

1

The Cloud

You are reading this book because you have some programming skills, and you want to learn how to write programs for “the cloud”. If you can program you can take a problem, figure out how to solve it, and then express that solution as a set of instructions in a programming language a computer can be made to understand.

Now you want to take your problem-solving ability and use it to make “cloud based” solutions. But what is “the cloud”? What does “the cloud” do? And what kinds of problems will you have to solve if you are writing a solution that uses “the cloud”?

In this part we will start to answer these questions. We are going to take look at the origins of the cloud and identify what it is that makes an application “cloud based”. Then we’ll move on to consider how to create and manage services in the cloud. Next, we’ll address security and safety, identifying the issues to worry about and steps you can take to reduce risk when you create for the cloud. Finally, we are going to explore how the TypeScript language builds on JavaScript to make programs more secure and manageable. As we explore each topic, we’ll also take a close

look at JavaScript language features by creating some useful and fun applications.

Chapter 1

Coding for the

cloud

What you will learn

In this chapter we are going to investigate the fundamentals of cloud computing and discover what makes an application “cloud based”. We are also going start our journey with the JavaScript language by exploring how JavaScript functions allow code running in the browser to interact with the JavaScript environment. We’ll see how programs run inside a web browser and how we can interact directly with code running in the browser via the Developer Tools, which will even let us view inside our programs as they run.

I’m assuming that you are familiar with programming but just in case there are things that you don’t know (or I have a different understanding of) I’ve added a glossary at the end of this book. Whenever you see a word highlighted like this: “**computer**” it means that the word is defined in the glossary. If something doesn’t quite make sense to you, go to the glossary and check on my definition of the word that I’m using.

What is the cloud?

The internet now underpins many of our daily activities. Things like booking a table at a restaurant, buying a book, or keeping in touch with our friends are now performed using networked services. Nowadays we refer to these services as "in the cloud". But what is the cloud? What does it do? And how can we use it? Let's start with a look at how things were done in before we had the cloud.

The World Wide Web

The **world wide web** and the **internet** are different things. The internet was invented to make it easy to connect software together over long distances. One early "killer app" for the internet was electronic mail. Internet connected computers acting as "mail servers" managed mailboxes for users, replacing paper messages with digital ones.

The world wide web was created some years after the internet to make it easier to work with documents. Rather than having to fetch and read a paper document you use a **browser** program to load an electronic copy from a **web server**. Documents can contain links to other documents, so that you can follow a reference without having to go and fetch another physical document.

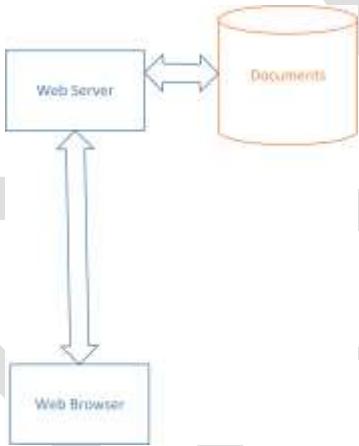


Figure 1.1 Ch01_Fig_01 Browser and Server

Figure 1.1 above shows how it works. The user sits at the browser and sends the web server computer requests for documents which are then sent back to the browser to be read. Later versions of the web added graphics so that a document could contain pictures.

Putting the web in the cloud

If you wanted a web site in the early days of the internet you would set up your own **server** computer. If your site became popular you had to increase the power of your server (or get extra ones) to handle the load. Then you might find that all your capacity was only used at times of peak demand. The rest of the time your expensive hardware was sat twiddling its digital thumbs.

The cloud addresses this problem by turning computing resources into a commodity that can be bought and sold. Rather than setting up your own server, you now rent space in the cloud and pay someone else to host your site. The amount you spend on computer resources is proportional to the demand for the service you are providing. You never have to pay for resources that you don't use. What's more, the cloud makes it possible to create and deploy new services without having to set up expensive servers to make the service available. Most of the popular cloud suppliers even have pricing plans that provide free tariffs to help you get started.

So, the server that you connect to when using a network service (including the world wide web) might be owned by the company providing the service (Facebook have invested considerable sums in setting up their own servers) but it is more likely that you will be connected to a system hosted by one of the cloud providers. Later in the book we will discover how to create an account on a cloud provider and set up a service in the cloud.

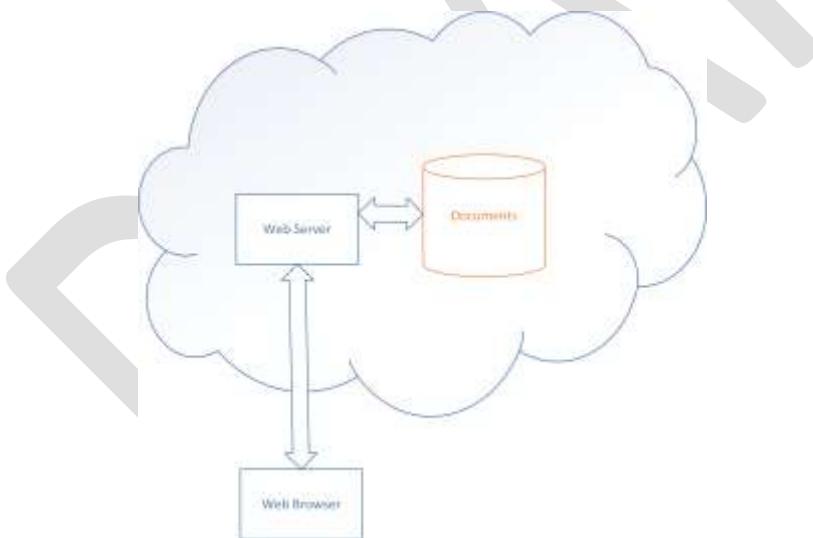


Figure 1.2 Ch01_Fig_02 HTML browser in the cloud

Figure 1.2 shows how this works. The web server and the documents are hosted in the cloud. Note that you could use a mix of different service providers, you could host the resources in one place and the server somewhere else. From the perspective of the service user, a cloud-based web site works in the same way as a server based one.



Figure 1.3 Ch01_Fig_03 Cloud application

Figure 1.3 shows the login page for a service we will be creating in Chapter 11 which connects with Internet of Things devices. My version of the service has the web address: <https://clbportal.azurewebsites.net/> If you enter that address into your browser you will be connected to a process in the cloud hosting this web site.

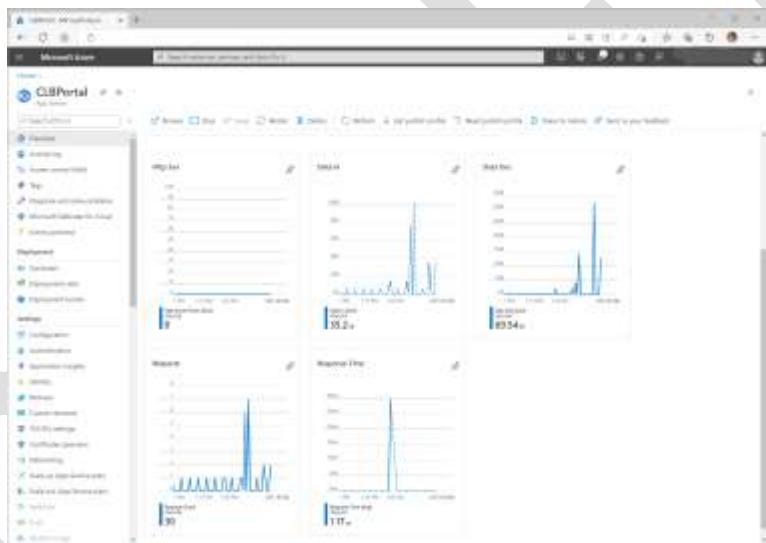


Figure 1.4 Ch01 Fig 04 Cloud management

You can manage your cloud services from the web. Figure 1.4 above shows the overview page on the Azure Portal for the service shown in Figure 1.3. It shows the amount of traffic the page is receiving and the time taken for the page to respond. There are also lots of options for service management and diagnostics. You can also use this page to increase the service provision to support many thousands of users. At the moment this service is using a free service level which supports enough users to allow demonstration and testing.

So, to answer our original questions: The cloud is a means by which companies can provide computing resources as a service. It hides the **physical** location of computing resources behind a **logical** address which users connect to. We can use the cloud to host our own services and make

them available for others to use via web addresses. A cloud service provide will provide a management interface for each service it hosts. Now, let's move on to take a look the JavaScript language.

Programmer's Point

The cloud makes it a great time to be a developer

When I was learning to program it was very nearly impossible to show people what I had done. I could send them my punched cards (each card was punched with holes that contained the text of one line of my code) but it would be unlikely the program would run on the recipient's computer. Today you can invent something and make it available to the whole world by writing some JavaScript and hosting it in the cloud for free. This is tremendously empowering.

Nowadays the hard part is not making your service available but making people aware that it exists. Take a look at the Programmers Points in chapter 2, "Open-Source projects are a great place to start your career" and "GitHub is also a social network" for hints on how to do this.

JavaScript

We now know what the cloud does. It provides a means of buying (or even getting for free) space on the internet where we can host our services. JavaScript has been described as "the programming language for the cloud". Let's look at what this means.

Originally the browser just displayed information that had been received from the server. Then it was decided that it would be useful if the browser could run programs that have been loaded from web sites. Putting a program inside a web page makes the page interactive without increasing the traffic to and from the web server. The user can interact with a program running in the browser without the server having to do anything. The program in the browser can animate the display or check user input to make sure it was correct before sending it to the server.

The language developed to run inside the browser was JavaScript. There have been several different versions of the language. Early ones suffered from a lack of standardization. Browsers from different companies provided different sets of features that were accessed in different ways. But this has now settled down. The specification of the language is now based on a worldwide standard managed by a standards organization called ECMA. We are going to be using version ES6 of the language.

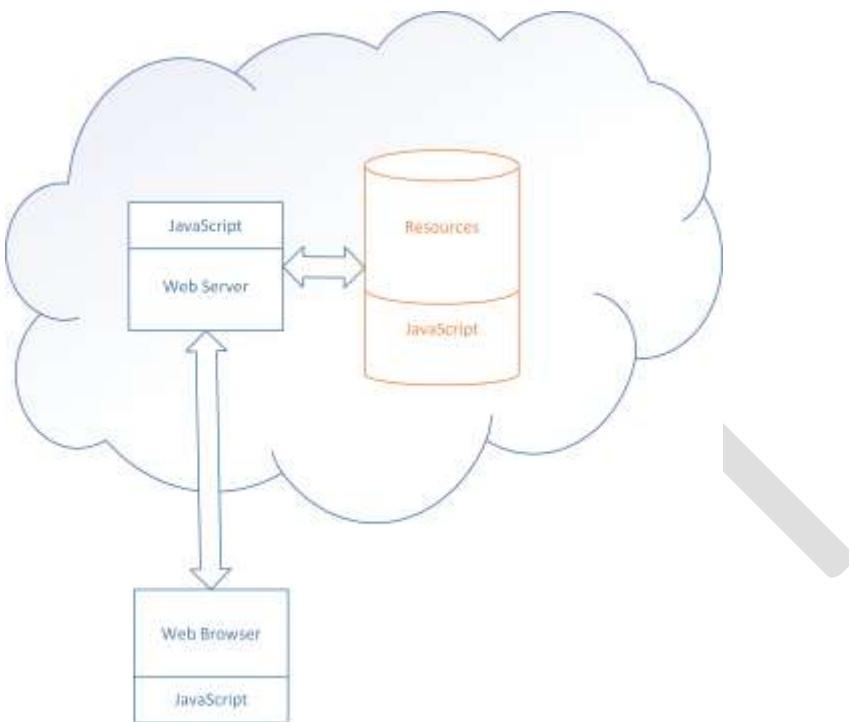


Figure 1.5 Ch01_Fig_05 JavaScript powered web

Figure 1.5 shows how a modern, JavaScript enabled browser and server work together. The web pages and the server program both contain JavaScript elements and the document store has been replaced with a range of resources which can also contain JavaScript code.

JavaScript has become extremely popular. Whenever you visit a website, you will almost certainly be running JavaScript code. JavaScript has also become very popular on web servers (the machines on the internet that deliver the information the browser requests). A technology called “**node.js**” allows JavaScript programs to run on a server to respond to requests from browsers. The web page shown in Figure 1.3 is produced by a JavaScript program running in the cloud using node.js. We’ll discover how to do this later, for the rest of this chapter we are going to be running JavaScript in the web browser. We’ll start by looking at a JavaScript hero, the JavaScript Function.

JavaScript Heroes: functions

This is the first of our “JavaScript heroes”. JavaScript heroes are features of the language which make it super useful for creating cloud applications. A cloud application is all about events. Events happen when the user clicks the mouse button, when a message arrives from a server and when a timer goes tick. In all these situations you need a way of connecting a behavior to what

has just happened. As we shall see, functions in JavaScript simplify the process of connecting code to events. You might think you already know about functions in a programming language. However, I'd advise you to work through this section very carefully anyway. I'm sure that there will be at least one thing here that you didn't know about functions in JavaScript. This is because JavaScript has a very interesting implementation of the function which sets it apart from other programming languages. Let's start with an overview of functions and then drill down into what makes them special in JavaScript.

The JavaScript function object

A JavaScript function is an object that contains a “body” made up of JavaScript statements that are performed when the function is called. A function object contains a name property which gives the name of that function. Functions can be called by their name, at which point the program execution is transferred into the statements that make up the function body. When the function completes the execution returns to statement after the one called the function. Functions can be made to accept values to work on and a function can also return a result value.

```
function doAddition(p1, p2) {  
    let result = p1 + p2;  
    alert("Result:" + result);  
}
```

The code above defines a function with the name `doAddition`. The definition is made up of the header (the part with the name of the function and the parameters it accepts) and the body (the block of two statements that are obeyed when the function is called). The function above calculates the sum of two values and displays the result in an alert box. The `alert` function is one of many “built-in” functions that are provided by the browser Application Programming Interface or API. Learning how to use the facilities provided by the API in a system is a huge part of learning how to be an effective developer.

Programmer's Point

You don't need to know about every JavaScript API function

There are thousands of different functions available to a JavaScript program. The `alert` function is one of them. You might think you need to know about all of them, but actually you don't. I don't know anyone who knows all the functions available to JavaScript programs. But I know lots of people who know how to use search engines to find things when they need them. Don't be afraid to look things up, and don't feel bad about not knowing everything.

```
doAddition(3,4);
```

We call the `doAddition` function as shown above. The values 3 and 4 are called the **arguments**. Argument values are mapped onto the parameters in the function. When the above call of `doAddition` starts to run the parameter `p1` will hold the value 3 and the parameter `p2` will hold the value 4. This leads to the display of an alert box that displays the value 7.



Figure 1.6 Alert box

Figure 1.6 above shows the alert box that is displayed when the function runs. An alert box always has the name of the originator at the top. In this case the function was running in a page on a website located at begintocodecloud.com. The `alert` function is the first JavaScript function that we have seen. It asks the browser to display a message and then waits for the user to click the OK button. From an API point of view, we can say that the `alert` function accepts a string of text and displays it.

Lifting the lid on JavaScript

Wouldn't it be nice if we could watch the `doAddition` function run? It turns out that we can. Modern browsers contain a "Developer Tools Console" which lets you type in JavaScript statements and view the results. How you start the console depends on the browser you are using:

Operating System	Browser	Sequence	Notes
Windows	Edge	F12 or CTRL+SHIFT+J	The first time you do this you will be asked to confirm the action.
Windows	Chrome	F12 or CTRL+SHIFT+J	
Windows	FireFox	F12 or CTRL+SHIFT+J	
Windows	Opera	CTRL+SHIFT+J	
Macintosh	Safari	CMD+OPTION+C	You need to go to Preferences->Advanced->Show Develop Menu and select "Show Develop Menu" to enable it.
Macintosh	Edge	F12 or CTRL+SHIFT+J	
Macintosh	Chrome	CMD+OPTION+J	
Macintosh	FireFox	CMD+SHIFT+J	
Macintosh	Opera	CMD+SHIFT+J	
Linux	Chrome	F12 or CTRL+SHIFT+J	This also works with the Chromium browser on the

Raspberry Pi.

The table above gives the shortcut keys for different browsers and operating systems. Note that the console will look slightly different on each browser, but the views that we are going to use are present on all of them. Let's do our first "Make Something Happen" and use the Developer Tools console to explore functions from our web browser.

Make Something Happen

Explore functions in the console

All the example code for this book is available in the cloud. Of course. You can find the sample pages at beginintocodecloud.com. Open this web site and scroll down to the Samples section on the page.

Begin to Code: Building Applications and Games with the Cloud

Welcome to the Begin to Code: Building Applications and Games with the Cloud web pages. You will be able to run any of the examples in your browser and investigate what they do. You can find all the sample code and resources in the GitHub repository for the book, which [here](#).

Sample Code

This is the sample code for all the chapters. There are references to the examples in each chapter. Open the link to see the web page containing the example program. The link contains executable instructions for getting your own copy of the sample code from GitHub. We can visit any of the sample web applications by clicking one of the links below.

• Ch01-What_is_the_cloud/Ch01-01_Explore_functions_in_the_console

Ch01_inset01_01 Web Site Samples

The sample code is presented as a list of links. There is a link for each sample page. We are going to do the very first sample so click [Ch01-What_is_the_cloud/Ch01-01_Explore_functions_in_the_console](#).

Make Something Happen : Explore functions in the console

This exercise demonstrates how functions are called, and how functions are actually objects in JavaScript. The JavaScript in this page contains some functions that you can call from the browser Developer Tools console. The Developer Tools Console is available in the Edge, Chrome and Chromium browsers.

To open Developer Tools in the browser press the F12 key. If your keyboard does not have that key, hold down the CTRL and SHIFT keys and press J. The console will appear on the right of the page. You may need to confirm that you want to open it.

You can find the steps to follow for the exercise in Chapter 1 of the book in the section "Make Something Happen - Explore functions in the console".

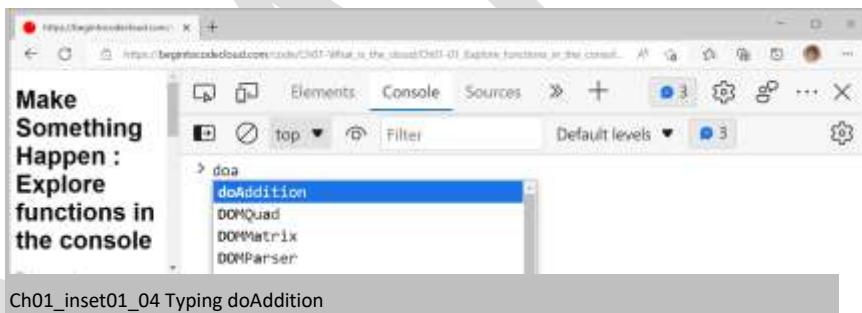
Ch01_inset01_02 Explore Functions

This is the web page for this exercise. Press the key sequence to open the Developer Tools in your browser. The screenshots for this section are from the Edge browser running in Windows.

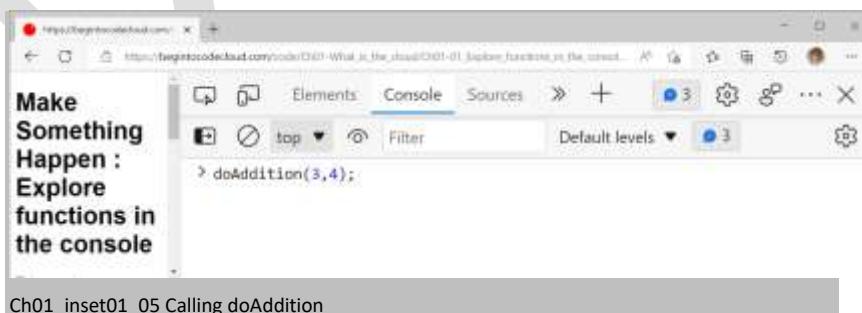


The Developer Tools will open on the right-hand side of your page. You can make the tools area larger by clicking the line separating the sample code from the tools and dragging it to the left as shown above. The web page will automatically resize. The Developer Tools contain several different tabs. We will look at the Elements and Sources tabs later. For now, click the Console tab to open the console.

We can type JavaScript statements into the console, and they will be performed in the browser and the results displayed. The console provides a ‘’ prompt which is where you type commands. Click the console and start typing “doAddition” and watch what happens.



As you type the text the console presents you with a menu of things you could type in. This is a very useful feature. It saves you typing, and it makes it less likely that you will type something incorrectly. You can move up and down the menu of items by using the arrow keys or the mouse. Click [doAddition](#) in the list or press the TAB key when it is selected. Now fill in the rest of the function call by adding 3 and 4, the two arguments:



When you have finished the console should look like the above. Now press enter to run the function.

A screenshot of a web browser window. The address bar shows the URL https://beginccodecloud.com/codes/Ch01-What_is_the_code/Ch01-01_Explore_functions_in_the_console. The main content area displays the text "Make Something Happen : Explore functions in the console". On the right, there is a developer tools sidebar with tabs for "Elements", "Console", and "Sources". The "Console" tab is active, showing the command `> doAddition(3,4);` and its result `< undefined`. A small alert dialog box is overlaid on the page, containing the text "Ch01_inset01_06 doAddtion alert".

The `alert` function has control and is displaying the alert box containing the result. Note that you can't enter new statements into the console at this point. The only thing you can do next is press the OK button in the alert to clear it. Do this.

A screenshot of a web browser window, identical to the previous one but with a key difference. The alert dialog box from the previous screenshot is now gone, replaced by a standard browser message box with an "OK" button. The text "Ch01_inset01_07 doAddtion completed" is displayed in a grey box at the bottom left. The developer tools sidebar on the right shows the command `> doAddition(3,4);` and its result `< undefined`.

When you click on OK the alert box disappears and the `doAddition` function completes. You can now enter further commands in the console.

