this:

```
// Notice that asyncHandler is being passed an arrow function, so this
// gets called when our server starts up, NOT when a request arrives.
app.get("/", asyncHandler((req, res) => {
  // In here:
  // We call some async function that might return an error, but it's okay
  // because we wrapped it in our async handler that catches them and calls
  // next();
}))
```

6. Use the morgan npm package to log requests to the terminal window to assist with auditing and debugging

install morgan:

`npm install morgan`

Then add it as middleware to your `app.js`

```
const morgan = require('morgan');

app.use(morgan('dev'));
```

And it magically logs your requests to the console!

7. Add support for the Bootstrap front-end component library to a Pug layout template

CSS into the head of the pug template

Follow the directions on the bootstrap site, but convert the HTML into pug.

```
link(rel='stylesheet'
    href='https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css'
    integrity='sha384-Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh'
    crossorigin='anonymous')
```

JS into bottom of the pug template

```
script(src="https://code.jquery.com/jquery-3.5.1.slim.min.js"
       integrity="sha384-DfXdz2htPH0lsSSs5nCTpuj/zy4C+OGpamoFVy38MVBnE+IbbVYUew+OrCXaRkfj" crossorigin="anonymous")
script(src="https://cdn.jsdelivr.net/npm/popper.js@1.16.1/dist/umd/popper.min.js"
       integrity="sha384-9/reFTGAW83EW2RDu2S0VKaIzap3H66lZH81PoYlFhbGU+6BZp6G7niu735Sk7lN"
       crossorigin="anonymous")
script(src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"
       integrity="sha384-B4gt1jrGC7Jh4AgTPSdUtOBvfO8shuf57BaghqFfPlYxofvL8/KUEfYiJOMMV+rV"
       crossorigin="anonymous")
```

8. Install and configure Sequelize within an Express application.

Install sequelize like normal

`npm install sequelize@^5.0.0 pg@^8.0.0`

`npm install sequelize-cli@^5.0.0 --save-dev`

A .sequelizerc is mostly optional, it changes the folder structure sequelize uses
and also allows us to have our `config.json` be `database.js` so we can
use it as a JavaScript file instead of a JSON file. (Which means we can now use
process.env to grab our environment variables)

```
// .sequelizerc

const path = require('path');

module.exports = {
  'config': path.resolve('config', 'database.js'),
  'models-path': path.resolve('db', 'models'),
  'seeders-path': path.resolve('db', 'seeders'),
  'migrations-path': path.resolve('db', 'migrations')
};
```

Now we need to setup our database and our user

```
create database reading_list;
create user reading_list_app with encrypted password '«a strong password for the reading_list_app user»';
grant all privileges on database reading_list to reading_list_app;
```

Now we can initialize sequelize

```
$ npx sequelize init
```

And create a `.env` file to hold our environment variables

```
// env
PORT=8080
DB_USERNAME=reading_list_app
DB_PASSWORD=«the reading_list_app user password»
DB_DATABASE=reading_list
DB_HOST=localhost
```

We create a config/index.js to hold any of our express configuration,
including our sequelize configuration values.

```
// ./config/index.js

module.exports = {
  environment: process.env.NODE_ENV || 'development',
  port: process.env.PORT || 8080,
  db: {
    username: process.env.DB_USERNAME,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_DATABASE,
    host: process.env.DB_HOST,
  },
};
```

Now we can make sequelize's `database.js` config file require
our main express `config/index.js` file to get the values it needs.
(This replaces the old config.json file we used to use with sequelize)

```
// ./config/database.js

const {
  username,
  password,
  database,
  host,
} = require('./index').db;

module.exports = {
  development: {
    username,
    password,
    database,
    host,
    dialect: 'postgres',
  },
};
```

9. Use Sequelize to test the connection to a database before starting the HTTP server on application startup

```
#!/usr/bin/env node

const { port } = require('../config');

const app = require('../app');
const db = require('../db/models');

// Check the database connection before starting the app.
db.sequelize.authenticate()
  .then(() => {
    console.log('Database connection success! Sequelize is ready to use...');

    // Start listening for connections.
    app.listen(port, () => console.log(`Listening on port ${port}...`));
  })
  .catch((err) => {
    console.log('Database connection failure.');
    console.error(err);
  });
```

10. Define a collection of routes (and views) that perform CRUD operations against a single resource using Sequelize

Reminder, CRUD means "Create, Read, Update, Delete"

A Resource is an entity within your application, usually this is directly related
to a model or a table in your database.

11. describe how an Express.js error handler function differs from middleware and route handler functions

Express middleware functions define three parameters (req, res, next) and route
handlers define two or three parameters (req, res, and optionally the next parameter):

```
// Middleware function.
app.use((req, res, next) => {
  console.log('Hello from a middleware function!');
  next();
});

// Route handler function.
app.get('/', (req, res) => {
  res.send('Hello from a route handler function!');
});
```

Error handling functions look the same as middleware functions except they
define four parameters instead of three—err, req, res, and next:

```
app.use((err, req, res, next) => {
  console.error(err);
  res.send('An error occurred!');
});
```

12. Define a global Express.js error-handling function to catch and process unhandled errors

Since this is to catch and process 'unhandled' errors, it needs to be the very
last error handler in your application.

```
// Custom error handler.
app.use((err, req, res, next) => {
  console.error(err);
  res.status(err.status || 500);
  res.send('An error occurred!');
});
```

13. Define a middleware function to handle requests for unknown routes by returning a 404 NOT FOUND error

First we setup a middleware to catch everything that routes didn't match.
This should be the last middleware before we start defining error handlers

It will just create a `new Error` and set the status to 404, so that we
can catch and handle it in the error handlers later.

```
app.use((req, res, next) => {
  const err = new Error('The requested page couldn\'t be found.');
  err.status = 404;
  next(err);
});
```

Then somewhere in our error handlers, we check to see if the status is 404
and if it is, we render a custom template for "Page not Found". Don't
forget to call `next(err)` and pass the error on down to other error handlers!

```
// Error handler for 404 errors.
app.use((err, req, res, next) => {
  if (err.status === 404) {
    res.status(404);
    res.render('page-not-found', {
      title: 'Page Not Found',
    });
    } else {
      next(err);
    }
  });
```