UNIT - 1

➔ Define Ajax ?

AJAX = Asynchronous JavaScript And XML.
AJAX is not a programming language.
Ajax is a means of using JavaScript to communicate with a web server <u>without</u> submitting a form or loading a new page.
AJAX just uses a combination of:

- A browser built-in XMLHttpRequest object (to request data from a web server)
- JavaScript and HTML DOM (to display or use the data)

➔ Basic Principles of Ajax ?

Usability
  - Ajax applications should be as intuitive, productive, and fun to use as possible.

◆ ***Developer productivity***
  - Development should be as efficient as possible, with a clean, maintainable code base.

◆ ***Efficiency***
  - Ajax applications should consume minimal bandwidth and server resources.

◆ ***Reliability***
  - Ajax applications should provide accurate information and preserve the integrity of data.

◆ ***Privacy***
  - While user-generated data can and should be used to improve the user experience, users' privacy should also be respected, and users should be aware of when and how their data is used.

◆ ***Accessibility***

        - Ajax applications should work for users with particular disabilities and of different ages and cultural backgrounds.
    - ***Compatibility***
        - As an extension to accessibility, Ajax applications should work on a wide range of browser applications, hardware devices, and operating systems.

➔ Drawbacks of Ajax

- No proper history is maintained.
- No back button or forward, difficult in navigation.
- Browser support- Not all browsers support Ajax JS.
- Dynamic updates are not easily noticeable.
- Failure of any one request can fail the load of the whole page.
- Browsers with JS disabled will not be able to use pages using ajax.
- Ajax Applications May Create New Security Concerns-Client Side Javascript is easily viewable and therefore inherently insecure.
- Ajax Applications Do Not Run on a Single Platform browsers that are 1 or 2 versions old.
- Data Exchange Behind the Scenes May Make Users Uncomfortable - Ex : Submit after filling out the form

---

➔ Traditional Model v/s Ajax Model

- Typical browsing behavior
  - consists of loading a web page, then selecting some action that we want to do, filling out a form, submitting the information, etc.

- Sequential manner,
  - requesting one page at a time, wait for the server to respond, loading a whole new web page before we continue.
  - limitations of web pages

- JavaScript
  - Cuts response time: verify.
  - Limitation

- Reloading limitation

- Ajax stands for "Asynchronous JavaScript and XML".
- The word "asynchronous" means that the user isn't left waiting for the server the respond to a request, but can continue using the web page.

- The typical method for using Ajax is the following:

1) A JavaScript creates an `XMLHttpRequest` object, initializes it with relevant information as necessary, and sends it to the server. The script (or web page) can continue after sending it to the server.
2) The server responds by sending the contents of a file or the output of a server side program (written, for example, in PHP).
3) When the response arrives from the server, a JavaScript function is triggered to act on the data supplied by the server.
4) This JavaScript response function typically refreshes the display using the DOM, avoiding the requirement to reload or refresh the entire page.

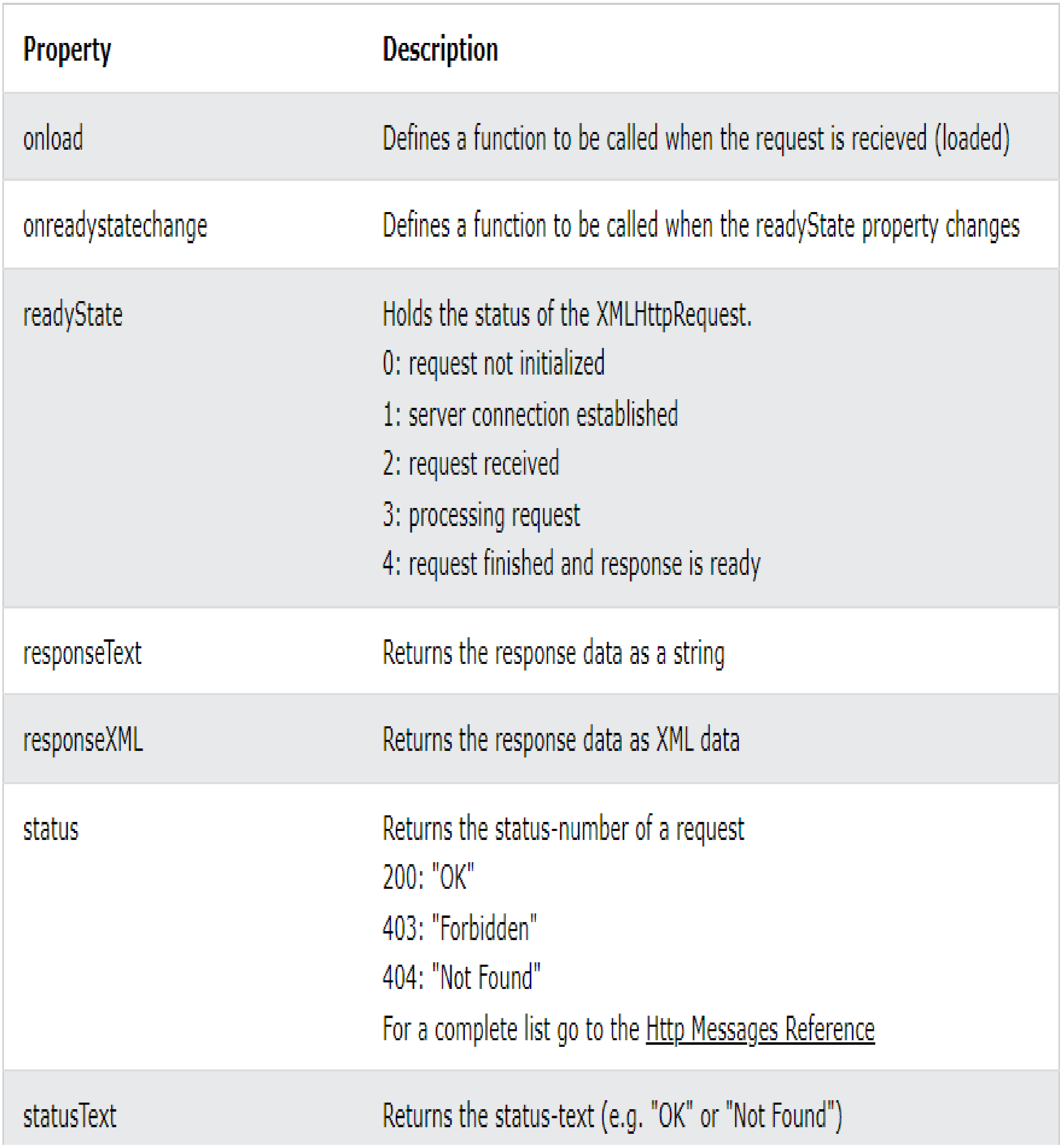
➔ XMLHttpRequest Object

- The XMLHttpRequest object is the backbone of every Ajax method. Each application requires the creation of one of these objects.
- The XMLHttpRequest object can be used to exchange data with a web server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.
- Syntax for creating an XMLHttpRequest object:
  variable = new XMLHttpRequest();

- XMLHttpRequest Methods

Method	Description
new XMLHttpRequest()	Creates a new XMLHttpRequest object
abort()	Cancels the current request
getAllResponseHeaders()	Returns header information
getResponseHeader()	Returns specific header information
open(*method*, *url*, *async*, *user*, *psw*)	Specifies the request *method*: the request type GET or POST *url*: the file location *async*: true (asynchronous) or false (synchronous) *user*: optional user name *psw*: optional password
send()	Sends the request to the server Used for GET requests
send(*string*)	Sends the request to the server. Used for POST requests
setRequestHeader()	Adds a label/value pair to the header to be sent

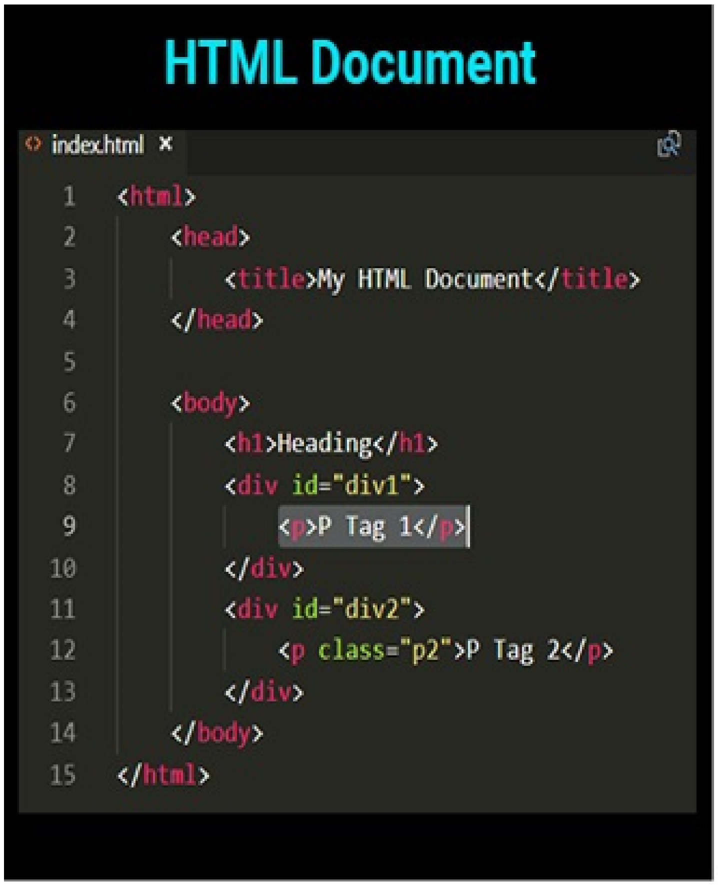
- XMLHttpRequest Object Props



Property	Description
onload	Defines a function to be called when the request is recieved (loaded)
onreadystatechange	Defines a function to be called when the readyState property changes
readyState	Holds the status of the XMLHttpRequest. 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready
responseText	Returns the response data as a string
responseXML	Returns the response data as XML data
status	Returns the status-number of a request 200: "OK" 403: "Forbidden" 404: "Not Found" For a complete list go to the Http Messages Reference
statusText	Returns the status-text (e.g. "OK" or "Not Found")

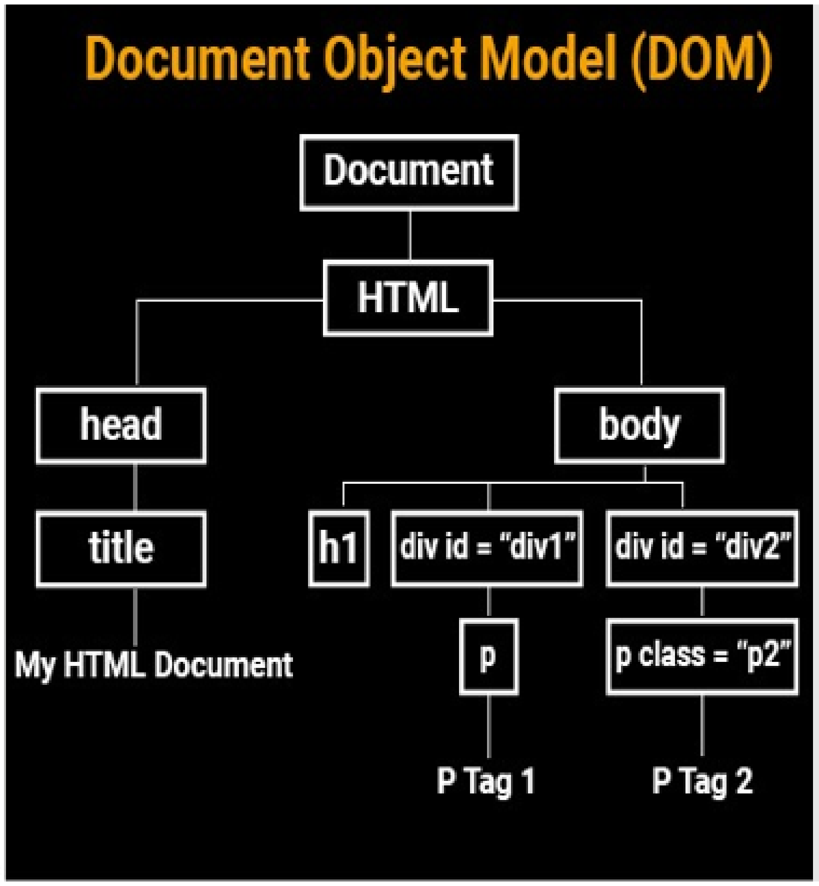
➔ DOM ? DOM Tree ?

The DOM is a W3C (World Wide Web Consortium) standard. The DOM defines a standard for accessing and manipulating documents. The HTML DOM model is constructed as a tree of Objects.
With the object model, JavaScript gets all the power it needs to create dynamic HTML:
JavaScript can change all the HTML elements in the page
JavaScript can change all the HTML attributes in the page
JavaScript can change all the CSS styles in the page
JavaScript can remove existing HTML elements and attributes
JavaScript can add new HTML elements and attributes
JavaScript can react to all existing HTML events in the page
JavaScript can create new HTML events in the page.
The HTML DOM is a standard for how to get, change, add, or delete HTML elements.

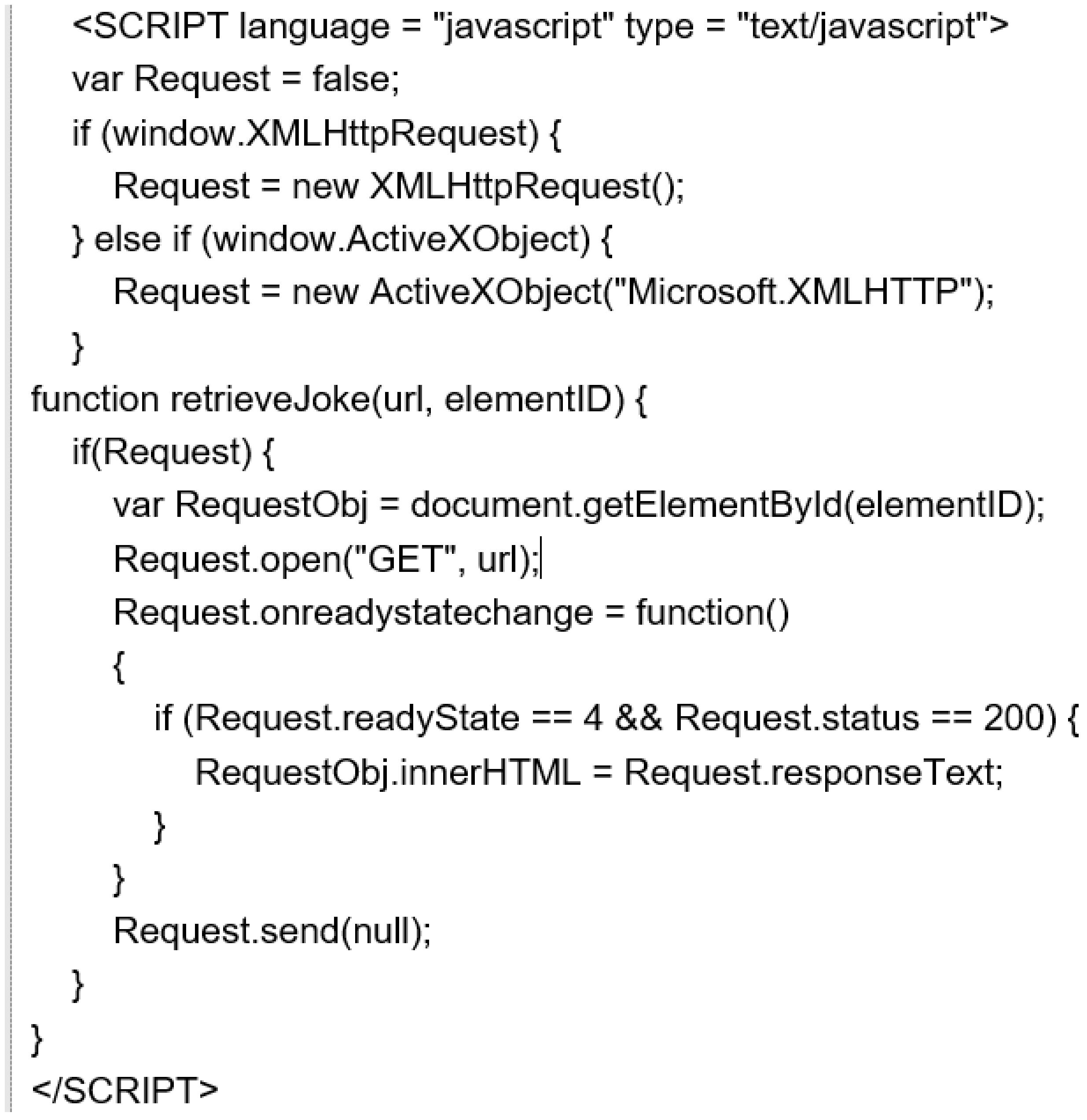


```html
<html>
    <head>
        <title>My HTML Document</title>
    </head>

    <body>
        <h1>Heading</h1>
        <div id="div1">
            <p>P Tag 1</p>
        </div>
        <div id="div2">
            <p class="p2">P Tag 2</p>
        </div>
    </body>
</html>
```



➔ Program to read from a text file and on click of a button. (Joke of the day shit)



```
    <SCRIPT language = "javascript" type = "text/javascript">
    var Request = false;
    if (window.XMLHttpRequest) {
        Request = new XMLHttpRequest();
    } else if (window.ActiveXObject) {
        Request = new ActiveXObject("Microsoft.XMLHTTP");
    }
function retrieveJoke(url, elementID) {
    if(Request) {
        var RequestObj = document.getElementById(elementID);
        Request.open("GET", url);
        Request.onreadystatechange = function()
        {
            if (Request.readyState == 4 && Request.status == 200) {
                RequestObj.innerHTML = Request.responseText;
            }
        }
        Request.send(null);
    }
}
</SCRIPT>
```

➔ Page to update content dynamically using DOM

```
<HTML>
<HEAD>
<TITLE>Demo: Accessing DOM Elements by ID</TITLE>
<SCRIPT language = "javascript" type = "text/javascript">
<!-- Start hiding JavaScript statements
function SetAlarm() {
document.getElementById('DivTrgt').innerHTML = "Charge!"
}
// End hiding JavaScript statements -->
</SCRIPT>
</HEAD>
<BODY onload = " SetAlarm()">
<H1>Knock Knock!</H1>
<DIV id="DivTrgt"> </DIV>
</BODY>
</HTML>
```

➔ DOM Props and Methods

TABLE 4.1 DOM PROPERTIES

Property	Description
`childNodes`	A collection (e.g., array) of child objects belonging to an object
`firstChild`	The first child node belonging to an object
`lastChild`	An object's last child node
`NodeName`	The name assigned to an object's HTML tag
`nodeType`	Identifies the type of HTML element (tag, attribute, or text) associated with the object
`nodeValue`	Retrieves the value assigned to a text node
`nextSibling`	The child node following the previous child node in the tree
`previousSibling`	The child node that comes before the current child node
`parentNode`	An object's parent object

TABLE 4.2 DOM METHODS

Property	Description
`appendChild()`	Adds a new child node to the specified element
`createAttribute()`	Creates a new element attribute
`createElement()`	Creates a new document element
`createTextNode()`	Creates a new text item
`getElementByTagName()`	Retrieves an array of item tag names
`getElementsById()`	Retrieves an element based on its ID
`hasChildNotes()`	Returns a `true` or `false` value depending on whether a node has children
`removeChild()`	Deletes the specified child node

UNIT - 2

- ➔ Define AngularJS

  AngularJS is a JavaScript-based open-source front-end web framework for developing single-page applications.
  Benefits of AngularJS :
  It provides the capability to create Single Page Application in a very clean and maintainable way.
  It provides data binding capability to HTML. ...
  AngularJS code is unit testable.
  AngularJS uses dependency injection and make use of separation of concerns.
  AngularJS provides reusable components.