CODE ANALYSIS

Calling functions

A code analysis section is where we examine something that we have just seen and answer some questions you might have about it. There are a few questions you might have about calling functions in JavaScript. Keep the browser open and displaying the console so you can find out the answers.

Question: What does the `undefined` message mean in the console after the call of `doAdditon`?

The console takes a JavaScript statement, executes it, and displays the value generated by the statement. If the statement calculates a result the statement will have the value of that

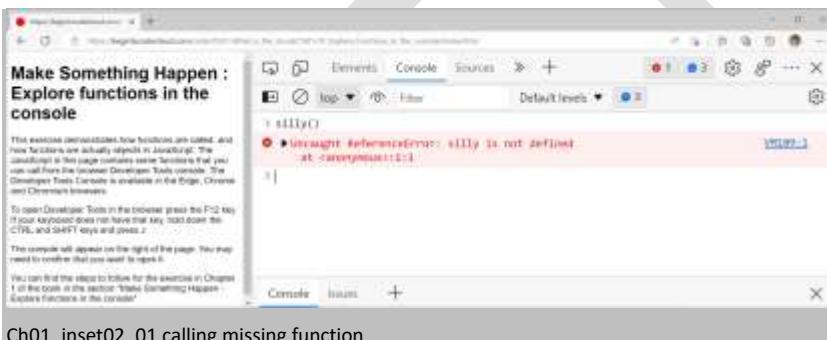
result. This means you can use the console as a calculator. If you type in `2+2` the console will display 4.

```
> 2+2  
< 4
```

The expression `2+2` is a valid JavaScript statement that returns the value of the calculation result. So the console displays 4.

However, the `doAddition` function does not deliver a result. It has no value to return, so it returns a special JavaScript value called **undefined**. Later we will discover JavaScript functions that do return a value.

Question: What happens if I try to call a function that is not available?



Above you can see what happened when I tried to call a function called "silly". JavaScript told me that the function has not been defined.

Question: What would happen if I added two strings together?

In JavaScript you can express a **string** of text by enclosing it in double quotes or single quotes. We will discuss JavaScript in detail strings later in the text.

```
> doAddition("hello","world");
```

The call of `doAddition` above has two string arguments. When the function runs the value of `p1` is set to "hello" and the value of `p2` is set to "world". The function applies the `+` operator between the parameters to get the result.

```
let result = p1 + p2;
```

Above you can see the statement in the `doAddition` function that calculates the value of the result of the function. The statement defines a variable called `result` which is then set to sum of the two parameters. We will be looking at the `let` keyword later in the book (or you can look up `let` in the Glossary). The JavaScript selects a `+` operator to use according to the **context** of the addition. If `p1` and `p2` are numbers JavaScript will use the numeric version of `+`.

If `p1` and `p2` are strings of text JavaScript will use the string version of `+` and set the value of `result` to a string containing the text “helloworld”. You might find it interesting to try adding strings to numbers and watching what JavaScript does. If you look in the `doAddition` function itself you will find a statement does this.

Question: What happens if I subtract one string from another?

Adding two strings together makes sense but subtracting one string from another is not sensible. We can investigate what happens if we do this because the web page for this exercise contains a function called `doSubtraction`. Normally you would give this function numeric arguments. Let’s discover what happens if we use text.

```
> doSubtraction("hello", "world");
```

If you make the above call of `doSubtraction` you get the following message displayed by the alert:

```
Result: Nan
```

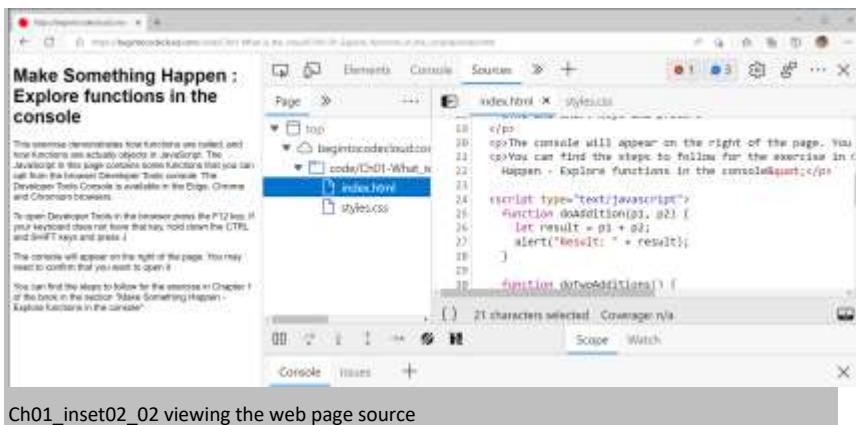
The value “Nan” means “not a number”. There is only one version of the `-` operator, the numeric one. However, this can’t produce a number as a result because it is meaningless to subtract one string from another. So the result of the operation is to set the value of `result` to a “special” value **NaN** to indicate that the result is not a number. Later we’ll discover more about the special values in JavaScript programs.

Question: Why do some strings have " around them and some have '.

When the debug console shows you a string value it will enclose the string in single quote characters. However, in some parts of the program strings are delimited by double quote characters. In JavaScript you can use either double or single quotes to mark the start and end of a string.

Question: Where do these functions come from?

That’s a good question. The function statements are in the web page loaded by the browser from the begintocodecloud.com server. We can use the Developer Tools to view this file. We must change the view from Console to Source to do this. Click the Sources tab which is next to the Console tab on the top row.



Ch01_inset02_02 viewing the web page source

The Sources view shows you all the files behind the website that you are visiting. This site has two files; a styles.css file which contains style definitions (of which more in the next chapter) and an index.html file which contains the text of the web page, along with the JavaScript programs. If you select the index.html file as shown above, you will see the contents of the file including the JavaScript for `doAddition`. There is another function called `doTwoAdditions` which calls `doAddition` twice.

Question: Can we watch the JavaScript run?

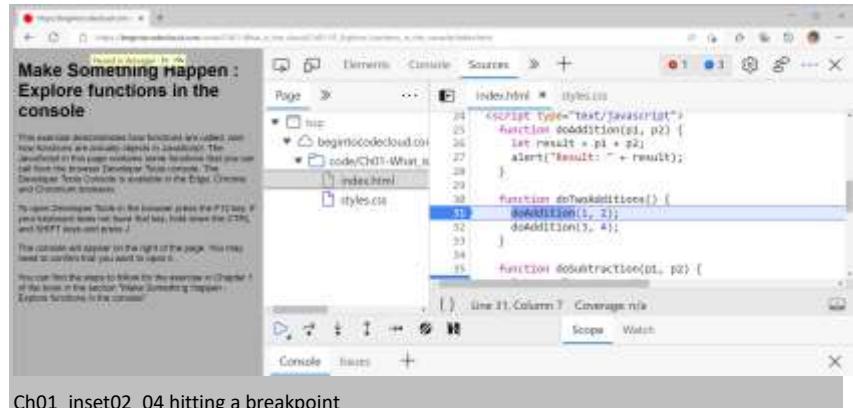
Yes. We can set a “breakpoint” at a statement and when that statement is reached the program will pause and we can go through it one step at a time. This is a wonderful way to see what a program is doing. Put a breakpoint at the first statement of `doTwoAdditions` by clicking in the left margin to the left of the line number:

```
29  
30  
31 function doTwoAdditions() {  
32     doAddition(1, 2);  
33     doAddition(3, 4);  
34 }
```

Ch01_inset02_04 setting a breakpoint

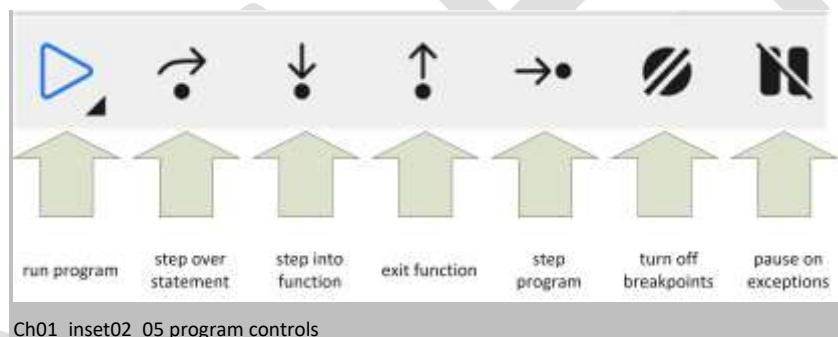
The breakpoint is indicated by the arrow highlighting the line number. It will cause the program to pause when it reaches line 31 in the program. Now we need to call the `doTwoAdditions` function. Select the Console tab, type in the following and press enter:

> doTwoAdditions();



Ch01_inset02_04 hitting a breakpoint

Above you can see what happens when a breakpoint is hit. The browser shows the program paused at the statement with the breakpoint. The most interesting part of this view is the control buttons you can see towards the bottom of the page:



Ch01_inset02_05 program controls

These controls might look a bit like cave paintings, but they are very useful. They control how the browser will work through your program. The one that we will use first is “step into function” (the one third from the left). Each time you press this control the browser will perform one statement in your program. If the statement is a function call the browser will step into the function. You can use the “step over statement” control to step over function calls and the “exit function” to leave a function that you have just entered. Press “step into function”.

The screenshot shows a browser's developer tools with the "Sources" tab selected. A file named "index.html" is open, containing the following code:

```
22    Happen - Explore functions in the console".</p>
23
24    <script type="text/javascript">
25        function doAddition(p1, p2) { p1 = 1, p2 = 2
26            let result = p1 + p2;
27            alert("Result: " + result);
28        }
29
30        function doTwoAdditions() {
31            doAddition(1, 2);
32            doAddition(3, 4);
33        }

```

A blue arrow-shaped cursor is positioned at line 31, indicating the current execution point. The line "let result = p1 + p2;" is highlighted in yellow, indicating it is the next statement to be executed. The status bar at the bottom of the debugger window displays the text "Ch01_inset02_06 viewing program execution".

The highlighted line has now moved to the first statement of the `doAddition` function. The debugger shows you the values in the parameters. If you keep pressing the “step into function” button in the control panel you can see each statement obeyed in turn. Note that you will have to click the OK button in the alert when you perform the statement on line 27 that calls `alert`. If you get bored, you can press the “run program” button at the left of the program controls to run the program. You can clear the breakpoint at line 31 by clicking it.

You can add as many breakpoints as you like and you can use this technique on any web page that you visit. It is interesting to see just how much complexity that there is behind a simple site. Leave the browser open at this page, we will be making some more things happen in the following text.

References to JavaScript function objects

We have seen that function in a JavaScript program is function is represented by a JavaScript function object. A JavaScript function **object** is managed by **reference**. Our programs can contain reference variables that can be made to refer functions.

```
function doAddition(p1, p2) {
    let result = p1 + p2;
    alert("Result:" + result);
}
```

We've seen the definition above before. It defines a function called `doAddition` that adds two parameters together and display the result. When JavaScript sees this it creates a function object to

represent the function and creates a variable called `doAddition` that refers to the function object.

```
doAddition(1,2);
```

The statement above calls the `doAddition` function which will display an alert containing a result of 3. JavaScript variables can contain references to objects so you can write statements like this in your program:

```
let x = doAddition;
```

This statement creates a variable called `x` and makes it refer to the same object as the `doAddition` function.

```
x(5,6);
```

This statement calls whatever `x` is referring to and passes it the arguments 5 and 6. This would call the same object that `doAddition` is referring to (because that is what `x` is referring to) resulting in an `alert` with the message “Result:11” being displayed. We can make the variable `x` refer to a different function.

```
x = doSubtraction;
```

The above statement only works if we have previously declared a function called `doSubtraction` which performs subtraction. The statement makes `x` refer to this function.

```
x(5,6);
```

When the statement above calls `x` it will run the `doSubtraction` function and display a result value of -1, because that is the result of subtracting 6 from 5. We can do evil things with function references. Consider the following statement:

```
doAddition = doSubtraction;
```

If you understand how evil this statement is, you can call yourself a “function reference ninja”. It is completely legal JavaScript that means that from now on a call of `doAddition` will now run the `doSubtraction` function.

Function expressions

You can create JavaScript functions in places where you might not expect to be able to. We are used to setting variables by assigning **expressions** to them:

```
let result=p1+p2;
```

The above statement assigns the expression `p1+p2` to a variable called `result`. However, you can also assign a variable to a *function expression*:

```
let codeFunc = function (p1,p2){ let result=p1+p2; alert("Result: "+result);};
```

The above statement creates a function object which does the same as the `doAddition` function we have been using. I’ve put the entire function on a single line but the statements are exactly the same. The function is referred to by a variable called `codeFunc`. We can call `codeFunc` in the same way as we used `doAddition`. The statement below calls the new function and would display a result of 17

```
codeFunc(10,7);
```

Function references as function arguments

This is probably the most confusing section title so far. Sorry about that. What we want to do is look at how you can pass function references into functions. In other words, a program can tell a function which function to call. Later in this chapter we are going to tell a timer which function to call when the timer goes tick. This is how it works.

An **argument** is something that which is passed into a function when it is called. We have passed two arguments (**p1** and **p2**) into the **doAddition** function each time we call it (these are the items to be added). We can also use references to functions as arguments to function calls.

```
function doFunctionCall(functionToCall, p1, p2){  
    functionToCall(p1,p2);  
}
```

The function **doFunctionCall** above has three parameters. The first (**functionToCall**) is a function to call, the second (**p1**) and third (**p2**) are values to be passed into that function when it is called. All that **doFunctionCall** does is call the supplied function with the given arguments. It's not particularly useful, but it does show that you can make a function that accepts function a reference as a parameter. We could call **doFunctionCall** like this:

```
doFunctionCall(doAddition,1,7);
```

This statement calls the **codeFunc** function. The first argument to the call is a reference to **doAddition**, the second argument is 1 and the third 7. The result would be an alert that displayed “Result:8”. We can get different behaviors by using **doFunctionCall** to call different functions.

We can take this further and use function expressions as arguments to function calls as shown in the statement below which defines a function which used as an argument in a call to the **doFunctionCall** function. A function created as an argument has no name and so it is called an anonymous function.

```
doFunctionCall(function (p1,p2){ let result=p1+p2; alert("Result: "+result);},1,7);
```

Make Something Happen

Fun with Function Objects

Function objects can be confusing. Let's use our debugging skills to take a closer look at how they work. If you have left the browser at the page for the previous “Make Something Happen” just continue with that. Otherwise need to go to the book web page at

begintocodecloud.com and then select the sample **Ch01-What_is_the_cloud/Ch01-01_Explore_functions_in_the_console**. Then press function key F12 (or CTRL+SHIFT J) to open Developer Tools and select the console.

```
> let x = doAddition;
```

Type the above into the console. Note that we don't put any arguments on the end of `doAddition` because we are not calling it, we are specifying the name of the reference to the function. Now press enter.

```
> let x = doAddition;  
< undefined
```

The console shows the value "undefined" because the console always displays the value returned by a statement and the act of assignment (which is what the program is doing) does not return a value. After this statement has been performed the variable `x` now refers to the `doAddition` function. We can check this by looking at the `name` property of `x`. A function object has a `name` property which is the name of the function. Type in "`x.name`;" press enter and look at what comes back:

```
> x.name;  
< 'doAddition'
```

The console displays the value returned by the statement. In this case the statement is accessing the `name` property of the variable `x` which is the string `doAddition`.

```
> x.name;  
< 'doAddition'
```

Now let's call `x` with some arguments Type in the following statement and press Enter.

```
> x(10,11);
```

Since `x` refers to `doAddition`, you will see an alert displaying the value 21. Next, we are going to feed the `x` function reference into a call of `doFunctionCall`, but before we do that we are going to set a breakpoint so that we can watch the program run. Select the Sources tab and scroll down the index.htm source file until you find the definition of `doFunctionCall`. Click the left margin near the line number 41 to set a breakpoint inside the function at statement 41.

The screenshot shows the Microsoft Edge browser's developer tools. The 'Sources' tab is selected, displaying the file 'index.html' with line numbers 37 through 41. A blue box highlights line 41, which contains the code 'function doFunctionCall(functionToCall, n1, n2) {'. This indicates a breakpoint has been set at this line. The 'Console' tab is visible at the top.

Ch01_inset03_01 breakpoint in doFunctionCall

Now return to the Console view, type the following statement and press Enter.

```
> doFunctionCall(x,11,12);
```

When you press Enter the console calls `doFunctionCall`. When it reaches the first statement in the function it hits the breakpoint and pauses.

The screenshot shows the Microsoft Edge developer tools with the 'Sources' tab active. Line 41 of 'index.html' is highlighted. The 'Scope' tab is open, showing the current scope variables: 'functionToCall' (with a tooltip showing its value is 'doAddition'), 'n1' (value: 11), and 'n2' (value: 12). The 'Watch' tab is also visible at the bottom.

Ch01_inset03_02 program paused in doFunctionCall

Above you can see the program paused. The `doFunctionCall` function has been entered and the parameters to the function have been set to the arguments that were supplied. If you hover the mouse pointer over `functionToCall` in line 41 you will see a full description of the value in the parameter. The description shows that the parameter refers to the `doAddition` function.

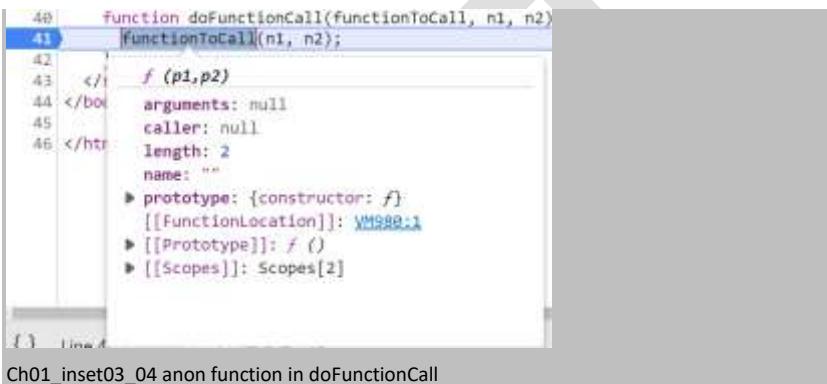
Repeatedly press the “step into function” button to watch the program go into the `doAddition` function, calculate the result and display the result in an alert. Then click OK in the alert to clear it and press the clear the alert and press the “Run Program” button (it’s the one on the left) to complete this call. Finally, return to the console for the “grand finale” of this Make Something Happen.

In the “grand finale” we are going to create an anonymous function and pass it into a call of

`doFunctionCall`. The function will be defined as an argument to `doFunctionCall`. The statement we are going to type is a bit long and you must get it exactly right for it to work. The good news is that the console will suggest sensible things to type. If you get an error, you can use the up arrow key to get the line back so that you can edit it and fix any mistakes.

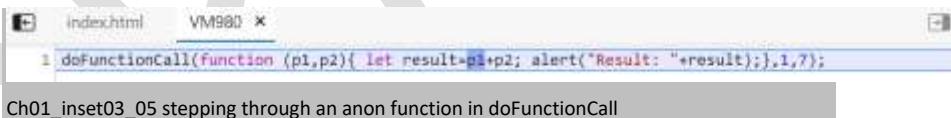
```
> doFunctionCall(function (p1,p2){ let result=p1+p2; alert("Result: "+result);},1,7);
Ch01_inset03_03 calling anonymous function
```

Now press enter to execute this statement. The program will hit the same breakpoint as before, but the display will be different:



Ch01_inset03_04 anon function in doFunctionCall

This time the name property of the function is an empty string. The function is anonymous. Press the “step into function” button to see what happens when an anonymous function is called.



Ch01_inset03_05 stepping through an anon function in doFunctionCall

The browser has created a temporary file to hold the anonymous function while it is in the debugger. The file is called VM980. When you do this exercise, you might see a different name. We can step through the statements in this file using “step into function”. If you want to just run the function to completion you can press the “Run program” button in the program controls.

Anonymous functions are used a lot in JavaScript, particularly when calling API functions to perform tasks. An object from the JavaScript API will signal that something has happened by calling a function. The quickest and most convenient way to create the function to be called is to declare it as an anonymous function.

Returning values from function calls

Up until now we have just called functions and they have done things. They have not returned a

value. They have returned the value undefined. Now we are going to investigate how a function can return a value and how a program can use the value that is returned.

```
function doAddSum(p1, p2) {  
    let result = p1 + p2;  
    return result;  
}
```

The function `doAddSum` above shows how a function can return a value. The `return` keyword is followed by an expression that gives the value to be returned when the function is called.

```
let v = doAddSum(4,5);
```

The statement above creates a variable called `v` and set the value of this variable to the result of the call of `doAddSum` – in this case the value 9 (the result of adding 4 to 5). The return from a function can be used anywhere you can use a value in a function.

```
let v = doAddSum(4,5) + doAddSum(6,7);
```

In the statement above the `doAddSum` function would be called twice and the value of `v` would be set to 22. We can also use function returns as arguments in function calls.

```
let v = doAddSum(doAddSum(4,5), doAddSum(6,7));
```

The code above looks a bit confusing but JavaScript would not have a problem performing it. The outer call of `doAddSum` would be called first, and then the two further calls would run to calculate the values of the two arguments. Then the outer call would run with these values. Note that the above statement is not an example of **recursion** (where a function calls itself). It shows how function calls can be used to produce values to be used as arguments to function calls.

A function can contain multiple `return` statements so it can return from different places in the function code. You can also use a `return` form a function to “escape” from inside the middle of deeply nested loops. However, if you use `return` like this you might end up making code that is

harder to debug.

Programmer's Point

Try to design your code to make it easy to debug and maintain

You will spend at least as much time debugging and maintaining code as you will writing it. Worse still, you will frequently be called on to debug and maintain programs that have been written by other people. Even worse still, six months after you've written a piece of code you become one of the "other people". I've occasionally asked myself "What idiot wrote this code?" only to find out that it was me.

