Setting up a Server with Node.js and Express

- There are many different frameworks for deploying web application servers, e.g.:
  - Spring (a Java framework)
  - Django (a Python framework)
  - NGINX
  - Apache
  - etc.

- In this course, we will use Node.js and a framework called Express to write servers in JavaScript.

- To that end, we will spend some time reviewing some of the basics of Node.js before we dive into building a server with Express.
  - Note that we will not spend time reviewing the JavaScript language in this course. It will be up to you to make sure you're comfortable with JS.

Node.js basics

- Node.js is a tool that is often thought of as a server framework. Node is more general than that, though: it is a tool that simply allows us to run JS outside the browser.

- Indeed, we can create any command-line tool using Node.js. For example, say we had a file named `hello.js` that contained a simple "hello world" program:

```
console.log("Hello world.");
```

- We could run that file from the command line using Node like this:

```
node hello.js
```

- Within any program run using Node, a global object called `process` is available to provide information about the running process. Some of the properties contained in this object are:

    - `process.argv` – an array containing command line arguments passed to the running process
    - `process.env` – an object containing the environment variables available to the running process
    - `process.stdout`, `process.stderr`, and `process.stdin` – objects representing the process's standard output, error, and input streams
    - `process.exit()` – a method to exit the running process
    - Lots of others: https://nodejs.org/api/process.html

- For example, if we had set the environment variable `PORT` before running a program with Node, we could access the value of that environment variable within the program using `process.env` as follows:

```
console.log("PORT:", process.env.PORT);
```

- A couple other important values available in all Node processes are:
    - `__dirname` – a string containing the path of the directory in which the currently executing code resides
    - `__filename` – a string containing the path of the file in which the currently executing code resides
    - Both `__dirname` and `__filename` are actually local to each individual module (more on modules below), not global like `process`.

Node.js modules

- Beyond some essentials, like `console`, `process`, `__dirname`, and `__filename`, which we saw above, there isn't much functionality available in the global scope in a Node application.
    - You can see all the globals here: https://nodejs.org/api/globals.html.

- Node has many other pieces of functionality available as importable ***modules***, which we can access using the global `require()` function.

- For example, we can import `fs`, Node's built-in module for working with the file system, like this:

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

- Then, we can use the methods of `fs`, e.g.:

```
fs.readFile(...);
```

- There are many important built-in modules available in Node:
  - `fs` – file system operations: https://nodejs.org/api/fs.html
  - `path` – utilities for working with file and directory paths: https://nodejs.org/api/path.html
  - `url` – utilities for working with URLs: https://nodejs.org/api/url.html
  - `http` – HTTP perver and client: https://nodejs.org/api/http.html
  - Lots of others: https://nodejs.org/api/

- We can also write our own modules, which can be imported using their path.

- Module functionality is specified (exported) using the `module.exports` property.

- For example, if I wanted to write a simple module to compute the circumference of a circle given its radius, I could write a file `circumference.js`, like this:

```
module.exports = function (r) {
     return 2 * Math.PI * r;
};
```

- Then if my module is stored in the same directory as my application file, I could use it like this:

```
var circumference = require('./circumference');
console.log("The circumference of a circle with
     radius 4 is:", circumference(4));
```

  - The path specified in `require()` is usually relative to the path of the current file, e.g.:
    - `require('./circumference')`
    - `require('./lib/circumference')`

Third-party modules

- Node's built-in modules provide some useful, but limited, functionality. Luckily, Node is set up so you can easily use third-party modules, and there are a huge number of these available.

- In particular, you can find lots of third-party Node modules at the website for npm, Node's package manager (which we'll talk more about in a second): https://www.npmjs.com/.

- For example, if I wanted to create ASCII art in a Node program, I could just search for "ASCII art" on npm. The first search result is called "figlet". If I wanted to use figlet, my first step would be to download and install it using npm in the terminal:

  ```
  npm install figlet
  ```

  - Note that npm will give us warnings about not being able to find a file called `package.json`. We can ignore these for now.