- ➔ One way data binding v/s Two way data-binding

  One way binding:
  In one-way binding, the data flow is one-directional.
  This means that the flow of code is from typescript file to Html file.
  In order to achieve a one-way binding, we used the property binding concept in Angular.
  In a two-way binding, the data flow is bi-directional.
  This means that the flow of code is from ts file to Html file as well as from Html file to ts file.Any changes to the view are propagated to the component class. Also, any changes to the properties in the component class are reflected in the view.

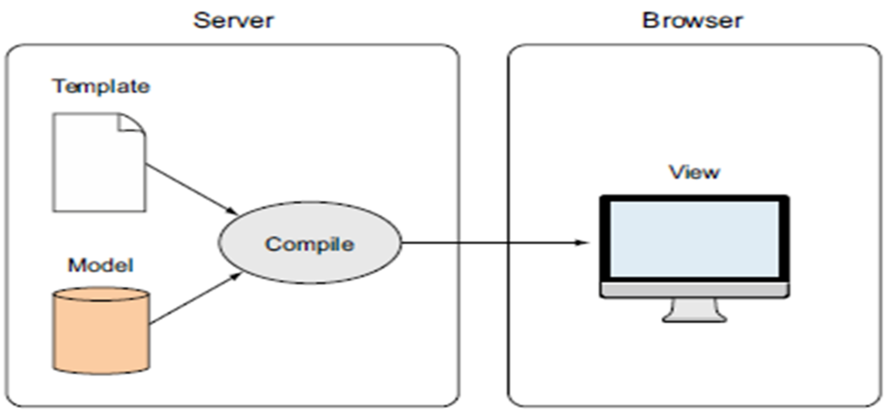


Figure 1.5 One-way data binding—the template and model are compiled on the server before being sent to the browser.

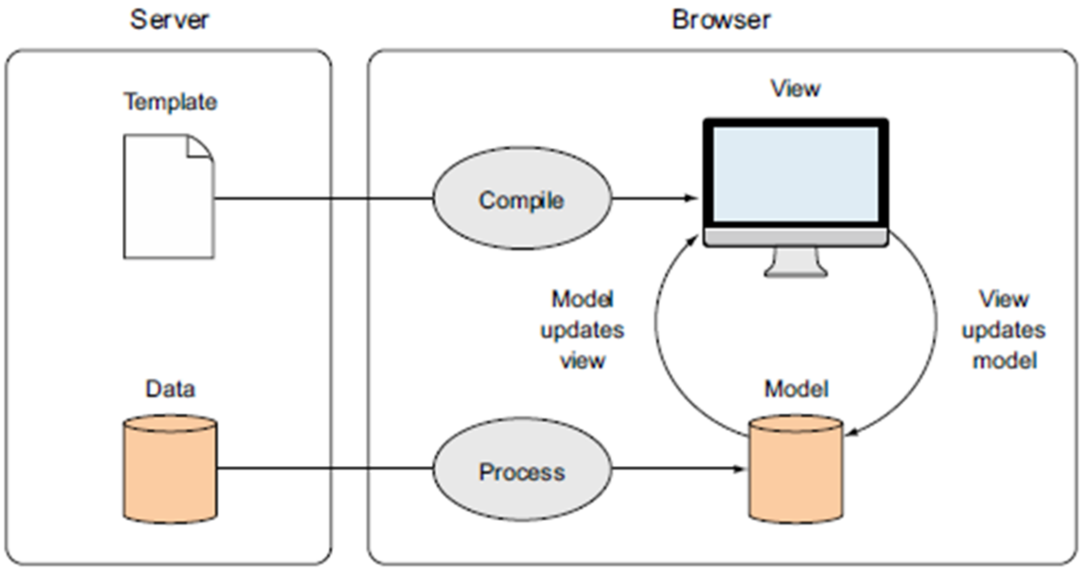


Figure 1.6 Two-way data binding—the model and the view are processed in the browser and bound together, each instantly updating the other.

➔ Define NodeJS

Node.js is an open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes

JavaScript code outside a web browser.
Benefits of NodeJS :
Efficient performance.
Easier development process.
Reusable code.
Ability to handle multiple requests.
Ability to scale smoothly.
Prompt code execution.
Asynchronous and event-driven.
Supported by leading companies.

➔ 5 stages of development of a website

STAGE 1: BUILD A STATIC SITE
The first stage is to build a static version of the application, which is essentially a number
of HTML screens. The aims of this stage are
■ To quickly figure out the layout
■ To ensure that the user flow makes sense
At this point we're not concerned with a database or flashy effects on the user interface;
all we want to do is create a working mockup of the main screens and journeys
that a user will take through the application.

STAGE 2: DESIGN THE DATA MODEL AND CREATE THE DATABASE
Once we have a working static prototype that we're happy with, the next thing to do is
look at any hard-coded data in the static application and put it into a database. The

aims of this stage are
■ To define a data model that reflects the requirements of the application
■ To create a database to work with the model

STAGE 3: BUILD OUR DATA API
After stages 1 and 2 we have a static site on one hand and a database on the other. This stage and the next take the natural steps of linking them together. The aim of stage 3 is
■ To create a REST API that will allow our application to interact with the database

STAGE 4: HOOK THE DATABASE INTO THE APPLICATION
When we get to this stage we have a static application and an API exposing an interface
to our database. The aim of this stage is
■ To get our application to talk to our API
When this stage is complete the application will look pretty much the same as it did
before, but the data will be coming from the database. When it's done, we'll have a
data-driven application!

STAGE 5: AUGMENT THE APPLICATION
This stage is all about embellishing the application with additional functionality. We
might add authentication systems, data validation, or methods for displaying error
messages to users. It could include adding more interactivity to the front end or tightening
up the business logic in the application itself.
So, really, the aims of this stage are

■ To add finishing touches to our application
■ To get the application ready for people to use
These five stages of development provide a great methodology for approaching a new
build project

OR

- ➔ Phase One – Information Gathering

  This is the most important phase of website design & development. The primary stage of website development is gathering information i.e analyzing clients needs & requirements also known as “Discovery phase”. In this phase, the designer portrays the client’s vision into the paper. In the discovery phase, it is important to understand the purpose of creating a website, it is also very important to know the goal of the website, which targets audience you want to get targeted, type of content your target audience will look for? These factors are very crucial to determine in the fundamental phase of website design.

  Phase Two – Planning

  “Good website is the result of good planning” is what we believe in. After the information gathering what comes is planning. Planning is nothing but prioritizing tasks for website completion. In this phase, we develop the sitemap of the website is developed. Here, we decide the menus, contents & navigational system for the website.

  Phase Three – Designing

  This is the creative phase of website design. This is the phase where designer put their heart & soul into it. Also, communication plays a major role here. The designer needs to understand each & every aspect of the client expectation & try to sketch it. From logo design to selecting templates everything is discovered in this phase.

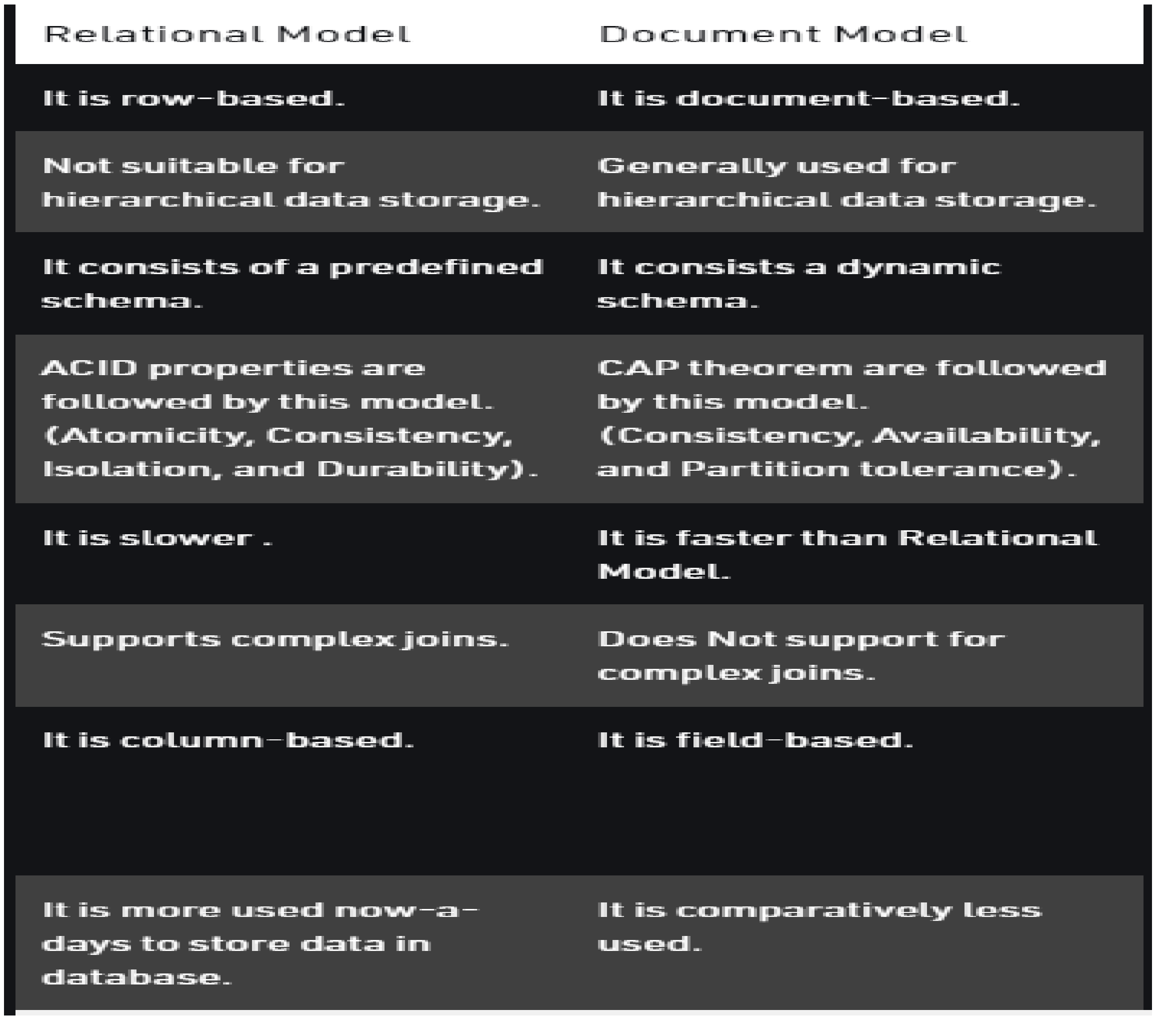
Phase Four – Development

After designing, there is a development phase also known as ‘implementing phase’. Now, this is the phase where your actual website starts its implementation. The development phase is also a very crucial phase for the website design. Here, we integrate all the information that we had collected from initial phases. Creating Database, logic & actual programming is done here.

Phase Five – Testing & Delivery

After the Development phase, there is a Testing & Discovery Phase. The testing is done by QA, also responsible for preparing the test cases. The following are the types of website testing

➔ Relational v/s Document DB



Relational Model	Document Model
It is row-based.	It is document-based.
Not suitable for hierarchical data storage.	Generally used for hierarchical data storage.
It consists of a predefined schema.	It consists a dynamic schema.
ACID properties are followed by this model. (Atomicity, Consistency, Isolation, and Durability).	CAP theorem are followed by this model. (Consistency, Availability, and Partition tolerance).
It is slower .	It is faster than Relational Model.
Supports complex joins.	Does Not support for complex joins.
It is column-based.	It is field-based.
It is more used now-a-days to store data in database.	It is comparatively less used.

➔ Benefits of FS

- Better view of the bigger picture
- Appreciation
- Team members
- End-to-end applications
- More skills, services, and capabilities.

➔Application Architecture of Mean Stack

A common way to architect a MEAN stack application is to have a representational

state transfer (REST) API feeding a single-page application. The API is typically built

with MongoDB, Express, and Node.js, with the SPA being built in AngularJS. This

approach is particularly popular with those who come to the MEAN stack from an

AngularJS background and are looking for a stack that gives a fast, responsive API. Figure

2.1 illustrates the basic setup and data flow.

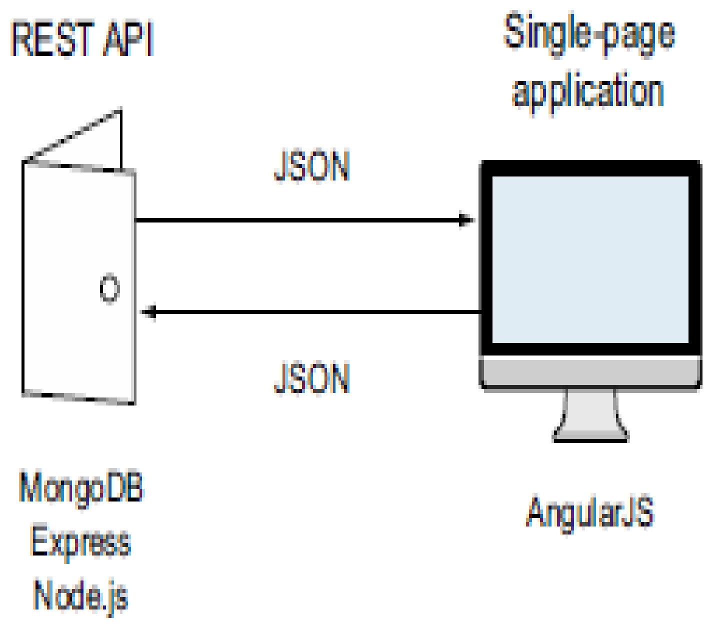


Figure 2.1 A common approach to MEAN stack architecture, using MongoDB, Express, and Node.js to build a REST API that feeds JSON data to an AngularJS SPA run in the browser

Figure 2.1 is a great setup, ideal if you have, or intend to build, an SPA as your userfacing

side. AngularJS is designed with a focus on building SPAs, pulling in data from a

REST API, as well as pushing it back. MongoDB, Express, and Node.js are also extremely capable when it comes to building an API, using JSON all the way through the stack

including the database itself.

This is where many people start with the MEAN stack, looking for an answer to the

question, “I’ve built an application in AngularJS; now where do I get the data?”

Having an architecture like this is great if you have an SPA, but what if you don’t

have or want to use an SPA? If this is the only way you think of the MEAN stack, you’re

going to get a bit stuck and start looking elsewhere. But the MEAN stack is very flexible.

All four components are very powerful and have a lot to offer.

