Response Methods

- The methods on the response object (res) in the following table can send a response to the client, and terminate the request-response cycle
  - If none of these methods are called, the client request will be left hanging

Method	Description
res.download()	Prompt a file to be downloaded.
res.end()	End the response process.
res.json()	Send a JSON response.
res.jsonp()	Send a JSON response with JSONP support.
res.redirect()	Redirect a request.
res.render()	Render a view template.
res.send()	Send a response of various types.
res.sendFile()	Send a file as an octet stream.
res.sendStatus()	Set the response status code and send its string representation as the response body.

res.send()

- **res.send(body)** Sends the HTTP response
- The body parameter can be a string, an object, an array or a Buffer object
- The method automatically assigns the Content-Length response header field (unless previously defined)
  - When the parameter is a string, the method sets the Content-Type to “text/html”:

```
res.send('<p>some html</p>');
```

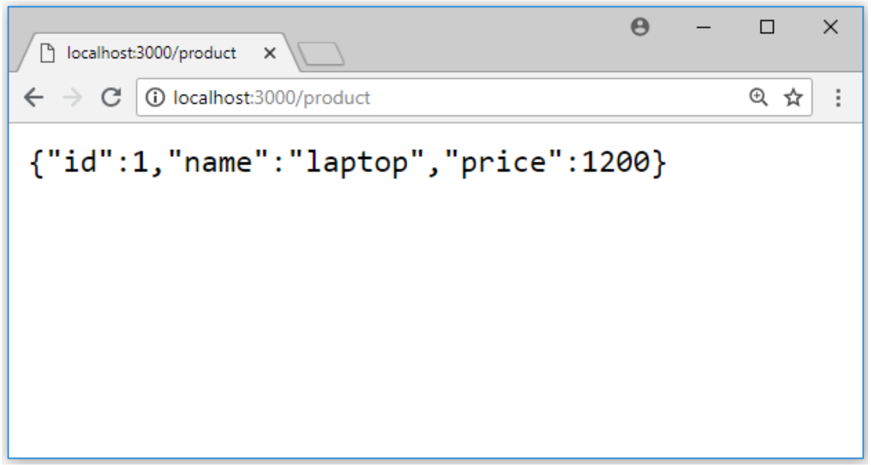
  - When the parameter is an array or object, Express responds with the JSON representation:

```
res.send({ user: 'tobi' });
res.send([1,2,3]);
```

res.json()

- **res.json(body)** sends a JSON response
- This method converts its parameter to a JSON string using JSON.stringify()
- The parameter can be any JSON type, including object, array, string, Boolean, or number
- Example:

```
app.get('/product', (req, res) => {
    let product = {
        id: 1,
        name: 'laptop',
        price: 1200
    }

    res.json(product);
});
```

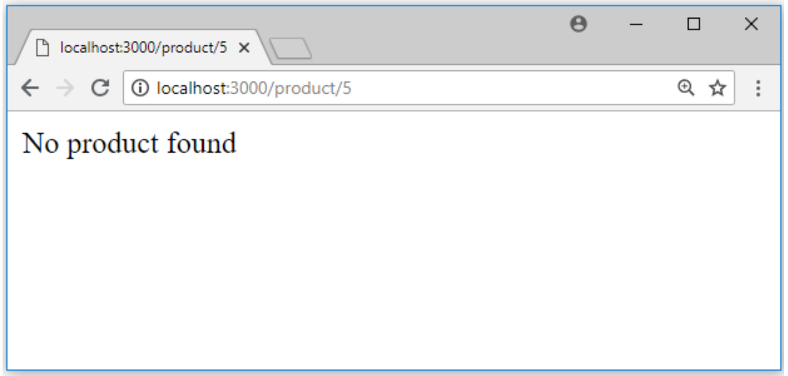


```
{"id":1,"name":"laptop","price":1200}
```

res.status()

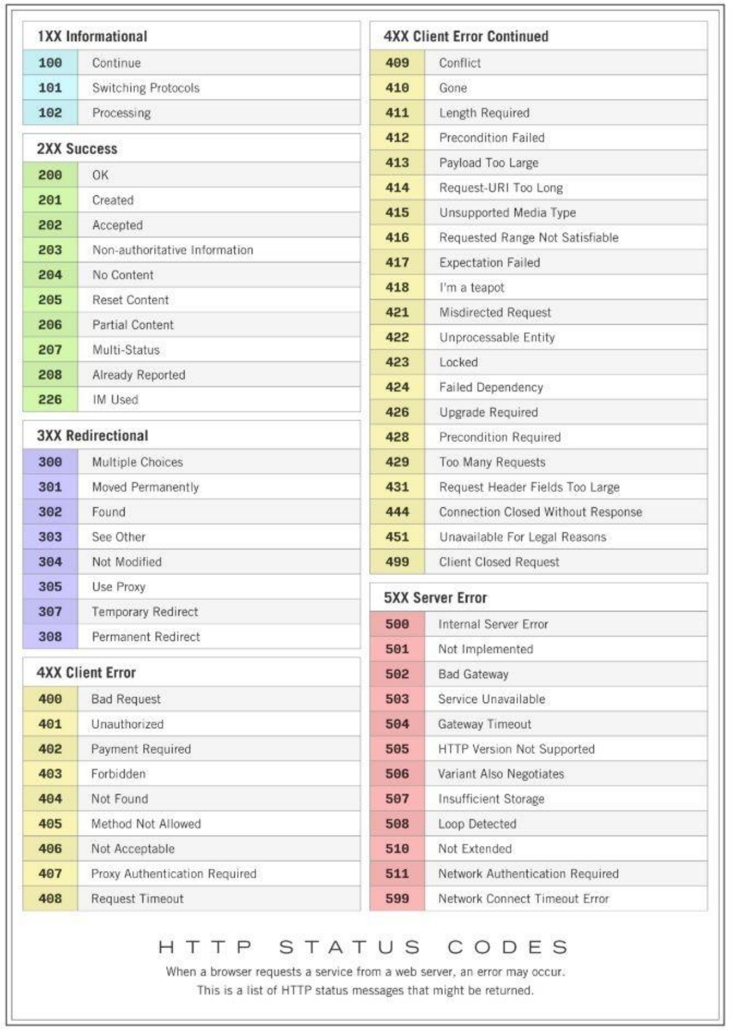
- Sets the HTTP status for the response
- For example, if the product id was not found on the server, we can set the status code to HTTP 404 (Not Found):

```
let products = [
    { id: 1, name: 'laptop', price: 1200 },
    { id: 2, name: 'chair', price: 200 },
    { id: 3, name: 'printer', price: 250 }
];

app.get('/product/:id', (req, res) => {
    let productId = req.params.id;

    let product = products.find(p => p.id == productId);
    if (!product) {
        return res.status(404).send("No product found");
    }
    else {
        res.json(product);
    }
});
```