- After running the above command, you will see a directory called `node_modules/` created in the current directory. This is the directory where Node keeps third-party modules by default. If you list that directory, you will see a subdirectory named `figlet/`, which is where figlet is installed.

- Now, I can import and use the figlet module using `require()`, just like with a Node build-in module or a module I wrote myself:

  ```
  var figlet = require('figlet');
  figlet('Hello world!', function (err, data) {
       if (!err) {
            console.log(data);
       }
  });
  ```

Packages and npm

- The Node package manager, npm, is actually many things:
    - A package registry, which is a big database with info about all publicly-shared packages.
    - A website, https://www.npmjs.com/, that helps find packages in the package registry.
    - A command-line tool that allows developers to work with and share packages.

- A Node package is simply a piece of JS code that is bundled up in such a way as to make it portable.

- A piece of code could be made into a package so that other developers can use it, so a single developer can use it in multiple places, or simply to gain some of the benefits of packaging code. These include:
    - Allowing you to specify your code's dependencies (with versions) so that they're very easy to install again, perhaps on a different machine than the one where you developed the code.
    - Allowing you to specify commands to perform common tasks associated with your code, e.g.:
        - Building your code into a runnable representation
        - Running setup steps
        - Running code in production or development mode
        - Running tests on your code
        - Etc.
    - Allowing you to make note of important info about your code, e.g. info about you, the author, info about the code's version control repository, info about a website associated with the code, etc.

- In general, packaging your code makes it more reusable and easier to share with other developers.

- To make your code into a package, you'd just need to create a file called `package.json` that provides information about your code.
    - As the extension implies, the `package.json` file is written in JSON syntax. You can read more about JSON here: https://en.wikipedia.org/wiki/JSON.

- The `package.json` file specifies metadata about your package. At a minimum, it contains a name and a version for your package:

```
{
      "name": "my-code",
      "version": "1.0.0"
}
```

- Instead of creating `package.json` from scratch, you can use npm's `init` command, which will walk you through several questions about your package and use your answers to construct a `package.json`:

```
npm init
```

- With `package.json` in place, we can begin to take advantages of some of the benefits of packaging code.

Managing dependencies with npm and `package.json`

- One of the prime benefits of packaging code is automated dependency management with npm and `package.json`.

- In particular, there are two fields you can use in `package.json` to specify dependencies:
    - `dependencies` – packages specified here are ones that are required by your application to run in production.
    - `devDependencies` – packages specified here are ones that are only needed for development and testing.

- You can manually specify packages in either `dependencies` or `devDependencies`. Each entry in either of these fields is a `"name": "version"` pair:

```
      "dependency-name": "^4.0.3"
```

    - The version is specified as a ***semver*** expression. In particular, the combination of the carat (`^`) and the version `4.0.3` indicate that any version of this package with major version 4 (i.e. version 4.x) can be used.
        - You can see more about semantic versioning here: https://docs.npmjs.com/getting-started/semantic-versioning.

- Here's an example of a package with dependencies:

```
{
      "name": "my-package",
      "version": "1.0.0",
      "dependencies": {
            "my-dependency": "^2.0.0"
      },
      "devDependencies": {
            "my-testing-dependency": "^3.2.0"
      }
}
```

- Luckily, you don't have to manually list out all of your dependencies. Instead, you can add the `--save` or `--save-dev` option when making an `npm install` command to automatically save the installed package to one of the dependency fields in `package.json`, e.g.:

```
npm install --save my-dependency
npm install --save-dev my-testing-dependency
```

- Once your packages are specified in `package.json`, you can install all of them by using npm's `install` command without arguments from the directory where `package.json` lives:

```
npm install
```

- This is most useful when you want to start work on your code on a different machine. For example, you can copy all of your code to the new machine, excluding the `node_modules/` directory, and then simply run `npm install` to make sure all of the dependencies are installed.

- This works perfectly, e.g., with git. You can simply add `node_modules/` to your `.gitignore` file (which is best practice anyway, since the `node_modules/` directory can consume a large amount of disk space and can also contain some machine-dependent code) and then simply clone your code (which includes `package.json`) to start working with it on a new machine.