UNIT - 3

Textbook Link

- ➔ Explain key interactions and processes that Express goes through when responding to the request of the default landing page.
  Page No. 64

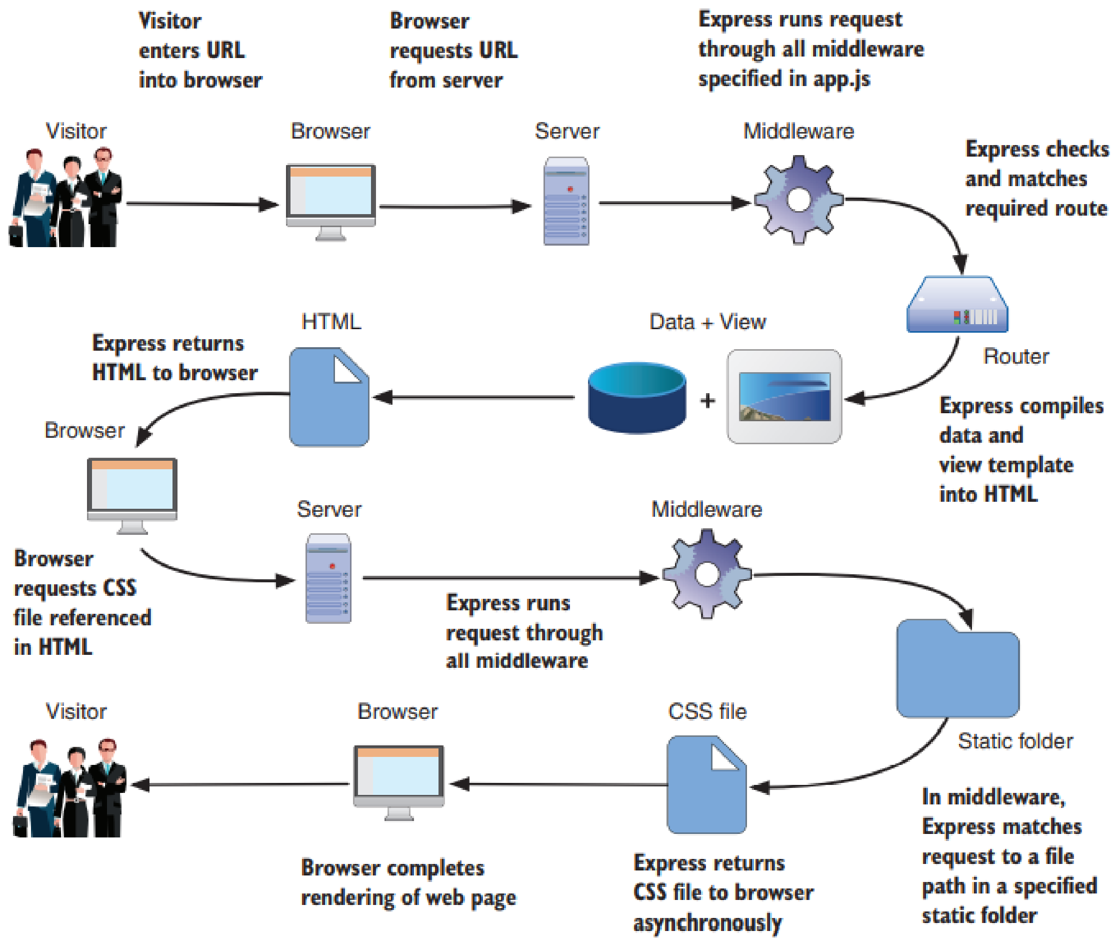
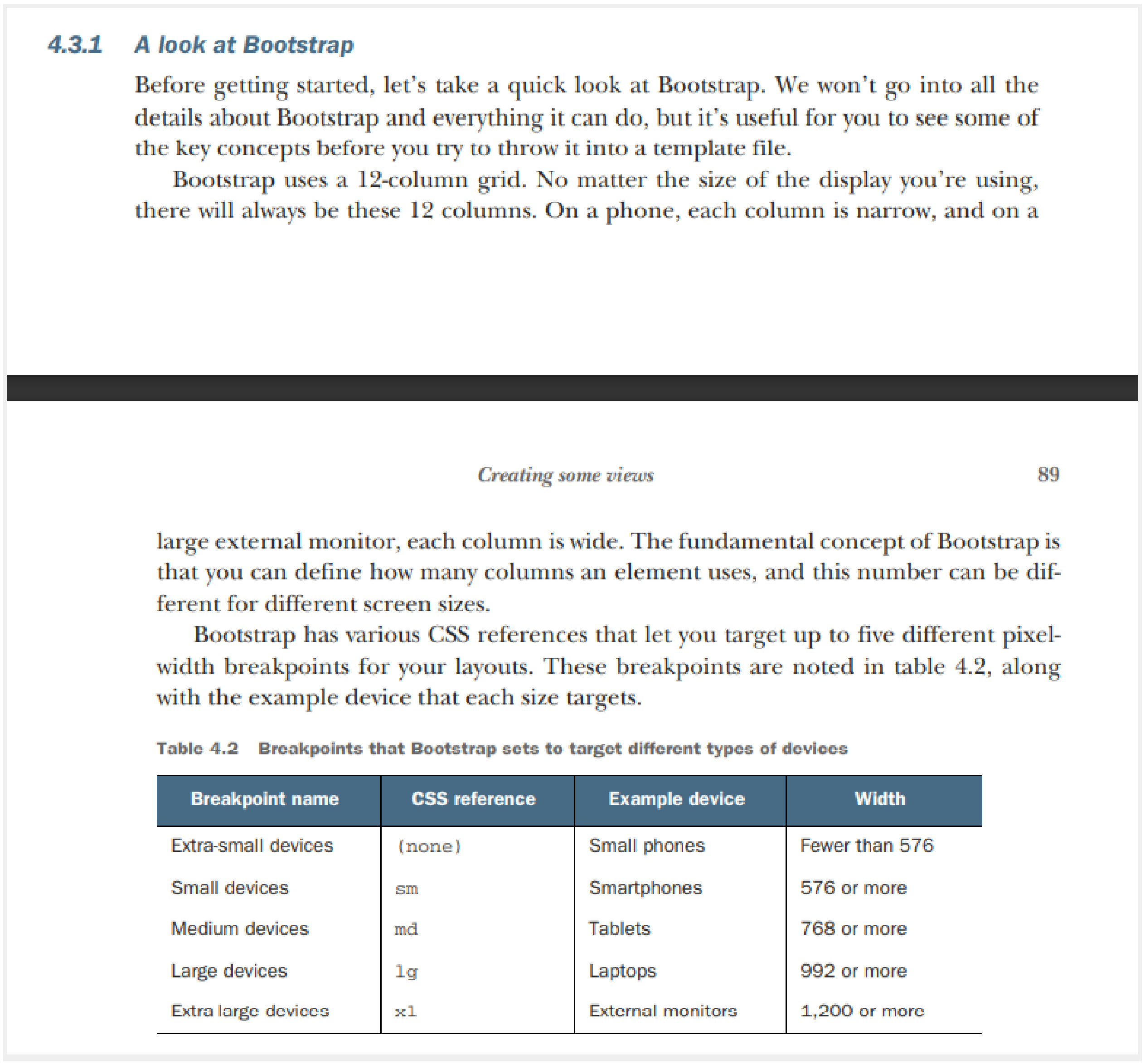

Visitor enters URL into browser
Browser requests URL from server
Express runs request through all middleware specified in app.js
Visitor
Browser
Server
Middleware
Express checks and matches required route
Router
Express returns HTML to browser
HTML
Data + View
+
Express compiles data and view template into HTML
Browser
Server
Middleware
Browser requests CSS file referenced in HTML
Express runs request through all middleware
Static folder
Visitor
Browser
CSS file
In middleware, Express matches request to a file path in a specified static folder
Browser completes rendering of web page
Express returns CSS file to browser asynchronously

Figure 3.4 The key interactions and processes that Express goes through when responding to the request for the default landing page. The HTML page is processed by Node to compile data and a view template, and the CSS file is served asynchronously from a static folder.

➔ Explain Bootstrap's responsive grid system with a diagram.

Page No. 88



4.3.1 A look at Bootstrap

Before getting started, let's take a quick look at Bootstrap. We won't go into all the details about Bootstrap and everything it can do, but it's useful for you to see some of the key concepts before you try to throw it into a template file.

Bootstrap uses a 12-column grid. No matter the size of the display you're using, there will always be these 12 columns. On a phone, each column is narrow, and on a

large external monitor, each column is wide. The fundamental concept of Bootstrap is that you can define how many columns an element uses, and this number can be different for different screen sizes.

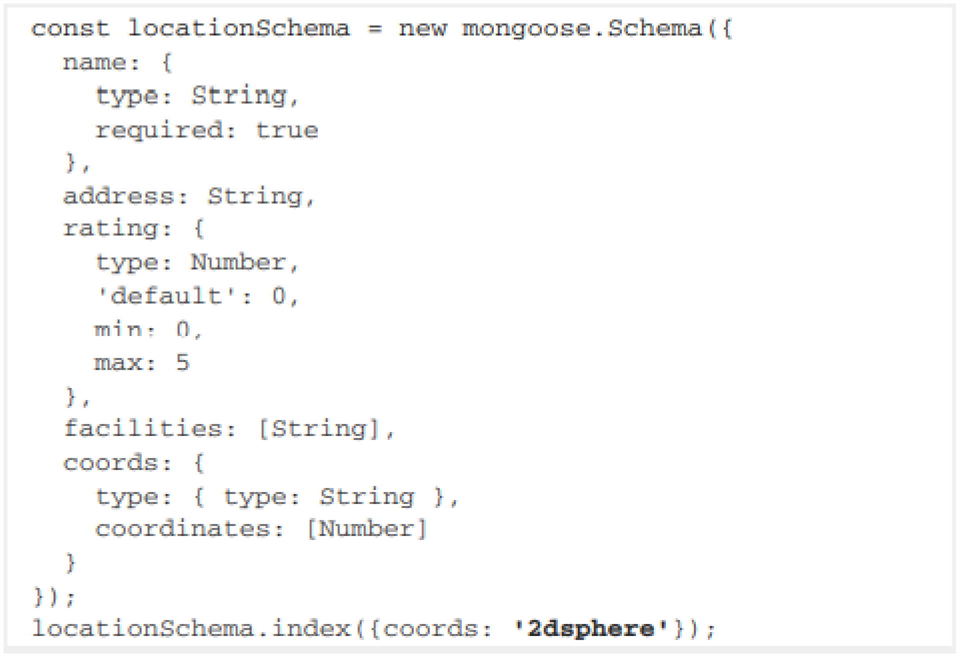
Bootstrap has various CSS references that let you target up to five different pixel-width breakpoints for your layouts. These breakpoints are noted in table 4.2, along with the example device that each size targets.

Table 4.2 Breakpoints that Bootstrap sets to target different types of devices

Breakpoint name	CSS reference	Example device	Width
Extra-small devices	`(none)`	Small phones	Fewer than 576
Small devices	`sm`	Smartphones	576 or more
Medium devices	`md`	Tablets	768 or more
Large devices	`lg`	Laptops	992 or more
Extra large devices	`xl`	External monitors	1,200 or more

➔ Design the mongoose schema for Details page of loc8r application with sample data.

Page No. 136 & 138



```
const locationSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true
  },
  address: String,
  rating: {
    type: Number,
    'default': 0,
    min: 0,
    max: 5
  },
  facilities: [String],
  coords: {
    type: { type: String },
    coordinates: [Number]
  }
});
locationSchema.index({coords: '2dsphere'});
```

Listing 5.2 Data in the controller powering the Details page

```
location: {
  name: 'Starcups',
  address: '125 High Street, Reading, RG6 1PS',
  rating: 3,
  facilities: ['Hot drinks', 'Food', 'Premium wifi'],
  coords: {lat: 51.455041, lng: -0.9690884},

    days: 'Monday - Friday',
    opening: '7:00am',
    closing: '7:00pm',
    closed: false
  },{
    days: 'Saturday',
    opening: '8:00am',
    closing: '5:00pm',
    closed: false
  },{
    days: 'Sunday',
    closed: true
  }],
  reviews: [{
    author: 'Simon Holmes',
    rating: 5,
    timestamp: '16 July 2013',
    reviewText: 'What a great place.
➥ I can\'t say enough good things about it.'
  },{
    author: 'Charlie Chaplin',
    rating: 3,
    timestamp: '16 June 2013',
    reviewText: 'It was okay. Coffee wasn\'t great,
    ➥ but the wifi was fast.'
  }]
}
```

Already covered with the existing schema

Data for opening hours is held as an array of objects.

Reviews are also passed to the view as an array of objects.

➔ Explain how you import Bootstrap for responsive layouts.
Page No. 400

Twitter Bootstrap

Bootstrap isn't installed as such, but is added to your application. This process is as simple as downloading the library files, unzipping them, and placing them in the application.

The first step is downloading Bootstrap. This book uses version 4.1, which currently is the official release. You can get it from https://getbootstrap.com. Make sure you download the "ready to use files" and not the source. The distribution zip contains two folders: css and js.

When you have the files downloaded and unzipped, move one file from each folder into the public folder in your Express application, as follows:

1. Copy bootstrap.min.css into your public/stylesheets folder.
2. Copy bootstrap.min.js into your public/js folder.

Figure B.1 shows how the public folder in your application should look.

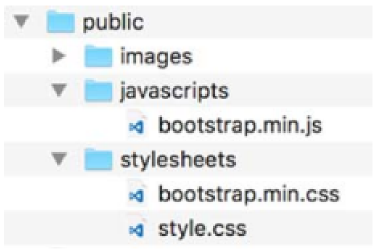


Figure B.1 The structure and contents of the public folder after Bootstrap has been added

That gives you access to the default look and feel of Bootstrap, but you probably want your application to stand out from the crowd a bit. You can do so by adding a theme or some custom styles.

➔ What is mongoose? Explain how mongoose models the data.

◆ Page 132

Mongoose was built specifically as a MongoDB Object-Document Modeler (ODM) for Node applications. One of the key principles is that you can manage your data model from within your application. You don't have to mess around directly with databases or external frameworks or relational mappers; you can just define your data model in the comfort of your application.

First off, let's get some naming conventions out of the way:

- In MongoDB each entry in a database is called a *document*.
- In MongoDB a collection of documents is called a *collection* (think "table" if you're used to relational databases).
- In Mongoose the definition of a document is called a *schema*.
- Each individual data entity defined in a schema is called a *path*.

How does Mongoose model data?

If we're defining our data in the application, how are we going to do it? In JavaScript, of course! JavaScript objects to be precise. We've already had a sneak peak in figure 5.7, but let's take a look at a simple MongoDB document and see what the Mongoose schema for it might look like. The following code snippet shows a MongoDB document, followed by the Mongoose schema:

```
{
  "firstname" : "Simon",
  "surname" : "Holmes",
  _id : ObjectId("52279effc62ca8b0c1000007")
}
```

Example MongoDB document

```
{
  firstname : String,
  surname : String
}
```

Corresponding Mongoose schema

As you can see, the schema bears a very strong resemblance to the data itself. The schema defines the name for each data path, and the data type it will contain. In this example we've simply declared the paths `firstname` and `surname` as strings.

5.1 Connecting the Express application to MongoDB by using Mongoose

You could connect your application directly to MongoDB and have the two interact by using the native driver. Although the native MongoDB driver is powerful, it isn't particularly easy to work with. It also doesn't offer a built-in way of defining and maintaining data structures. Mongoose exposes most of the functionality of the native driver, but in a more convenient way, designed to fit into the flow of application development.

Where Mongoose really excels is in the way it enables you to define data structures and models, maintain them, and use them to interact with your database, all from the comfort of your application code. As part of this approach, Mongoose includes the ability to add validation to your data definitions, meaning that you don't have to write validation code in every place in your application where you send data back to the database.

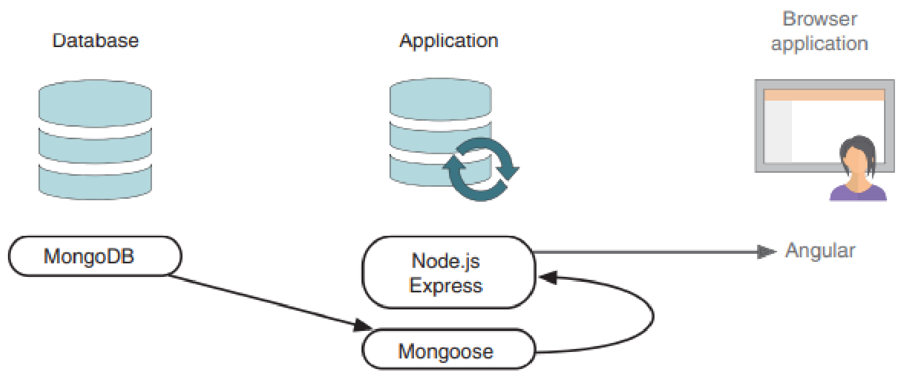


Figure 5.4 The data interactions in the MEAN stack and where Mongoose fits in. The Node/ Express application interacts with MongoDB through Mongoose; Node and Express can also talk to

➔ Explain how a relational database and document database store repeating information relating to a parent element.
Page No. 138

MongoDB offers the concept of *subdocuments* to store this repeating, nested data. Subdocuments are much like documents in that they have their own schema; each is

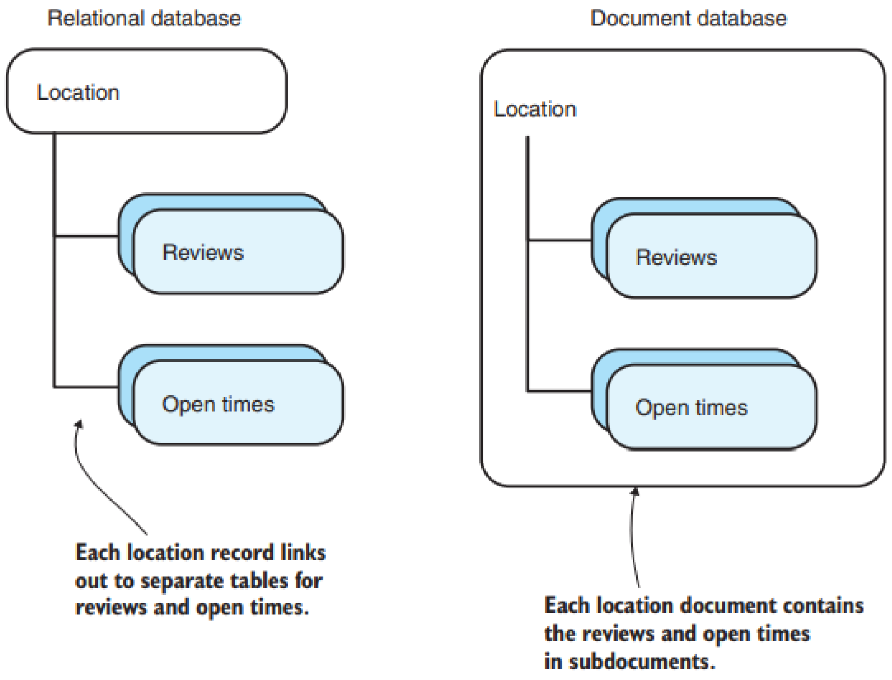

Relational database
Location
Reviews
Open times
Each location record links out to separate tables for reviews and open times.
Document database
Location
Reviews
Open times
Each location document contains the reviews and open times in subdocuments.