No product found

HTTP Status Codes



1XX Informational	
100	Continue
101	Switching Protocols
102	Processing

2XX Success	
200	OK
201	Created
202	Accepted
203	Non-authoritative Information
204	No Content
205	Reset Content
206	Partial Content
207	Multi-Status
208	Already Reported
226	IM Used

3XX Redirectional	
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
307	Temporary Redirect
308	Permanent Redirect

4XX Client Error	
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout

4XX Client Error Continued	
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Payload Too Large
414	Request-URI Too Long
415	Unsupported Media Type
416	Requested Range Not Satisfiable
417	Expectation Failed
418	I'm a teapot
421	Misdirected Request
422	Unprocessable Entity
423	Locked
424	Failed Dependency
426	Upgrade Required
428	Precondition Required
429	Too Many Requests
431	Request Header Fields Too Large
444	Connection Closed Without Response
451	Unavailable For Legal Reasons
499	Client Closed Request

5XX Server Error	
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported
506	Variant Also Negotiates
507	Insufficient Storage
508	Loop Detected
510	Not Extended
511	Network Authentication Required
599	Network Connect Timeout Error

HTTP STATUS CODES

When a browser requests a service from a web server, an error may occur.
This is a list of HTTP status messages that might be returned.

res.sendStatus()

- **res.sendStatus(statusCode)** sets the response HTTP status code to statusCode and sends its string representation as the response body

```
res.sendStatus(200); // equivalent to res.status(200).send('OK')
res.sendStatus(403); // equivalent to res.status(403).send('Forbidden')
res.sendStatus(404); // equivalent to res.status(404).send('Not Found')
res.sendStatus(500); // equivalent to res.status(500).send('Internal Server Error')
```

res.sendFile()

- **res.redirect([status,] path)** redirects to the URL derived from the specified path, with the specified HTTP status code
  - If not specified, status defaults to 302 “Found”

res.redirect()

- **res.redirect([status,] path)** redirects to the URL derived from the specified path, with the specified HTTP status code
  - If not specified, status defaults to 302 “Found”
- Redirects can be relative to the root of the host name
  - For example, if the application is on http://example.com/admin/post/new, the following would redirect to the URL http://example.com/admin:

```
res.redirect('/admin');
```

- Redirects can be relative to the current URL
  - For example, from http://example.com/blog/admin/ (notice the trailing slash), the following would redirect to the URL http://example.com/blog/admin/post/new

```
res.redirect('post/new');
```

- Redirects can be a fully-qualified URL for redirecting to a different site:

```
res.redirect('http://google.com');
```

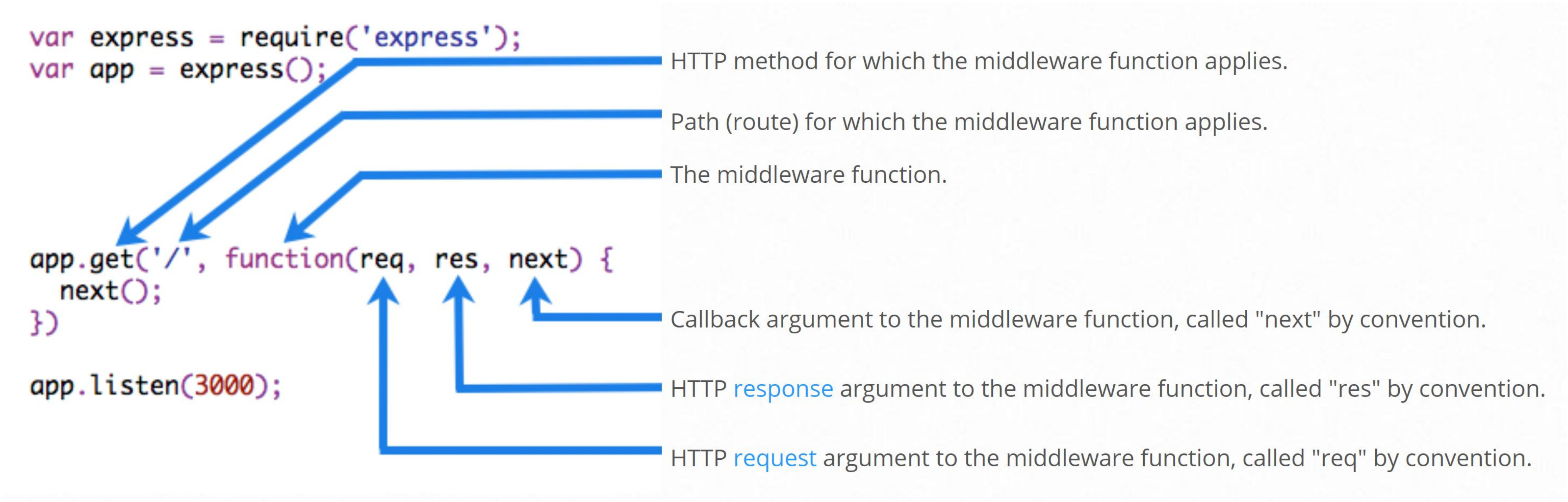
Exercise (5)

- Change the products server from previous exercise to return JSON instead of strings
- The server should support the following operations:
  - Get all products – return a JSON with all the products
  - Get product by id – return a JSON with the product’s details
  - Get product by name – return a JSON with the product’s details
  - Add a new product – unchanged
  - Delete a product by id – unchanged
- In case of an error (e.g., product id not found) send the appropriate HTTP status code
- Test the methods by using PostMan

Middleware

- An Express application is essentially a series of middleware function calls
- **Middleware** functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle (next)
- Middleware functions can perform the following tasks:
  - Execute any code
  - Make changes to the request and the response objects
  - End the request-response cycle
  - Call the next middleware function in the stack
- If the current middleware function doesn't end the request-response cycle, it must call next() to pass control to the next middleware function
  - Otherwise, the request will be left hanging