When you write a program, try to make sure that it is going to be easy to debug. Consider the implementation of `doAddSum` below.

```
function doAddSum(p1, p2) {  
    let result = p1 + p2;  
    return result;  
}
```

You might think that it would be more efficient to return the result directly and get rid of the `result` variable:

```
function doAddSum(p1, p2) {  
    return p1 + p2;  
}
```

The above version of the function works fine. And it might even save a few millionths of a second when it runs (although I doubt this because browsers are very good at optimizing code). However, the second one will be harder to debug because you can't easily view the value of the result that it returns. With the original code I can just look at the contents of the `result` variable. In the "improved" version I'll have to mess around a bit to find out what value is being returned to the caller.

It's a similar issue with lots of `return` statements in a function. If the function containing lots of returns delivers the wrong result you have to step through it to work out the route it is following to deliver that particular result. If the function only has one return statement you know exactly where the result is coming from.

If the function just performs a task a good trick is to make a function return a status code so that the caller knows exactly what has happened. I often use the convention that an empty string means that the operation worked, whereas a string contains a reason why it failed.

If the function is supposed to return a value you can use the JavaScript values `null` and `undefined` to indicate that something has not worked

Oh, and if you find yourself thinking “What idiot wrote this code?”, don’t be so hard on the “idiot”. They were probably in a hurry, or lacked your experience or maybe, just maybe, there might a reason why they did it that way that you don’t know.

Returning multiple values from a function call

A problem with functions is that they can only return one value. However, sometimes we would like a function that returns multiple values. Perhaps we need a function to read information about a user. The function returns a name and an address along with status which indicates whether or not the function has succeeded. If the status is an empty string it means that the function worked. Otherwise, the status string contains an error message.

```
function readPerson() {  
    let name = "Rob Miles";  
    let address = "House of Rob in the city of Hull";  
    let status = "";  
}
```

Above is an implementation of a `readPerson` function that sets up some return values but doesn't return anything. What we want now is a way the function can return these values to a caller.

Returning an array from a function call

```
function readPersonArray() {  
    let name = "Rob Miles";  
    let address = "House of Rob in the city of Hull";  
    let status = "";  
    return [status, name, address];  
}
```

The above code creates a function called `readPersonArray` that returns an array containing the status, name and address values. We create an array in JavaScript by enclosing a list of values in brackets.

```
let reply = readPersonArray();
```

The statement above shows how we would create a `reply` variable that holds the result of a call to `readPersonArray`. We can now work with the values in the array by using an index value to specify which element we want to use in our program.

```
let status = reply[0];
if(status != "") {
    alert("Person read failed:" + status);
}
```

The above code puts the element at the start of the `reply` array (JavaScript arrays are indexed starting at 0) into a variable called `status`. This should be the status value of the call that has just been made. If this element is not an empty string the code displays an alert containing the status so that the user can see that something has gone wrong. If you look at the code for `readPersonArray` you'll see that the element at the start of the array is the status value, so this code would display an alert if the `status` variable contains an error message.

This code works, but it has an obvious disadvantage. The person making the call of `readPersonArray` needs to know the order of the values returned by the function. For this reason, I don't think using an array here is a very good idea. Let's look at a better one.

Programmer's Point

Use extra variables to make code clearer

You might look at the code above and decide that I've used a variable when I don't need to. I've created a variable called `status` that contains a copy of the value in `reply[0]`. I've done this because it makes the code that follows much clearer. The test of the status and the display in the alert make a lot more sense to the reader than they would if the code contained the variable `reply[0]`. This won't slow the program down or make it larger because the JavaScript engine is very good at optimizing statements like these.

Returning an object from a function call

```
function readPersonObject() {
    let name = "Rob Miles";
    let address = "House of Rob in the city of Hull";
    let status = "";
    return {status:status, name:name, address:address};
}
```

The above code creates a function called `readPersonObject` that returns an object containing status, name and address properties.

```
let reply = readPersonObject();
if(reply.status != "") {
    alert(reply.status);
}
```

The code above shows how the `readPersonObject` function would be called and the status property tested and displayed. Note that this time we can specify the parts of the reply that we want by using their name. If you want to experiment with these functions you can find them in the example web page we have been using for this chapter: [Ch01-What_is_the_cloud/Ch01-01_Explore_functions_in_the_console](#)

Programmer's Point

Make good use of object literals

This way of creating objects in JavaScript programs is called an object literal. I'm a big fan of them. They are a great way to create data structures which are easy to use and understand, right at the point you want to use them. You can also make an object literal to supply as an argument to a function call. If I want to supply name and address values to a function, I can do this because a function can have multiple arguments.

```
function displayPersonDetails(name, address) {
    // do something with the name and address here
}
```

The function `displayPersonDetails` has two parameters so that it can accept the incoming information. However, I'd have to be careful when calling the function because I don't want to get the arguments the wrong way round:

```
displayPersonDetails("House of Rob", "Rob Miles");
```

This would display the details of someone called "House of Rob" living at "Rob Miles". A much better way would be to have the function accept an object which contains name and address properties:

```
function displayPersonDetails(person) {
    // do something with the person.name and person.address here
}
```

When I call the function I create a object literal to deliver the parameters.

```
displayPersonDetails({ address:"House of Rob", name:"Rob Miles"});
```

This call of the function creates an argument which is an object literal containing the name and address information for the person. It is now impossible to get the properties the wrong way round in the function.

Make a Console Clock

We are now going to apply what we have learned to create a clock that we can start from the console. The clock will display the hours, minutes and seconds on a web page. At the moment we don't know how to display things on web pages (that is a topic for Chapter 2) so I've provided a helper function that we can use. You can use the Developer Tools to see how it works.

Getting the date and time

Our clock is going to need to know the date and time so that it can display it. The JavaScript environment provides a [Date](#) object we can use to do this. When a program makes a new [Date](#) object the object is set to the current date and time.

```
let currentDate = new Date();
```

The statement above creates a new [Date](#) object and sets the variable [currentDate](#) to refer to it. The keyword **new** tells JavaScript to find the definition of the [Date](#) object and then construct one. We will look at objects in detail later in the text. Once we have our date object we can call **methods** on the object to make it do things for us.

```
function getTimeString() {  
    let currentDate = new Date();  
    let hours = currentDate.getHours();  
    let mins = currentDate.getMinutes();  
    let secs = currentDate.getSeconds();  
    let timeString = hours + ":" + mins + ":" + secs;  
    return timeString;  
}
```

The function [getTimeString](#) above creates a [Date](#) object and then uses the methods called

`getHours`, `getMinutes` and `getSeconds` to get those values out of it. The values are then assembled into a string which the function returns. We can use this function to get a time string for display.

Make Something Happen

Console Clock

Open begintocodecloud.com and scroll down to the Samples section. Click **Ch01-What_is_the_cloud/Ch01-02_Console_Clock** to open the sample page. Then open the Developer Tools. Select the console tab.



Ch01_inset04_01 console clock page

The page shows an “empty” clock display. Let’s start by investigating the `Date` object. Type in the following:

```
> let currentDate = new Date();
```

Now press enter to create the new `Date` object.

```
> let currentDate = new Date();
< undefined
```

This statement creates a new `Date` and sets the variable `currentDate` to refer to it. As we have seen before, the `let` statement doesn’t return a value, so the console will display “undefined”. Now we can call methods to extract values from the object. Type in the following statement:

```
> currentDate.getMinutes();
```

When you press Enter the `getMinutes` method will be called. It returns the minutes value of the current time and the console will display it.

```
> currentDate.getMinutes();
< 17
```

The code above was run at seventeen minutes past the hour, so the value returned by `getMinutes` will be 17. Note that `currentDate` holds a “snapshot” of the time. To get an

updated date you will have to make a new `Date` object. You can also call methods to set values in the date too. The date contents will automatically update. You could use `setMinutes` to add 1000 to the minutes value and discover what the date would be 1000 minutes in the future.

The web page for the clock has the `getTimeString` function built in, so we can use this to get the current time as a string. Try this by entering a call of the function and pressing enter:

```
> getTimeString();
< '13:18:17'
```

Above you can see a call of the function and the exact time returned by it. Now that we have our time we need a way of displaying it. The page contains a function called `showMessage` which displays a string of text. Let's test it out by displaying a string. Type the statement below and press enter

```
> showMessage("hello");
```



The web page now displays the string that was entered. Now we need a function that will display a clock tick. We can define this in the console window. Enter the following statement and press Enter:

```
> let tick = function(){showMessage(getTimeString());};
< undefined
```

Take a careful look at the contents of the function and see if you can work out what they do. If you're not clear about this, remember that we want to get a time string and display it. The `getTimeString` function delivers a time string, and the `showMessage` function displays a string. If we have typed it in correctly, we should be able to display the time by calling the function `tick`. Type the following statement and press Enter

```
> tick();
```



Ch01_inset04_03 time display

The time is displayed. The final thing we need for our ticking clock is a way of calling the `tick` function at regular intervals so that the clock keeps time. It turns out that JavaScript provides a function called `setInterval` that will do this for us. The first parameter to `setInterval` is a reference to the function to call. The second parameter is the interval between calls in thousandths of a second. We want to call the function `tick` every second so type in the statement below and press Enter.

```
> setInterval(tick,1000);
```

This should start the clock ticking. The `setInterval` function returns a value as you can see below:

```
> setInterval(tick,1000);
< 1
```

The return from `setInterval` is a value which identifies this timer. You can use multiple calls of `setInterval` to set up several timers if you wish. You can use the `clearInterval` function to stop a particular timer:

```
> clearInterval(1);
```

If you perform the statement above you will stop the clock ticking. You can make another call of `setInterval` to start the clock again. At the moment we have to enter commands into the console to make the clock. In the next chapter we'll discover how to run JavaScript programs when a page is loaded so that we can make the clock start automatically.

Arrow functions

If you look at lots of popular cartoon figures you will notice that some only have three fingers on each hand, not five. You might be wondering why this is. It is to reduce the “pencil miles” for the animators. The first cartoons were made from frames that were all hand-drawn by animators. They discovered that they could make their lives easier by reducing the number of fingers they had to draw. Fewer fingers meant fewer “pencil miles”.

The JavaScript arrow function is a way of reducing the “keyboard miles” of a developer. The character sequence `=>` provides a way to create a function without using the word `function` in the definition.

```
doAdditionArrow = (p1, p2) => {  
    let result = p1 + p2;  
    alert("Result: " + result);  
}
```

The JavaScript above creates a function called `doAdditionArrow` which is exactly the same as the original `doAdditon`. However, it is much quicker to type in. If the body of the arrow function only contains one statement you can leave off the braces that mark the start and end of the function body. And a single statement arrow function returns the value of the statement, so you can leave off the return keyword too. The code below creates a function called `doSum` which returns the sum of the two arguments.

```
doSum = (p1,p2) => p1 + p2;
```

We could call the `doSum` function as we would any other:

```
let result = doSum(5,6);
```

This would set the value of `result` to 11. You see the true power of the arrow notation when you start using it to create functions that are to be used as arguments to function calls.

```
setInterval(()=>showMessage(getTimeString()),1000);
```

This innocent looking statement repays careful study. It makes our clock tick. In the “Make Something Happen: Console clock” above we used the `setInterval` function to make the clock tick. The `setInterval` function accepts two arguments, a function to call and the interval between each function call. I’ve implemented the function to call as an arrow function containing a single statement which is a call to `showMessage`. The `showMessage` function has a single argument which is the message to be displayed. This is provided by a call to the `getTimeString` function.

Arrow functions can be confusing

JavaScript is not the first programming language that I've learned. It might not be your first language either. When I was learning JavaScript I found the arrow function one of the hardest things to understand. If you have seen C, C++, C#, or even Python programs before you will know about functions, arguments, parameters and return values already. This means that "traditional" JavaScript functions will be easy to grasp.

But you will not have seen arrow functions before because they are unique to JavaScript. And you can't easily work out what they do from seeing them in code. If you don't know what an arrow function does you will find quite tricky to understand what the creation of `doSum` above is doing. One way to deal with this would be to just regard arrow functions as a little extra provided by the language to make your life easier. If you don't mind doing the extra typing you can create all your programs without using the arrow notation. However, I think you should spend extra time learning how they work so that you can understand JavaScript written by other people.

There is one other aspect of arrow functions that can be confusing. There is a specific behavior that they have involving the `this` reference which it is important to know about. We will cover this in the section "The meaning of this" in Chapter 3.

When we started talking about JavaScript functions we noted that they are used to attach JavaScript code to events. The arrow function makes this very easy to do.

Make Something Happen

Arrow function Console Clock

You should already have the sample [Ch01-What_is_the_cloud/Ch01-02_Console_Clock](#) open. If not, open it. Then open the console tab and use the arrow function above to get the clock ticking. If the clock is already ticking you can reset everything by reloading the example page in the browser.

What you have learned

At the end of each chapter, we have a "what you have learned" section which sets out the major points covered in the text and poses some questions that you can use to reinforce your understanding.

- A web browser is an application that requests data from a web server in

the form of web pages. The connection between the two applications is provided by the internet.

- Web servers were originally single machines connected to the internet which were owned and operated by the owner of the site. The cloud made computing power into a resource that can be bought and sold. We can pay to have our web sites hosted in the internet. Page requests sent to the address of our site will be processed on machines at our service provider.
- The JavaScript programming language was invented to allow a browser to run program code that has been download from a website. It has since developed into a language that can be used to create web servers and free-standing applications.
- A JavaScript function is a block of code and a header which specifies the name of the function and any parameters that it accepts. Within the function the parameters are replaced by values which were supplied as arguments to the function call.
- When a running program calls a function the statements in the function are performed and then the running program continues from the statement after the function call. A function can return a value using the return keyword.
- When a JavaScript program runs inside the browser it uses an Application Programming Interface (API) to interact with the services the browser provides. The API is provided by many JavaScript functions.
- Operators in JavaScript statements perform an action according to the context established by the operands they are working on. As an example, adding two numbers together will result in an arithmetic addition, but adding two strings together will result in the strings being concatenated. If a numeric operation is attempted with operands that are not compatible the result will be set to the value “not a number”.
- Variables in JavaScript can hold values that indicate specific variable state. A variable that has not been assigned anything has the value “undefined”. A calculated value that is not a number (for example the result of adding a number to a string of text) will have the value NaN – short for Not A Number.
- Modern browsers provide a Developer Tools component that contains a console that can be used to execute JavaScript statements. The Developer Tools interface also lets you view the JavaScript being run inside the page and add breakpoints to stop the code. You can step through individual statements and view the contents of variables.

- A JavaScript function is represented by a function object. JavaScript manages these by reference so variables can contain references to functions. Function references can be assigned between variables, used as arguments to a function call and returned by functions.
- Function expressions allow a function to be created and assigned to a reference at any point in a program. A function expression used as an argument to a function call is called an *anonymous function* because it is not associated with any name.
- The JavaScript API provides a `Date` object which can be used to determine the current date and time and also allows date and time manipulation. The API also provides the functions `setInterval` and `clearInterval` which can be used to trigger functions at regular intervals.
- Functions can be defined using “arrow notation” which is shorter than the normal function definition. This is especially useful when creating functions to be used as arguments to function calls.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions" about the cloud and what we do with it.

What is the difference between the internet and the web?

The internet is the technology that allows computers to communicate. The web is a service that uses the internet to link web browsers and web servers.

What is the difference between the cloud and the internet?

The internet is the networking technology that allows a program running on one computer to exchange data with a program running on another computer. We don't need the cloud to make an internet-based application. We just need two computers with internet connections. The cloud lets you replace a computer connected to the internet with a service that you have purchased from a cloud service provider. The cloud based server will have a network address which is used by the cloud service provider to locate the required service.

How does the cloud work?

A server hosted by a cloud service provider runs an operating system that allows it to switch between processes running services for different clients. At the front of the cloud service is a component which accepts requests and routes them to the process that provides the required service. The computing resources used by each process are monitored so that the services can be billed for the services they have provided.

What is the difference between a function and a method?

A function is declared outside of any objects. We have created lots of functions in this chapter. A method is a function which is part of an object. A `Date` object provides a `GetMinutes` method that returns the minutes value for a given date. This is called a method because it is part of the `Date` object. Methods themselves look like functions. They have parameters and can return values.

What is the difference between a function and a procedure?

A function returns a value. A procedure does not.

Can you store functions in arrays and objects?

Yes you can. A function is an object and is manipulated by reference. You can create arrays of references and objects can contain references.

What makes a function anonymous?

An anonymous function is one which is created in a context where it is not given a name. Let's take a look at the `tick` function we created for the clock:

```
let tick = function(){showMessage(getTimeString());};
```

You might think that this function is anonymous. However JavaScript can work out that the function is called "tick". If you look at the `name` property of the function you will find that it has been set to `tick`. But instead of creating a `tick` function we might use a function expression as an argument to the call of `setInterval`:

```
setInterval(()=>showMessage(getTimeString()),1000);
```

The statement above feeds a function expression into `setInterval`. The function expression does the same job as `tick`, but now it is an anonymous function. There is no name attached to the function. Note that this statement uses the arrow notation to define the function.

Why do we make functions anonymous?

We don't have to use anonymous functions. We could create every function with a name and then use the name of that function. However, anonymous functions make life a lot easier. We can bind behaviors very tightly to the place they are needed. Also, if we are only ever going to perform a behavior once it is rather tedious to have to invent a function name for it.

Can an anonymous function accept parameters and return a result?

Yes it can.

Are arrow functions always anonymous?

An arrow function is simply a quick way of creating a function definition. Arrow functions can have names.

What happens if I forget to return a value from a function?

A function that returns a value should contain a return statement followed by the value to be returned. However, if the function contains multiple return statements you might forget to return a value from one of them. The program will still run, but the value returned by the function at that point would be set to “undefined”.

What happens if I don't use the value returned by a function.

There is no need for a program to use the value returned by a function.

What does the let keyword do?

The let keyword creates a variable which is local to the block of code in which it is declared. When execution leaves the block the variable is discarded.

Do JavaScript programs crash?

This is an interesting question. With some languages the program text is carefully checked for consistency before it runs to make sure that it contains valid statements. With JavaScript, not so much. Using the wrong type of value in an expression, giving the wrong numbers of arguments to a function call or forgetting to return a value from function are all examples of program mistakes which JavaScript does not check for. In each of the above situations the program would not fail when it ran, instead the errors would cause variables to set to values like undefined or Not a Number. This means that you need to be careful to check the results of operations before using them in case a program error has made them invalid.

So the answer is that your JavaScript program probably won't crash, but it might display the wrong results.

Begin to Code: Chapter 2 Get Building Apps into the Cloud and Games in What you will learn

In the last chapter we discovered the origins of the cloud and ran some JavaScript code in a web page using the browser Developer Tools. We also learned a lot about JavaScript functions how to connect them to events

the Cloud

In this chapter we are going to take our JavaScript code and put it into the cloud for anyone to access. We are going start by getting the tools we are going to use and then move on to look at the format of the documents that underpin web pages. Then we are going to use JavaScript to add programmed behaviors to pages and discover how we can put our active pages into the

cloud.

Rob Miles

Don't forget that you can use the Glossary to look up any terms that seem unfamiliar. Words defined in the glossary will be formatted like **this** in the text. Note that both **this** and **in** are in the glossary.

Working in the cloud

Before you start to build your applications, you need to find a nice place to work. There is a physical aspect to this. You work best if you are comfortable so a decent screen, nice keyboard and a responsive computer are all great things to have. However, there is also a "logical" element too. You need to find a place to store all the materials that you're going to generate. You could just store all the files on your computer, but you might lose them if your machine fails. What's more, if you make the wrong modifications to the only copy of a crucial file you can lose a lot of work very quickly (I have done this many times). So, let's look at how we can manage our stuff. Starting with git.

Git

Git was created in 2005 by Linus Torvalds who was writing the Linux operating system at the time. He needed a tool that could track what he was doing and make it easy for him to share work with other people. So, he created his own. Git organizes data into “repositories”. A **repository** can be as simple as a folder containing a single file or it can be a **hierarchy** of nested folders containing thousands of documents. Git doesn’t care what the documents contain, or even what type of data they are. A repository can contain pictures, songs, and 3D designs along with software code.

You ask Git to “commit” changes you have made to the contents of a repository. When you do this the Git program searches through all the files in the repository and takes copies of the ones that have changed. These files, along with records of changes are then stored by Git in a special folder. The great thing about Git is that you can return to any of your commits at any time. You can also send people a copy of your repository (including the special folder). They can work on some of the files in the repository, commit their changes and send the repository back to you.

Git can identify which files they’ve changed and handle contentions. If both of you have changed a particular file Git can show the changes each person has made. These changes can be used to establish a definitive version of the file which is then stored back in the repository. If you think Git was a hard program to write, you are correct. It turned out to be tricky to manage file synchronization and change resolution. However, it makes it much easier for people to work together on shared projects.

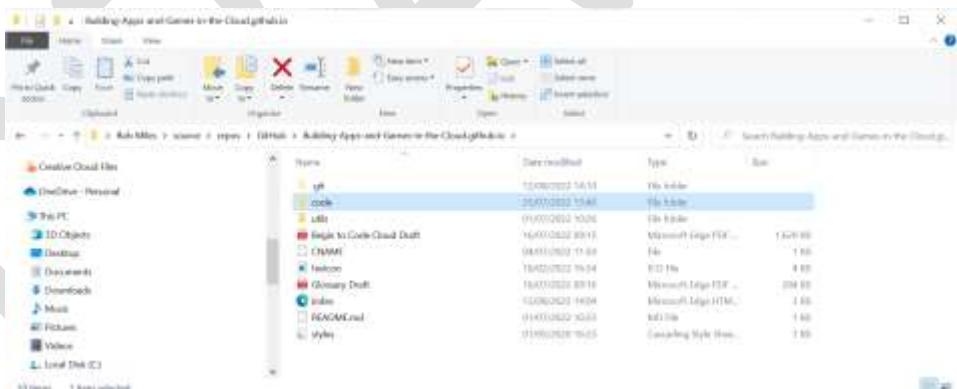


Figure 2.7 Ch02_Fig_01 Sample Repository

Figure 2.1 shows the contents of the repository containing all the sample code for this book. I have set the ‘**Hidden items**’ option in the **View** tab so that File Explorer shows hidden files. At the top the folder you can see a folder called **.git** which is the special folder created and managed by the git program. You can have a look inside if you wish, but you really shouldn’t change anything in it. When you copy a repository it is important that this folder be copied too.