Figure 5.10 Difference between how a relational database and a document database store repeating information relating to a parent element

given a unique _id by MongoDB when created. But subdocuments are nested inside a document, and they can be accessed only as a path of that parent document.

➔ Explain request response flow of MVC architecture.

Page No. 65

Most applications or sites that you build are designed to take an incoming request, do something with it, and return a response. At a simple level, this loop in an MVC architecture works like this:

1. A request comes into the application.
2. The request gets routed to a controller.
3. The controller, if necessary, makes a request to the model.
4. The model responds to the controller.
5. The controller merges the view and the data to form a response.
6. The controller sends the generated response to the original requester.

In reality, depending on your setup, the controller may compile the view before sending the response to the visitor. The effect is the same, though, so keep this simple flow in mind as a visual for what will happen in your Loc8r application. See figure 3.5 for an illustration of this loop.

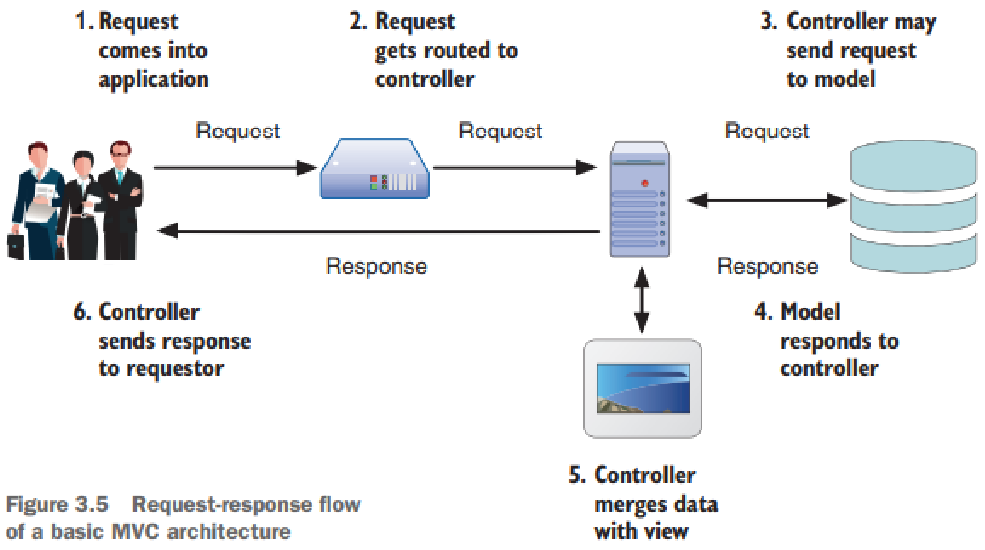

1. Request comes into application
2. Request gets routed to controller
3. Controller may send request to model
Request
Request
Request
Response
Response
6. Controller sends response to requestor
4. Model responds to controller
5. Controller merges data with view

Figure 3.5 Request-response flow of a basic MVC architecture

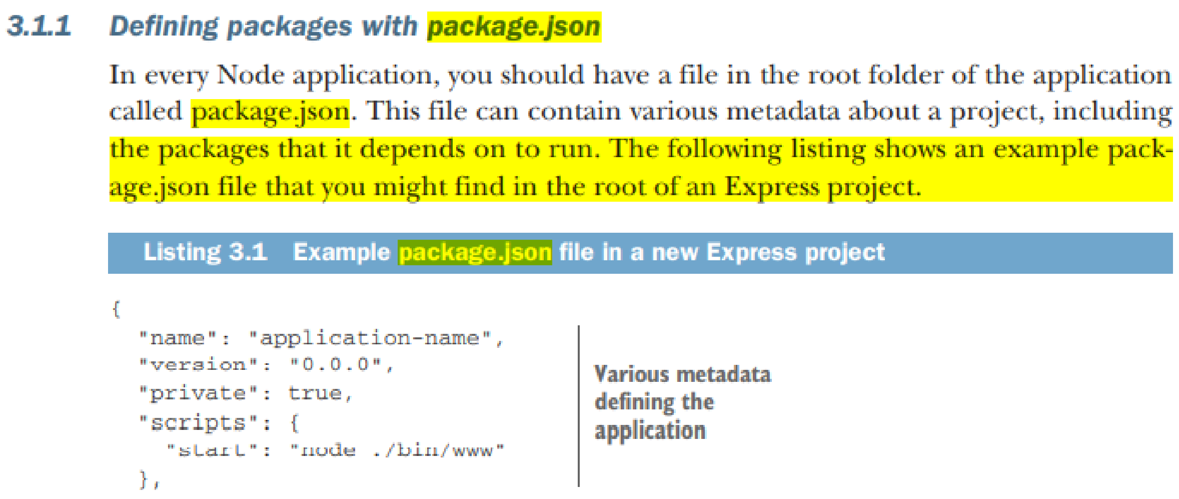
Figure 3.5 highlights the parts of the MVC architecture and shows how they link together. It also illustrates the need for a routing mechanism along with the model, view, and controller components.

Now that you've seen how you want the basic flow of your Loc8r application to work, it's time to modify the Express setup to make this happen.

➔ Explain the three options for designing the architecture of the *Loc8r* application with proper diagram

➔ Define *package.json.* Explain how to install node dependencies using npm.

Page No. 55



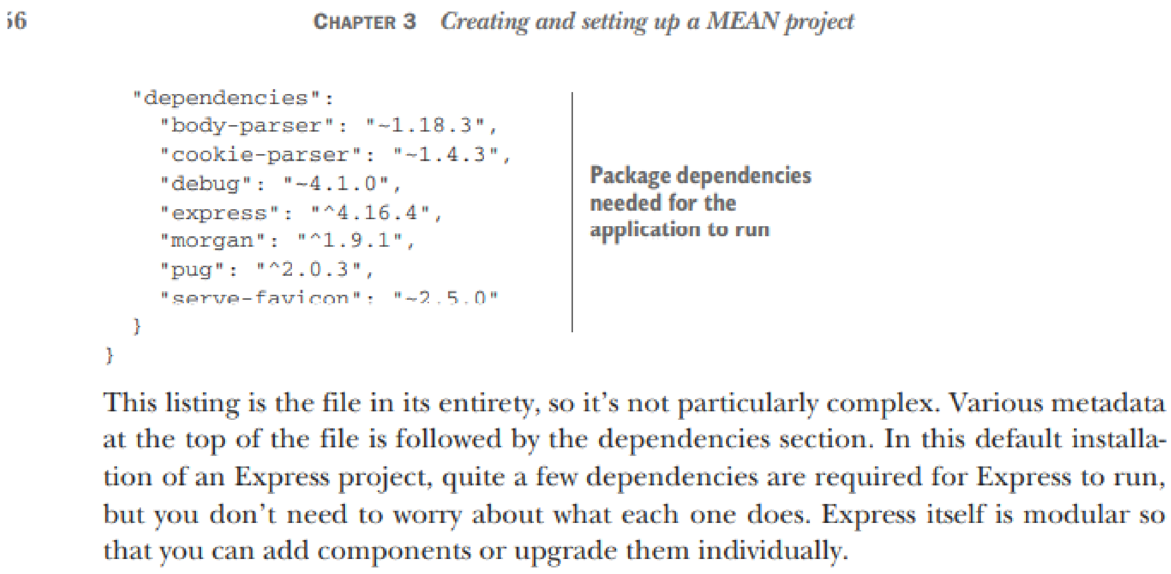
3.1.1 *Defining packages with package.json*

In every Node application, you should have a file in the root folder of the application called package.json. This file can contain various metadata about a project, including the packages that it depends on to run. The following listing shows an example package.json file that you might find in the root of an Express project.

Listing 3.1 Example package.json file in a new Express project

```
{
  "name": "application-name",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
```

Various metadata defining the application



```
  "dependencies":
    "body-parser": "~1.18.3",
    "cookie-parser": "~1.4.3",
    "debug": "~4.1.0",
    "express": "^4.16.4",
    "morgan": "^1.9.1",
    "pug": "^2.0.3",
    "serve-favicon": "~2.5.0"
  }
}
```

Package dependencies needed for the application to run

This listing is the file in its entirety, so it's not particularly complex. Various metadata at the top of the file is followed by the dependencies section. In this default installation of an Express project, quite a few dependencies are required for Express to run, but you don't need to worry about what each one does. Express itself is modular so that you can add components or upgrade them individually.

3.1.3 Installing Node dependencies with npm

Any Node application or module can have dependencies defined in a package.json file. Installing them is easy and is done the same way regardless of the application or module.

Using a terminal prompt in the same folder as the package.json file, run the following command:

```
$ npm install
```

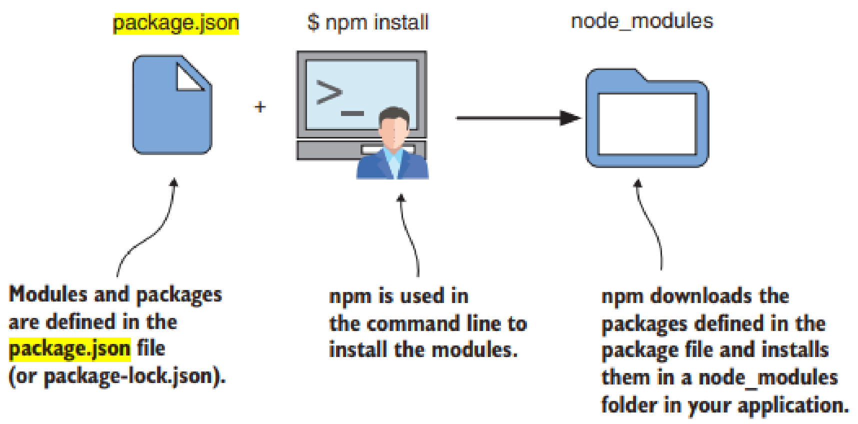

package.json
+
$ npm install
node_modules
Modules and packages are defined in the package.json file (or package-lock.json).
npm is used in the command line to install the modules.
npm downloads the packages defined in the package file and installs them in a node_modules folder in your application.

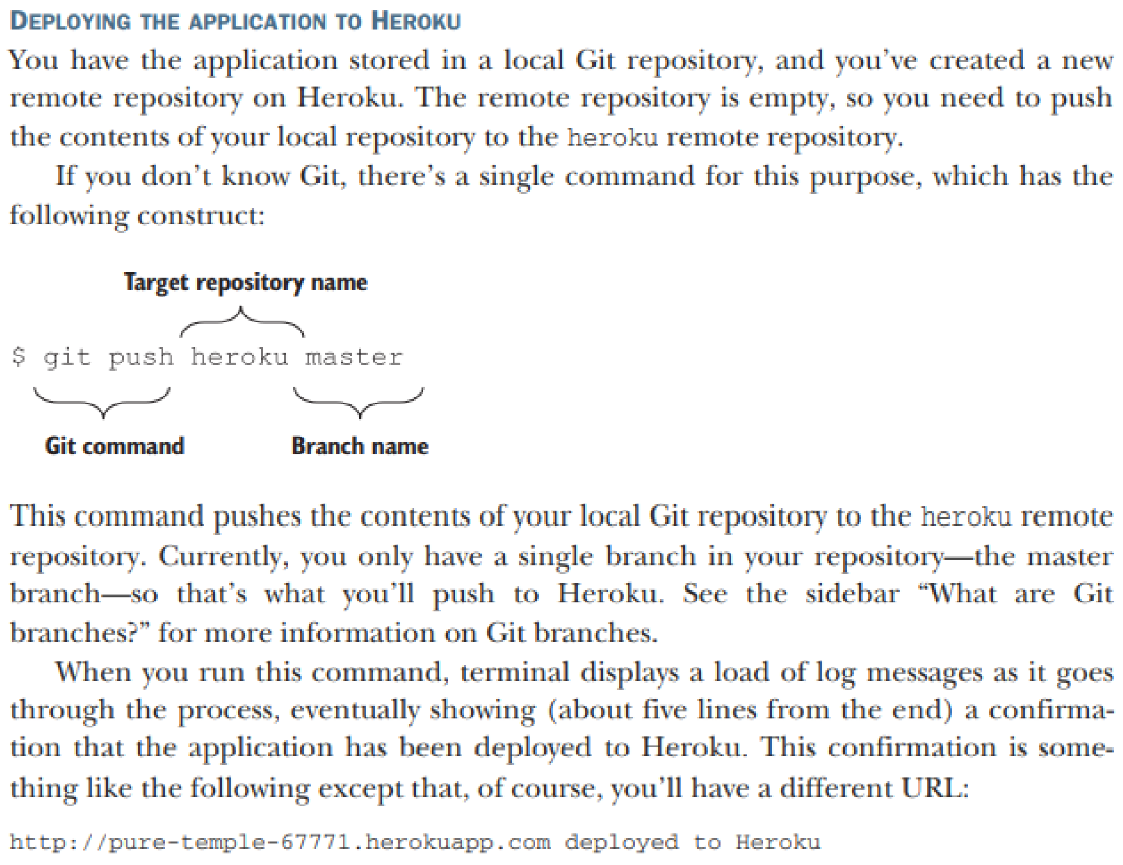
Figure 3.2 The npm modules defined in a package.json file are downloaded and installed in the application's node_modules folder when you run the `npm install` terminal command.

This command tells npm to install all the dependencies listed in the package.json file. When you run it, npm downloads all the packages listed as dependencies and installs them in a specific folder in the application, called node_modules. Figure 3.2 illustrates the three key parts.

npm installs each package into its own subfolder because each one is effectively a Node package in its own right. As such, each package also has its own package.json file defining the metadata, including the specific dependencies. It's quite common for a package to have its own node_modules folder. You don't need to worry about manually installing all the nested dependencies, though, because this task is handled by the original `npm install` command.

➔ Explain with appropriate commands how to deploy the Node application to heroku using git.

Page No. 78



DEPLOYING THE APPLICATION TO HEROKU

You have the application stored in a local Git repository, and you've created a new remote repository on Heroku. The remote repository is empty, so you need to push the contents of your local repository to the `heroku` remote repository.