Adding scripts to `package.json`

- Another benefit of packaging code is that it allows you to write out commonly-used commands as ***scripts*** in your `package.json` file.

- In particular, one of the top-level fields in the `package.json` object is named `"scripts"`. The value of this field is itself an object that contains terminal commands with labels assigned to them. You can run a given terminal command from the `"scripts"` field using `npm run`.

- For example if you were writing a server and wanted to use nodemon to run your server in development mode with automatic restarts when you changed your source files, you could install nodemon using npm (e.g. `npm install --save-dev nodemon`), and then add the following to the `"scripts"` field in `package.json`:

```
{
      ...
      "scripts": {
            "dev": "NODE_ENV=development nodemon server.js"
      },
      ...
}
```

- Then, to run your development-mode server, you could simply run this command:

```
npm run dev
```

Serving with Express

- ***Express*** is a widely-used framework for creating web applications and APIs with Node.js. We will use Express in this course to build our API servers.

- Express centered around the concept of ***middleware***. A middleware function is simply a function that participates in the HTTP request-response cycle.

- In particular, every Express-based server is specified as a sequence of middleware functions. This sequence of functions is executed, in order, every time a request comes into the server.

- Each middleware function is passed three arguments:
    - An object representing a single HTTP request (`req`).
    - An object representing the corresponding HTTP response (`res`).
    - A reference to the next middleware function in the in the sequence (`next`).

- With access to these three things, a middleware function can do the following:
    - Execute any code it needs to in order to help process the response.
        - E.g. fetch data from a database, perform a computation based on the request object, log info about the request to the console, etc.
    - Update/modify the request object and/or the response object.
    - Terminate the request-response cycle by sending the response back to the client.
    - Call the next middleware function.

- We will implement API servers by writing middleware functions that specify what the server should do in response to every HTTP request it receives.

- We'll look at a few different ways to incorporate middleware into an Express app in order to provide some basic HTTP server functionality. First, however, let's look at how we simply get an Express server up and running.
- To be able to use Express, we first need to install it using npm. Note that express is a runtime dependency, so we save it there in our `package.json`:

```
npm install --save express
```

- Once we have Express installed, we can start writing our server. The first thing we'll do is require the Express module and create a new Express application object, which will serve as the representation of our Express server:

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

- Once we have our application object `app`, we can start the server listening for requests. To do this, we specify two things, one of which is optional:
    - The number of a TCP port on which to listen for requests.
    - An optional callback function to be called when the server is successfully started.

- Here's what it might look like to start our server listening:

```
app.listen(port, function () {
     console.log("== Server is listening on port", port);
});
```

- Of course, at this point, we haven't told our server what to do when it actually gets a request, so it won't do anything right now. It won't even send a response back for any requests it gets. Let's look at how to add functionality to the server to make it respond to requests. We'll do this by adding middleware functions.

Routing middleware

- ***Routing*** is the process of determining how to respond to an HTTP request to a specific URL path with a specific HTTP method. This is a key component of a RESTful API server, since, as we've seen, restful APIs are designed around request path/HTTP method pairs.

- Going forward, we'll refer to a specific path/method pair as a ***route***.

- Express allows one or more middleware functions to be assigned to any given route. A middleware function can be assigned to a route as follows:

```
app.METHOD(PATH, MIDDLEWARE);
```

- The values in this call are as follows:
    - `METHOD` – This indicates the specific HTTP method associated with the route. For example we use `get` for GET requests, `post` for POST requests, `put` for PUT requests, `delete` for DELETE requests, etc.
    - `PATH` – This indicates the request path associated with the route.
    - `MIDDLEWARE`– This is the middleware function being assigned to the route.

- For example, if we wanted to specify a middleware function to handle HTTP GET requests on the root path `/`, we could do that using `app.get()`:

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

- Note again here that the middleware function we're assigning will receive three arguments:
  - `req` – an object representing the HTTP request
  - `res` – an object representing the corresponding HTTP response
  - `next` – a function that can be called to invoke the next middleware function

- Importantly, every middleware function must end *either* by calling `next` or by terminating the request-response cycle by sending the response back to the client.