The sample repository is exposed by GitHub as a web page, so it contains an index.html file. This is so you can view any of the sample code in your browser without downloading any samples onto your machine. Not all GitHub repositories are web pages, but it is very useful to be able to host a web site in this way. We will be doing this at the end of this chapter.

If we are going to work with repositories, we need to install the git software. This is a free download and there are versions for all types of computers.

Make Something Happen

Install Git

First you need to open your browser and visit the web page:

<https://git-scm.com>



Now follow the installation process selecting all the default options.

Storing git repositories

You can use the git program on a single computer to manage your work. In this case you will put each repository in a folder somewhere. If you want to store your files in a central location, you can also set up a computer as a “git server”. This is a bit like a “web server for software developers”. You use the git program to send copies of your repositories over the network to the server and have them stored there. You can also “clone” a repository from the git server (this is called “checking out” a repository), work on it and then sync your changes with the original (this is called “checking in”). Git provides user management so individuals can have their own login names and be formed into teams that have access to particular repositories.

All the changes are tracked by git so that they can be reversed or examined in detail. This makes it much easier for developers to collaborate (which is what Linus Torvalds wanted at the start) as git can also detect multiple changes to the same file and insist that they are resolved to create a single definitive version.

GitHub and Open-Source Software

A git server makes it easy for programmers in a company to work together. But what if you want anyone in the world to be able to work on your project? Lots of important software used today, including operating systems, network tools and even games are now developed as **open-source** projects. All the software code that comprises these applications is stored openly, frequently in a git repository which is accessible to people who want to help with the project. Anyone can check out the repository, make some changes and then submit a “pull request” to the project owners. The project owners can “pull” in a copy of the changes. The changes could be a fix to a problem, a new feature, or even just improved error messages. If the changes check out these are then incorporated into the application which then gets that bit better.

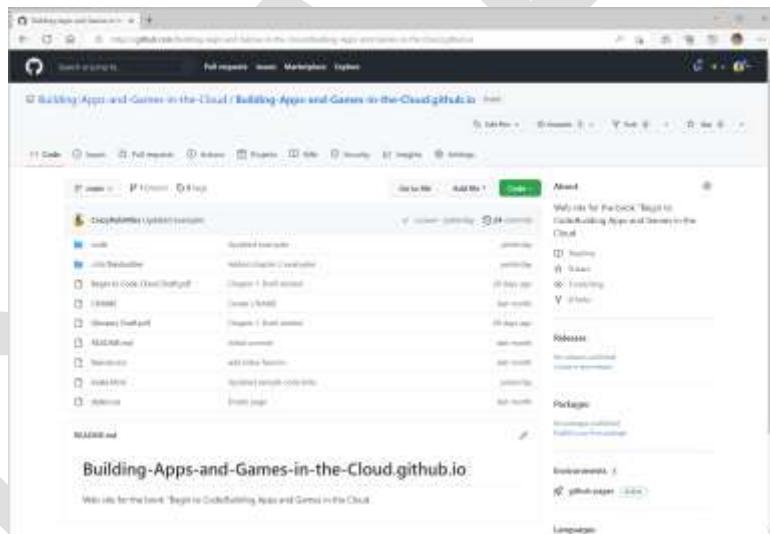


Figure 2.8 Ch02_Fig_02 Sample Repository on GitHub

Figure 2.2 above shows the sample repository as it appears on GitHub. GitHub also provides **organizations** as a way of managing projects. An organization can be created by a user and can contain multiple repositories. I've created one called “Building-Apps-and-Games-in-the-Cloud” for this book. The clock repository is one of several that are held in this organization. If you are storing repositories for a group or a large project which should not be directly associated with a particular GitHub user, you can create an organization to hold those repositories.

The clock repository is public, so anyone can look inside the files. They can even clone the

repository onto their computer, make some changes and send me a pull request. I have also added a license file which sets out what other people can do with the code, so you could regard the clock as a mini Open-Source Project. You can make private repositories which will not be visible to other GitHub users if you need them.

Programmer's Point

Open-Source projects are a great place to start your career

You might think that you must become a great programmer before you can start to make a name for yourself in software development. This is not true. You can add value to an open-source project well before you can create complete programs. You can learn a huge amount just by looking at code written by other people. And working out how the internal pieces of a system fit together is very satisfying, even if you don't know how the whole thing works.

An important part of being a successful developer is being able to work well with other people, so being part of an open-source project prepares you well for this. Many developers can also remember what it was like to start out and will be happy to help you improve, as long as you keep your input constructive and focused.

In addition, there are lots of things a project needs that have nothing to do with programming. A large project will need people to test things, write documentation, make artwork, create different language versions and so on. If you've got any of those skills, you could find yourself in high demand just for those. And that alone could open a totally different career path for you.

Many open-source projects (and lots of commercial ones) are hosted by GitHub. Companies rent space on GitHub rather than set up their own git server, but GitHub also offers comprehensive free services for open-source developers. It also offers a good service to individual developers; you can host your own private repositories on GitHub for your personal projects.

I strongly advise you to set up an account on GitHub and start using it for your projects. It will not cost you anything to do this. Whenever I start a new project one of my first thoughts is "what happens if I lose everything?". GitHub is a good answer to that question. If I put things into a GitHub repository they will be as safe as they can be. And if I make good use of the ability to commit changes at regular intervals I can save myself from losing work. Furthermore, once the repository is on GitHub I can invite other people into my project to work on it with them, or make it public so anyone can use it.

You don't need a GitHub account to make use of everything in this book, but if you want to get the most out of the contents you really should make one. It can completely change the way you work. If I want to do anything these days, from organizing a party to writing a book, I start by creating a GitHub repository for the project.

GitHub is a great place to store your data. However, it is also a great place to network with others. Each repository has a Wiki. This is a space for collaboratively creating documentation. A repository also has issue tracking discussions that people can use to report bugs and request features and you can create and manage projects within the repository.

GitHub also has a final ace up its sleeve. It can host web sites. You can take your web pages, put them into a GitHub repository and then make them available for anyone in the world to see. We will be doing this at the end of this chapter.

Make Something Happen

Join GitHub

If you don't want to join GitHub you can skip this section. You can still grab the sample repositories and work on the code but you won't be able to create your own repositories or use GitHub to host websites.

You will need an email address to create a GitHub account. Start by opening your browser and visit the web page: <https://github.com/join>



Ch02_inset02_01 GitHub join page

Enter your email address and click "Sign up for GitHub". It takes a few steps, and you have to wait for a message to validate your email address, but eventually you will end up at the dashboard for your GitHub account.

You are now logged in to the GitHub website and can work with your repositories and clone those of other GitHub users. If you want to use GitHub from the browser from another machine you will have to log in to the service on that machine.

You can work with your repositories using the web page interface. You can also enter git commands into your Windows terminal or MacOS console. However, we are going to use the Visual Studio Code program to edit and debug our programs and it can also talk to git and manage our repositories for us.

Programmer's Point

GitHub is also a social network.

You can regard GitHub as a social network. If you make something that you think would be useful to other people, you can share it by creating a public GitHub repository. Other GitHub members can download your repository and use it. They can also make changes and send you “pull requests” to signal that they have made a new version you might like.

Members can award repositories “stars” and comment on their contents. There are also lots of discussions going on. This means that GitHub is another place you can start to make a name for yourself. However, just as you should be cautious when you post details on other social network sites, you should also be aware of the potential for misuse that social networking frameworks provide.

Make sure not to give out too much personal information and ensure that anything you put into a public repository does not contain personal data. Don’t put personal details, usernames, or passwords into program code that you put on GitHub. Later in the text we will discover how to remove personal information from projects that we want to make public.

Get Visual Studio Code

Now that we have git installed and working, we are going to install Visual Studio Code which we are going to use to create all our programs. It is free to download and versions are available for Windows, Macintosh and Linux based computers, including the Raspberry Pi. It also supports lots of extensions that can be used to extend its capabilities.

Make Something Happen

Install Visual Studio Code and clone a repository

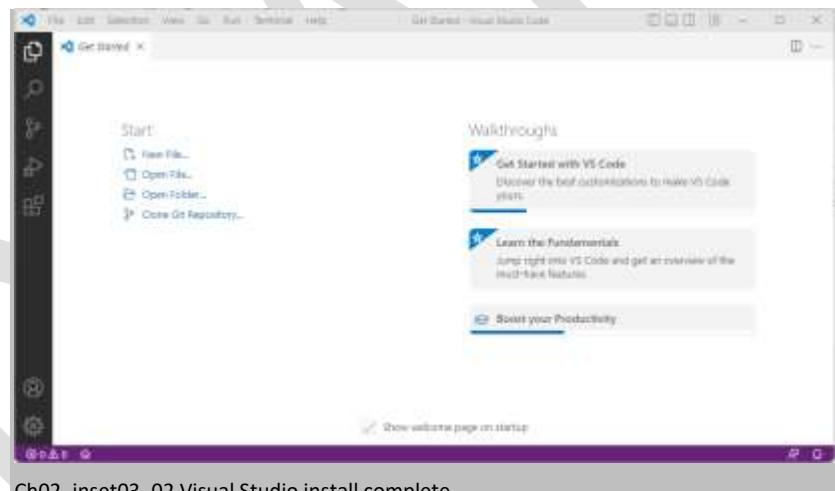
I'm going to give you instructions for Windows. The instructions for macOS are very similar. First you need to open your browser and visit the web page:

<https://code.visualstudio.com/Download>



Ch02_inset03_01 Visual Studio Install page

Click the version of Visual Studio Code that you want and follow the instructions to install it. Once it is installed you will see the start page.



Ch02_inset03_02 Visual Studio install complete

Now that you have Visual Studio installed the next thing you need to do is fetch the sample files to work on. To do this click the Clone Git Repository link which is about halfway up the left-hand side of the page.



Ch02_inset03_03 Clone examples repository

<https://github.com/Building-Apps-and-Games-in-the-Cloud/Building-Apps-and-Games-in-the-Cloud.github.io>

Enter the address of the repository into the dialog box as shown above. Then click the “Clone from URL” button underneath the address. If you click the “Clone from GitHub” button underneath “Clone from URL” you will be prompted to log in to GitHub. This can be useful if you want send local repositories from your machine into GitHub, but you don’t need to do this to just fetch files.

The repository will be copied into a folder on your machine. The next thing that Visual Studio needs to know is the location of that folder. I have a special GitHub folder where I store my repositories. This is not in my OneDrive folder. Since I’m using GitHub to keep my files safe, I don’t need to use OneDrive to synchronize things.



Ch02_inset03_04 Clone complete

Once the files have been cloned Visual Studio Code offers you the chance to open the new repository. Click **Open** to do this. You will be asked to confirm that you trust the author. It is best to say yes here. Then Visual Studio will open the repository and show you the contents. On the far left are the tools you can use to work on the repository. Click the top one to select Explorer. The examples are in the code folder, organized by chapter. Click on **Ch02-Get_into_the_cloud** to open the folder and then open the "**Ch02-01_Simple HTML**" folder and then click the **index.html** file to open it in the editor.



```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Ch02-01 Simple HTML</title>
</head>
<body>
    <p>This is <strong><b>Hello World</b></strong></p>
    <p>This is <b><strong>Hello World</strong></b></p>
</body>
</html>
```

Ch02_inset03_05 editing code

This is the first example file in this chapter. We will be looking at it later. Leave Visual Studio Code running, we will be using it again in a moment. The next thing to do is install our first Visual Studio Code extension.

Install the Live Server Extension

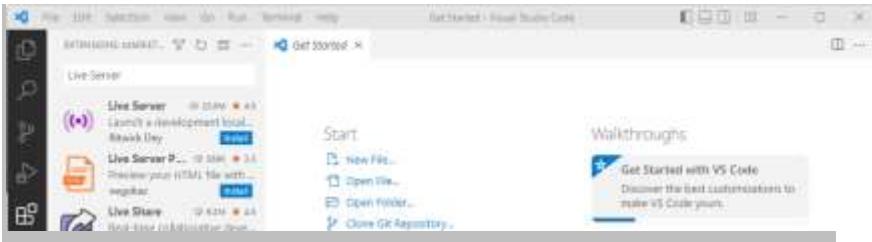
Before we can work with our web pages there is something we can do to make our job much easier. If you think about it, what we are going to do is edit a web page with Visual Studio Code, view the page in our web browser, edit it again and so on. We could do this by repeatedly saving the web page to a file and opening it by hand each time. However, this would be hard work and programmers hate hard work. What programmers do when faced with a problem like this is create a tool that will do the work for them.

Ritwick Dey is a programmer who solved this problem by creating the Live Server extension for Visual Studio Code. An extension is something you can add to Visual Studio Code to add a new feature. There are thousands of extensions available for download, we are going to use the Live Server extension. Then, when we want to view HTML in our browser we can just press a button to do this.

Make Something Happen

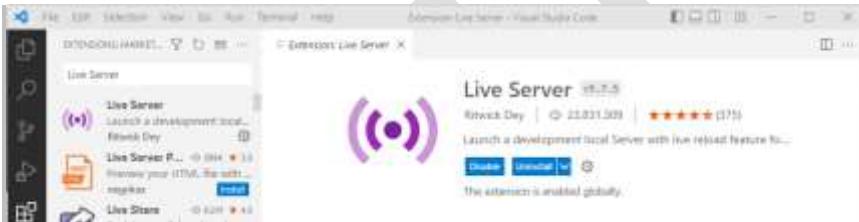
Install the Live Server extension

To begin, open Visual Studio Code. Then click the extensions button on the left-hand toolbar (it is the fifth one down as highlighted below. The Extensions Marketplace opens. Type Live Server into the search box.



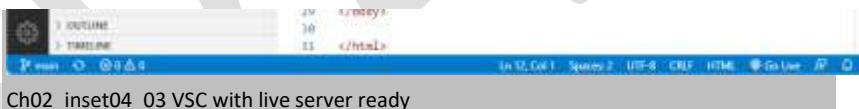
Ch02_inset04_01 Installing Live Server

The marketplace will show all the extensions with this name. Click the Install button on the one written by Ritwick Dey.



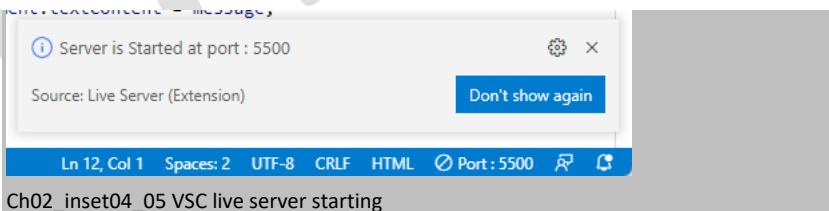
Ch02_inset04_02 Live Server installed

The extension will be installed and will start up each time Visual Studio Code is loaded. We can test it by using the clock page. If you have not left Visual Studio Code open from earlier, you should open the "**Ch02-01_Simple HTML**" folder in the **clock** and then click the **index.html** file to open it in the editor.



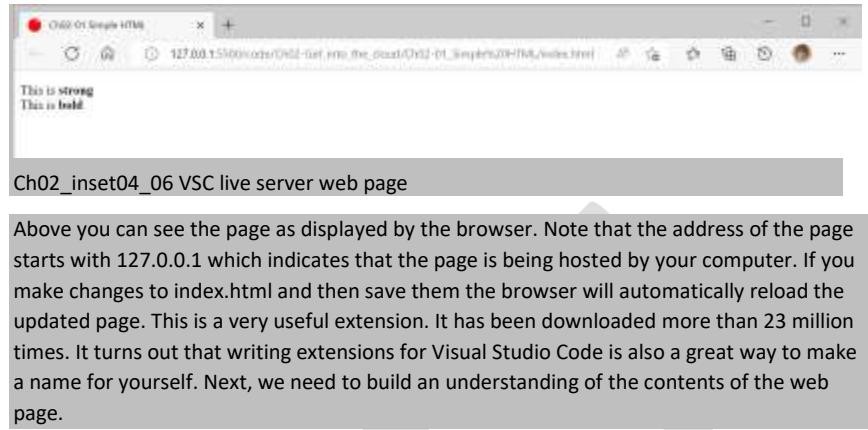
Ch02_inset04_03 VSC with live server ready

You can see the Go Live button on the bottom left of the screen in the blue border. Click the button and Go Live will now open the index file in your browser. You might see some messages from the firewall on your machine the first time you do this. You should allow Visual Studio Code access to the ports that it needs.



Ch02_inset04_05 VSC live server starting

You will also see a message from Live Studio telling you the server has started and specifying the network port that it is using. Then the browser opens and displays the page.



How a web page works

We have been using web pages in our browsers for years. In chapter 1 we saw that a web page can contain JavaScript code. Now it is time to dig deeper and consider what makes a web page. We are not going to go into too much detail, there are books specifically about these topics that are much thicker than this one. However, you may think you already know what a web page is, but I'd be most grateful if you would read this section anyway. You may find a few things you didn't know.

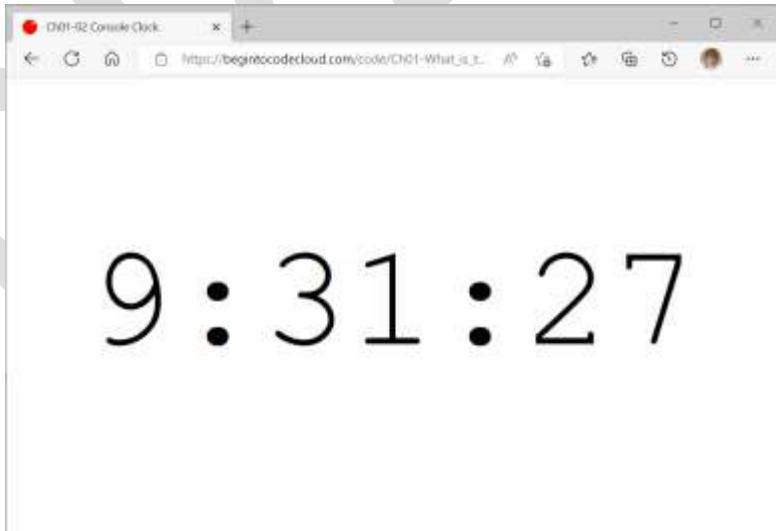


Figure 2.9 Ch02_Fig_03 Ticking Clock

A web page is a “logical document”. What do I mean by that? Well, a book is a **physical** document. It exists in the real world. We can read it, add annotations, leave it on the bus and do everything else that you can do with a physical book in the real world. An electronic book (or e-book) is a **virtual** document. It is something created to play the part of a physical book in a virtual environment created by a computer. We can read it, annotate it, and if we delete the file containing the book, we could even manage to lose it. A web page is a **logical** document. It is a thing created by software and hosted on a computer. A web page has no physical counterpart. There is no physical version of a book that contains a ticking clock, as you can see in Figure 2.3 the best we can do is a stationary image.

Loading a page and displaying it

The starting point for a web page is a file of text which is hosted on a server. When you visit a website, for example www.robmiles.com, the browser looks for a file called index.html at that location which contains the start page of the site. The text in this file describes the logical document that makes up the web page. The browser reads the file of text from the server and uses it to build the logical document that is the web page. At this point you might say “Aha! The file of text on the server describes the web page so a web page is just a file of text.” Well, no. A web page could contain JavaScript code that runs when the document is loaded into the browser. That code can change the contents of the logical document and even add new things. The logical document that the HTML describes is held by the browser in a structure called the **Document Object Model** (DOM). The DOM is a very important part of web programming. Our JavaScript programs will interact with the DOM to display output.

Once the browser has built the DOM it draws it on the display. The browser program then repeatedly checks the DOM for changes and redraws it if any changes are found. This is how the ticking clock we created in chapter 1 works. There is a JavaScript function running which changes something in the document and the browser redraws the page to reflect the changes. We don’t have to do anything to trigger a redraw, our code can change the contents of the logical document and the updated version of the page is displayed automatically.

A logical document can contain images, sounds and videos which are all updated automatically. The initial contents of the document objects are expressed using a language called Hypertext Markup Language or HTML. Let’s look at that next.

Hypertext Markup Language (HTML)

We can discover what Hypertext Markup Language really is by unpicking its name. Let’s start with Hypertext. Remember that we can call things **logical** to indicate that they have no counterpart in real life. **Hypertext** is a logical version of text. Normal text, whether it is printed or displayed on a screen, is something you read from beginning to end. Hypertext can only be displayed by a computer. This is because hypertext can contain **hyperlinks** which refer to other documents. You can start reading one document, open a hyperlink and be moved to a completely different one, perhaps served by a completely different computer in a different country. When hypertext and

hyperlinks were invented, it was thought cool to put the word hyper in front of them to make them sound impressive. So that is where the name came from. At least, that's what I think.

So, the word hyperlink in HTML means that the aim of the language is to express a page that can contain hyperlinks. HTML was invented to make it easier to navigate reports. Before hyperlinks, if a report contained a reference to another report you would have to go and find that report and open it to read it. After hyperlinks you could just follow a link in the original report. Hypertext was designed to be extensible, so that it would be easy to add new features. The features provided by modern web pages are way beyond any foreseen by Tim Berners-Lee, the inventor of HTML, but the fundamental content of a page remains the same.

The word **markup** refers to the way that an HTML document separates the intent of the author from the content in the page. Let's look at a tiny web page to discover how this works.