If you don't know Git, there's a single command for this purpose, which has the following construct:

```
               Target repository name
$ git push heroku master
  Git command        Branch name
```

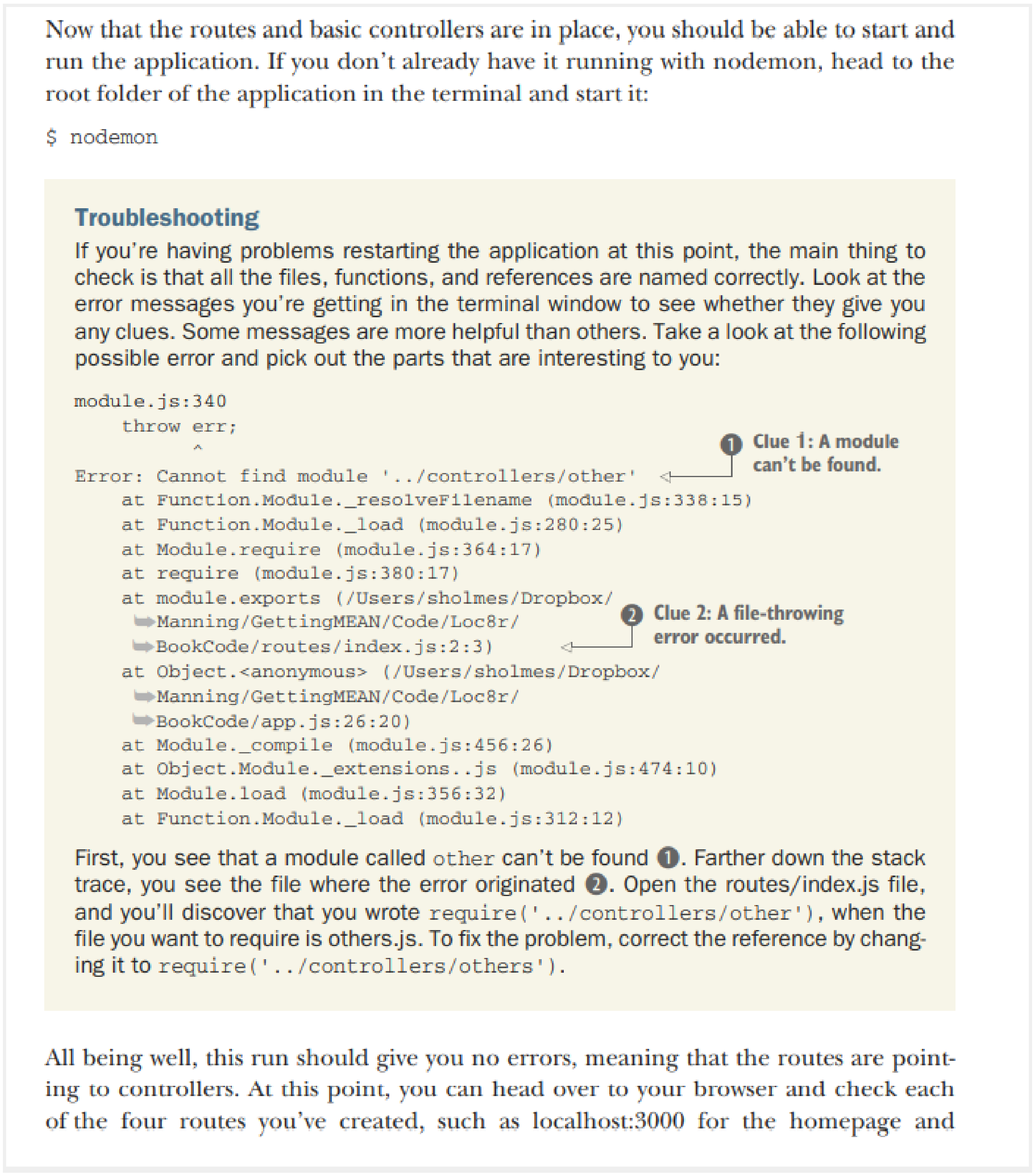
This command pushes the contents of your local Git repository to the `heroku` remote repository. Currently, you only have a single branch in your repository—the master branch—so that's what you'll push to Heroku. See the sidebar "What are Git branches?" for more information on Git branches.

When you run this command, terminal displays a load of log messages as it goes through the process, eventually showing (about five lines from the end) a confirmation that the application has been deployed to Heroku. This confirmation is something like the following except that, of course, you'll have a different URL:

```
http://pure-temple-67771.herokuapp.com deployed to Heroku
```

➔ Explain how you test routers and controllers.

Page No. 87



Now that the routes and basic controllers are in place, you should be able to start and run the application. If you don't already have it running with nodemon, head to the root folder of the application in the terminal and start it:

```
$ nodemon
```

Troubleshooting

If you're having problems restarting the application at this point, the main thing to check is that all the files, functions, and references are named correctly. Look at the error messages you're getting in the terminal window to see whether they give you any clues. Some messages are more helpful than others. Take a look at the following possible error and pick out the parts that are interesting to you:

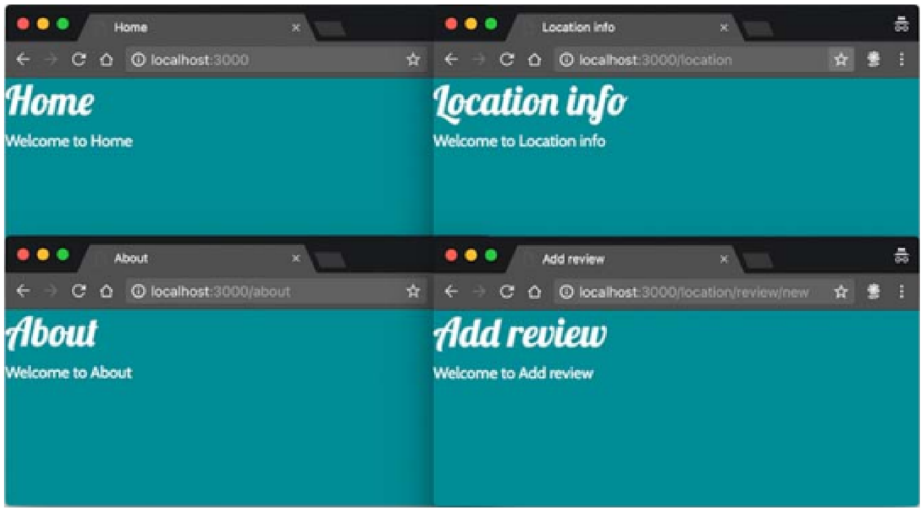
```
module.js:340
    throw err;
          ^
Error: Cannot find module '../controllers/other'
    at Function.Module._resolveFilename (module.js:338:15)
    at Function.Module._load (module.js:280:25)
    at Module.require (module.js:364:17)
    at require (module.js:380:17)
    at module.exports (/Users/sholmes/Dropbox/
➥Manning/GettingMEAN/Code/Loc8r/
➥BookCode/routes/index.js:2:3)
    at Object.<anonymous> (/Users/sholmes/Dropbox/
➥Manning/GettingMEAN/Code/Loc8r/
➥BookCode/app.js:26:20)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
```

1 Clue 1: A module can't be found.

2 Clue 2: A file-throwing error occurred.

First, you see that a module called `other` can't be found 1. Farther down the stack trace, you see the file where the error originated 2. Open the routes/index.js file, and you'll discover that you wrote `require('../controllers/other')`, when the file you want to require is others.js. To fix the problem, correct the reference by changing it to `require('../controllers/others')`.

All being well, this run should give you no errors, meaning that the routes are pointing to controllers. At this point, you can head over to your browser and check each of the four routes you've created, such as localhost:3000 for the homepage and localhost:3000/location for the location information page. Because you changed the data being sent to the view template by each of the controllers, you can easily see that each one is running correctly—the title and heading should be different on each page. Figure 4.4 shows a collection of screenshots of the newly created routes and controllers. You can see that each route is getting unique content, so you know that the routing and controller setup has worked.



Home

Welcome to Home

Location info

Welcome to Location info

About

Welcome to About

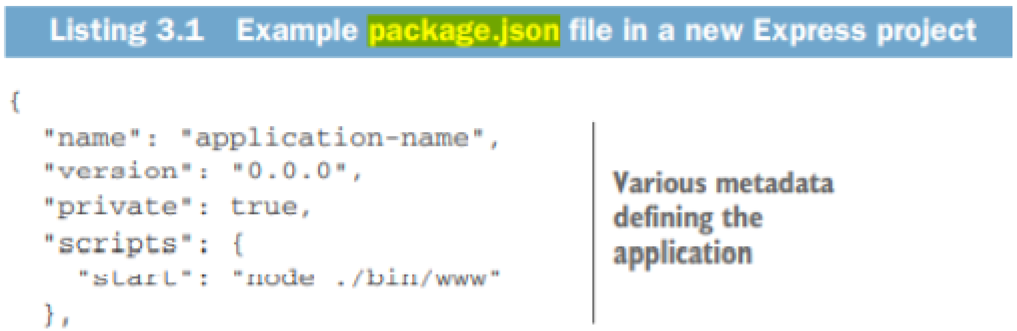
Add review

Welcome to Add review

Figure 4.4 Screenshots of the four routes created so far, with different heading text coming through from the specific controllers associated with each route

The next stage in this prototyping process is putting some HTML, layout, and content on each screen. You'll do this by using views.

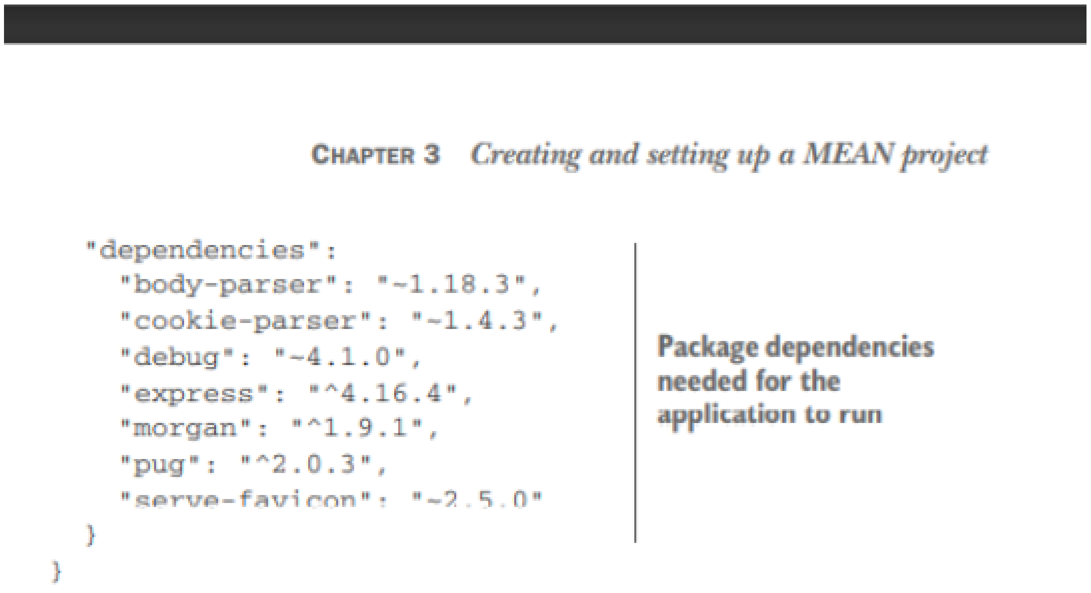
➔ Illustrate the definition of the package.json file.



Listing 3.1 Example package.json file in a new Express project

```
{
  "name": "application-name",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
```

Various metadata defining the application

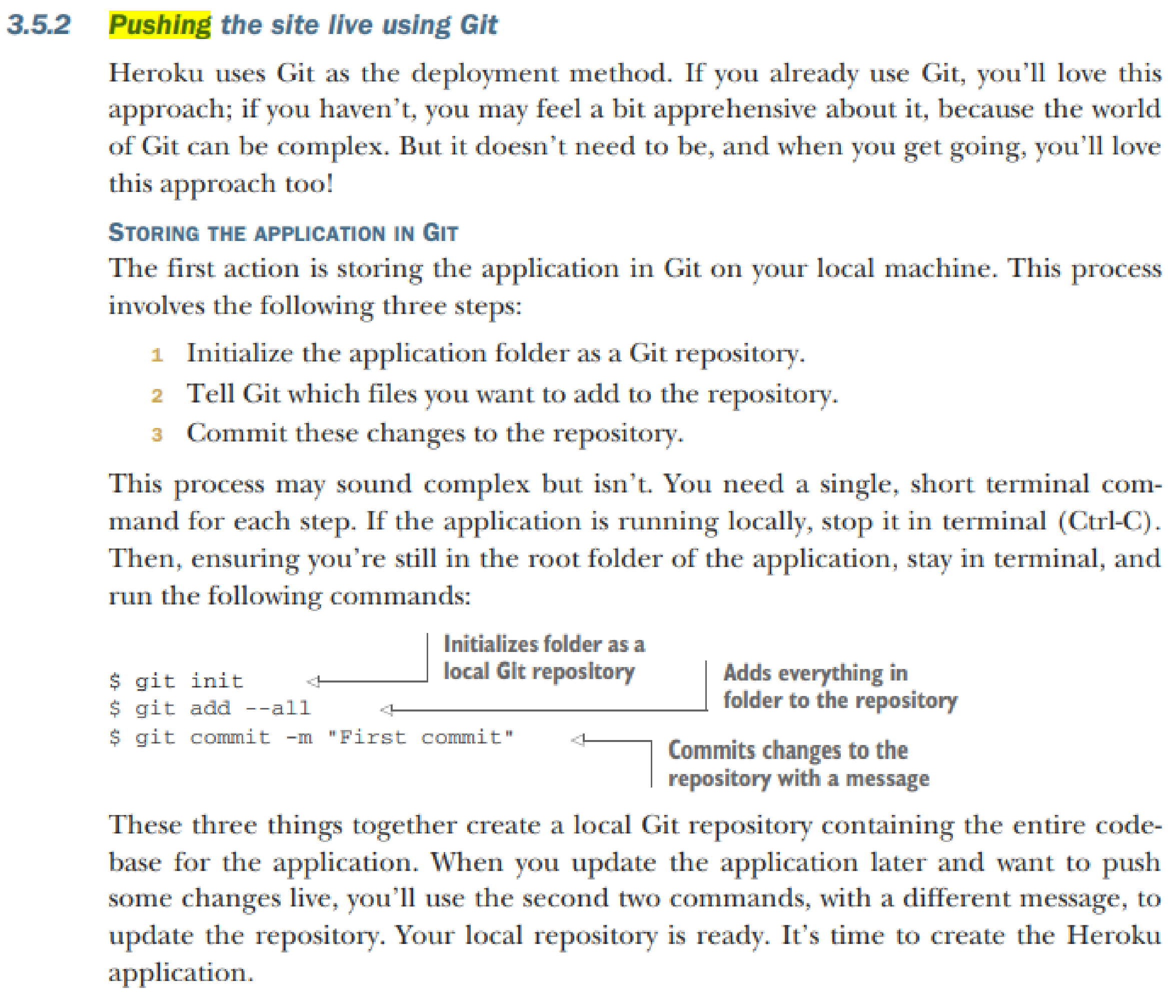


```
  "dependencies":
    "body-parser": "~1.18.3",
    "cookie-parser": "~1.4.3",
    "debug": "~4.1.0",
    "express": "^4.16.4",
    "morgan": "^1.9.1",
    "pug": "^2.0.3",
    "serve-favicon": "~2.5.0"
  }
}
```

Package dependencies needed for the application to run

➔ Explain the steps in pushing the site live using git.

Page No. 77



3.5.2 Pushing the site live using Git

Heroku uses Git as the deployment method. If you already use Git, you'll love this approach; if you haven't, you may feel a bit apprehensive about it, because the world of Git can be complex. But it doesn't need to be, and when you get going, you'll love this approach too!

STORING THE APPLICATION IN GIT

The first action is storing the application in Git on your local machine. This process involves the following three steps:

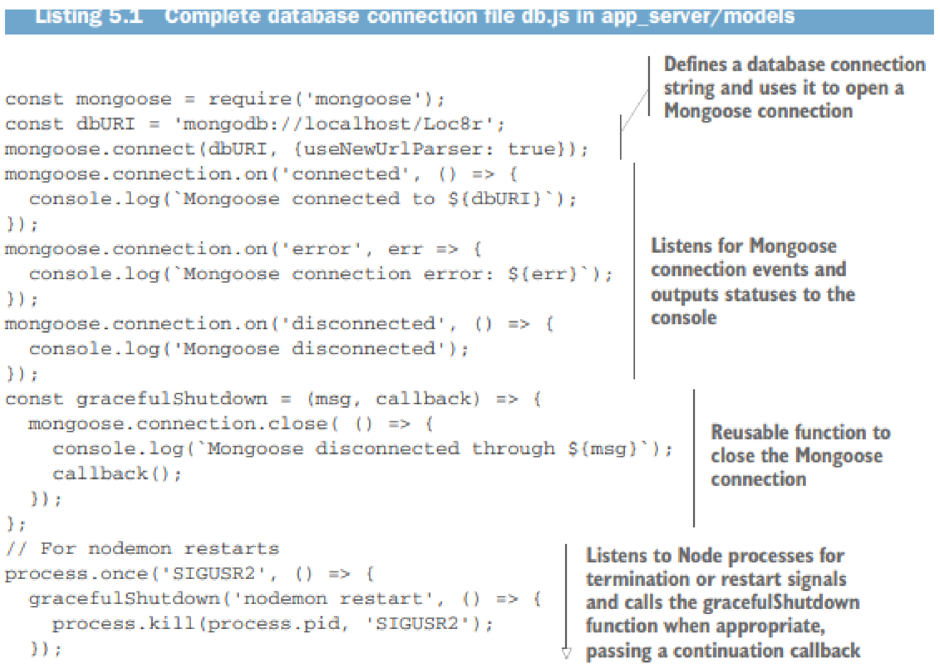
1. Initialize the application folder as a Git repository.
2. Tell Git which files you want to add to the repository.
3. Commit these changes to the repository.

This process may sound complex but isn't. You need a single, short terminal command for each step. If the application is running locally, stop it in terminal (Ctrl-C). Then, ensuring you're still in the root folder of the application, stay in terminal, and run the following commands:

```
$ git init                      <- Initializes folder as a local Git repository
$ git add --all                 <- Adds everything in folder to the repository
$ git commit -m "First commit"  <- Commits changes to the repository with a message
```

These three things together create a local Git repository containing the entire codebase for the application. When you update the application later and want to push some changes live, you'll use the second two commands, with a different message, to update the repository. Your local repository is ready. It's time to create the Heroku application.