Middleware Function



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

app.get('/', function(req, res, next) {
  next();
})

app.listen(3000);
```

HTTP method for which the middleware function applies.

Path (route) for which the middleware function applies.

The middleware function.

Callback argument to the middleware function, called "next" by convention.

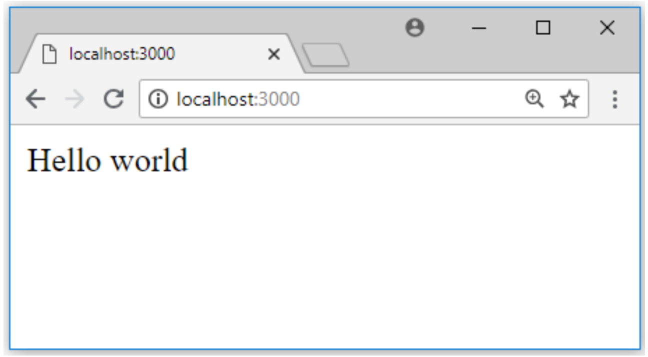
HTTP response argument to the middleware function, called "res" by convention.

HTTP request argument to the middleware function, called "req" by convention.

Application-Level Middleware

- Application-level middlewares are bound to the **app** object by using the app.use() and app.METHOD() functions, where METHOD is the HTTP method of the request that the middleware function handles (such as GET, PUT, or POST)
- The following example shows a middleware function that is executed every time the app receives a request:

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

let myLogger = function(req, res, next) {
    console.log('Logging');
    next();
}

app.use(myLogger);
app.get('/', function(req, res) {
    res.send('Hello world');
});
app.listen(3000);
```



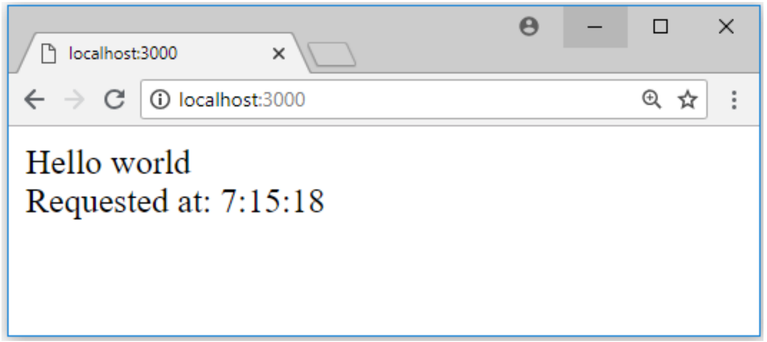
Hello world

```
C:\NodeJS\middlewares>nodemon app.js
[nodemon] 1.18.3
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node app.js`
Logging
```

Application-Level Middleware

- Next, we'll create a middleware function that adds a property called requestTime to the request object:

```
let requestTime = function (req, res, next) {
    time = new Date();
    req.requestTime = time.getHours() + ':' +
time.getMinutes() + ':' + time.getSeconds();
    next();
}
app.use(requestTime)

app.get('/', function(req, res) {
    let responseText = 'Hello world<br/>';
    responseText += 'Requested at: ' + req.requestTime;
    res.send(responseText);
});
```



Hello world
Requested at: 7:15:18

Configurable Middleware

- If you need your middleware to be configurable, export a function which accepts an options object or other parameters, which, then returns the middleware implementation based on the input parameters:

```
// my-middleware.js
module.exports = function(options) {
    return function(req, res, next) {
        // Implement the middleware function based on the options object
        next();
    }
}
```

- The middleware can now be used as shown below:

```
let mw = require('./my-middleware.js');
app.use(mw({ option1: '1', option2: '2' }));
```

Error Handling Middleware

- Define error-handling middleware functions in the same way as other middleware functions, except with four arguments instead of three:

```
app.use((err, req, res, next) => {
    console.error(err.stack)
    res.status(500).send('Something broke!')
});
```

- You should define the error-handling middleware last, after other app.use() and routes calls

Built-in Middleware

- Express has the following built-in middleware functions:
  - express.static serves static assets such as HTML files, images, and so on
  - express.json parses incoming requests with JSON payloads
    - NOTE: Available with Express 4.16.0+
  - express.urlencoded parses incoming requests with URL-encoded payloads
    - NOTE: Available with Express 4.16.0+

Static Files

- To serve static files such as images, CSS files, and JavaScript files, use the **express.static()** built-in middleware function:

```
express.static(root)
```

  - The root argument specifies the root directory from which to serve static assets
- For example, use the following code to serve static files from a folder named public:

```
app.use(express.static('public'));
```

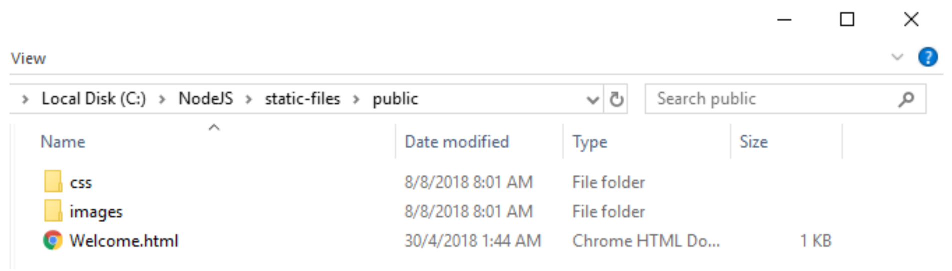
- Now, you can load the files that are in the public directory, e.g.
  - http://localhost:3000/welcome.html
  - http://localhost:3000/js/app.js
  - http://localhost:3000/images/kitten.jpg

Static Files - Using Absolute Path

- The path that you provide to the express.static() is relative to the directory from where you launch your node process
- If you run the express app from another directory, it's safer to use the absolute path of the directory that you want to serve:

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

app.use(express.static(path.join(__dirname, 'public')));
```

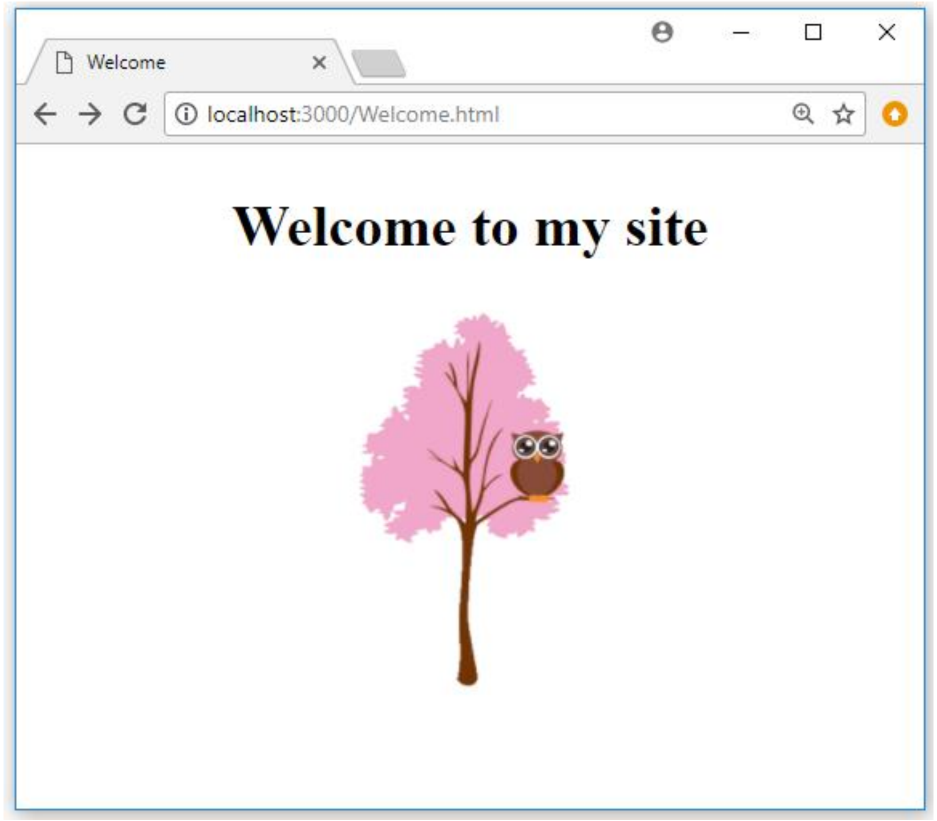
Static Files


View

Local Disk (C:) › NodeJS › static-files › public

Search public

Name	Date modified	Type	Size
css	8/8/2018 8:01 AM	File folder	
images	8/8/2018 8:01 AM	File folder	
Welcome.html	30/4/2018 1:44 AM	Chrome HTML Do...	1 KB


Welcome

localhost:3000/Welcome.html

Welcome to my site

Virtual Path Prefix

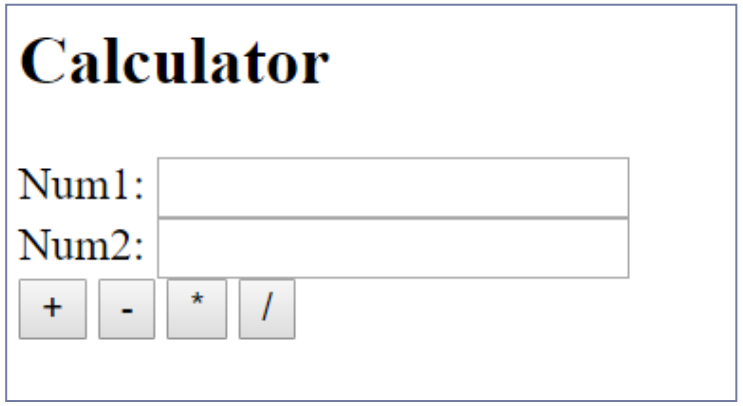
- To create a virtual path prefix (where the path doesn't actually exist in the file system) for files that are served by the express.static() function, specify a mount path for the static directory:

```
app.use('/static', express.static('public'));
```

- Now, you can load the files that are in the public directory from the /static path prefix, e.g.
  - http://localhost:3000/static/welcome.html
  - http://localhost:3000/static/js/app.js
  - http://localhost:3000/static/images/kitten.jpg

Exercise (6)

- Build an HTML page that displays a simple calculator, such as the following:



- The calculator should submit the exercise to your web server, passing the following params:
  - num1 – the first operand
  - num2 – the second operand
  - op – the operator
- The server should send back an HTML with the result of the computation

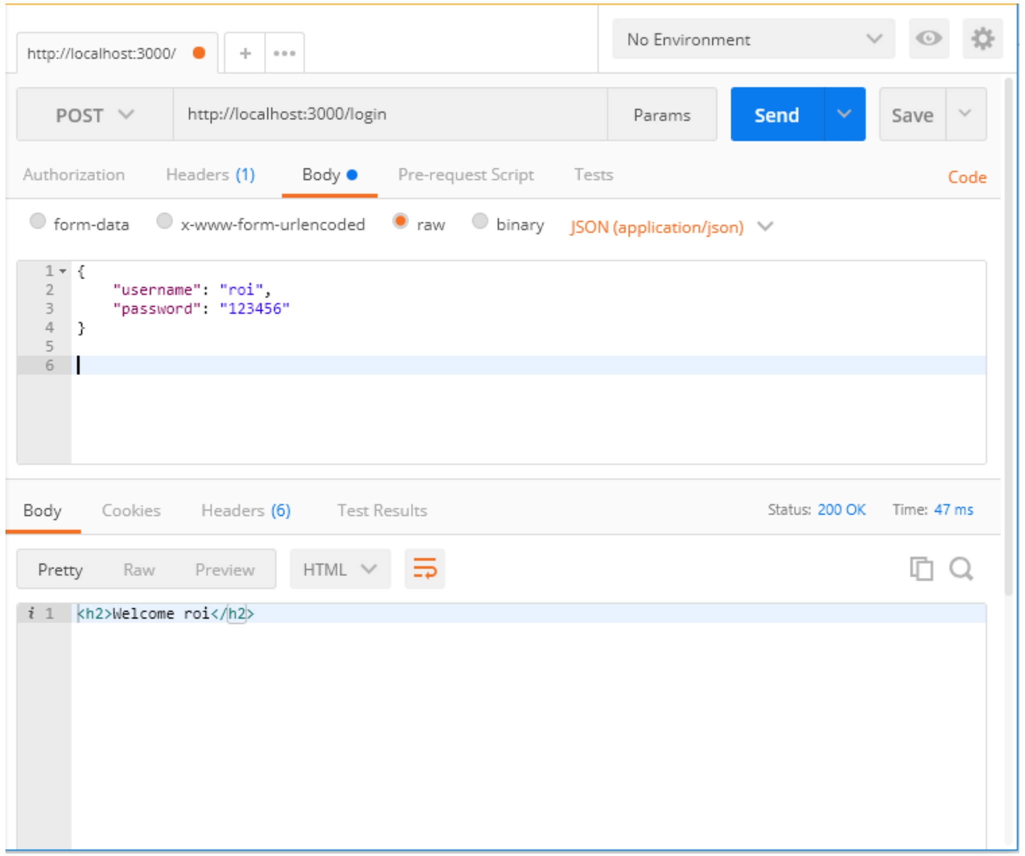
express.json()

- This is a built-in middleware function, which parses requests with JSON payloads
- A new body object containing the parsed data is populated on the request object after the middleware (i.e., req.body)
  - or an empty object ({}) if there was no body to parse, or an error occurred

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

app.use(express.json());

// POST /login gets JSON bodies
app.post('/login', (req, res) => {
    if (!req.body)
        return res.sendStatus(400);
    let user = req.body;
    res.send(`<h2>Welcome ${user.username}</h2>`);
});

app.listen(3000);
```