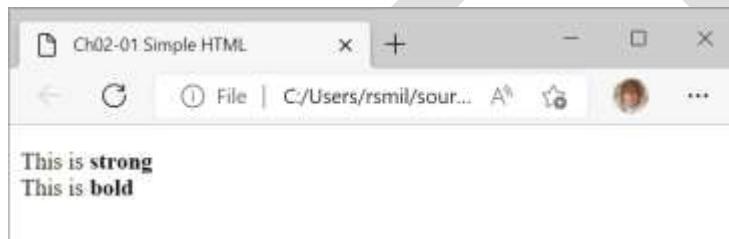


Figure 2.10 Ch02_Fig_04 Simple HTML

Figure 2.4 shows a tiny web page. It only contains six words. Let's look at the HTML file behind it and discover the role of markup in creating the page contents:

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Ch02-01 Simple HTML</title>
</head>
<body>
<p>
    This is <strong>strong</strong><br>
    This is <b>bold</b><br>
</p>
</body>
</html>
```

This is the html file that describes the page in Figure 2.4. The most important characters in the file are the < and > characters that mark the start and end of HTML element names. The < and >

are called **delimiters**. They *define the limits* of something. An **element** is a thing in the document that the browser knows how to work with. Elements can have attribute values. These are name-value pairs included in the definition of the element. The `<html lang="en">` element above has an attribute called `lang` which gives the language of the html page. The language code `en` means English.

Elements can be containers. A container starts with the name of the element and ends with the name preceded by a forward slash (/). You can see that the `<title>` element contains the text that is to be used as the title of the web page. You can see this title on the top of the web page in Figure 2.4. The title information is part of the header for the page, which is why it is enclosed in the `<head>` element.

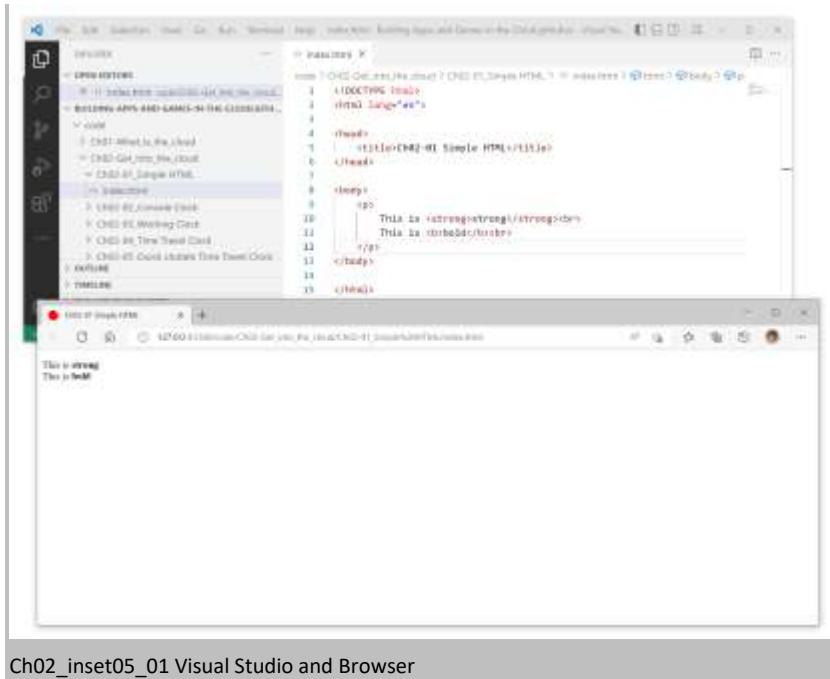
The `<body>` element contains all the elements to be drawn by the browser. The `<p>` element (short for paragraph) groups text into a paragraph. The `<bold>` and `` elements give formatting information to the browser. However, an element can exist as the starting element name, without another to mark the end of the element. The `
` element tells the browser to add a line break in the text. You don't need to add a `</br>` element to a page to mark the end of a line break.

Now we can look at the last word in the phrase “Hypertext Markup Language”. The word language means that we are going to use HTML to express things. HTML is very specific (it is being used to tell a browser how to build a logical document) but it is a language.

Make Something Happen

Web page editing

If you have been following this exercise you will already have the first sample file open in your browser. If not, use Visual Studio Code to open the “**Ch02-01_Simple HTML**” in the examples and then click the **index.html** file to open it in the editor. Then click **Go Live** to open the page in the browser.

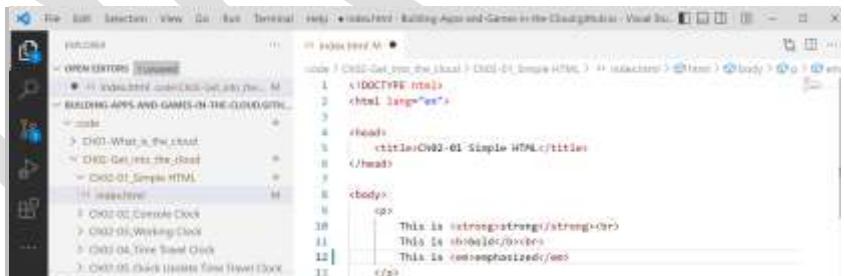


Ch02_inset05_01 Visual Studio and Browser

Your desktop should now have both Visual Studio and your browser on it, as shown above. Now go back to Visual Studio Code and add the following element into the body element of the page:

This is emphasized

Visual Studio Code should now look like this:



Ch02_inset05_02 web page edits

Now select **File>Save** from the Visual Studio Code menu bar to save the updated file

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Ch02-B1 Simple HTML</title>
</head>
<body>
    <p>
        This is <strong>strong</strong><br>
        This is <b>bold</b><br>
        This is <em>emphasized</em>
    </p>
</body>
</html>
```

Ch02_inset05_03 edit save

Because you are using Go Live you should see the page in the browser update automatically and display the new text on the page.



Ch02_inset05_04 updated page

This is a very nice way to work on web pages. Each time you save your file the web page updates automatically.

CODE ANALYSIS

Investigating HTML

If this is your first brush with HTML you may have some questions.

Question: What is the difference between bold and strong?

The HTML file marks one piece of text as bold and another as strong. But in Figure 2.2 they both look the same. You might think that this means that bold and strong are the same thing. However, they are intended to be used for different kinds of text. Text that needs to stand out should be formatted bold. Text that is more important than the text around it should be formatted strong. I would print my name in bold (because I would like it to stand

out). I would print “Do not put your head out of the window when the train is moving” in strong because that is probably more important than the text around it.

Remember that HTML just contains instructions to the browser to display things in certain ways. The browser has a default behavior (to display bold and strong as bold text) but you can change this for your web pages by adding styles, of which more later.

Question: How do I enter a < or a > into the text?

Good question. HTML uses another character to mark the start of a **symbol** entity. The & character marks the start of a symbol. Symbols can be identified by their name. Some useful ones are:

< > &

You can find a list of all the symbols here: <https://html.spec.whatwg.org/multipage/named-characters.html> You can also use symbols to add emoticons to your pages. Take a look here for the codes to use <https://emojiguide.org/>

Question: What happens if I mis-spell the name of an element?

If the browser sees an element it doesn't know it will just ignore it.

Question: What happens if I get the nesting of the elements wrong?

If you look at the sample code, you will see that some elements are inside others. The <p> element is inside the <body> for example. If you get the nesting wrong (for example put the </body> element inside the <p> the browser will not complain and the page will display, but it might not look how you were expecting.

Question: What does the <!DOCTYPE html> element mean?

Good question. The very first line of a resource loaded from a web server should describe what it contains. The resource could contain an image, a sound file, or any number of other kinds of data. The !DOCTYPE element is used to deliver this information. The browser doesn't always use this. A browser will try to display any file of text it is given, but it is very useful to add the information.

Question: How do I put JavaScript into a web page?

You use the <script> element to embed JavaScript into a page.

Question: Are there other kinds of markup language?

Yes there are. There is XML (eXtensible Markup Language) which is used for expressing structures of data. There are also lots of others which have been developed for particular applications.

Question: How do I stop the Go Live server?

You might want to stop the Go Live server and open a web page from a different index file. You can do this by restarting Visual Studio Code, but you can also press the close button next to the port number on the bottom right of the Visual Studio Code window.

Make an active web page

We have reached a pretty powerful position. We have tools that we can use to create and store web resources and we are building an understanding of the way that web pages are structured. Now we are going to take another big step forwards. We are going to discover how a JavaScript program can modify the contents of the document object and change the appearance of the web page. This is how JavaScript programs running in the browser communicate with the user.

Interact with the document object

Make Something Happen

Interact with a web page

If you have been following this exercise you will already have the console clock open in your browser. If Go Live is displaying the page, stop it from running by pressing the close button next to the port number at the bottom of the Visual Studio Code window. If not, use Visual Studio Code to open the “Ch02-02_Console Clock” folder in the examples and then click the **index.html** file to open it in the editor. Then click **Go Live** to open the page in the browser. Now open the Developer Tools and then select the **Elements** tab. This shows us the elements in the document:



Ch02_inset06_01 Clock page elements

In the figure above I've resized the Developer Tools part of the screen to give me more space. On the left of the window, you can see the page as displayed by the browser. On the right you can see a display of the contents of the web page. One item has been selected in

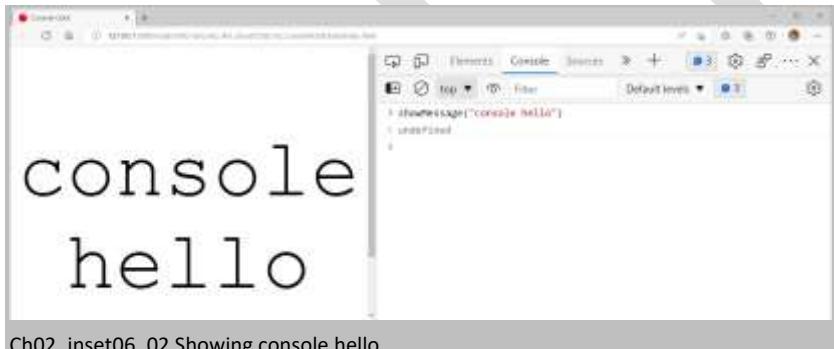
the view. This is a paragraph which has an attribute called `id` which is set to the value `timePar`. This page looks a lot like the source code of the page, but it is actually a view of the elements in the **Document Object Model (DOM)** that was created from the html file.

```
<p id="timePar" class="clock">0:0:0</p>
```

Above you can see the element that is highlighted in the image. The clock program changes the content of this element to display the time. The program looks for the element with an `id` attribute of `timepar` and then displays the time in this element. In the last chapter we used the `showMessage` function to display a message on the page. Lets look at how it works.

```
showMessage("console hello");
```

This is how we call `showMessage`. We give it an argument which is the message to be shown on the screen.



Ch02_inset06_02 Showing console hello

You can see the effect of the call of `showMessage` above. The message is displayed because the browser redrew the document and the data in the document object has changed. Now let's take a look at the `showMessage` function itself.

```
function showMessage(message) {  
    let outputElement = document.getElementById("timePar");  
    outputElement.textContent = message;  
}
```

You can see the function above. It contains just two statements. The first statement finds an HTML element in the document and the second statement sets the `textContent` property of this element to the message that was supplied as a parameter. That sounds simple enough, especially if we say it very quickly. Or not. Let's break it down into a series of steps and enter them into the console. Select the Console tab in the Developer Tools window and enter the following statement:

```
let outputElement = document.getElementById("timePar");
```

This statement finds the element in the document that displays the output (that's what

`getElementById` does). The `getElementById` method is provided by the DOM to search for elements by name. The element we have asked it to look for is a paragraph that has an `id` property set to `timePar`. The value returned by `getElementByID` is assigned to a variable called `outputElement`. Press enter to perform the statement.

The screenshot shows a browser's developer tools console tab. The code entered is:

```
> showMessage("console hello")
< undefined
> let outputElement = document.getElementById("timePar");
< undefined
>
```

The page content displays the text "console hello".

Ch02_inset06_03 Finding the output element

We know that `let` does not return a value, so the console displays “undefined”. The display has not changed because all we have done is obtain a reference to a paragraph in the document. We need to change the text in that paragraph. Enter the following statement:

```
outputElement.textContent = "element hello";
```

This statement uses the `outputElement` reference that we have just created. It puts the string “hello” into the `textContent` property of the element that `outputElement` refers to. Press enter to see what that does.

The screenshot shows a browser's developer tools console tab. The code entered is:

```
> showMessage("console hello")
< undefined
> let outputElement = document.getElementById("timePar");
< undefined
> outputElement.textContent = "element hello";
< undefined
>
```

The page content displays the text "element hello".

Ch02_inset06_04 element hello

The page now shows the message “element hello” because the `textContent` property of the `outputElement` is now “element hello”. The `showMessage` function performs these two steps and sets the `textContent` property to a parameter it has been supplied with, so we can use `showMessage` to show any message we like.

CODE ANALYSIS

Documents objects and JavaScript

The way that JavaScript programs can interact with document is very powerful. But it can also be very confusing. You might have some questions.

Question: Why is the text that we are displaying so large?

This is a good question. All the other text on the web pages we have made has been tiny. But the time message in our clock is huge, and in a different font. How does this work? It works because we are using a feature called a **stylesheet**. Let's take a look at the HTML that defines the paragraph containing the clock output:

```
<p id="timePar" class="clock">0:0:0</p>
```

This is a paragraph that initially contains "0:0:0". It has the id **timePar** (this is how our JavaScript finds the paragraph). It also contains another attribute. It has a **class** attribute which is set to the value of **clock**. The class attribute tells the browser how to display an item when it is drawn. Alongside the HTML file containing the clock web page is a field which defines the styles for this page. There is a link element in the **head** element for the clock web page that tells the browser which stylesheet file to use.

```
<link rel="stylesheet" href="styles.css">
```

This document is using the **styles.css** file. Inside this file you can find the following style definition:

```
.clock {  
    font-size: 10em;  
    font-family: 'Courier New', Courier, monospace;  
    text-align: center;  
}
```

This style information defines the **clock class**, which can then be assigned to elements in the web page. It sets the size, the font, and the alignment. When a web page is loaded the browser also fetches the stylesheet file that goes with it. You can use stylesheet files to separate the formatting of a page from the code that creates it. If the designer and the programmer agree on the names the classes to use the designer can create the styles and the programmer can create the software.

It is possible for a running program to change the style class of an element. You see this happen when invalid items on a web form are highlighted in red. What has happened is that the JavaScript processing the form has changed the style class of an element to one which has a color setting of red.

Question: What if we try to find a document element that is not there?

```
let outputElement = document.getElementById("timeParx");
```

The statement above is valid JavaScript and is intended to get a reference to the element with the id `timePar`. However, it will not do what we want because we have mis-typed the id and there is no element in the web page with the id `timeParx`. When the statement runs the `getElementById` function returns the value of `null`. This means that `outputElement` is set to `null`. The value `null` is another JavaScript special value. We've seen ones called `nan` (not a number) and `undefined` (I have not been given a value). The value of `null` is how `getElementById` can say "I looked for it, but I couldn't find it – therefore I'm giving you a reference back that explicitly means that it was not found". If a program tries to set a property on a `null` reference we get an error that stops the program:

```
> let outputElement = document.getElementById("timeParx");
< undefined
> outputElement
< null
> outputElement.textContent = "hello";
✖ ► Uncaught TypeError: Cannot set properties of null VM542:1
  (setting 'textContent')
    at <anonymous>:1:27
>
```

Ch02_inset07_01 null reference error

Above you can see the effect of trying to use a `null` reference. The message in red means that the program would stop at this point. In other words, any statements after the one that tries to set the `textContent` property on a null reference would not be performed. Later we will discover how we can detect null references and deal with statements that fail like this.

Question: What if we try to use an element property that is not there?

```
outputElement.textContentx = "hello";
```

The statement above is also legal JavaScript. But it won't display the message hello. This is because it sets a property called `textContextx` rather than the correct `textContent` one. What happens next is really very interesting. You don't get hello displayed because you've written to the wrong property. Instead, JavaScript creates a new property on the `outputElement` object which is called `textContextx`. It then sets the content of this property to "hello". This is a powerful feature of the JavaScript language. It means that code running in the web page can attach its own data to elements on the page. We will explore this feature later in the book.

Leave this page open because we are going to be working on it in the next "Make Something Happen".

Web pages and events

We are going to make a web page that contains active content which runs when the page is

loaded. The previous pages have contained JavaScript functions, but we have had to run them from the Developer Tools console. Now we are going to give them control to make truly interactive web sites. To do this we are going to bind a function to the event that is fired when a web page is loaded by the browser. In chapter 1 in the section “JavaScript Heroes: Functions” we saw how we can pass an event generator a function reference so that the function is called when the event occurs. We used it to make a ticking clock. The clock contained a function `tick` which could get the time and display it.

```
function tick(){
    let timeString = getTimeString();1
    showMessage(timeString);2
}
```

You can see the `tick` function above. I’ve expanded it slightly from the version in chapter 1 to make it clear how it works. The first statement in the function gets the time into a string called `timeString` and the second statement shows it on the screen. Each time `tick` is called it will show the latest time value. To make the clock display update continuously we need a way of calling the `tick` function at regular intervals. We can use the `setInterval` function to do this:

```
setInterval(tick,1000);
```

The `setInterval` function is called with two arguments. The first argument is a reference to `tick`, and the second argument is the interval between calls, in this case 1000 milliseconds. This will make our clock tick every second. The `setInterval` function causes an event to be created every second, so the clock will update every second. What we need next is a way of calling this function to start the clock each time the clock web page is loaded.

```
function startClock(){
    setInterval(tick,1000);
}
```

¹ Get the time

² Display the time

Above you can see a function called `startClock`. If we can find a way of calling this function when the clock web page is loaded we can make a clock that starts when the page is loaded. When we began learning JavaScript in chapter 1 we saw the importance of learning the Application Programming Interface (API) that provides functions you can perform. Now we are starting to see the importance of another kind of interface; the one provided by properties of elements in the web page itself. We've seen how we can add properties to elements in an HTML document. Each element supports a set of properties. Some of the properties can be bound to snippets of JavaScript.

```
<body onload="startClock();">
```

The body element above now has an attribute called `onload` which is the string “`startClock();`”. Properties with names that start with the word “on” are event properties that will be triggered when the event occurs. The `onload` event is triggered when the body of a page is loaded. When the page is loaded the string of JavaScript is performed by the browser in just the same way as the browser performs commands that we have typed into the console.

Make Something Happen

Make a ticking clock

If you have been following this exercise you will already have the console clock open in your browser. If not, use Visual Studio Code to open the “**Eg 01 Console Clock**” folder in the `clock` repository and then click the `index.html` file to open it in the editor. You are going to add some functions to the JavaScript code in the web page, so scroll to that part of the document.

```
function tick(){
    let timeString = getTimeString();
    showMessage(timeString);
}

function startClock(){
    setInterval(tick,1000);
}
```

These are the two functions that make the clock work. Type them into the `index.html` page inside the `<script></script>` part of the page, near the two existing functions.

Now we need to modify the `body` element to add the `onload` attribute that will call the `startClock` function. Navigate to line 10 in the file and modify the statement as follows:

```
<body onload="startClock();>
```

Now the body element has an `onload` attribute which will call the `startClock()` function when the page is loaded. Now press the Go Live button to open the page in the browser. You should see the clock start ticking.

CODE ANALYSIS

Events and web pages

Events are great fun, but you might have some questions about them:

Question: My clock is not ticking. Why is this?

There are a number of reasons why your clock might not tick. If you mis-spell the name of any of the functions they might not be called correctly. For example, if you call the starting function `StartClock` (with an upper-case S at the beginning) then this will not work, because the `onload` event is expecting to call a function called `startClock`. If you get completely stuck, head over to the **Eg 02 Working clock** folder where there is a working version of the clock.

Question: What is the difference between an attribute and a property?

Good question. A property is associated with a software object. We have seen how objects can have properties that we can access when our programs run. For example, we looked at the `name` property of a function object in chapter 1 in the section Make Something Happen - Fun with Function Objects. An attribute is associated with an element in a web page. The `body` element can have an `onload` attribute.

Question: Can you run more than one function when a page loads?

You can do this, but you wouldn't do it by adding multiple `onload` attributes. Instead you would write a single function that runs all the functions in turn, and connect that to the `onload` event.

Making a time machine clock

Now that we know how to make a clock, we are going to make a clock that lets us travel through time. Sort of. In chapter 1 in the section Make Something Happen - Console Clock we noticed that the `Date` object provides methods that could be used to set values in a date as well as read them back. If we use this to add 1000 minutes to the minute value, the `Date` object will work out the date and time 1000 minutes into the future.

```
let d = new Date();3
let mins = d.getMinutes();4
let mins = mins + 1000;5
d.setMinutes(mins);6
```

The code above shows how this would work. The first statement creates a variable called `d` that refers to an object containing the current date. The second statement creates a variable called `mins` that holds the number of minutes in the date stored in `d`. The third statement adds 1000 to the value of `mins`. The fourth statement sets the minutes value of `d` to the value in `mins`. This moves the date in `d` 1000 minutes into the future. The `Date` object sorts out the date value, updating the hours and even the day, month and year if required.

We can use this to make a time travel clock which is fast or slow by an amount that we can nominate. At some times of the day, perhaps the morning, we can make the clock go fast so we are not late for anything. At other times, perhaps when it is time to go to bed, we can make the clock go slow, so we can go to bed a little later.



Figure 2.11 Ch02_Fig_05 Time Travel Clock

Figure 2.5 above shows how it would be used. The user clicks the buttons to select a fast clock, a

³ Make `d` refer to a new `Date` object

⁴ Extract the minutes from the date

⁵ Add 1000 to the minutes value

⁶ Set the minutes in the future

slow clock, or a normal clock.

Add buttons to a page

The first thing we need to do is add some buttons to the page for the user to press. We do this with the `button` element.

```
<p>
  <button onclick="selectFastClock()">Fast Clock</button>
</p>
```

Above you can see a paragraph that contains a `button`. The button encloses the text that will appear on the button on the page. The `button` element can have an `onclick` attribute which contains a string of JavaScript to be performed when the button is clicked. When this button is clicked a function called `selectFastClock` is called. Now we need to create some code to put inside this function that will make the clock five minutes fast when it runs. We can do this by creating a **global** variable.