➔ Write the code for a complete database connection file using mongoose.
Page No. 127



Listing 5.1 Complete database connection file db.js in app_server/models

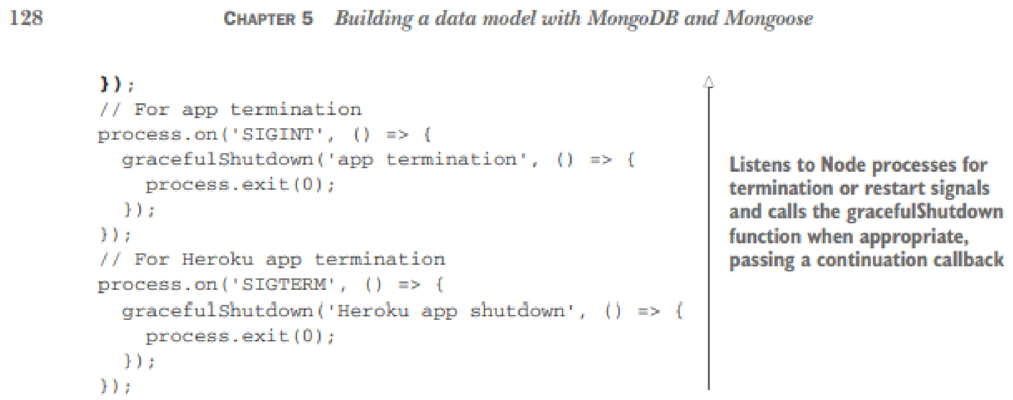
```
const mongoose = require('mongoose');
const dbURI = 'mongodb://localhost/Loc8r';
mongoose.connect(dbURI, {useNewUrlParser: true});
mongoose.connection.on('connected', () => {
  console.log(`Mongoose connected to ${dbURI}`);
});
mongoose.connection.on('error', err => {
  console.log(`Mongoose connection error: ${err}`);
});
mongoose.connection.on('disconnected', () => {
  console.log('Mongoose disconnected');
});
const gracefulShutdown = (msg, callback) => {
  mongoose.connection.close( () => {
    console.log(`Mongoose disconnected through ${msg}`);
    callback();
  });
};
// For nodemon restarts
process.once('SIGUSR2', () => {
  gracefulShutdown('nodemon restart', () => {
    process.kill(process.pid, 'SIGUSR2');
  });
```

Defines a database connection string and uses it to open a Mongoose connection

Listens for Mongoose connection events and outputs statuses to the console

Reusable function to close the Mongoose connection

Listens to Node processes for termination or restart signals and calls the gracefulShutdown function when appropriate, passing a continuation callback



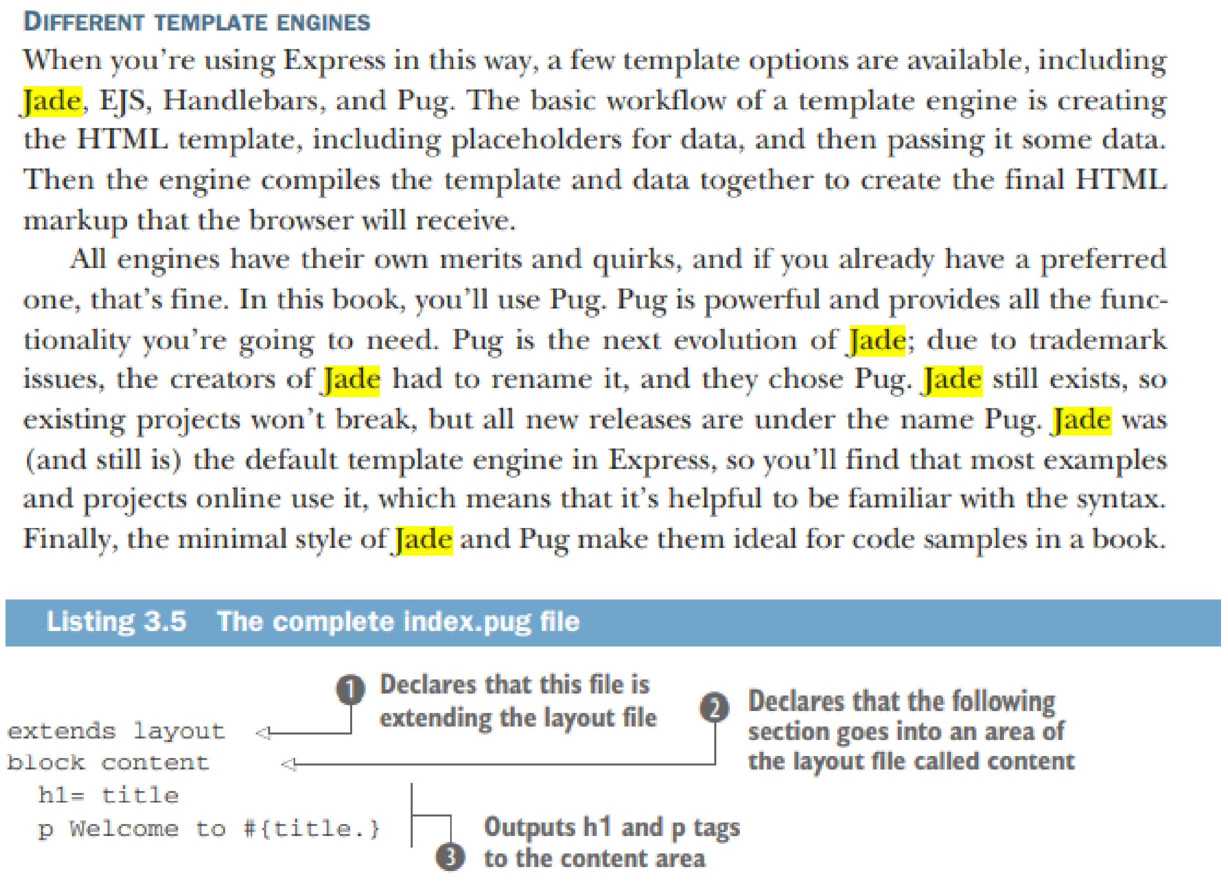
128 CHAPTER 5 *Building a data model with MongoDB and Mongoose*

```
});
// For app termination
process.on('SIGINT', () => {
  gracefulShutdown('app termination', () => {
    process.exit(0);
  });
});
// For Heroku app termination
process.on('SIGTERM', () => {
  gracefulShutdown('Heroku app shutdown', () => {
    process.exit(0);
  });
});
```

Listens to Node processes for termination or restart signals and calls the gracefulShutdown function when appropriate, passing a continuation callback

➔ Explain the usage of jade templates. Also explain the index.jade and layout.jade template files with sample code.

Page No. 60



DIFFERENT TEMPLATE ENGINES

When you're using Express in this way, a few template options are available, including Jade, EJS, Handlebars, and Pug. The basic workflow of a template engine is creating the HTML template, including placeholders for data, and then passing it some data. Then the engine compiles the template and data together to create the final HTML markup that the browser will receive.

All engines have their own merits and quirks, and if you already have a preferred one, that's fine. In this book, you'll use Pug. Pug is powerful and provides all the functionality you're going to need. Pug is the next evolution of Jade; due to trademark issues, the creators of Jade had to rename it, and they chose Pug. Jade still exists, so existing projects won't break, but all new releases are under the name Pug. Jade was (and still is) the default template engine in Express, so you'll find that most examples and projects online use it, which means that it's helpful to be familiar with the syntax. Finally, the minimal style of Jade and Pug make them ideal for code samples in a book.

Listing 3.5 The complete index.pug file

```
extends layout
block content
  h1= title
  p Welcome to #{title.}
```

1 Declares that this file is extending the layout file

2 Declares that the following section goes into an area of the layout file called content

3 Outputs h1 and p tags to the content area

Listing 3.7 Updated layout.pug including Bootstrap references

```
doctype html
html
  head
    meta(name='viewport', content='width=device-width,
      ➥initial-scale=1.0')
    title= title
    link(rel='stylesheet',  href='/stylesheets/bootstrap.min.css')
    link(rel='stylesheet', href='/stylesheets/all.min.css')
```

Sets the viewport metadata for better display on mobile devices

Includes Bootstrap and Font Awesome CSS

```
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
    script(src='https://code.jquery.com/jquery-3.3.1.slim.min.js',
    ➥integrity='sha384-
    ➥q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo',
    ➥crossorigin='anonymous')
    script(src='https://cdnjs.cloudflare.com/ajax/libs/
    ➥popper.js/1.14.3/umd/popper.min.js',integrity='sha384-
    ➥ZMP7rVo3mIykV+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPIPm49',
    crossorigin='anonymous')
    script(src='/javascripts/bootstrap.min.js')
```

Brings in the Bootstrap JavaScript file

Brings in jQuery and Popper, needed by Bootstrap. Make sure that the script tags are all at the same indentation.

➔ Explain how you would change the folder structure of an Express setup to MVC architecture.
Page No. 66

3.3.2 Changing the folder structure

If you look inside the newly created Express project in the loc8r folder, you should see a file structure including a views folder and even a routes folder, but no mention of models or controllers. Rather than cluttering the root level of the application with some new folders, keep things tidy by creating one new folder for all your MVC architecture. Follow these three quick steps:

1. Create a new folder called app_server.
2. In app_server, create two new folders called models and controllers.
3. Move the views and routes folders from the root of the application into the app_server folder.

Figure 3.6 illustrates these changes and shows the folder structures before and after modification.

Now you have an obvious MVC setup in the application, which makes it easier to separate your concerns. But if you try to run the application now, it won't work, as you've just broken it. So fix it. Express doesn't know that you've added some new folders or have any idea what you want to use them for, so you need to tell it.

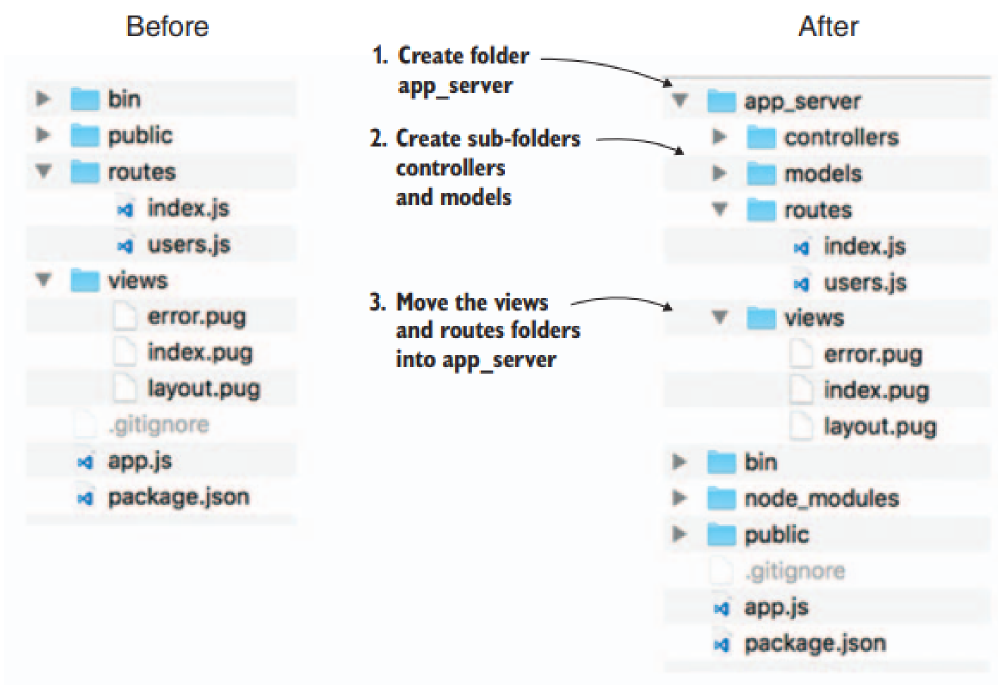


Figure 3.6 Changing the folder structure of an Express project into an MVC architecture

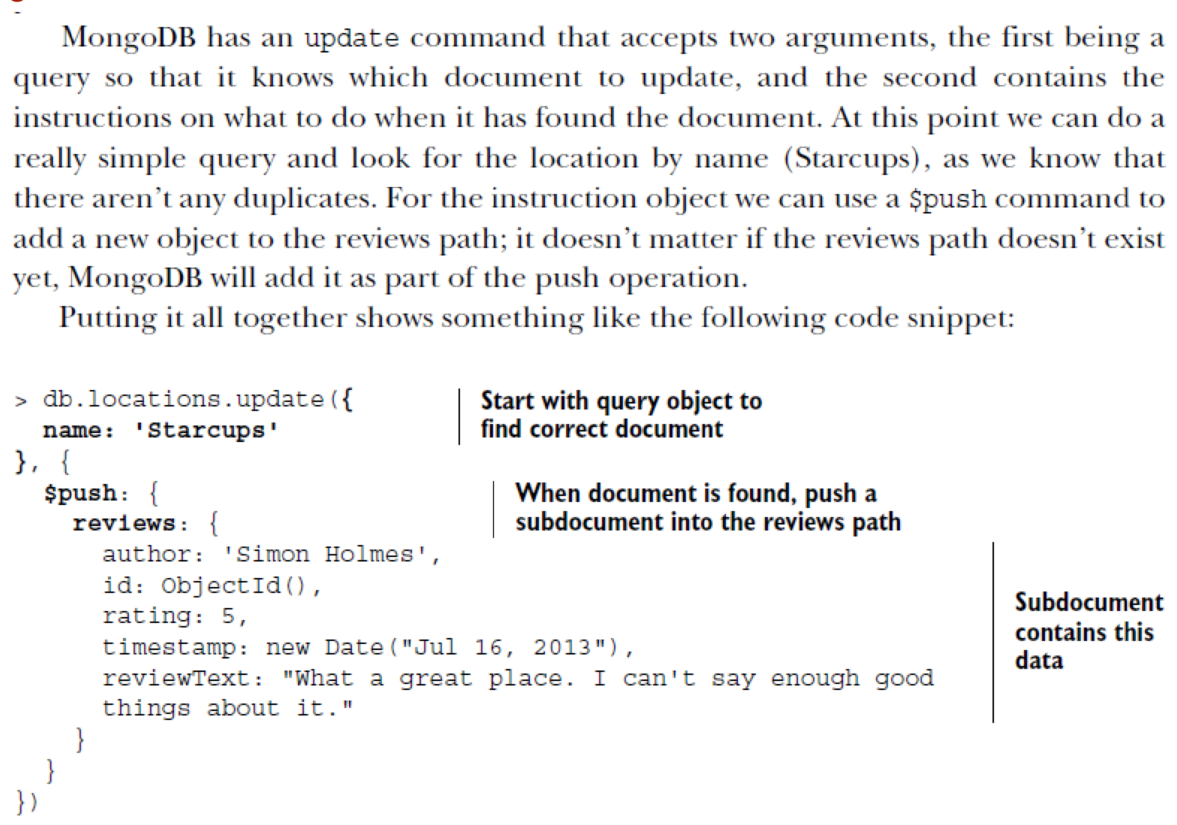
UNIT - 4

➔ Explain how to update an existing subdocument in MongoDB

Updating a subdocument is exactly the same as updating a document, with one exception.
After finding the document you then have to find the correct subdocument to
make your changes. After this, the save method is applied to the document, not the
subdocument. So the steps to updating an existing subdocument are

1 Find the relevant document.
2 Find the relevant subdocument.
3 Make some changes to the subdocument.
4 Save the document.
5 Send a JSON response.

Page 151



MongoDB has an `update` command that accepts two arguments, the first being a query so that it knows which document to update, and the second contains the instructions on what to do when it has found the document. At this point we can do a really simple query and look for the location by name (Starcups), as we know that there aren't any duplicates. For the instruction object we can use a `$push` command to add a new object to the reviews path; it doesn't matter if the reviews path doesn't exist yet, MongoDB will add it as part of the push operation.