Sending JSON in PostMan

- Under Body choose raw option to send URL encoded form data
- Define all the parameters inside a JSON object



```
{
    "username": "roi",
    "password": "123456"
}
```

Status: 200 OK Time: 47 ms

```
<h2>Welcome roi</h2>
```

express.urlencoded()

- This is a middleware function, which parses requests with urlencoded payloads
- Urlencoded payloads use the same encoding as the one used in query string parameters (key-value pairs)
- When you submit a HTML form with method="POST", the Content-Type of the request is application/x-www-form-urlencoded by default, and it looks like this:

```
POST /some-path HTTP/1.1
Content-Type: application/x-www-form-urlencoded

foo=bar&name=John
```

  - Whereas a request with a JSON payload is typically submitted via AJAX call, and looks like this:

```
POST /some-path HTTP/1.1
Content-Type: application/json

{ "foo" : "bar", "name" : "John" }
```

express.urlencoded()

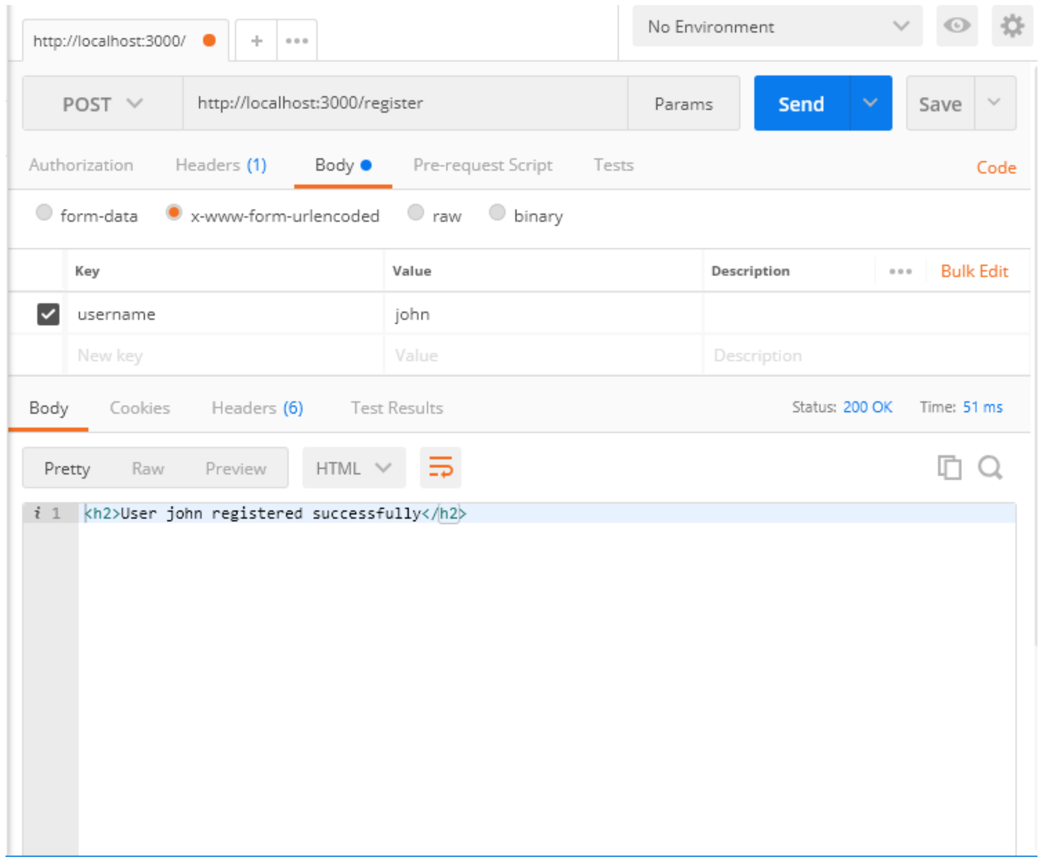
- Example for using the urlenocded body parser:

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

// POST /register gets urlencoded bodies
app.post('/register', (req, res) => {
    if (!req.body)
        return res.sendStatus(400);

    let user = req.body;
    res.send(`User ${user.username} registered successfully`);
});
```

  - The option extended allows to choose between parsing the URL-encoded data with the querystring library (when false) or the qs library (when true)
  - The “extended” syntax allows for rich objects and arrays to be encoded into the URL-encoded format, allowing for a JSON-like experience with URL-encoded

Sending Form Data in PostMan

- Under Body choose x-www-form-urlencoded option to send URL encoded form data
- Then enter the parameters as key/value pairs



http://localhost:3000/register

form-data [✓ x-www-form-urlencoded] raw binary

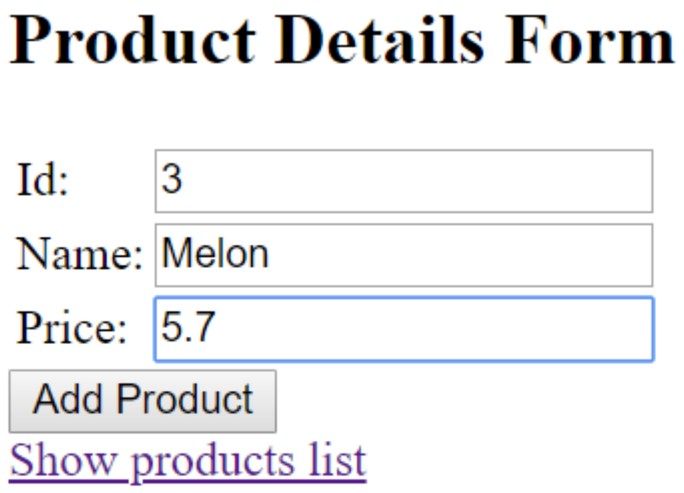
	Key	Value	Description
✓	username	john	
	New key	Value	Description

Status: 200 OK Time: 51 ms