Share values with global variables

Up until now every variable we have created has been used inside a function body. We have used `let` to create the variable. A variable declared with `let` ceases to exist when the program execution exits the block in which the variable was declared. So when the function completes the variable is discarded. Most of the time this is just what you want. It is best if variables don't "hang around" after you have finished with them. I'm very partial to using a variable with the identifier `i` for counting. This is because I am a very old programmer. However, I don't want an `i` used in one part of the program to be confused with one used somewhere else. I like the idea of a variable disappearing as soon as the program leaves the block where it was declared.

However, in the case of the minutes offset value we want to share the value between functions. The `minutesOffset` mustn't disappear when the program exits a function where it is used. We can't declare `minutesOffset` in just one function, we must declare it so that it can be shared between all functions. We must make it **global**.

```
var minutesOffset = 0;
```

The variable `minutesOffset` above is not declared inside any code block. It is declared outside all

the functions. It is also declared using `var` rather than `let`. This means that the variable can be used in any of the functions that follow it. The value in the variable will be shared by all the functions, which is exactly what we want. If we change the value in `minutesOffset` the next time the clock is updated by the tick `function` it will draw the new time.

Programmer's Point

Global variables are a necessary evil

When you make a variable global by declaring it outside any function, you lose control of it. What do I mean by this? Well, suppose I'm working with a bunch of programmers, each of them writing some of the functions in my JavaScript application. If I declare all the variables in my functions by using `let` I can be sure that those variables can only be changed by me. I can also be sure that I won't change any values in other functions.

However, if a variable is made global it is possible for code in any of the functions view it and change it, which might lead to mistakes and makes the program less secure.

Some things must be global. It would be very hard to make the clock work without creating a global variable called `minutesOffset`. But when you write code you should start by making the variables as local as possible (by declaring them using `let`) and then make things global if you have a need for this.

JavaScript give you control of variable visibility. We will discover how to do this in the next chapter in the section "JavaScript Heroes: `let`, `var` and `const`".

```
function selectFastClock() {  
    minutesOffset = 5;7  
}
```

Above you can see the code for the `selectFastClock` function. When the function runs it sets the variable `minutesOffset` to 5. The contents of the `minutesOffset` variable are added to the minutes value when the time is displayed.

⁷ Set the minutes offset to 5

```

function getTimeString() {
    let currentDate = new Date();8
    let displayMins = currentDate.getMinutes()
        + minutesOffset;9
    currentDate.setMinutes(displayMins);10
    let hours = currentDate.getHours();
    let mins = currentDate.getMinutes();
    let secs = currentDate.getSeconds();

    let timeString = hours + ":" + mins + ":" + secs;11
    return timeString;
}

```

Above is a modified version of the `getTimeString` function. This adds the value of `minutesOffset` onto the minutes in the time string. This means that when the fast button is clicked the clock will display the time five minutes into the future. The page also has button handlers for `selectSlowClock` and `selectNormalClock` which set the value of `minutesOffset` to the appropriate values. The complete HTML file for the Time Travel Clock is shown below. You can view the code running in the example [Eg 03 Time Travel Clock](#).

```

<!DOCTYPE html>
<html>

<head>
    <title>Time Travel Clock</title>
    <link rel="shortcut icon" type="image/x-icon" href="favicon.ico">
    <link rel="stylesheet" href="styles.css">
</head>

<body onload="startClock();">
    <p id="timePar" class="clock">0:0:0</p>

    <p>

```

⁸ Get the date

⁹ Calculate the new minutes

¹⁰ Set the new minutes value

¹¹ Build the time string

¹² Return the time string

```
<button onclick="selectFastClock();>Fast Clock</button>
</p>
<p>
  <button onclick="selectSlowClock();>Slow Clock</button>
</p>
<p>
  <button onclick="selectNormalClock();>Normal Clock</button>
</p>

<script type="text/javascript">

  var minutesOffset = 0;

  function selectFastClock() {
    minutesOffset = 5;
  }

  function selectSlowClock() {
    minutesOffset = -5;
  }

  function selectNormalClock() {
    minutesOffset = 0;
  }

  function tick() {
    let timeString = getTimeString();
    showMessage(timeString);
  }

  function startClock() {
    setInterval(tick, 1000);
  }

  function getTimeString() {
    let currentDate = new Date();
    let displayMins = currentDate.getMinutes() + minutesOffset;
    currentDate.setMinutes(displayMins);
    let hours = currentDate.getHours();
    let mins = currentDate.getMinutes();
    let secs = currentDate.getSeconds();
    let timeString = hours + ":" + mins + ":" + secs;
    return timeString;
  }
}
```

```

        function showMessage(message) {
            let outputElement = document.getElementById("timePar");
            outputElement.textContent = message;
        }
    </script>
</body>

</html>

```

CODE ANALYSIS

Time Travel Clock

You may have some questions about the time travel clock:

Question: Can we make the display update immediately after a button is pressed?

There is a problem with the time travel clock. It takes a while to “catch up” when you click a button. You have to wait up to a second before you see the time change to reflect a new offset value. We can fix this by making the `selectFastClock`, `selectSlowClock` and `selectNormalClock` functions call the `Tick` function once they have updated the offset.

```

function selectFastClock() {
    minutesOffset = 5;
    tick();
}

```

Now, when the user clicks the button the clock updates instantly. You can find this version in the example [Eg 04 Quick Update Time Travel Clock](#)

Question: Can we make the clock display change color to indicate whether the clock is fast or slow?

Yes we can. We could have a different style class of the display paragraph for each of the different clock options.

```

.normalClock,.fastClock,.slowClock {
    font-size: 10em;
    font-family: 'Courier New', Courier, monospace;
    text-align: center;
}

.normalClock{

```

```
    color: black;
}

.fastClock {
    color: red;
}

.slowClock {
    color: green;
}
```

Above you can see a stylesheet that creates three styles, `normalClock`, `fastClock` and `slowClock`. You can see which settings are shared by all the styles and which just set the specific colors. The fast clock is displayed in red and the slow one in green. We can now set the style class to the appropriate style when the selection button is pressed.

```
function selectFastClock() {
    let outputElement = document.getElementById("timePar");
    outputElement.className = "fastClock";
    minutesOffset = 5;
    tick();
}
```

An HTML element has a `className` property which is set to the name of the class. We can change the style class by changing this name. The `selectFastClock` function sets the `className` for the `outputElement` to "fastClock" so that the `fastClock` style class is used to display it. So the text now turns red when the clock is fast.

Question: Is there a way to write this program without using a global variable?

At the moment the time travel clock uses a global variable called `minutesOffset` to determine whether the clock is fast or slow. Global variables are something to be avoided if possible, but how can we do this? It turns out that we can.

We've just made a modification that changes the `className` property of the time paragraph to select a different display style depending on whether the clock is fast, slow or normal. We can use the value of the `className` property on the time paragraph to set the time offset as well.

```
function getMinutesOffset(){
    let minutesOffset = 0;
    let outputElement = document.getElementById("timePar");
    switch(outputElement.className) {
        case "normalClock": minutesOffset = 0;
        break;
        case "fastClock": minutesOffset = 5;
        break;
        case "slowClock": minutesOffset = -5;
        break;
    }
}
```

```
    return minutesOffset;  
}
```

The function `getMinutesOffset` above uses the JavaScript **switch** construction to return an offset value which is 0 if the `className` property of the `timePar` element is `normalClock`, 5 if the `className` is `fastClock` and -5 if the `className` is `slowClock`. This can be used in `getTimeString` to calculate the time to be displayed.

```
function getTimeString() {  
    let currentDate = new Date();  
    let minutesOffset = getMinutesOffset();  
    let displayMins = currentDate.getMinutes() + minutesOffset;  
    currentDate.setMinutes(displayMins);  
    let hours = currentDate.getHours();  
    let mins = currentDate.getMinutes();  
    let secs = currentDate.getSeconds();  
    let timeString = hours + ":" + mins + ":" + secs;  
    return timeString;  
}
```

This version gets the value of the minutes offset and then uses it to create a time string with the required offset. There is now no need for a global `minutesOffset` variable.

This is a very good way of solving the problem. The setting is held directly on the element that will be affected by it. There is also no chance that the color of the display can get out of step with the `minutesOffset` value. This version is in the example folder **Eg 06 No Globals Clock**

Host a website on GitHub

You now have something you might like to show off to the world. What better way to do this than putting it on a website for everyone to see? Then anyone who wants a time travelling clock can just go to your site and start it running. One way to do this is to create a website repository on GitHub. To do this you must have a GitHub account. You can only host one website on your account, but the site can contain multiple pages with links between them. We are going to start with a really simple web site which just contains the time travel clock we have just made. If you want to experiment with styles you can change the color, size and font of the text in the clock. In the next chapter we'll discover how to add images to pages too.

Make Something Happen

Host a web page on GitHub



Ch02_inset08_01 clock online

Above you can see the endpoint of this exercise. This shows the clock program we have just created running in a website hosted by GitHub. Covering the steps to get this would take more pages than would fit in this chapter. You need to create an empty repository, use Visual Studio Code to clone the repository onto your PC, add the clock files to the repository, check in the changes and then synchronize the changes from the PC up to the repository. Then you just need to configure GitHub to tell it which part of the repository to share on the web and you have your new web page. The good news is that you won't be doing this very often. The better news is that I've made a step-by-step video that you can watch that takes you through the process.

Authors note: I tried to make a guided section for this, but it got much too large and convoluted. My thinking is that if they want to do this they'd be happy to watch a video.

What you have learned

This has been a very busy chapter. We've covered a lot of ground. Here is a recap plus some points to ponder.

- Git is a tool that was created to make it easier to work on large projects. It organizes units of work into repositories. A repository is a folder full of files plus a special folder managed by the git tool to track changes by making copies of changed files. You "commit" changes to the repository at which a snapshot is taken of their contents. You can return to the snapshot at any time. You can also compare the snapshot with the current

files.

- Git can be used on one machine by one person, or a git server can be set up for network access to repositories by several people. Git also provides a means of resolving changes to the same file by multiple people.
- GitHub is a cloud-based service that hosts git repositories. Users can take copies of (clone) repositories, work on them and then check them back over the network. A repository can be private to a particular user or public. Public repositories are the basis of open-source projects which have managers to accept contributions and then commit them after testing. A GitHub repository can be exposed as a web page, making GitHub a good way to host a simple web site.
- Git (and by extension GitHub) support can be added to software tools which can then make use of repository storage and management. The Visual Studio Code integrated development environment (IDE) works in this way. We can check repositories in and out as we work on them.
- Visual Studio Code also provides an extension mechanism which can be used to add extra features. The “Live Server” extension allows you to deploy websites on your PC so that you can test them.
- A web page is expressed by a file of text containing Hypertext Markup Language (HTML). The contains elements on the page that will be drawn by the browser when the page is displayed. The names of the elements are distinguished from text to be displayed by the use of < and > characters to delimit the element names, for example `<head>` is how the start of the header element would be expressed. The end of the header is expressed by an element containing the name preceded by a forward slash: `</head>`. The `
` element (line break) does not need a corresponding `</br>` element.
- HTML elements can be nested. The `<body>` element contains all the elements which are to be displayed in the body of a web page.
- HTML elements can have attributes which give information about the element. Attributes are added as named values in the definition of the element. The HTML `<p id="timePar">` marks the start of a paragraph. This paragraph has an `id` attribute which is set to the value `timePar`.
- Elements in an HTML document can be given a `class` attribute which maps back to a style definition in a stylesheet file which is loaded by the browser. The source HTML file specifies the stylesheet file location using a link element to the `head` of the document.

- The browser uses the HTML file to create a Document Object Model (DOM). The DOM is a software object that describes the web page structure. The DOM contains references to objects that represent the elements described in the HTML Page.
- Element objects in the DOM contain property values which are mapped onto the attributes assigned in the HTML. For example, the `class` attribute on an element in an HTML document is mapped onto the `className` property in the element object in the DOM. This makes it possible for JavaScript programs to change the values of properties and their appearance on the page. We used this ability to change the color of the text in the time travel clock.
- You can use the Elements tab in the Developer Tools for the browser to look at the elements in the DOM.
- An element in an HTML document can be given an id attribute that allows a JavaScript program to find it in the DOM. The method provided by a document that will get an element by its id is called, not surprisingly, `getElementById`. If the method can't find an element with the requested id it will return a null reference.
- You can change the `textContent` property of a paragraph element by assigning a string to it. The element will then display this new text on the web page.
- Some HTML elements can generate events. An event is identified by a name (usually starting with the word on) and contains a string of JavaScript code to be performed when the event occurs. The `body` element can have an `onload` attribute that specifies JavaScript to run when a web page is loaded by the browser.
- A web page can contain button elements that generate events when the user clicks on them. When the user of the web page clicks the button a JavaScript function can be called to respond to this event.
- GitHub can be made to host web pages as well as store repositories.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions".

What is the difference between Git and GitHub?

Git is the program that you run to manage your repositories. GitHub is a cloud based service that hosts repositories and is accessed using the git program.

Can I use Git on my own machine?

Yes you can. You can use it to manage a repository stored on your machine. You can also set up your own Git server and use it to store private projects on your home network.

Will I ever use the git program directly?

I try to avoid doing this too much. Most of the time Visual Studio Code will hide a lot of the complexities of git, but every now and then, particularly if you work in group projects, you might find that you have to type in a git command to fix something.

Are there different kinds of Open-Source projects?

Yes there are. This is controlled by the terms of the license agreement assigned to a particular project. It is worth reading through these to find out what permissions you can give people on the projects that you make open source. You should also check when you use an open-source project that you are not breaking any of their conditions.

Why is HTML called a markup language?

HTML is used to express the design of a view of a web page. In the days before computers a printer would write instructions on the original copy of a document they were printing. The instructions would specify the fonts and type sizes for the text to be printed. This was called "marking up". HTML can be used to express how text is to be formatted and so it was given the name markup.

What would happen if I just gave a browser a file of text?

The browser tries very hard to display something, even if it doesn't look like a proper HTML document. It would display a file of text as a single line however, as it ignores line feeds and reduces multiple spaces to 1. If you want to display separate lines and paragraphs you have to add the appropriate formatting elements.

Where does the browser store the Document Object Model?

The Document Object Model is stored in the memory of the computer by the browser when a web page is loaded. A software object provides data storage and methods that are used to interact with the data stored in the object. The HTML file sets the initial elements in the object and their initial values.

What is the difference between HTML and HTTP?

HTML is a way of expressing the content of a web page. HTTP (Hypertext Transport Protocol) is all about getting that page from the server to the browser. The browser uses the HTTP protocol to get resources and the server delivers them. The browser sends an HTTP get request (which actually includes the word GET). The server finds the resource and sends it

back. A get request always returns a status value. A status of 200 means “all is well, here is the data” whereas 404 is the infamous “file not found” error that has become so notorious that it now appears on T-shirts.

Is HTML a programming language?

No. A programming language expresses a solution to a problem. Solving a problem may involve making decisions and repeating behaviors. HTML does not have constructions for doing either of these things. HTML is used to express the contents of a logical document, not to solve problems.

What happens if a JavaScript program changes a visible property on an element very quickly?

We have seen that a JavaScript program can display messages in the browser by updating a property on an element in the document object. We set the `textContent` property on a paragraph to update the clock. If we do this very quickly the browser will not be able to keep up. The browser updates at a particular rate, usually 60 times a second. There is a way that code in a web page can get an event each time that the browser wants to update the page. We will be using this in the last chapter of this text when we look at games.

Does every element on a web page need to have an id attribute?

We added an `id` attribute to the time paragraph on the web page so that the clock program could find that paragraph and change the text on it when the clock ticks. We only need to add an `id` to the elements that the JavaScript code needs to find.

What happens if an event function gets stuck?

Events can be assigned to JavaScript functions by attributes in web pages. We have seen how the `onload` attribute of the `body` element lets us call a JavaScript function when a page loads. But what happens if the function never returns? We've all had that experience where you visit a web site and your browser stops. This can be caused by a stuck JavaScript function which is triggered by an event in the web page. If you write a function that never returns (perhaps because it gets stuck in a loop) and then call this from an `onload` attribute you will find that the web page will never complete loading. If a function called from `setInterval` never returns the web page will not stop immediately, but it will progressively slow down as the computer memory fills up with more and more unterminated processes. If a JavaScript function assigned to a button gets stuck the user will not be able to press any other buttons until that function returns.

What is the web address of a page hosted by GitHub web server?

It is an address made from the GitHub username and the name of the repository that holds the site files. However, if you register your own domain name you can configure GitHub to use it. In other words, you can set the address of the page to any domain that you can get hold of.

3

Chapter 3 Make an active site

What you will learn

We now know how to create JavaScript applications and deploy them into the cloud as part of HTML formatted web pages. We have seen how a JavaScript code can communicate with the user by changing the properties of document elements which are held in “document object model” (DOM) which is created by the browser from the original HTML. We did this by making a clock which ticked when our JavaScript code changed the `textContent` property of a paragraph that displays the time. We can also deploy our web pages and their applications into the cloud so that they can be used by anyone with a web browser.

In chapter 2 we also discovered how a program can receive input from the user in the form of button presses, in this chapter we are going to discover how a JavaScript program can accept numbers and text from the user and store their values between browsing sessions. Then we are

going to move on and start writing code that can generate web page content dynamically. And on the way we are going to learn about a bunch of JavaScript heroes.

Don't forget that you can use the Glossary to look up unfamiliar things and find the cloud meaning of things you thought were familiar.

Get input from a web page

You might find it strange, but people seem to quite like the idea of our time travel clock. However, like most people who like something you've done, they also have suggestions to make it better. In this case they think it would be a good idea to be able to set the amount the clock is fast or slow. We know that web pages can read input from the user. We have been entering numbers, text and passwords into web pages ever since we started using them. Let's see how we can do this to make an "adjustable" clock.



Figure 3.12 Ch-03_Fig_01 Adjustable Clock

Figure 3.1 above shows what we want. The time offset is entered as an input at the bottom left of the page. The entered value is set when the "Minutes offset" button is pressed. The user has entered 10 and pressed the button. Now the clock is now 10 minutes fast.

The html input element

The first thing we will need is something we can display on the web page for the user to enter text into. HTML provides the `input` element for this:

```
<input type="number" id="minutesOffsetInput" value="">
```

The html statement above creates an `input` element. The element has been given three attributes. The first attribute specifies the type of the input. This has been set to “number” which tells the browser to only accept numeric values in this input. The second attribute is an `id` for the element. This will allow the JavaScript program to find this element and read the number out of it. The third element is the initial value of the input, which we have set as an empty string.

The next thing we need is a button to press to set the new offset value. This will call a function to set the value when the button is pressed:

```
<button onclick="doReadMinutesOffsetFromPage();">Minutes offset</button>
```

We’ve seen buttons before. We used them to set the modes of the clock in chapter 1. A button can have an `onclick` attribute that specifies JavaScript to be run when the button is pressed. This button will call the function `doReadMinutesOffsetFromPage` when it is pressed.

```
function doReadMinutesOffsetFromPage (){  
    let minutesOffsetElement =  
        document.getElementById("minutesOffsetInput");13  
    let minutesOffsetValue = minutesOffsetElement.value;14  
    minutesOffset = Number(minutesOffsetValue);15  
}
```

You can see the `doReadMinutesOffsetFromPage` function above. It does you might expect from its rather long name. It finds the input element into which the user has typed the number, gets the `value` property of that element which contains the text of the number entered, converts the text of the number into a number and then sets a global variable called `minutesOffset` to hold the new value.

¹³ Get the input element

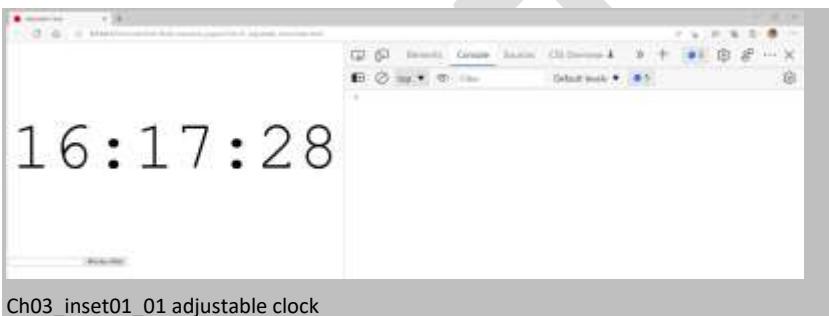
¹⁴ Get the value out of the input element

¹⁵ Convert the string into a number

Make Something Happen

Adjustable time travel clock

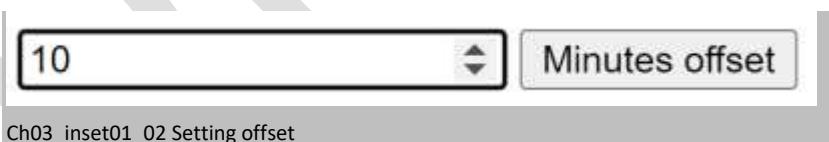
The code for this exercise is in the folder **Ch03-01_Adjustable_Clock** in the **Ch03-Build_interactive_pages** examples. Use Visual Studio to open the **index.html** file for this example, start Go Live to open the page in a browser and open the developer view for the page. Select the console view of the application.



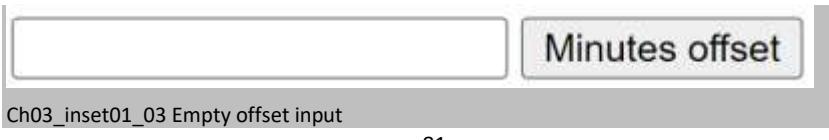
On the left you will see the ticking clock. On the right you will have the console. We can check on the current value of `minutesOffset` by using the console. Enter `minutesOffset` and press enter to ask the console to show you the contents of the variable.

```
> minutesOffset  
0
```

The listing above shows what happens. The variable is initially set as zero so the clock will show the current time. Now enter an offset value into the text box and click the Minutes offset button:



You should see the clock time change to 10 minutes in the future. Now try setting the offset value to an empty string. Clear the contents of the input box and click the Minutes offset button.