Putting it all together shows something like the following code snippet:

```
> db.locations.update({
  name: 'Starcups'
}, {
  $push: {
    reviews: {
      author: 'Simon Holmes',
      id: ObjectId(),
      rating: 5,
      timestamp: new Date("Jul 16, 2013"),
      reviewText: "What a great place. I can't say enough good
      things about it."
    }
  }
})
```

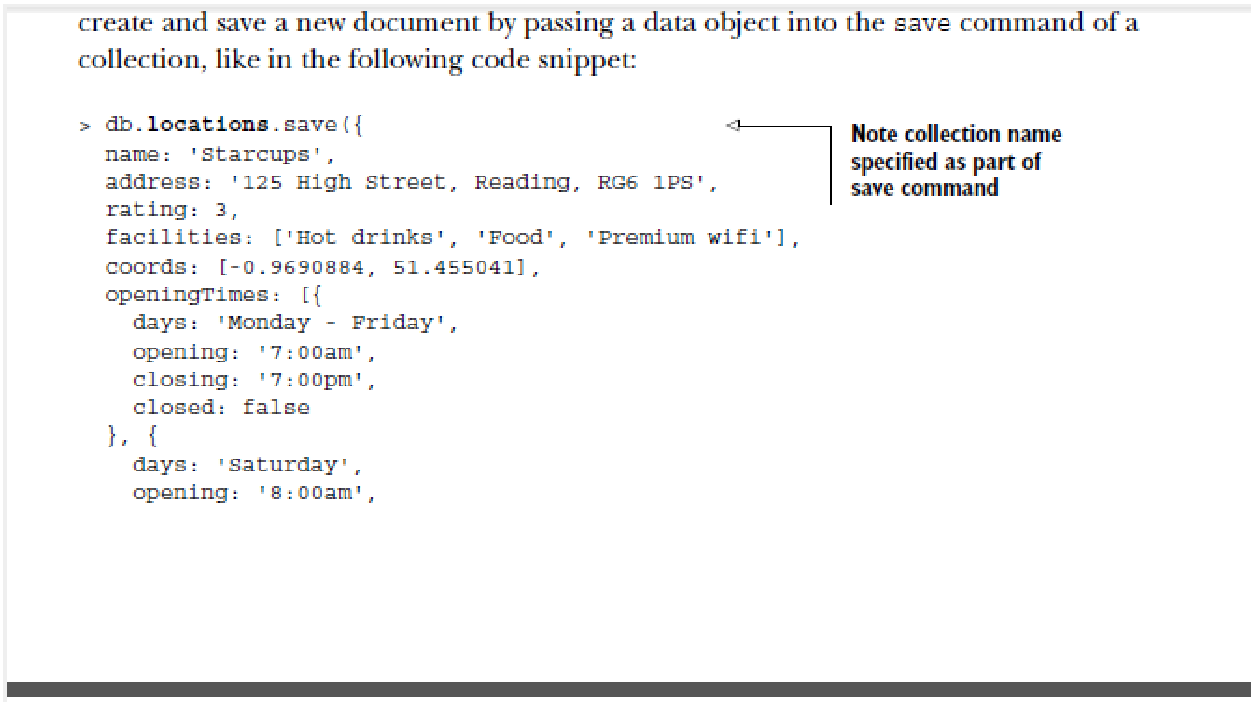
Start with query object to find correct document

When document is found, push a subdocument into the reviews path

Subdocument contains this data

➔ Creation of new document in MongoDB AND subdocument in MongoDB

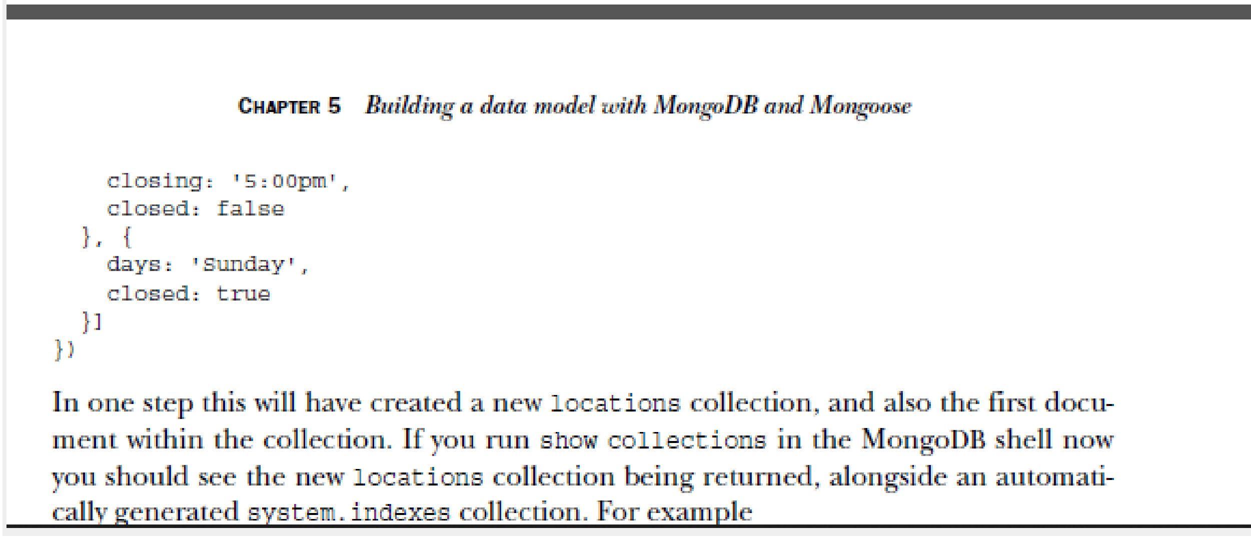
Page 149



create and save a new document by passing a data object into the save command of a collection, like in the following code snippet:

```
> db.locations.save({
  name: 'Starcups',
  address: '125 High Street, Reading, RG6 1PS',
  rating: 3,
  facilities: ['Hot drinks', 'Food', 'Premium wifi'],
  coords: [-0.9690884, 51.455041],
  openingTimes: [{
    days: 'Monday - Friday',
    opening: '7:00am',
    closing: '7:00pm',
    closed: false
  }, {
    days: 'Saturday',
    opening: '8:00am',
```

Note collection name specified as part of save command



```
    closing: '5:00pm',
    closed: false
  }, {
    days: 'Sunday',
    closed: true
  }]
})
```

In one step this will have created a new `locations` collection, and also the first document within the collection. If you run `show collections` in the MongoDB shell now you should see the new `locations` collection being returned, alongside an automatically generated `system.indexes` collection. For example

PTO

Page 141

MongoDB offers the concept of *subdocuments* to store this repeating, nested data. Subdocuments are very much like documents in that they have their own schema and each is given a unique `_id` by MongoDB when created. But subdocuments are nested inside a document and they can only be accessed as a path of that parent document.

USING NESTED SCHEMAS IN MONGOOSE TO DEFINE SUBDOCUMENTS

Subdocuments are defined in Mongoose by using nested schemas. So that's one schema nested inside another. Let's create one to see how that works in code. The first step is to define a new schema for a subdocument. We'll start with the opening times and create the following schema. Note that this needs to be in the same file as the `locationSchema` definition, and, importantly, must be *before* the `locationSchema` definition.

```
var openingTimeSchema = new mongoose.Schema({
  days: {type: String, required: true},
  opening: String,
  closing: String,
  closed: {type: Boolean, required: true}
});
```

➔ Explain validation at schema level with mongoose

Page 136

ADDING SOME BASIC VALIDATION: REQUIRED FIELDS

Through Mongoose you can quickly add some basic validation at the schema level. This helps toward maintaining data integrity and can protect your database from problems of missing or malformed data. Mongoose's helpers make it really easy to add some of the most common validation tasks, meaning that you don't have to write or import the code each time.

The first example of this type of validation ensures that required fields aren't empty before saving the document to the database. Rather than writing the checks for each required field in code, you can simply add a `required: true` flag to the definition objects of each path that you decide should be mandatory. In the `locationSchema`, we certainly want to ensure that each location has a name, so we can update the name path like this:

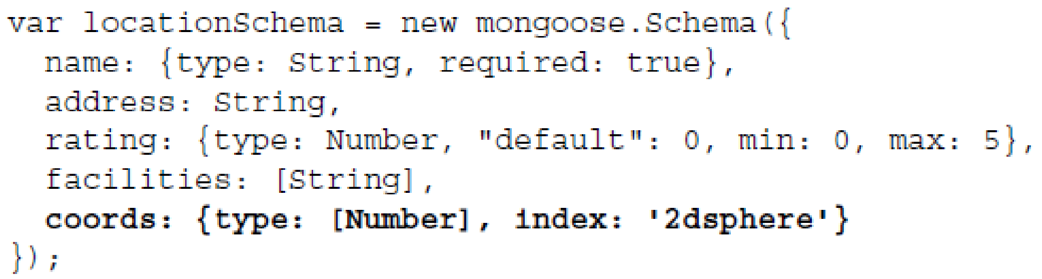
```
name: {type: String, required: true}
```

If you try to save a location without a name, Mongoose will return a validation error that you can capture immediately in your code, without needing a roundtrip to the database.

➔ Define a mongoose schema for a new location in *Loc8r* application.

➔ Write the complete mongoose Schema for a location, with name, address, rating, facilities, opening times, reviews

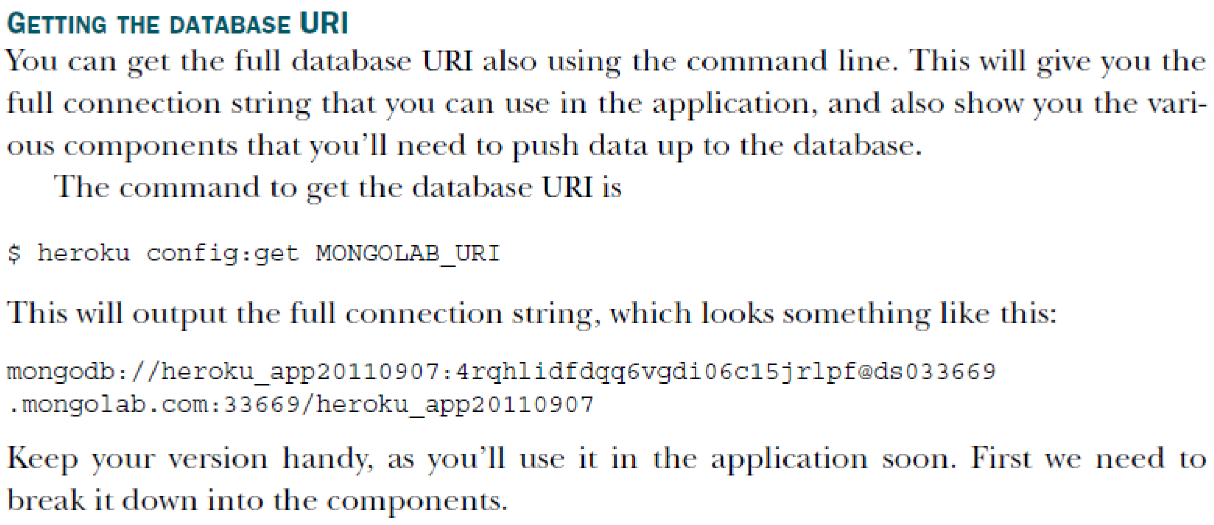
Page 138



```
var locationSchema = new mongoose.Schema({
  name: {type: String, required: true},
  address: String,
  rating: {type: Number, "default": 0, min: 0, max: 5},
  facilities: [String],
  coords: {type: [Number], index: '2dsphere'}
});
```

➔ How to monitor the mongoose connection events? Write database URI for mongoose connection

Page 153



GETTING THE DATABASE URI

You can get the full database URI also using the command line. This will give you the full connection string that you can use in the application, and also show you the various components that you'll need to push data up to the database.

The command to get the database URI is

```
$ heroku config:get MONGOLAB_URI
```

This will output the full connection string, which looks something like this:

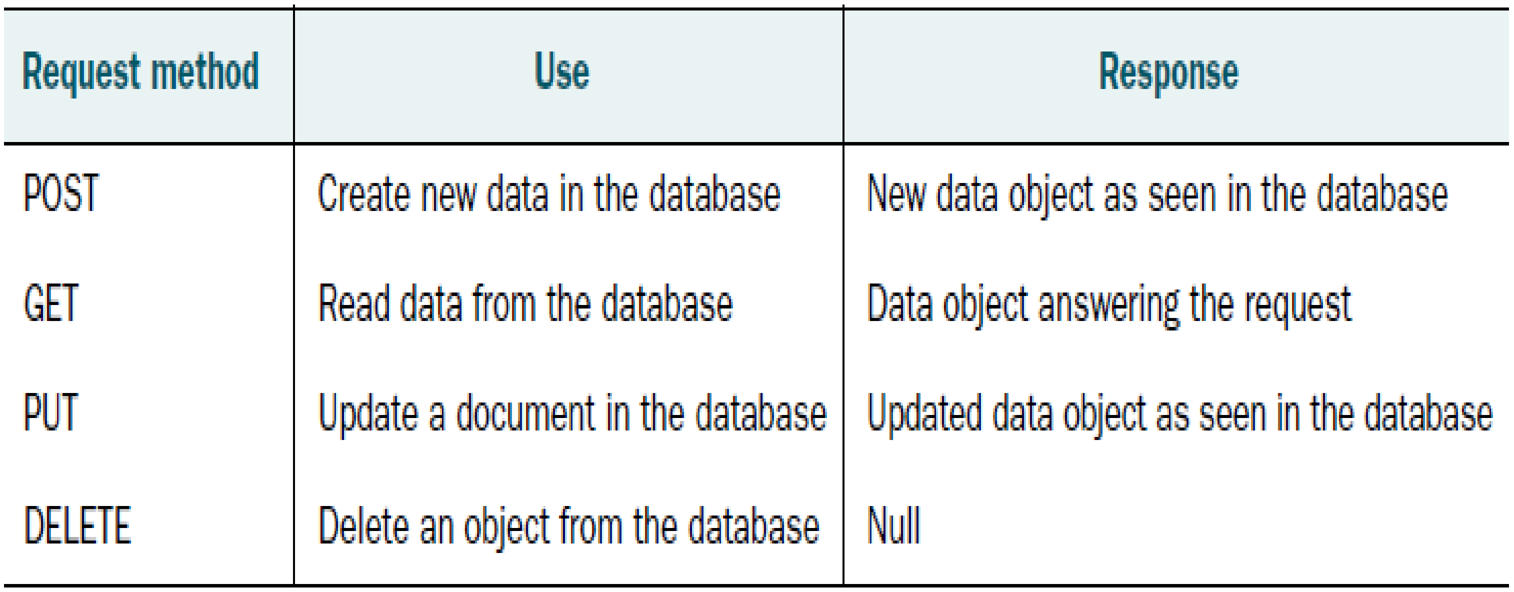
```
mongodb://heroku_app20110907:4rqhlidfdqq6vgdi06c15jrlpf@ds033669
.mongolab.com:33669/heroku_app20110907
```

Keep your version handy, as you'll use it in the application soon. First we need to break it down into the components.

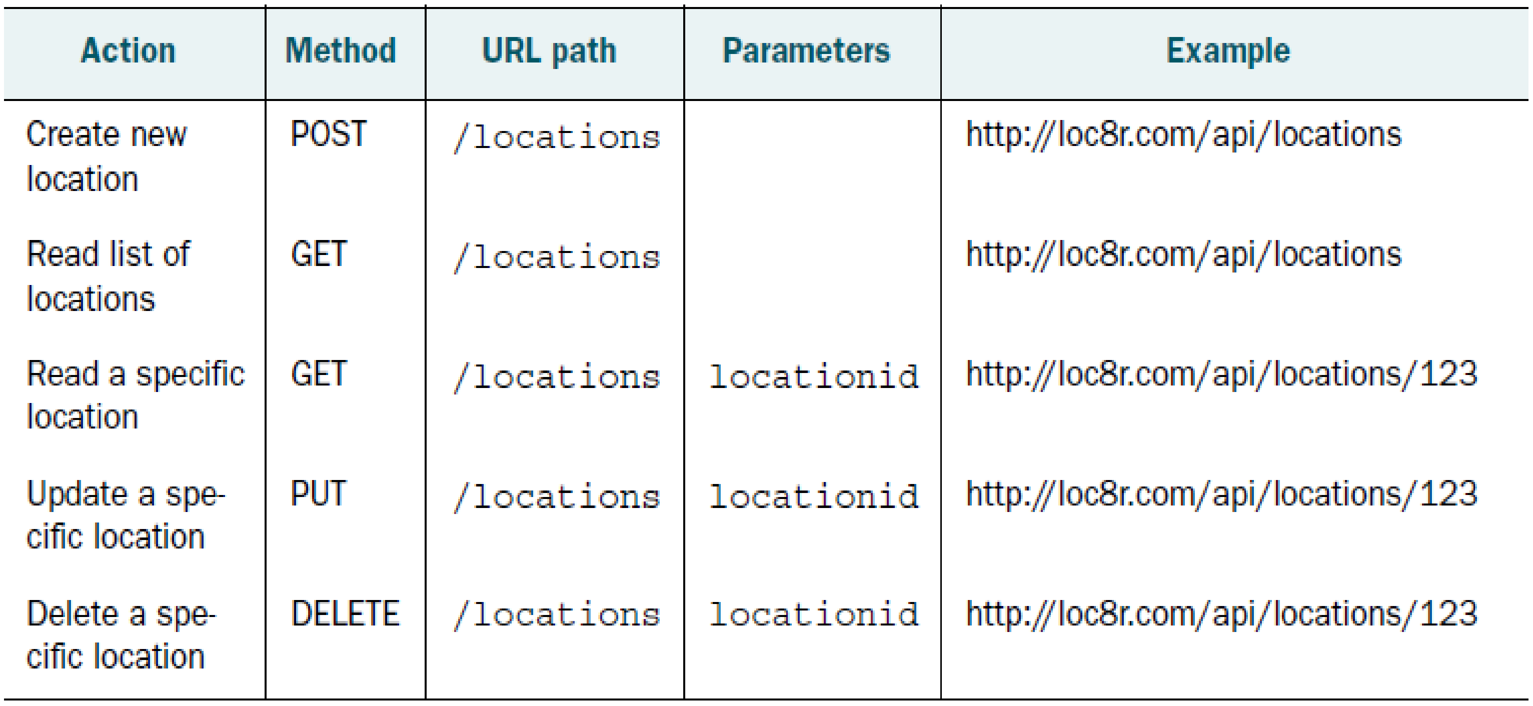
UNIT - 5

➔ Explain the different request methods and any *five* HTTP status codes available for REST API