- For example, we could implement our route above to simply use the response object's `status()` and `send()` methods to send a plaintext response back to the client with a 200 status code:

```
app.get('/', function (req, res, next) {
     res.status(200).send("Hello world");
});
```

  - Note that `res.send()` automatically sets response headers like the `Content-Type` and `Content-Length` based on the argument passed. Here, since we pass a string argument, the `Content-Type` header will be set to `text/html`. If the argument was an object or an array, the content type `application/json` would have been used instead. We'll see later how to more explicitly send JSON content.

Routes matching multiple paths

- Note that the middleware functions of routes specified like we did above will only be called if the specified HTTP method and request path exactly match those specified in the incoming HTTP request. If our server gets a request for a method/path combination that's not specified in our routes, no routing middleware function will be called, and thus no response will be sent for that request.

- This seems to place a heavy burden on us as server developers. How can we possibly write an individual route to handle each of the many individual resources associated with an API? Moreover, what can we do to handle erroneous URLs? We can't possibly anticipate all of the typos our API clients could make.

- Luckily, we don't have to do these things, because Express provides us with a few ways to write a single route that handles requests for many different paths.

- The first approach we'll look at to handling multiple paths with a single route in Express is the use of paths containing ***string patterns***, which allow us to match multiple request paths that all take a certain specified form.
    - In addition to string patterns, Express also allows the use of regular expressions in a route's path specification. Though they are extremely useful, a discussion of regular expressions is beyond the scope of this course. If you're interested to learn more about how regular expressions can be used in JS, check out the MDN page on regular expressions.

- The most familiar string pattern is the asterisk (`*`), which matches any number of any character.
    - For example the string pattern `c*t` matches the following strings:
        - cat
        - cot
        - cut
        - c123t
        - cANY_NUMBER_OF_ANY_CHARACTERt
        - etc.

- Specifying a route path that contains a string pattern allows the route to handle requests for many different paths. For example, a common practice when is to include a catch-all route by specifying the path `'*'` and using this route to respond with a 404 (not found) status and some content that lets the client know the resource they requested doesn't exist, e.g.:

```
app.use('*', function (req, res) {
     res.status(404).send({
          err: "The requested resource doesn't exist"
     });
});
```

    - Note that we're specifying middleware here using `app.use()`. This results in the middleware being called for each request to the specified path *regardless* of the associated HTTP method.

- Importantly, a catch-all route like this must be specified *after* any other routes we have specified. This is because Express tries to match requests against routes in the order in which the routes were specified. A route with the path `'*'` will match any request, so any other routes specified below that one would never be matched by Express.

Parameterized routes

- Another way we can use a single route to respond to multiple request paths is to include parameters in our route path. ***Route parameters*** allow us to use a single route to match multiple request paths that follow the same pattern, and they allow us to capture values at a specified position in the request path.

- When implementing an API server, route parameters are particularly useful for representing the item IDs in request paths of the form `/collection/itemID`.

- For example, in the API we designed earlier in the course for an Airbnb-like application, we created route the client could use to get information about a specific lodging:

  ```
  GET /lodgings/{lodgingID}
  ```

- For situations like this, Express allows us to create ***parameterized routes*** that contain placeholders to allow us to capture the parameterized portion of the URL.

- To create a parameterized route, we must include a colon-prefixed identifier (e.g. `:anIdentifier`) at each location in the path the corresponds to a route parameter like `lodgingID` above.

- For example, we could assign middleware to the route above like this:

  ```
  app.get('/lodgings/:lodgingID', function (req, res, next) {
    ...
  });
  ```

- When a client makes a request for a specific path that follows this pattern, the values specified by the client for each route parameter is available within the middleware function via the `req.params` object. Specifically `req.params` will be an object that contains one entry for each colon-prefixed identifier specified in the route path, where the entry's key is the identifier itself, and the entry's value is the particular value specified at the corresponding location in the request path.