Watch what happens to the time displayed by the clock. You will notice that it goes back to the correct time. The offset has been set to zero. This is strange. You've not set it to zero, you've set it to an empty string. Some programming languages would give you an error if you tried to convert an empty string into a number. JavaScript doesn't seem to mind. Let's have a look at what is going on. The interesting function here is the one called `Number`. This takes something in and converts it into a number. We can give it a string with a number in it, and `Number` will give you a value back. Let's try a few different inputs, starting with a string that holds a value. Open the Developer Tools, select the console and type the following statement, which will show us the result provided by the `Number` function when it is fed the string "99".

```
> Number("99")
```

Now press enter. Remember that the console always shows us the value returned by the JavaScript that is executed. In this case it will show us the value returned by `Number`. Press enter.

```
> Number("99")
99
```

The result of calling `Number` with the string "99" is the numeric value 99. Now let's try a different string:

```
> Number("Fred")
NaN
```

For brevity I'm showing the call of `Number` and the result it displays from now on. You can check these results yourself in the console if you like. Converting the text "Fred" to a number is not possible. `Number` returns `NaN` (Not a number), which makes sense because "Fred" does not represent a number. Now let's try one last thing:

```
> Number("")
0
```

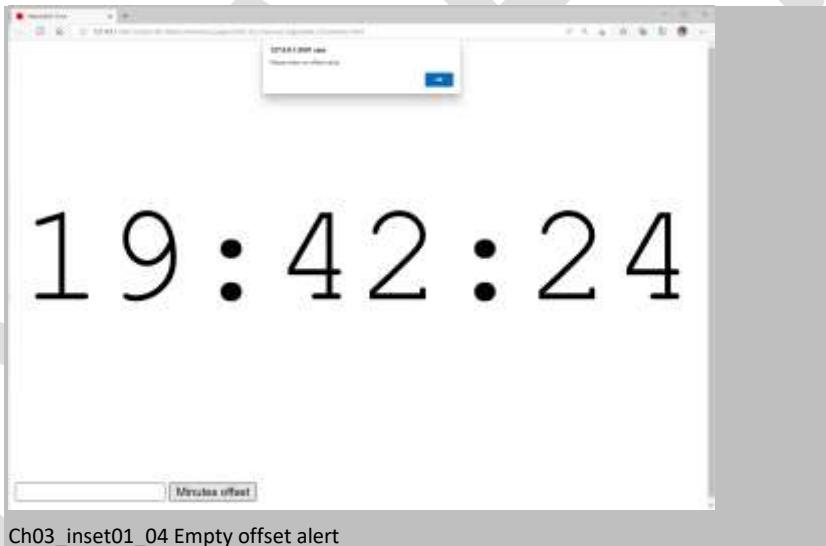
Now the `Number` function is working on an empty string. From with it returns the value 0. You might expect to see `NaN` again here, but you don't. This is one part of the reason that we get a `minutesOffset` of zero when we enter an empty field. The other part has to do with using a `type of number` for an input element. As you will have noticed when you tried to set the offset value, a number input tries not to let you enter text. On a device with a touch screen it may even display a numeric keyboard rather than a text one. If you do manage to type in an invalid number (or you leave the text out) the `input` element `value` property is an empty string. This empty string then gets fed into the `Number` function which then produces a result of 0.

This might not be what you want. You might take the view that if the user doesn't enter a number you want the `minutesOffset` to stay the same rather than go back to 0. We can fix

this by adding some extra code:

```
function doReadMinutesOffsetFromPage() {  
    let minutesOffsetElement =  
        document.getElementById("minutesOffsetInput");  
    let inputString = minutesOffsetElement.value;16  
    if (inputString.length==0){17  
        alert("Please enter an offset value");  
    }  
    else {  
        minutesOffset = Number(inputString);  
    }  
}
```

This version of the function `doReadMinutesOffsetFromPage` checks the length of the value received from the input element. If the length is zero (which means that a number wasn't entered or an empty string was entered) the function displays an alert. Otherwise, it sets the `minutesOffset` value.



Ch03_inset01_04 Empty offset alert

Above you can see what happens if you press the Minutes offset button with an empty string in the input element. You can find this version of the clock in the sample folder **Ch03-02_Improved_Adjustable_Clock**

¹⁶ Get the value from the input element

¹⁷ Check for an empty string

JavaScript input types

passwordInput : topsecret

The figure displays a vertical list of nine input fields, each with its corresponding type label in a button next to it. The input fields are:

- hello (text)
- 1234 (number)
- ***** (password)
- mail@someaddress.com (email)
- (12345) 567890 (tel)
- 19/08/2022 (date)
- 19:26 (time)
- www.robmiles.com (url)
- [color swatch] (color)

Figure 3.13 Ch-03_Fig_02 input types

Figure 3.2 above shows some of the input types available and what they look like on a web page when you enter data into them. Each input item in Figure 3.2 is has a paragraph that contains an input field with the appropriate type and a button which is used to call a function that will display the input that the browser will receive. You can enter input data into an input and then press the button next to it to display the value that is produced. The password input element has just been used to enter `topsecret` and the `password` button has been pressed to display the value in the input element. Note that the browser doesn't display the characters in the password as it is entered.

```
<p>
  <input type="password" id="passwordInput">
  <button onclick='showItem("passwordInput"); ''>password</button>
</p>
```

Above you can see the HTML that accepts the password input. The outer paragraph encloses an input element and a button element. The `onclick` event for the button element calls a function called `showItem` with the argument of "`passwordInput`". Note that the program uses two kinds of quotes to delimit items in the string that contains the JavaScript to be performed when the

button is pressed. The outer single quotes delimit the entire JavaScript text and the inner double quotes delimit the string “passwordInput” which is the argument to the function call. There are similar paragraphs for each of the different input types. The `showItem` function must find the value that was input and then display it on the output element.

```
function showItem(itemName){  
    let inputElement = document.getElementById(itemName);18  
    let outputElement = document.getElementById("outputPar");19  
    let message = itemName + " : " + inputElement.value20  
    outputElement.textContent = message;21  
}
```

The `showItem` function uses the `document.getElementById` method to get the input and output elements. It then builds the message to be displayed by adding the value in the input element onto the end of the item name. This message is then set as the content of the `outputElement`, causing it to be displayed.

The input types work differently in different browsers. Some browsers offer to auto-fill email addresses or pop up a calendar when a date is being entered. However, it is important to remember that all these inputs generate a string of text which your program will need to validate before using it. The email input doesn’t stop a user from entering an invalid email address. You can investigate the behavior of the of these types by using the page the sample folder **Ch03-03_Input_Types**

Storing data on the local machine

You show everyone your adjustable clock and they are very impressed. For a while. Then they complain that the clock doesn’t remember the offset that has been entered. Each time the clock web page is opened the offset is zero and the clock shows the correct time. What they want is a clock that retains the minutes offset value so that each time it is started it has the same offset

¹⁸ Get the source element

¹⁹ Get the destination element

²⁰ Build the message

²¹ Display the message

that they set last time it was used. They assumed that you would know that was what they wanted, and now you must provide it.

Programmer's Points

Engaged users are a great source of inspiration

It is very hard to find out from a user exactly what they want a system to do. Even when you think you have agreed on what needs to be provided you can still encounter problems like these. The solution is to provide a workflow which makes it very easy for the users to let you know what is wrong with your system and then be constructive in situations like these.

If you store your solution in a GitHub repository you get an issue tracker where users can post issues and you can respond to them. If you do this correctly you can build a team of engaged (rather than enraged) users who will help you improve your solution and even serve as evangelists for it.

JavaScript provides a way that a web page can store data on a local machine. It works because a browser has access to the file storage on the host PC. The browser can write small amounts of data into this storage and then recover it when requested.

```
function storeMinutesOffset(offset){  
    localStorage.setItem("minutesOffset", String(offset));  
}
```

The function `storeMinutesOffset` shows how we use this feature from a JavaScript program. The `storeMinutesOffset` function accepts a parameter called `offset` which holds the offset value to be stored in the browser local storage. The value is stored by the `setItem` method provided by the `localStorage` object. This method accepts two arguments, the name of the storage location and the string to be stored there.

```
function loadMinutesOffset(){  
    let offsetString = localStorage.getItem("minutesOffset");22  
    if (offsetString==null){23
```

²² Get the stored value

²³ Check for a missing value

```
    offsetString = "0";24  
}  
return Number(offsetString);25  
}
```

The `loadMinutesOffset` function fetches a stored offset value. It uses the `getItem` method which is provided by the `localStorage` object. The `getItem` method is supplied with a string which identifies the local storage item to return. If there is no item in the local storage with the given name `getItem` returns `null`. We've seen `null` before. It is a value which means "I can't find it". The code above tests for a return value of `null` from the `getItem` function and sets the offset to 0 if this happens. This behavior is required because the first time the clock is loaded into the browser there will be nothing in local storage. You can find this code in the example **Ch03-04_Storing_Adjustable_Clock**.

Make Something Happen

Software sleuthing

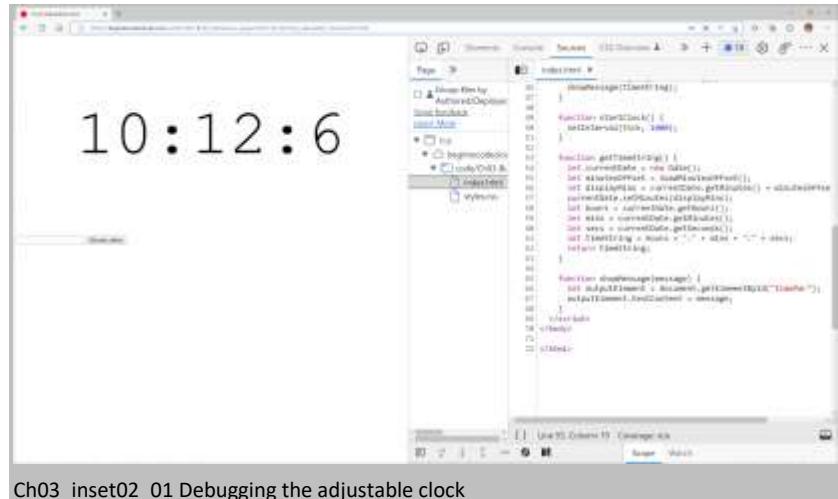
The storing adjustable clock works fine. But there is no way that the user can see the value of the minutes offset when they use the clock. It is not displayed on the page. But does this mean that it is impossible for anyone to discover this value? Let's see if we can use the debugger to get that value out of the browser. We can start by loading the clock page from the web. You can find it at the location:

https://begintocodecloud.com/code/Ch03-Build_interactive_pages/Ch03-04_Storing_Adjustable_Clock/index.html

Once you have loaded the page open the Developer Tools window, select the Sources view and open the file index.html. Then scroll down the listing until you find the `getTimeString` function. This function is called every second to display the time.

²⁴ Set the offset to 0 if nothing is stored

²⁵ Return a Number



Ch03_inset02_01 Debugging the adjustable clock

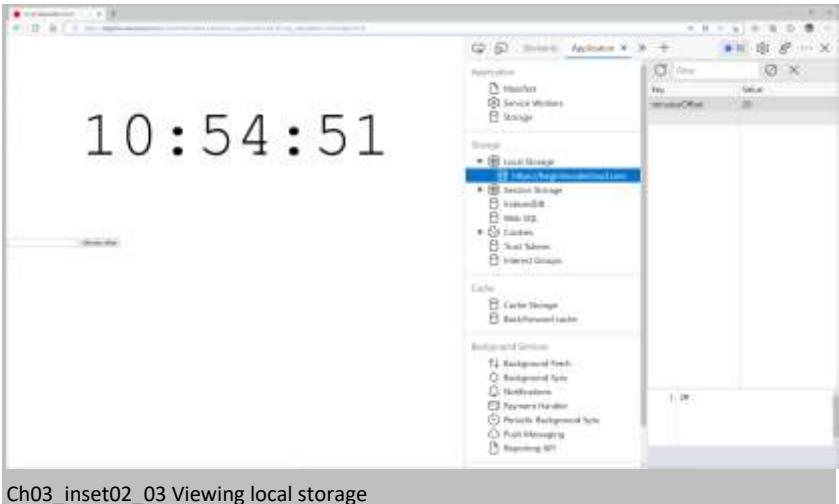
Set a breakpoint at line 56 by clicking on the left margin to the left of the line number. This function is called every second, and so the breakpoint will be hit almost instantly.

```
53     funct 20 getTimeString() {
54         let currentDate = new Date(),
55         let minutesOffset = loadMinutesOffset();
56         let displayMins = currentDate.getMinutes();
57         currentDate.setMinutes(displayMins);
58         let hours = currentDate.getHours();
59         let mins = currentDate.getMinutes();
60         let secs = currentDate.getSeconds();
```

Ch03_inset02_02 Viewing the minutes offset

You should see a display like the one above in the code window. If you hold the mouse pointer over the `minutesOffset` variable you can see the value 20 has been loaded from local storage. This shows how easy it is to view the values in a program as it runs.

However, if you want to see the values stored in local storage it turns out that there is an even easier way to do this.



Ch03_inset02_03 Viewing local storage

If you open the **Application** tab in the Developer Tools you can view the contents of Local Storage. As you can see from the image above, the value `minutesOffset` is stored as 10. Note that this local storage is shared for all the pages underneath the **beginintocode.com** domain. In other words, any of our example JavaScript applications can view and change that value. You can use this view to investigate the things that web pages are storing on your computer.

Programmer's Points

Think hard about security when writing JavaScript

It's not really a problem if someone reads and changes the minutes offset for our clock. But I hope you now appreciate just how open JavaScript is. A badly written application that stores password strings in local storage would be very vulnerable to attack, although the attacker would need to get physical access to the machine. When you write an application you need consider how exposed the application is to attacks like these. Tucking something into a local store might seem a good idea, but you need to consider how useful it would be to a malicious person. And you should make sure that variables are visible only where they are used.

JavaScript Heroes: let, var and const

Some features of a programming language are provided so you can use the language to make a working program. For example, a program needs to be able to calculate answers and make

decisions, so JavaScript provides assignments and if constructions. However, `let`, `var` and `const` are not provided to make programs work, they are provided to help us more secure code. They help us control the **visibility** of variables we use in our programs. Let's look at why variable visibility is important and how we can use `let`, `var` and `const` to manage it in JavaScript.

Making a variable in a program **global** is rather like writing your name and phone number on the notice board in the office. It makes it easy for your colleagues to contact you, but it also means that anyone seeing the notice board can call you. And someone else could erase the number you wrote and put up a different one if they wanted to redirect your phone calls to the speaking clock. In real life we need to take care of how much data we make public. It is the same in JavaScript programs. Let's look at the how we manage the scope of variables using our new JavaScript heroes.

Make Something Happen

Investigate `let`, `var` and `const`

The functions for this make something happen are in the example **Ch03-01_Variable Scope**. Use Visuals Studio to open the `index.html` file for this example, start Go Live to open the page in a browser and open the Developer Tools for the page. Now open the console tab.



Ch03_inset03_01 Investigate `let` `var` and `const`

The page shows a list of sample functions you can call from the Developer Tools Console to learn more about variables and scope. The scope of a variable is that part of a program where the variable can be accessed. JavaScript has three kinds of scope: global, function and block. A variable with global scope can be accessed anywhere in the program. A variable with function scope can be accessed anywhere in a block. A variable with block scope can be accessed anywhere in a block, except where it is “scoped out”. Let's discover what all this means with some code examples.

```
function letScopeDemo() {
    let i = 99;
    {
        let i = 100;
        console.log("let inner i:" + i);
    }
}
```

```
        console.log("let outer i:" + i);
    }
```

The above function creates two variables, both with the name `i`. The first version of `i` is assigned the value 99. This variable is declared in the body of the function. The second version of `i` is declared in the inner block and set to the value 100. Let's have a look at what happens when we run the function in the console:

```
> letScopeDemo()
let inner i: 100
let outer i: 99
```

Above you can see the call of `letScopeDemo` and the results displayed when it ran. You can run the function on your machine. When we run this function the value of each `i` is printed out. Note that within the inner block the outer variable called `i` (the one containing 99) is not accessible. We say that it is “scoped out”. When the program exits the inner block the inner `i` (the one containing 100) is discarded and the outer `i` becomes accessible again. You use `let` to create a variable that does not need to exist outside the block in which it was created. These are called global variables.

```
function varScopeDemo() {
    var i = 99;
    {
        var i = 100;
        console.log("var inner i:" + i);
    }
    console.log("var outer i:" + i);
}
```

The function `varScopeDemo` above is similar to `letScopeDemo` except that `i` is now declared using `var`. When we run it we get a different result:

```
> varScopeDemo()
var inner i: 100
var outer i: 100
```

Variables declared using `var` have function scope if declared in a function, or global scope if declared outside all functions. The second declaration of `i` replaces the original one with a new value which persists until the end of the `varScopeDemo` function. So variables declared using `var` within a block exist all the way to the end of the block and can be over written with new ones. Let's build on our understanding of scope by trying some things that might not work.

```
function letDemo() {
    {
        let i = 99;
    }
    console.log(i);
}
```

The `letDemo` function body contains a block of code nested inside it. Within this block the `let` keyword is used to declare a variable called `i` and set its value to 99. Then the block ends and the value of `i` is displayed on the console. We can run the program by just typing `letDemo()` on the console:

```
> letDemo()
Uncaught ReferenceError: i is not defined
    at letDemo (<index.html:58:21>
    at <anonymous>:1:1
```

Ch03_inset03_02 let demo

The `letDemo` function fails because the variable `i` only exists within the inner block in the function. As soon as execution leaves that block the variable is discarded, which means attempts to access that variable will fail as it no longer exists.

```
function varDemo() {
{
    var i = 99;
}
console.log(i);
```

The function `varDemo` is very similar to `letDemo`, but this time `i` is declared using `var`. Let's see what happens when we run this function.

```
> varDemo()
99
index.html:65
```

Ch03_inset03_03 var demo

This time the function works perfectly. Variables declared using `var` remain in existence from the point of declaration all the end of the enclosing scope, which in this case means the body of function `varDemo`. So, if we try to use the variable `i` from the console we will find that it no longer exists, because the function `varDemo` has finished.

```
> varDemo()
99
< undefined
> i
Uncaught ReferenceError: i is not defined
    at <anonymous>:1:1
VM428:1
```

Ch03_inset03_04 undefined i

So far everything makes perfect sense. You use a `let` if you want the variable to disappear when the program leaves the block where the variable is declared. You use a `var` if you want the variable to exist in the entire enclosing scope. So, let's try something weird.

```
function globalDemo() {
  {
    i = 99;
  }
  console.log(i);
}
```

The `globalDemo` function uses neither `let` or `var` to declare the variable `i`. You might think that this would cause an error. But it doesn't. Even stranger, the variable `i` still exists after the function has completed.

```
> globalDemo()
99
< undefined
> i
< 99
>

Ch03_inset03_05 global i
```

This is a “JavaScript Zero”. It is one of the things about JavaScript that I really don't like. If you don't use `let` or `var` to declare a variable you get a variable that has global scope, i.e. it exists everywhere. This is perhaps the worst thing that could happen. It means that if I forget the `var` or the `let` I don't get an error, I get something which exists right through my program and is open to prying and misuse.

This behavior goes back to the very first JavaScript version which was intended to be easy to learn and use. At the time it seemed a good idea to create variables automatically. Nowadays JavaScript is used to create applications which need to be highly secure and resilient, and this behavior is a bad idea. To solve the problem the latest versions of JavaScript have a `strict` mode which you can turn on by adding this statement to your program.

```
function strictGlobalDemo() {
  'use strict';
  i = 99;
}
```

The `strictGlobalDemo` function sets strict mode and then tries to create a global variable.

```
> strictGlobalDemo()
✖ Uncaught ReferenceError: i is not defined
  at strictGlobalDemo (index.html:1:107:11)
  at <anonymous>:1:1

Ch03_inset03_06 strict global
```

This function fails when it tries to automatically create the variable `i`. Note that `strict` mode is only enforced in the body of the function `strictGlobalDemo`. If you want to enforce `strict` mode on all the code in your application you should put the statement at the top of your

program, outside any functions.

Strict mode disallows lots of dangerous JavaScript behaviors, including the automatic declaration of variables. I add it to the start of all the JavaScript programs that I write.

The final hero we're going to meet is `const`. We use this when we don't want our program to change the value in a variable.

```
function constDemo() {  
  {  
    const i = 99;  
    i = i + 1;  
  }  
  console.log(i);  
}
```

In the `constDemo` function above the variable `i` is declared as a `const`. This means that the statement that tries to add 1 to the value of `i` will fail with an error. If you have a value in your program that shouldn't be changed you can declare it as constant. Variables declared using `const` inside a block have the same scope as `let`. Variables declared using `const` outside any function have global scope.

It seems obvious when we need to use `let` or `var`. We use `let` when we want to create a variable that will disappear when a program exits the block where it was declared, and we try hard not to use `var` at all (unless we have something we really want to share over the whole program. But what about `const`? When I write code I try to look out for situations where a bug can happen and then try to remove it. Look at these two statements from the adjustable clock that we created that store the time offset value in the browser:

```
localStorage.setItem("minutesOffset", String(offset));  
...  
offsetString = localStorage.getItem("minutesOffset");
```

The first statement creates a local storage item called "minutesOffset" which contains a string of text specifying the offset value. The second statement gets this value back from local storage. Can you spot anything wrong with this code? The thing I don't like about it is the way that I'm having to type the string "minutesOffset" twice. This string gives the name of the storage location that will be written to and then read back from later.

If I type one of the strings as "MinutesOffset" by mistake (I've made the first letter upper case rather than lower case) the program will either store the value in the wrong place or fail to find it when it looks for it. This means that the code has the potential for a bug. I can solve this problem completely by creating a constant variable that holds the name of the stored item:

```
const minutesOffsetStoreName = "minutesOffset";  
  
localStorage.setItem(minutesOffsetStoreName, String(offset));  
...  
offsetString = localStorage.getItem(minutesOffsetStoreName);
```

The code above shows how I would do this. This makes it impossible to miss-type the name of the store. The `minutesOffsetStoreName` is declared at global scope outside every function so that it is available over the entire program. I don't mind constant values being global as they are not vulnerable to being changed. You can find this code in the example [Ch03-06_Variable_Storage](#).