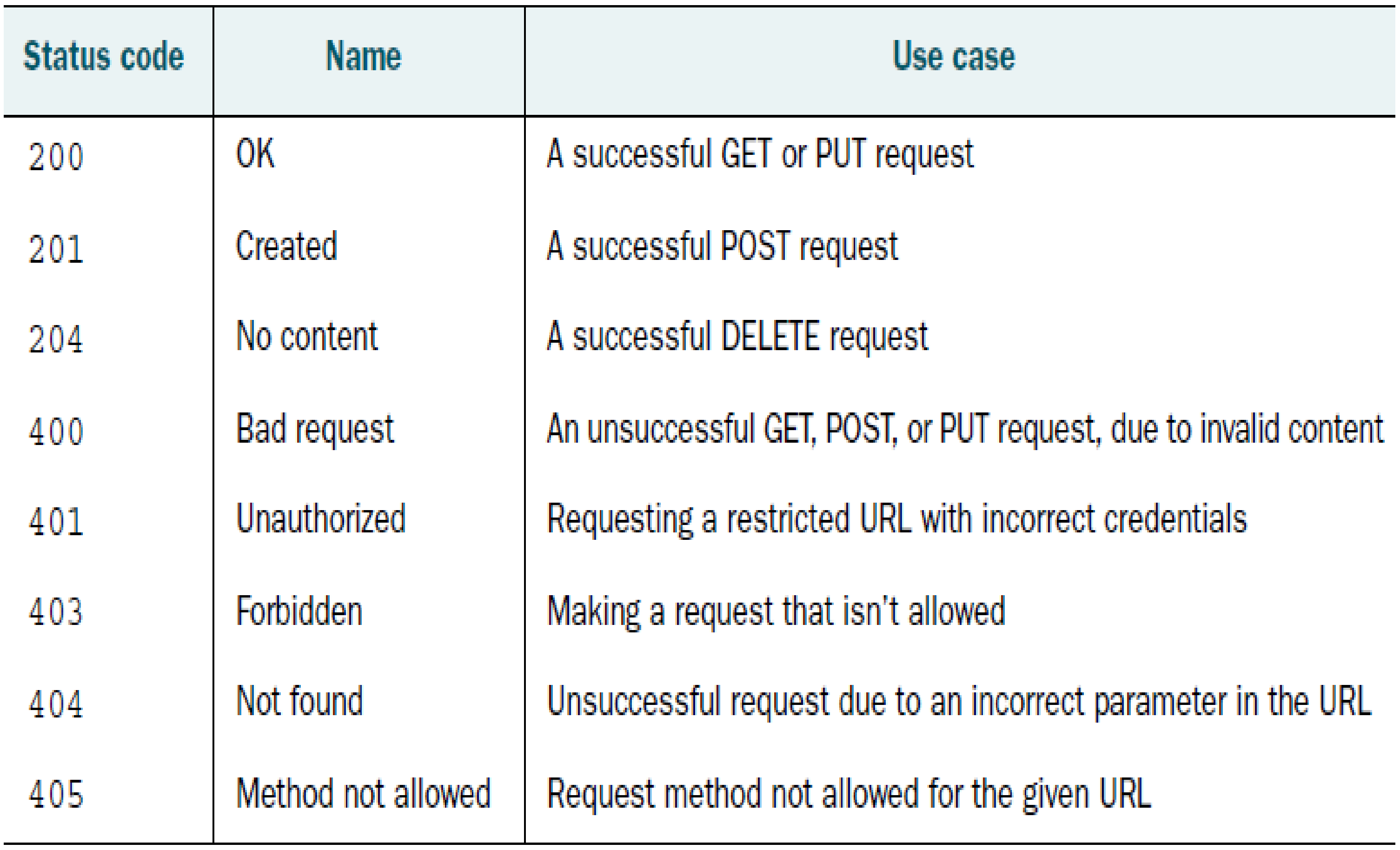
HTTP requests can have different methods that essentially tell the server what type of action to take. The most common type of request is a GET request—this is the method used when you enter a URL into the address bar of your browser. Another common method is POST, often used when submitting form data.



Request method	Use	Response
POST	Create new data in the database	New data object as seen in the database
GET	Read data from the database	Data object answering the request
PUT	Update a document in the database	Updated data object as seen in the database
DELETE	Delete an object from the database	Null



Action	Method	URL path	Parameters	Example
Create new location	POST	`/locations`		http://loc8r.com/api/locations
Read list of locations	GET	`/locations`		http://loc8r.com/api/locations
Read a specific location	GET	`/locations`	`locationid`	http://loc8r.com/api/locations/123
Update a specific location	PUT	`/locations`	`locationid`	http://loc8r.com/api/locations/123
Delete a specific location	DELETE	`/locations`	`locationid`	http://loc8r.com/api/locations/123



Status code	Name	Use case
200	OK	A successful GET or PUT request
201	Created	A successful POST request
204	No content	A successful DELETE request
400	Bad request	An unsuccessful GET, POST, or PUT request, due to invalid content
401	Unauthorized	Requesting a restricted URL with incorrect credentials
403	Forbidden	Making a request that isn't allowed
404	Not found	Unsuccessful request due to an incorrect parameter in the URL
405	Method not allowed	Request method not allowed for the given URL

➔What are the rules of REST API? Explain How REST API processes HTTP requests with proper diagram

REST stands for REpresentational State Transfer, which is an architectural style rather than a strict protocol. REST is stateless—it has no idea of any current user state or history.

■ API is an abbreviation for application program interface, which enables applications to talk to each other.

So a REST API is a stateless interface to your application. In the case of the MEAN stack the REST API is used to create a stateless interface to your database, enabling a way for other applications to work with the data.

REST APIs have an associated set of standards. While you don’t have to stick to these for your own API it’s generally best to, as it means that any API you create will follow the same approach. It also means you’re used to doing things in the “right” way if you decide you’re going to make your API public.

In basic terms a REST API takes an incoming HTTP request, does some processing, and always sends back an HTTP response

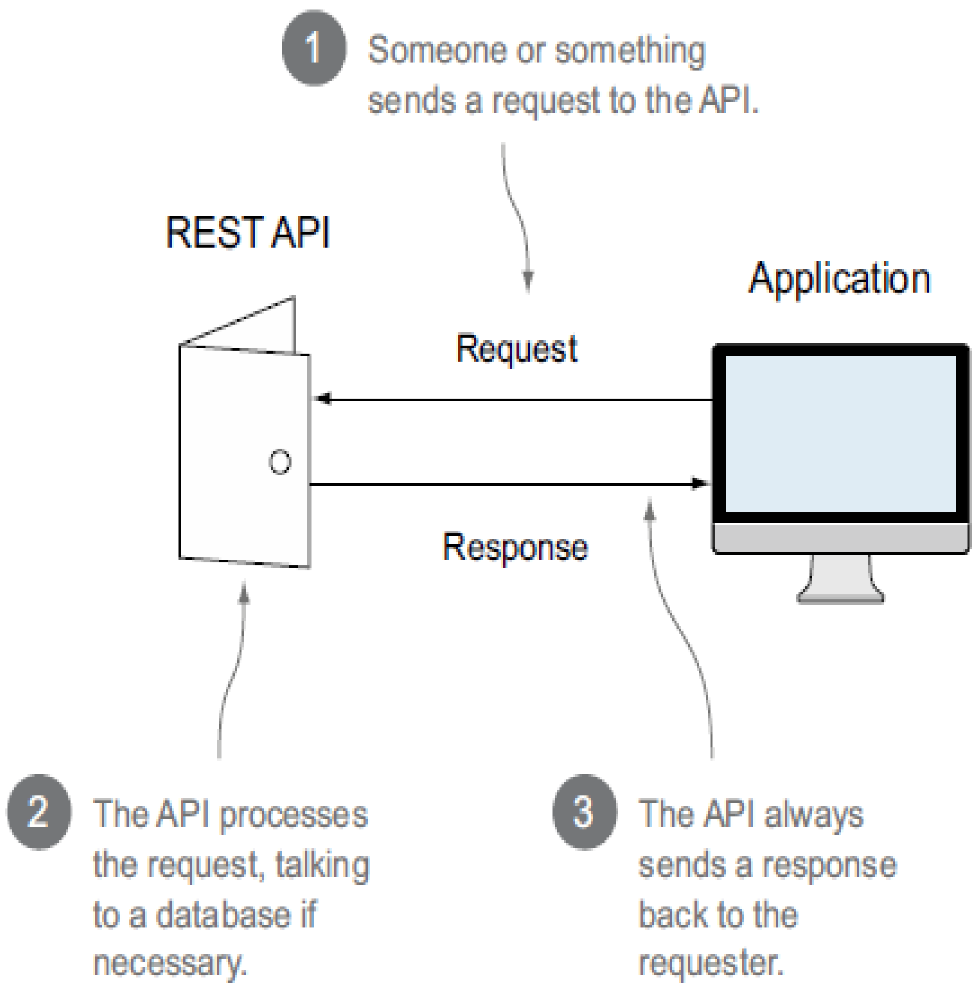

1 Someone or something sends a request to the API.
REST API
Application
Request
Response
2 The API processes the request, talking to a database if necessary.
3 The API always sends a response back to the requester.

Figure 6.2 A REST API takes incoming HTTP requests, does some processing, and returns HTTP responses.

➔How do you find a single document and a sub-document in MongoDB using Mongoose

Mongoose interacts with the database through its models, which is why we imported the Locations model as Loc at the top of the controller files. A Mongoose model has several associated methods to help manage the interactions.
For finding a single database document with a known ID in MongoDB, Mongoose has the findById method.

Mongoose query methods

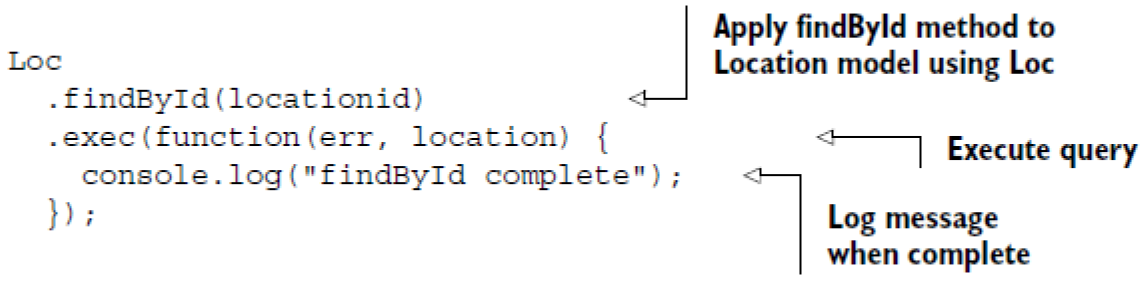
Mongoose models have several methods available to them to help with querying the database. Here are some of the key ones:
■ findGeneral search based on a supplied query object
■ findByIdLook for a specific ID
■ findOneGet the first document to match the supplied query
■ geoNearFind places geographically close to the provided latitude and longitude
■ geoSearchAdd query functionality to a geoNear operation

➔ Explain how to find a single document based in IDs using GET method. Develop the snippet of code by using the findById and exec methods.

The findById method is relatively straightforward, accepting a single parameter, the ID to look for. As it's a model method, it's applied to the model like this:
Loc.findById(locationid)
RUNNING THE QUERY WITH THE EXEC METHOD

The exec method executes the query and passes a callback function that will run when the operation is complete. The callback function should accept two parameters, an error object and the instance of the found document. As it's a callback function the names of these parameters can be whatever you like.

The methods can be chained as follows:



```
Loc
  .findById(locationid)
  .exec(function(err, location) {
    console.log("findById complete");
  });
```

Apply findById method to Location model using Loc
Execute query
Log message when complete

➔ How do you find a single document and a sub-document in MongoDB using Mongoose

To find a subdocument you first have to find the parent document to find a single location by its ID. Once you've found the document you can look for a specific subdocument.

This means that we can take the locationsReadOne controller as the starting point and add a few modifications to create the reviewsReadOne controller. These modifications are

- ■ Accept and use an additional reviewid URL parameter.
- ■ Select only the name and reviews from the document, rather than having MongoDB return the entire document.
- ■ Look for a review with a matching ID.
- ■ Return the appropriate JSON response.

➔ Deleting document in MongoDB

Mongoose makes deleting a document in MongoDB extremely simple by giving us the method findByIdAndRemove. This method expects just a single parameter—the ID of the document to be deleted. The API should respond with a 404 in case of an error and a 204 in case of success.

That's the quick and easy way to delete a document, but you can break it into a two-step process and find it then delete it if you prefer. This does give you the chance to do something with the document before deleting if you need to. This would look like the following code snippet:

```
Loc
  .findById(locationid)
  .exec(
    function (err, location) {
      // Do something with the document
      Loc.remove(function(err, location){
```

```
        // Confirm success or failure
      });
    }
  );
```

➔ Deleting a subdocument from MongoDB

The process for deleting a subdocument is no different from the other work we've done with subdocuments—everything is managed through the parent document. The steps for deleting a subdocument are
1 Find the parent document.
2 Find the relevant subdocument.
3 Remove the subdocument.
4 Save the parent document.
5 Confirm success or failure of operation.
Actually deleting the subdocument itself is really easy, as Mongoose gives us another helper method. You've already seen that we can find a subdocument by its ID with the id method like this:
location.reviews.id(reviewid)
Mongoose allows you to chain a remove method to the end of this statement like so:
location.reviews.id(reviewid).remove()
This will delete the subdocument from the array. Remember, of course, that the parent document will need saving after this to persist the change back to the database.

➔Explain in brief how to call an API from Express using:
(i)Adding the request module
(ii)Setting up default options
(iii)Using the request module

Adding the request module to our project
The request module is just like any of the other packages we've used so far, and can be added to our project using npm. To install the latest version and add it to the package.json file, head to terminal and type the following command:

```
$ npm install --save request
```

When npm has finished doing its thing, we can include request into the files that will use it.

Setting up default options

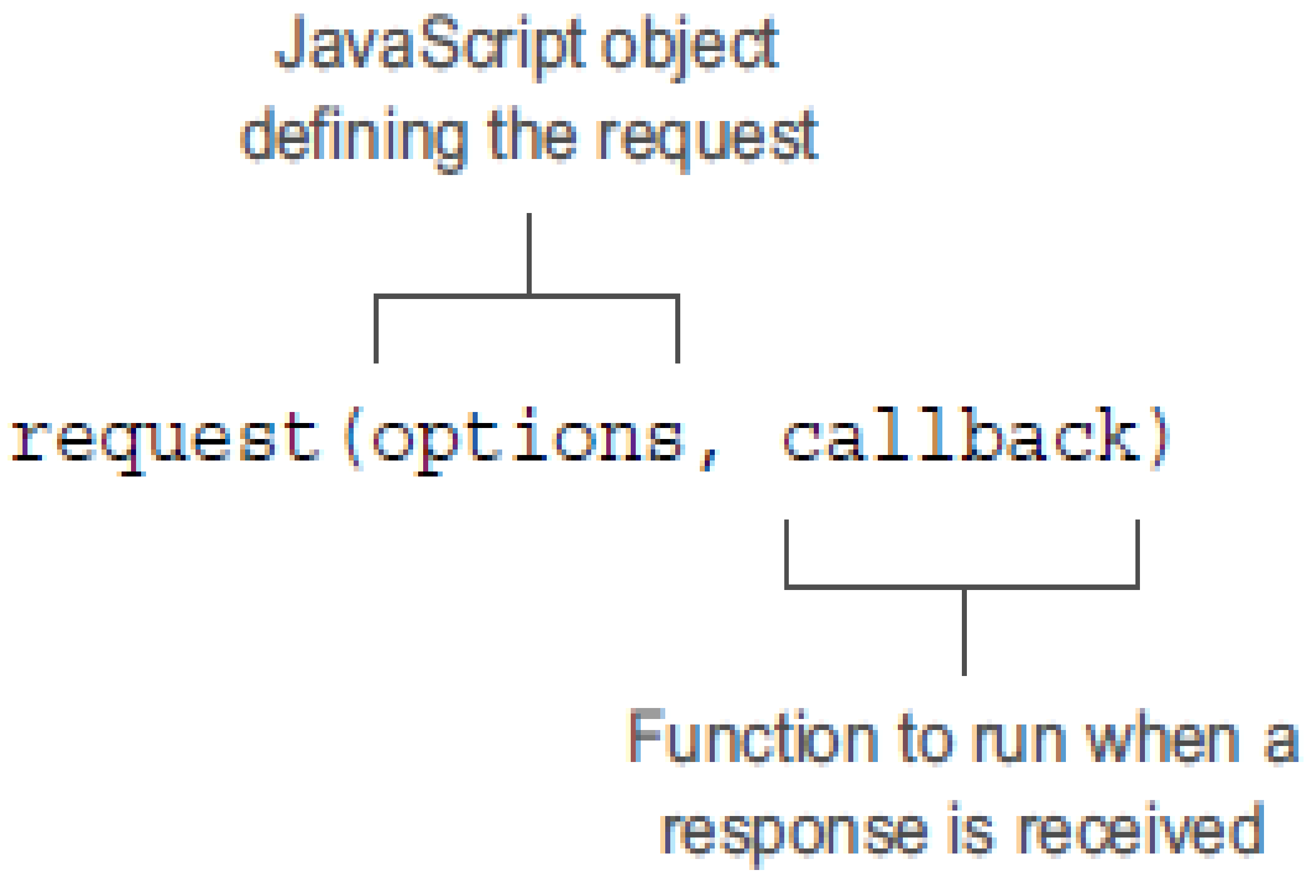
Every API call with request must have a fully qualified URL, meaning that it must include the full address and not be a relative link. But this URL will be different for development and live environments.

To avoid having to make this check in every controller that makes an API call, we can set a default configuration option once at the top of the controllers file. To use the correct URL depending on the environment we can use our old friend the NODE_ENV environment variable.

Putting this into practice, the top of the controllers file should now look something like the following listing.

Using the request module

The basic construct for making a request is really simple, being just a single command taking parameters for options and a callback like this


JavaScript object defining the request
request(options, callback)
Function to run when a response is received

The options specify everything for the request, including the URL, request method, request body, and query string parameters.

➔