- For example, if a client made a GET request to our server to the path `/lodgings/12345`, the `req.params` object would look like this inside our route's middleware function:

  ```
  { lodgingID: "12345" }
  ```

- In turn, we could use this data to fetch the needed information about the requested resource, e.g. by making a query to a database.

Non-routing middleware

- Often, we'll want our server to do something for each request that's not directly related to routing for that request. For example, we might simply want to log information about the request, which we can later aggregate with other request information to compute statistics about how people are using our API.

- To accomplish tasks like this, Express allows us to specify middleware functions that are not tied to a specific HTTP method or request path. We do this by calling the `use()` method on the application object:

  ```
  app.use(function (req, res, next) {...});
  ```

- Middleware functions registered this way (along with routing middleware functions) will be called for every request in the order in which they were registered.

- For example, if we wanted to accomplish our logging task, we could register a middleware function to do that:

```
app.use(function (req, res, next) {
     console.log("== Request received");
     console.log("  - Method:", req.method);
     console.log("  - URL:", req.url);
     next();
});
```

- Again, note that if a middleware function registered this way does not terminate the request-response cycle, e.g. by calling `req.send()`, it *must* call the next middleware function in the sequence using `next()`, as is done in the logging example above.
    - If a middleware function doesn't do either of these things, the server process will hang when it reaches that middleware function. No response will be sent, and no further middleware functions will be called.

- Note that middleware functions do not need to be anonymous, nor do they need to be be defined in the same file as the Express application. For example, we could define our logging function in an external file named `logger.js`:

```
module.exports = function (req, res, next) {
     console.log("== Request received");
     console.log("  - Method:", req.method);
     console.log("  - URL:", req.url);
     next();
};
```

- And then we could import that function in our server script and register it with Express:

```
var logger = require('./logger');
...
app.use(logger);
```

- There are also many third-party modules available on NPM that already implement useful middleware functions, and these can also be used in the same way as our logger just above.

Implementing an API using Express

- To implement an API using Express, we simply need to specify routing middleware for each API endpoint.

- Let's walk through a simple example of this. In particular, we can assume we have a basic Express server set up as described above, and we'll implement some of the API endpoints for the Airbnb-like application we designed earlier in the course.

- We'll specifically implement the API endpoints associated with lodgings. As a refresher, here's a list of the API endpoints we designed related to lodgings:
  - `GET /lodgings` – fetch a list of all lodgings
  - `POST /lodgings` – create a new lodging
  - `GET /lodgings/{lodgingID}` – fetch data about a single lodging
  - `PUT /lodgings/{lodgingID}` – modify a single lodging
  - `DELETE /lodgings/{lodgingID}` – delete a single lodging

- Since we haven't talked about storing data yet, for now, we'll represent our collection of lodgings as an in-memory array:

```
var lodgings = [
     {
          name: "Nice place",
          description: "A nice place to stay",
          ...
     },
     ...
];
```

`GET /lodgings`

- We'll implement the API endpoints in the order in which they're listed above, starting with `GET /lodgings`. We can start by just writing the skeleton for this endpoint:

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

- This endpoint is one for which we'll likely want to paginate the response, since our entire collection of lodgings could be quite large. The typical way to implement pagination is to allow the client to pass a query string parameter specifying the specific page they'd like to fetch, e.g.:

```
GET /lodgings?page=2
```

- If the client doesn't specify a page, we simply return the first one. The response will contain pagination information and links to relevant pages, e.g.:

```
{
     "pageNumber": 1,
     "totalPages": 127,
     "pageSize": 10,
     "totalCount": 1264,
     "lodgings": [ ... ],
     "links": {
          "nextPage": "/lodgings?page=2",
          "lastPage": "/lodgings?page=127"
     }
}
```

- Since we're not using a database yet, we'll have to do some extra work to perform the pagination. Our first step, though, will be to simply extract the requested page number from the URL's query parameters.

- Express actually provides automatic query string parsing for us, storing each specified query parameter in an object called `req.query`. For example, if the client specifies the query parameter `page`, its value will be available at `req.query.page`.