```
<h2>User john registered successfully</h2>
```

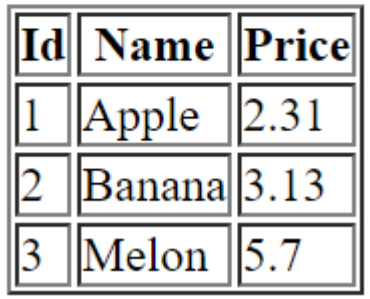
Exercise (7)

- Create a site for managing the products list of a store
- Each product has an id, name and price
- The site should contain two pages:
  - The first page displays a form for entering the details of a new product to be added to the store
  - Clicking the Add Product button sends the product details to the server using HTTP POST, and lets the user enter another product
  - The second page shows the list of products in the store (saved in the server's memory)



Product Details Form

Id:	3
Name:	Melon
Price:	5.7



Id	Name	Price
1	Apple	2.31
2	Banana	3.13
3	Melon	5.7

express.Router

- Use the **express.Router** class to create modular, mountable route handlers
- A Router instance is a complete middleware and routing system
  - For this reason, it is often referred to as a “mini-app”
- Router allows you to separate the route definitions from the main app.js file

- The following example creates a router as a module, loads a middleware function in it, defines some routes, and mounts the router module on a path in the main app
- Create a new package named express-router
- Run npm init
- Create a routes sub-folder inside your package folder

express.Router

- Create a file named users.js in the routes sub-folder and add the following code to it:

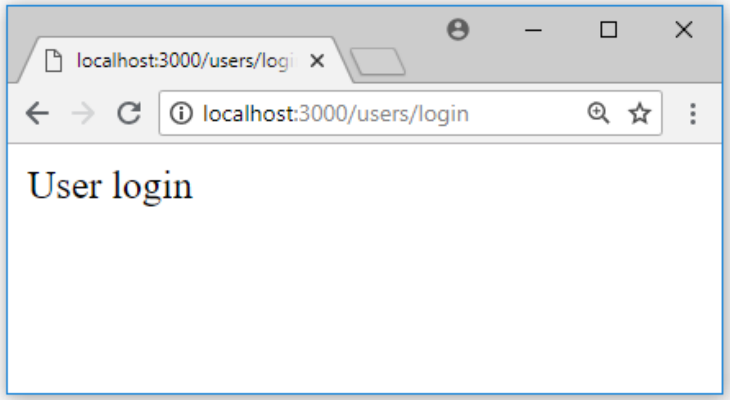
```
const express = require('express');
const router = express.Router();

// middleware that is specific to this router
router.use(function (req, res, next) {
    console.log('Time: ', Date.now());
    next();
});

// define the login route
router.get('/login', function(req, res) {
    res.send('User login');
});

// define the register route
router.get('/register', function(req, res) {
    res.send('User registration');
});

module.exports = router;
```

express.Router

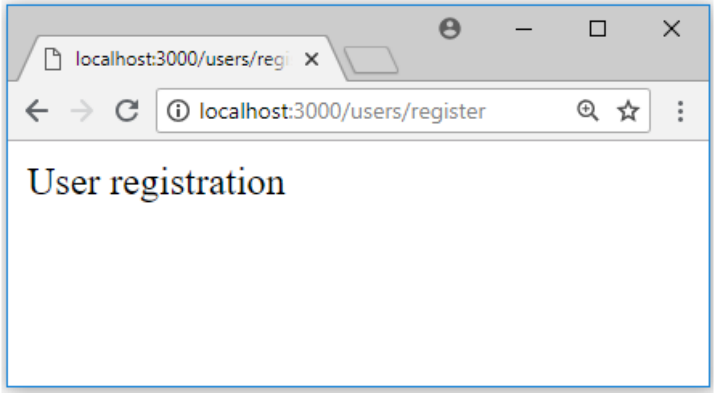
- Then, load the router module in the app:

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

const users = require('./routes/users');
app.use('/users', users);

app.listen(3000);
```

- The app will now be able to handle requests to /users/login and /users/register, as well as call the middleware function that is specific to the route


localhost:3000/users/login

User login


localhost:3000/users/register

User registration

Third-Party Middleware

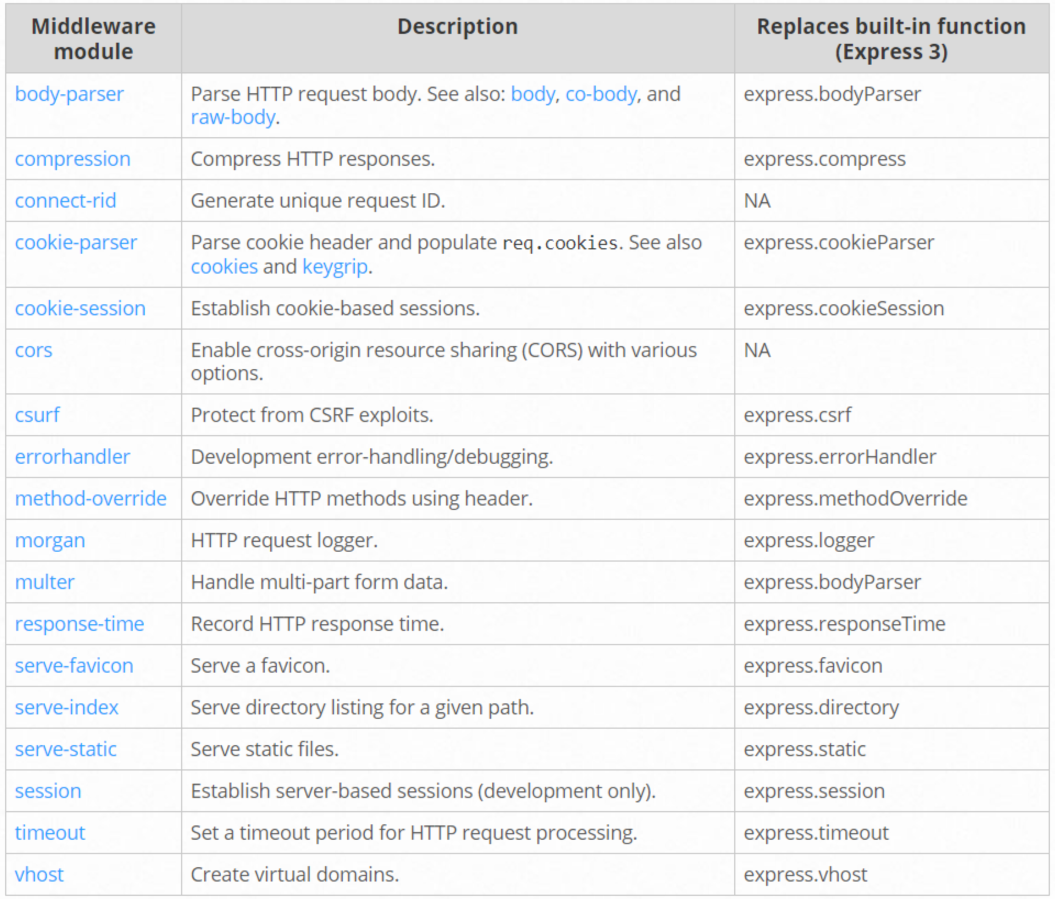
- Use third-party middleware to add functionality to your Express apps
- Install the Node.js module for the required functionality, then load it in your app at the application level or at the router level
- For exmaple, to work with cookies, you can install and load the cookie-parser middleware

```
$ npm install cookie-parser
```

```
const express = require('express');
const app = express();
const cookieParser = require('cookie-parser');

// load the cookie-parsing middleware
app.use(cookieParser());
```