- We'll want page to be an integer value, so we'll parse it as such, using the value 1 if the page doesn't parse as an integer (including if it isn't specified):

```
var page = parseInt(req.params.page) || 1;
```

- Next, we'll want to make sure the specified page is between 1 and the last page. For now, we'll hard-code 10 lodgings per page:

```
var numPerPage = 10;
var lastPage = Math.ceil(lodgings.length / numPerPage);
page = page < 1 ? 1 : page;
page = page > lastPage : lastPage page;
```

- With the page number figured out, we can compute the indices of the start and end of the page, and we can grab the sub-array of lodgings we'll return:

```
var start = (page - 1) * numPerPage;
var end = start + numPerPage;
var pageLodgings = lodgings.slice(start, end);
```

- Finally, we'll start to assemble our response. First, we'll build an object of HATEOAS links to pages surrounding the current one:

```
var links = {};
if (page < lastPage) {
     links.nextPage = '/lodgings?page=' + (page + 1);
     links.lastPage = '/lodgings?page=' + lastPage;
}
if (page > 1) {
     links.prevPage = '/lodgings?page=' + (page - 1);
     links.firstPage = '/lodgings?page=1';
}
```

- With the links assembled, we can send the response with a 200 status code. To explicitly return JSON data in a response, we can use the `res.json()` method:

```
res.status(200).json({
     pageNumber: page,
     totalPages: lastPage,
     pageSize: numPerPage,
     totalCount: lodgings.length,
     lodgings: pageLodgings,
     links: links
});
```

POST /lodgings

- Let's next implement `POST /lodgings`, which is the endpoint the client will use to create a new lodging We'll have to do some work here, too.

- Lodging information will be sent by the client to this API endpoint within the body of a POST request. Earlier in the course, we decided that request bodies for this API would be JSON-encoded. Thus, one of the things we'll need to be able to do when implementing this API (e.g. for this particular API endpoint) is parse JSON request bodies.

- Luckily, there is a third-party package called `body-parser` that provides Express middleware to parse request bodies. We can start out by installing `body-parser` as a dependency of our package:

  ```
  npm install --save body-parser
  ```

- With the `body-parser` dependency installed we can start to use it in our Express server. The first step to be able to do this is to `require()` the dependency:

  ```
  var bodyParser = require('body-parser');
  ```

- Then, we need to tell our Express server to use the JSON-parsing middleware:

  ```
  app.use(bodyParser.json());
  ```

- With this middleware in place, every time a request comes into our server with the `Content-Type` header set to `application/json`, the middleware will parse the body of that request into a JavaScript object, and it will store that object in the field `req.body`. For example, say our server got a request with the following JSON-encoded body:

  ```
  {"name":"Nice place","description":"A nice place to stay"}
  ```

- Then, `req.body` would look like this for that request:

```
{
     name: "Nice place",
     description: "A nice place to stay"
}
```

- With JSON body parsing in place, we can start with the skeleton of our `POST /lodgings` API endpoint:

```
app.post('/lodgings', function (req, res) {
     ...
});
```

- Inside this API endpoint we'll want to extract lodging information from the (parsed) request body and use it to add a new lodging to our lodgings collection. We might also want to do some verification on the information provided in the request body, like making sure required information about the lodging is included.

- For now, let's just say that every lodging needs a name. We'll respond with an error and a 400 (bad request) status to requests that either don't have a body or whose body doesn't contain a `name` field:

```
if (req.body && req.body.name) {
     ...
} else {
     res.status(400).json({
          err: "Request needs a JSON body with a name field"
     });
}
```

- Inside the `if` block here, we know that the client sent a JSON-formatted request body with (at minimum) a `name` field, and we know that request body is now parsed and stored as a JS object in `req.body`. Let's assume for now that we can just take that object and store it directly in our array of lodgings:

```
lodgings.push(req.body);
```

- Finally, we'll respond to the client with a 201 (created) status and a response body that provides the client with the ID of the newly-created lodging resource (its index in the array) as well as a HATEOAS link to that lodging resource (an API endpoint we'll implement later):

```
var id = lodgings.length - 1;
res.status(201).json({
     id: id,
     links: {
          lodging: '/lodgings/' + id
     }
});
```

GET /lodgings/{lodgingID}

- Next, we'll implement the endpoint `GET /lodgings/{lodgingID}`. The client will use this endpoint to fetch information about a single specific lodging.

- This is an endpoint where we'll want to use a parameterized route. Our endpoint skeleton will look like this:

```
app.get('/lodgings/:lodgingID', function (req, res, next) {
     ...
});
```

- Importantly, we'll have to be able to verify that the lodging ID specified by the client is a valid one. In our case, that means verifying that it is within the bounds of the `lodgings` array, since we're using array index as the lodging ID. We can write a simple function to perform this task:

```
function verifyLodgingID(lodgingID) {
     return lodgingID === 0 || lodgingID && lodgingID > 0
          && lodgingID < lodgings.length;
}
```

- Back in our API endpoint, we can access the specified lodging ID via the object `req.params`. If that ID (parsed as an integer) is valid, we can simply return the corresponding lodging. Otherwise, we'll respond with a 404 error:

```
var lodgingID = parseInt(req.params.lodgingID);
if (verifyLodgingID(lodgingID) && lodgings[lodgingID]) {
     res.status(200).json(lodgings[lodgingID]);
} else {
     next();
}
```

- Note here that we simply call `next()` if the lodging ID was invalid. This will result in the middleware function assigned as a catch-all 404 handler to be called (eventually), assuming we have one.

- Note also that even after we verify that the specified lodging ID is valid, we check to make sure the corresponding lodging is defined. This is because later, we'll implement deletion of lodgings by simply replacing a lodging with `null` in the lodgings array.

PUT /lodgings/{lodgingID}

- Next, we'll implement the endpoint `PUT /lodgings/{lodgingID}`, which the client will use to update information about a single specific lodging. Here's the skeleton for this endpoint:

```
app.put('/lodgings/:lodgingID', function (req, res, next) {
     ...
});
```

- When making a PUT request, a client must specify the *entire* replacement for the specified resource. Thus, we can expect the body of each request coming into this endpoint to contain complete lodging information (at minimum, a name).

- Since we've already set up JSON body parsing, we don't need to do that work again here, and we can simply expect our lodging information to be stored in `req.body`.

- When updating information about a lodging, we'll have to perform two different checks on the incoming data:
    - First, we'll have to verify that the specified lodging ID is valid and that the corresponding lodging exists.

    - If the specified lodging is valid, we'll have to make sure the request body is correctly formatted.

- We can start by performing the same check we did just above to verify the specified lodging ID:

```
var lodgingID = parseInt(req.params.lodgingID);
if (verifyLodgingID(lodgingID) && lodgings[lodgingID]) {
     ...
} else {
     next();
}
```

- Inside the `if` block, if the lodging ID is valid, we'll perform the same check on the request body that we performed above in our POST endpoint:

```
if (req.body && req.body.name) {
     ...
} else {
     res.status(400).json({
          err: "Request needs a JSON body with a name field"
     });
}
```

- If the body is valid here, we can replace the specified lodging with the new data and return a response with a link to the updated lodging resource:

```
lodgings[lodgingID] = req.body;
res.status(200).json({
     links: {
          lodging: '/lodgings/' + lodgingID
     }
});
```

DELETE /lodgings/{lodgingID}

- Finally, we'll implement the endpoint `DELETE /lodgings/{lodgingID}`, which the client will use to delete a single specific lodging.

- This endpoint will combine pieces of the endpoints we used above to verify that the lodging the client is trying to delete exists. If it does, it will simply replace it with `null` in the `lodgings` array and use `res.end()` to send back an empty response with status 204 (no content):

```
app.delete(
     '/lodgings/:lodgingID',
     function (req, res, next) {
          var lodgingID = parseInt(req.params.lodgingID);
          if (verifyLodgingID(lodgingID)) {
               lodgings[lodgingID] = null;
               res.status(204).end();
          } else {
               next();
          }
     }
);
```