Third-Party Middleware

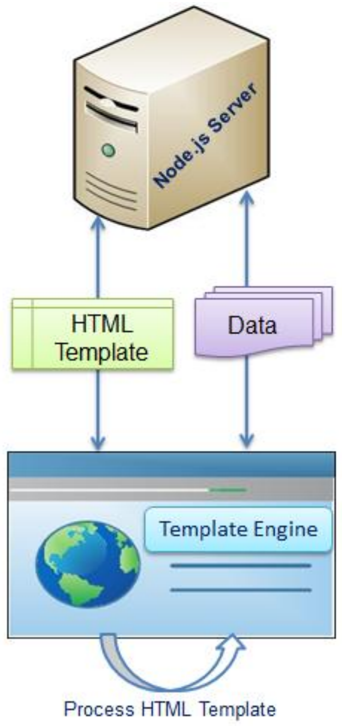
- A partial list of third-party middleware functions that are commonly used with Express:



Middleware module	Description	Replaces built-in function (Express 3)
body-parser	Parse HTTP request body. See also: body, co-body, and raw-body.	express.bodyParser
compression	Compress HTTP responses.	express.compress
connect-rid	Generate unique request ID.	NA
cookie-parser	Parse cookie header and populate `req.cookies`. See also cookies and keygrip.	express.cookieParser
cookie-session	Establish cookie-based sessions.	express.cookieSession
cors	Enable cross-origin resource sharing (CORS) with various options.	NA
csurf	Protect from CSRF exploits.	express.csrf
errorhandler	Development error-handling/debugging.	express.errorHandler
method-override	Override HTTP methods using header.	express.methodOverride
morgan	HTTP request logger.	express.logger
multer	Handle multi-part form data.	express.bodyParser
response-time	Record HTTP response time.	express.responseTime
serve-favicon	Serve a favicon.	express.favicon
serve-index	Serve directory listing for a given path.	express.directory
serve-static	Serve static files.	express.static
session	Establish server-based sessions (development only).	express.session
timeout	Set a timeout period for HTTP request processing.	express.timeout
vhost	Create virtual domains.	express.vhost

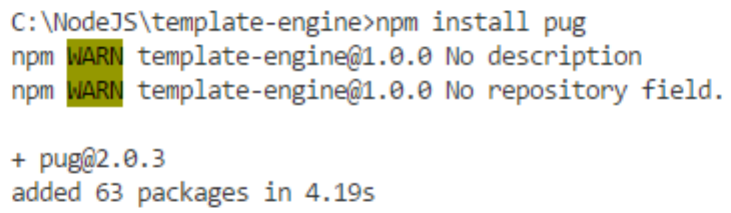
Template Engines

- Template engine helps us create an HTML template with minimal code
- At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client



Pug

- There are plenty of template engines to use with Node.js
- Some popular template engines that work with Express are Pug (formerly known as Jade), Mustache, and EJS
- To install a template engine, you need to install the corresponding npm package
- For example, to install Pug:



```
C:\NodeJS\template-engine>npm install pug
npm WARN template-engine@1.0.0 No description
npm WARN template-engine@1.0.0 No repository field.

+ pug@2.0.3
added 63 packages in 4.19s
```

Template Page

- Create a directory views, the directory where the template files are located
- Create a Pug template file named index.pug in the views directory, with the following content:

```
html
    head
        title= title
    body
        h1= message
```

- The equals sign (=) is used to evaluate JavaScript expressions and output the result in the HTML code

Using Template Engines with Express

- To render template files, first set the following application setting properties:

```
app.set('views', './views');
app.set('view engine', 'pug');
```

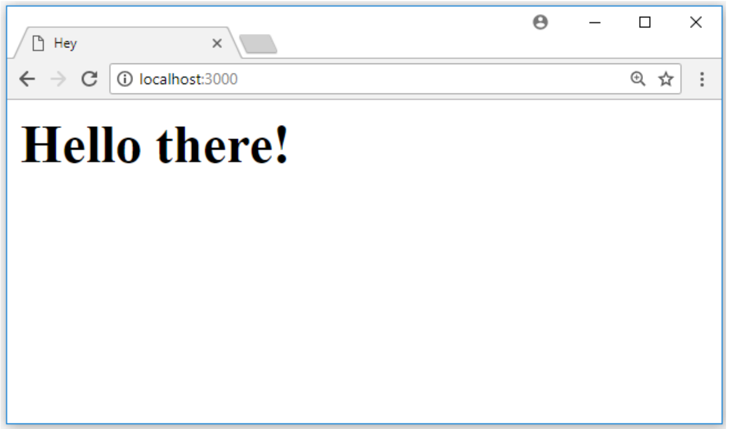
  - **views** is the directory where the template files are located
    - This defaults to the views directory in the application root directory
  - **view engine** is the name of the template engine to use
- Then create a route to render index.pug
- Use **res.render(view, [locals])** to return the rendered HTML of the view
  - It accepts an optional parameter that is an object containing local variables for the view

```
app.get('/', function (req, res) {
    res.render('index', { title: 'Hey', message: 'Hello there!' });
});
```

Using Template Engines with Express

- The final app.js looks like this:

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

app.set('view engine', 'pug');

app.get('/', function (req, res) {
    res.render('index', { title: 'Hey', message: 'Hello there!' });
});

app.listen(3000);
```



Hello there!

HTML Tags

- Text at the start of a line (or after only white space) represents an HTML tag
- Everything after the tag and one space will be the text contents of that tag
- Indented tags are nested, creating the tree structure of HTML

```
ul
    li Item A
    li Item B
    li Item C
```

→

```
<ul>
    <li>Item A</li>
    <li>Item B</li>
    <li>Item C</li>
</ul>
```

- Pug also knows which elements are self-closing:

```
img
```

→

```
<img/>
```

- To save space, Pug provides an inline syntax for nested tags:

```
a: img
```

→

```
<a><img/></a>
```

Tags with Blocks

- Often you might want large blocks of text within a tag
- A good example is writing JavaScript and CSS code in the script and style tags
- To do this, just add a . right after the tag name and indent the text contents of the tag one level:

```
script.
    let usingPug = true;
    if (usingPug)
        console.log('you are awesome');
    else
        console.log('use pug');
```

```
<script>
    let usingPug = true;
    if (usingPug)
        console.log('you are awesome');
    else
        console.log('use pug');
</script>
```

JavaScript Code

- Pug allows you to write inline JavaScript code in your templates
- Lines that start with - contain JavaScript code, which is not rendered to the output

```
- for (let i = 0; i < 3; i++)
    li item
```

$\Rightarrow$

```
<li>item</li>
<li>item</li>
<li>item</li>
```

- Code between #{ and } is evaluated, escaped, and the result rendered into the output

```
- let title = "I, Robot";
- let author = "Issac Asimov";

p The book #{title} was written by
#{author}.
```

$\Rightarrow$

```
<p>The book I, Robot was written by Issac
Asimov.</p>
```

```
- let num1 = 5;
- let num2 = 8;

p num1 * num2 = #{num1 * num2}
```

$\Rightarrow$

```
<p>num1 * num2 = 40</p>
```

Tag Attributes

- Tag attributes look similar to HTML (with optional commas), but their values are just regular JavaScript

```
a(href="http://www.google.com") Google
```

```
<a href="http://www.google.com">Google</a>
```

```
a(class="button",
href="http://www.google.com") Google
```

```
<a class="button"
href="http://www.google.com">Google</a>
```

- Normal JavaScript expressions work fine, too:

```
- let url = "http://www.example.com";
a(href=url) Example
```

```
<a href="http://www.example.com">Example</a>
```

```
- let authenticated = true
div(class=authenticated ? 'authed' : 'anon')
```

```
<div class="authed"></div>
```

Style, Class and Id Attributes

- The style attribute can be a string, or an object, which is handy when styles are generated by JavaScript:

```
a(style={color: 'red', background: 'green'})
```
➡
```
<a style="color:red;background:green;"></a>
```

- Classes may also be defined using a .classname syntax:

```
a.button
```
➡
```
<a class="button"></a>
```

- IDs may be defined using a #idname syntax:

```
a#main-link
```
➡
```
<a id="main-link"></a>
```

Conditions

- Like in JavaScript, you can use if statements for checking conditions
  - The parentheses around the logical expression are optional
  - You may also omit the leading -

```
- let num = 15

if num > 10
    h2.green num is big
else
    h2.red num is small
```

```
<h2 class="green">num is big</h2>
```

Iteration

- Pug supports two primary methods of iteration: each and while

```
ul
    each val in [1, 2, 3, 4, 5]
        li= val
```

```
<ul>
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
</ul>
```

```
- let n = 0;
ul
    while n < 4
        li= n++
```

```
<ul>
    <li>0</li>
    <li>1</li>
    <li>2</li>
    <li>3</li>
</ul>
```

- You can also use for as an alias for each

Comments

- JavaScript comments produce HTML comments in the rendered page

```
// just some paragraphs
p First paragraph
p Second paragraph
```

```
<!-- just some paragraphs-->
<p>First paragraph</p>
<p>Second paragraph</p>
```

- Comments that start with a hyphen (-) are only for commenting on the Pug code itself, and *do not* appear in the rendered HTML

```
//- this comment will not appear in the output
p First paragraph
p Second paragraph
```

```
<p>First paragraph</p>
<p>Second paragraph</p>
```

- Block comments work too

```
body
  //-
    Comments for your template writers.
    Use as much text as you want.
```

Includes

- Includes allow you to insert the contents of one Pug file into another
- This is useful for sharing some HTML code between different pages

```
//- home.pug
doctype html
html
    include includes/head.pug
    body
        h1 My Site
        p Welcome to my amazing site.
        include includes/footer.pug
```

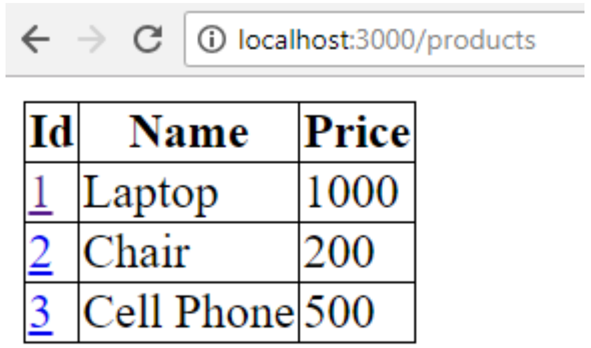
```
//- includes/head.pug
head
    title My Site
    script(src='/scripts/jquery.js')
    script(src='/scripts/app.js')
```

```
//- includes/foot.pug
footer#footer
    p Copyright (c) Roi Yehoshua
```

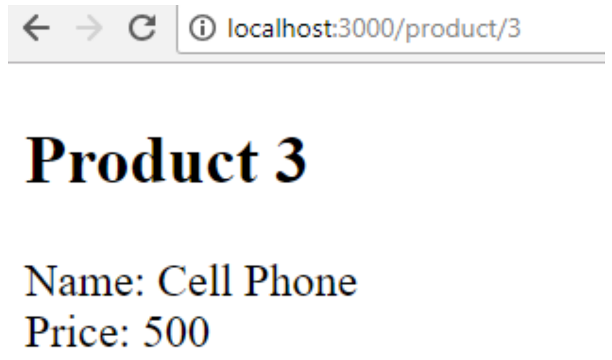
```
<!DOCTYPE html>
<html>
<head>
    <title>My Site</title>
    <script src="/scripts/jquery.js"></script>
    <script src="/scripts/app.js"></script>
</head>
<body>
    <h1>My Site</h1>
    <p>Welcome to my amazing site.</p>
    <footer id="footer">
        <p>Copyright (c) Roi Yehoshua</p>
    </footer>
</body>
</html>
```

Exercise (8)

- Continue from the previous exercise
- Convert the products list into a template page (instead of building its HTML in the code)
- Add another template page that displays the details of a selected product
- In the products table, add a link for each product id, that will lead to the product's details page



Id	Name	Price
1	Laptop	1000
2	Chair	200
3	Cell Phone	500



Product 3

Name: Cell Phone
Price: 500