Programmer's Point

Use language features to make your code better

The `let`, `var`, `const` and `strict` features of JavaScript are not there to allow you to do things, they are there to help you make programs safer. When I create a new variable, I consider how visible it needs to be. If I need to make the value widely available I'll try to find ways to do it without creating a global variable. I also use the strict mode at all times. You should too.

Making page elements from JavaScript

We have seen how the Document Object Model (DOM) is built in memory by the browser which uses the contents of the HTML file that defines the web site. The DOM is then rendered by the browser to display contents of the pages for the user. We've also seen how a JavaScript program can interact with the elements in the DOM by changing their properties and how these changes are reflected in what the user sees on the page. We used this to change the time displayed by a paragraph in the clock.

Now we are going to discover how a JavaScript program can create elements when it runs. This is a very important part of JavaScript programming. Some web pages are built from HTML files that are entirely JavaScript code. When the page loads the JavaScript runs and creates all the elements that are used in the display. We are going to show how this works by creating a little game called "Mine Finder". It turns out to be quite compelling.

Mine Finder

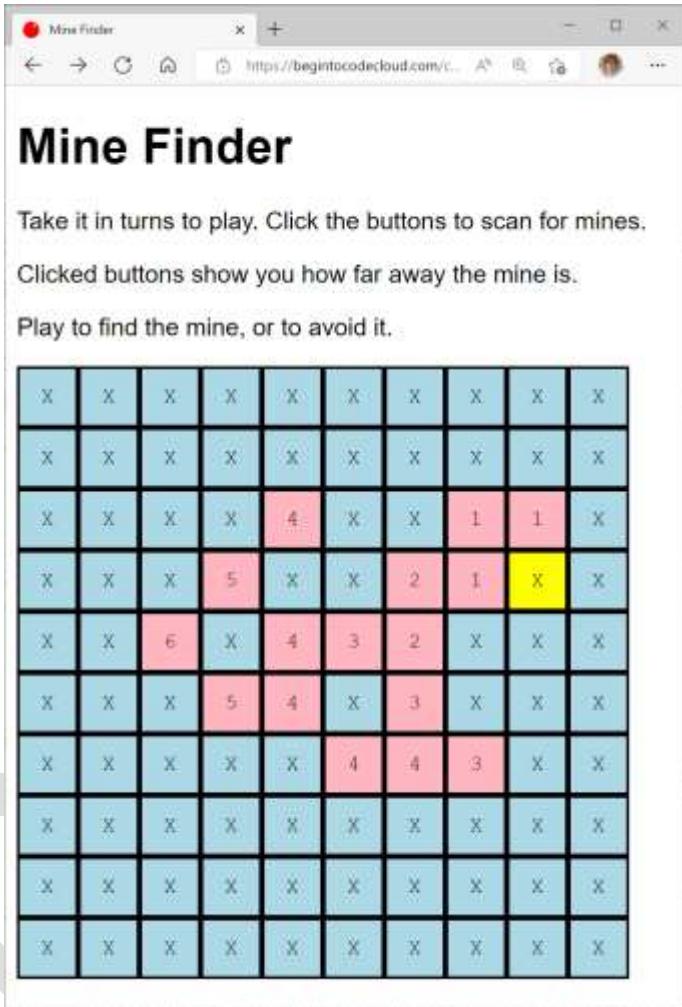


Figure 3.14 Ch-03_Fig_03 Mine Finder Game

Figure 3.3 shows the Mine Finder game. It is played on a 10x10 grid of buttons. One of the buttons contains a mine. Before you start you agree whether you are playing to find the mine or avoid it. Then each player in turn presses a button. If the button does not contain the mine it turns pink and displays the distance that square is from the mine. If the button is the mine a message is displayed, the mine button turns yellow and the game is over. Reloading the page creates a brand-new minefield and moves the mine to a new location. You can have a go at the game by visiting the example page at https://begintocodecloud.com/code/Ch03-Build_interactive_pages/Ch03-07_Mine_Finder/index.html

Place the buttons

To make the game work we need a web page that contains 100 buttons. It would be very hard to make all these buttons by hand. Fortunately, we can use loops in a JavaScript program to make the display for us.

```
<p id="buttonPar"> </p>
```

Above you can see the paragraph that will contain the buttons. In the HTML file this paragraph is empty. The buttons will be added by a function which is called when the page is loaded. The paragraph has the id `buttonPar` so that our code can locate it in the document.

```
function playGame(width, height) {  
  
    let container = document.getElementById("buttonPar");26  
  
    for (let y = 0; y < height; y++) {27  
        for (let x = 0; x < width; x++) {28  
            let newButton = document.createElement("button");29  
            newButton.className = "upButton";30  
            newButton.setAttribute("x", x);31  
            newButton.setAttribute("y", y);32  
            newButton.textContent = "X";33  
            newButton.setAttribute("onClick", "doButtonClicked(this)");34
```

²⁶ Find the destination paragraph

²⁷ Work through each row

²⁸ Work through each column in a row

²⁹ Make a button

³⁰ Set the style to “upButton”

³¹ Store the x position in the button

³² Store the y position in the button

³³ Draw an X in the button

³⁴ Add an event handler

```
    container.appendChild(newButton);35  
}  
  
let lineBreak = document.createElement("br");36  
container.appendChild(lineBreak);37  
}  
  
mineX = getRandom(0, width);38  
mineY = getRandom(0, height);39  
}
```

This is the function that creates the buttons and sets the game up. Let's work through what it does.

```
let container = document.getElementById("buttonPar");
```

This statement creates a local variable called `container` which refers to the paragraph that will contain all the buttons on the page. The paragraph has the id `buttonPar`.

```
for (let y = 0; y < height; y++) {  
  for (let x = 0; x < width; x++) {
```

These two statements create pair of for loops, one nested inside the other. The outer loop will be performed for each row of the button grid. The inner loop will be performed for each column in each row. The variable `y` keeps track of the row number, and the variable `x` keeps track of the column number.

³⁵ Append the button to the destination

³⁶ Create a line break

³⁷ Add the line break to the paragraph

³⁸ Set the X position for the mine

³⁹ Set the Y position for the mine

```
let newButton = document.createElement("button");
```

This is something we've not seen before. The document object provides a method called `createElement` that creates a new HTML element. We specify the kind of element we want by using a string. In this case we want a `button`. Note that creating an element does not add it to the DOM, we must do that separately.

```
newButton.className = "upButton";
```

The statement above sets the `className` for the button. This determines the style that will be used to display the button.

```
.upButton,.downButton,.explodeButton {  
    font-family: 'Courier New', Courier, monospace;  
    text-align: center;  
    min-width: 3em;  
    min-height: 3em;  
}  
  
.upButton{  
    background: lightblue;  
}  
.downButton {  
    background: lightpink;  
}  
.explodeButton {  
    background: yellow;  
}
```

These are the styles that are used for the buttons. There are some common style items (the font family, alignment and minimum width and height) along with different colors for each of the states of the button.

```
newButton.textContent = "X";
```

This statement sets the initial text content of the button. This will be replaced by the distance value when the button is clicked.

```
newButton.setAttribute("x", x);  
newButton.setAttribute("y", y);
```

These two statements set up a couple of attributes on the new button that give the location of the button in the grid. We are going to bind a function to the `onclick` event of the button. We don't want to create a different function for each button press. That would mean creating 100 functions. Instead we want to store location values in each button so that a single button function can work the position of a particular button. We have already written code that sets existing attributes on an element (to change the `class` or the `textContent` of a paragraph). The two statements above create attributes called `x` and `y` that contain the `x` and `y` positions of the button. This is a very powerful technique. It makes elements in the DOM into an extension of your variable storage.

```
newButton.setAttribute("onClick", "doButtonClicked(this);");
```

Above is the last statement that sets up the button. It binds a method called `doButtonClicked` to the `onClick` event for the button. If the button is clicked this function will run. All the buttons will call the same function when they are clicked. You might be wondering how the `doButtonClicked` function will know which button has been clicked. Let's take a look at the JavaScript statement that is being assigned to `onClick` to find out how this works.

```
doButtonClicked(this);
```

When executed in the context of a JavaScript statement running from HTML the value of the `this` is a reference to element generating the event. So, each time `doButtonClicked` is called it will be given an argument which is a reference to the button that has been clicked. This is terribly useful. It makes it very easy for an event handler to know which element caused the event.

If you are having bother understanding what is happening here, remember the problem that we are trying to solve. We have 100 buttons. Each button can generate an `onClick` event. We don't want to make 100 functions to deal with all these `onClicks`. We would much prefer just to write one function. But if we only have one function; it needs to know which button it has been called from. The `this` reference is an argument to the call of `doButtonClicked` which is fed into the

function when the button is clicked. In this context, the value of `this` refers to the button that has been pressed. So `doButtonClicked` is always told the button that has been clicked.

This will make more sense when we look at what the `doButtonClicked` function does. And we have a whole JavaScript Hero description for the `this` keyword coming up. For now, if you are happy with the value of `this` delivering a reference to the button that was pressed, we can move on to the next part of the button setup.

```
container.appendChild(newButton);
```

Way back at the start of this description we set up a variable called `container` which was a reference to the paragraph which is going to hold all our buttons. The `container` provides a method called `appendChild` which is given a reference to the new element and adds it. This means that the paragraph now contains the newly created button. New elements are appended in order. So the first element will be button (0,0) and the second (0,1) and so on.

```
let lineBreak = document.createElement("br");
container.appendChild(lineBreak);
```

These two statements are performed after we have added all the buttons in a row. They create a break element (`br`) and then append it to the paragraph container. This is how we separate successive rows in the grid.

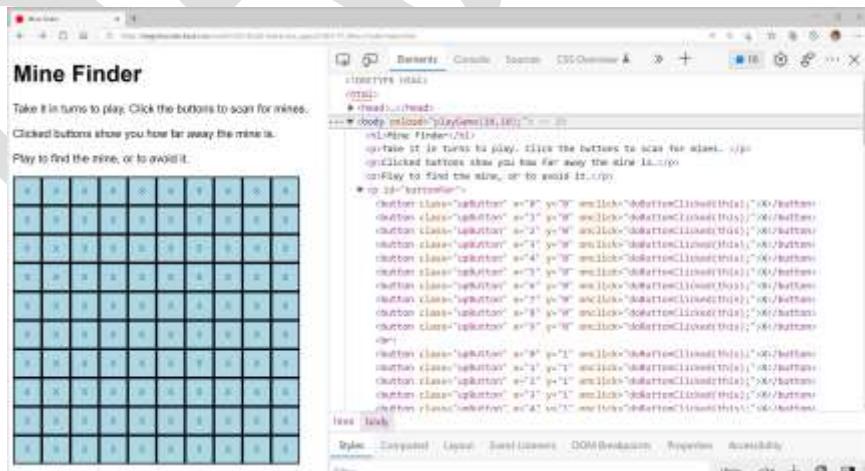


Figure 3.15 Ch-03_Fig_04 Mine Finder buttons

Figure 3.3 above shows how we can use the Elements tab from the Developer Tools to look at all the buttons that have been created by our code. Remember that the original HTML for the page did not contain any buttons. These have all been created by our code. You can see that all the buttons have the properties that you would expect.

Place the mine

The next thing the game needs to do is place the mine somewhere on the grid. For this we need random numbers. JavaScript has a random number generator which can produce a random value between 0 and 1. It lives in the [Math](#) library and is called [random](#). We can use this in a helper function to generate random integers in a particular range:

```
function getRandom(min, max) {  
    var range = max - min;  
    var result = Math.floor(Math.random() * (range)) + min;  
    return result;  
}
```

The [getRandom](#) function is given the minimum and maximum values of the random number to be produced. It then creates a value which between the two values. The maximum value is an exclusive upper limit. It is never produced. The function uses [Math.random](#) to create a random number between 0 and 1 and [Math.floor](#) to truncate the fractional part of a number and generate the integer value that we need.

```
mineX = getRandom(0, width);  
mineY = getRandom(0, height);
```

These two statements set the variables [mineX](#) and [mineY](#) to the position of the mine. These variables have been made global so that they are shared between all of the functions in the game.

```
var mineX;  
var mineY;
```

Making these values global makes the game a bit less secure, but it also keeps the code simple.

Respond to button presses

The final behavior that we need is the function that responds to a button press. If you look at the buttons definitions in Figure 3.4 above you will see that the `onClick` attribute of each button makes a call of the `doButtonClicked` function. Let's have a look at this function.

```
function doButtonClicked(button) {  
    let x = button.getAttribute("x");40  
    let y = button.getAttribute("y");41  
    if (x == mineX && y == mineY) {42  
        button.className = "explodeButton";43  
        alert("Booom! Reload the page to play again");44  
    }  
    else {45  
        let dx = x - mineX;46  
        let dy = y - mineY;47  
        let distance = Math.round(Math.sqrt((dx * dx) + (dy * dy)));48  
        button.textContent = distance;49  
        button.className = "downButton";50  
    }  
}
```

The `doButtonClicked` function has a single parameter, which is a reference to the button that has

⁴⁰ Get the x position of the button

⁴¹ Get the y position of the button

⁴² Check to see if this is the mine button

⁴³ Set the button style to “explode”

⁴⁴ Tell the player they have found the mine

⁴⁵ Do this part if the mine was not found

⁴⁶ Get the x distance to the mine

⁴⁷ Get the y distance to the mine

⁴⁸ Work out the distance

⁴⁹ Put the distance value into the button

⁵⁰ Set the button to the “down” style

been clicked. This is obtained from the `this` reference which is added when the event is bound in the element definition.

```
let x = button.getAttribute("x");
let y = button.getAttribute("y");
```

The first two statements in the function read the values in the `x` and `y` attributes on the button. These give the location of the button in the grid.

```
if (x == mineX && y == mineY) {
    button.className = "explodeButton";
    alert("Booom! Reload the page to play again");
}
```

The next set of statements checks to see if this button is at the location of the mine. If both the `x` and the `y` values match the statements set the class style for the button to “`explodeButton`”. This causes the button to turn yellow. Then an alert is displayed to tell the players the game is over.

```
else {
    let dx = x - mineX;
    let dy = y - mineY;
    let distance = Math.round(Math.sqrt((dx * dx) + (dy * dy)));
    button.textContent = distance;
    button.className = "downButton";
}
```

The final part of this function is the behavior that is performed if the button is not the mine. The first three statements use the laws of Pythagoras (the square of the hypotenuse is equal to the sum of the squares on the other two sides of a right-angled triangle) to work out the distance from this button to the mine. It then sets the text content of the button to this value and changes the style to `downButton`, which turns the button red.

Playing the game

The game is quite fun to play, particularly with two or more opponents. If you want to make the

game larger (or smaller) you just change the call of `playGame` which is bound to the `onload` event in the body of the html. This is where the number of rows and columns is set.

```
<body onload="playGame(10,10);">
```

The grid is made up of rows of buttons separated by line breaks. If the user makes the browser window too small the rows of buttons wrap round. We could fix this by positioning the button absolutely or by displaying the buttons in a table construction. We could create the table programmatically as we have created the buttons and then add elements to the table to make the required rows and columns.

Using events

The present version of Mine Finder works perfectly. But it turns out that there is a neater way of connecting events to JavaScript functions. At the moment we are using this statement to connect an event to an object:

```
newButton.setAttribute("onClick", "doButtonClicked(this);");
```

This works by creating an `onClick` attribute on a new button and then setting it to a string of JavaScript which calls the method that we want (and uses `this` to provide a pointer to the button being clicked). This works because it is exactly what we would do if we were setting the event handler of an element in the HTML file. However, this is not the neatest way of doing it if we are creating an element in software.

The major limitation of this technique is that we can only connect one event handler this way. We might have a situation where we want several events to fire when the button is clicked. This is not possible because an HTML element can only have one of each attribute. However, we can use a different mechanism to connect the button click handler.

```
newButton.addEventListener("click", buttonClickedHandler);
```

The statement above uses the `addEventListener` method provided by the `newButton` object to add an event listener function to a new button. The name of the event is specified by the string, in this case the event we want is “click”. The second parameter is the name of the method to be called when the button is clicked. We are using a method called `buttonClickedHandler`. After the

event listener has been added the function `buttonClickedHandler` will be called when the button is clicked.

In the earlier event handler code we used this to deliver a reference to the button that has been pressed. How does the `buttonClickedHandler` function know which button it is responding to? Let's take a look at the code of the function:

```
function buttonClickedHandler(event){  
    let button = event.target,51  
    . . .  
}
```

The `buttonClickedHandler` function is declared with a single parameter called `event` which describes the event that has occurred. One of the properties of the event object is a called `target`. This is a reference to the element that generated the event. The `buttonClickedHandler` function extracts this value from the event and sets the value of `button` to it. The function then works in the same way as the earlier version. You can find this code in the example **Ch03-08_Mine_Finder Events**.

Improve Mine Finder

The game is quite fun, but you might like to make some improvements. Here are some ideas for things that you might like to do:

- You could add a counter that counts the number of squares visited. They you could have a version where the aim is to find the mine in the smallest number of tries.
- You could add a timer which counts down. A player must find the mine in the shortest time (clicking as many squares as they like).
- You could change the way that the distance to the mine is displayed. Rather than putting a number in the square you could use a different color. You would need to create 10 or so new styles (one for each color) and then you could use an array of style names that you index with the distance value to get the style for the square. This might make for some nice-looking displays as the game is played.

⁵¹ Get the button reference from the event description

What you have learned

This has been another busy chapter. We've covered a lot of ground. Here is a recap plus some points to ponder.

- A web page can contain input elements that are used to read values from the user. An input tag can have different types, for example text, number, password, date and time. The value of an input tag is always delivered as a string and needs to be checked for validity before it is used. The input tag behaves differently in different browsers.
- The `Number` function is used to convert a string of text into a number. If the text does not contain a valid number the function will return `NaN`. If the text is empty the function returns 0.
- A browser provides local storage where a JavaScript application can store values that persist when the web page is not open. Local storage is provided on a “per site” basis, i.e. each top domain has its own local storage. Local storage is implemented as named strings of text. The Application tab of the browser Developer Tools can be used to view the contents of local storage. Local storage is specific to one browser on one machine.
- JavaScript variables can be declared local to a block of code using the keyword `let`. These variables are discarded when program execution leaves the block where they are declared. Using `let` to declare a variable in an inner block “scopes out” a variable with the same name which was declared in an enclosing block. An attempt to use a variable outside its declared scope will generate an error and stop the program.
- JavaScript variables can be declared global using the keyword `var`. Variables declared using `var` in a function body are global to that function but not visible outside it. Variables declared using `var` outside all functions are global and are visible to all functions in the program.
- Global variables represent risk. A global variable can be viewed by any code in the program (which represents a security risk) and it can be changed by any code in the program, which represents a risk of unintentional change or vulnerability to attack.
- Variables that are not explicitly declared (i.e. not declared using `let` or `var`) are global to the entire program. This dangerous default behavior can be disabled by adding a “use strict;” statement to the function or at the start of the entire program.
- You can declare a variable using `const`, which prevents the value assigned

to the variable being changed.

- A JavaScript program can add elements to the Document Object Model. This makes it possible for the contents of a web page to be created programmatically, rather than being defined in the HTML file that describes the page. You can view the elements in a web page (including those created by code) by using the Elements view in the Developer Tools.
- An element created in a JavaScript program can have additional attributes added to it. We used this to allow a button in the Mine Finder game to hold its x and y position in the grid.
- Creating a new HTML element in a JavaScript program does not automatically add it to the page. The `appendChild` function can be used on the container element (for example a paragraph) to add the new element. When the element is added the page will be re-drawn and the new element will appear on the display.
- The JavaScript `this` keyword can be used in the string of JavaScript bound to an event handler. In this context the `this` keyword provides a reference to the object generating the event. In the case of the Mine Finder program we use this to allow 100 buttons to be connected to the same event handler. By passing the value of `this` into the event handler we can tell the handler which button has been pressed.
- It is also possible to use the `addEventListener` method provided by an element instance (in our case a button in the Mine Finder game) to specify a function to be called when the event occurs. The event handler function is provided with a reference to event details when it is called. These details include a reference to the object that caused the event.
- JavaScript provides a `Math.random` function which produces a random number in the range 0 to 1. We can multiply this value by a range to get a number in that range.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions".

Does the user always have to press a button to trigger the reading of an input?

You can use the `oninput` event to specify a function to be called each time the content of an input box changes. This means that if the user was typing in a number the `oninput` function would be called each time a digit was entered.

Is Number the only way to convert a string to a number?

No. JavaScript provides functions called `parseInt` and `parseFloat` which can be used to parse a string and return a value of the requested type. These behave slightly differently from `Number`. A string that starts with a number would be regarded as that number. For example parsing "123hello" would return the value 123, whereas `Number` would regard this as `NaN`. The parse functions also regard an empty string as `NaN`. It doesn't matter whether you use `Number` or the parse functions, just be mindful of the slight differences in behavior.

How much local storage can I have on a web site?

The limit is around 5Mb for a browser on a PC.

How long are variables stored in local storage?

There is no limit to the time that a value will be stored.

Can I delete something from local storage?

Yes you can. The `removeItem` method will do this. However, once deleted it is impossible to get the data back.

How do I store more complex items in local storage?

Local storage stores a string of data. You can convert JavaScript objects into strings of text encoded using the JavaScript Object Notation (JSON). This allows you to store complex items in a single local storage location. We will be doing this later in the text

How do I protect items stored in local storage?

You can't. They are public. The solution is never to store important data in local storage. You should store such data on the server (which users don't have access to). We will be doing this in the next part of the book.

Can I stop someone looking through the JavaScript code in my web page?

No. There are tools you can use that will take your easy-to-understand code and make it much more difficult to read. These are called obfuscators. However, there are also tools that can unpick obfuscated code. The only way to make a properly secure application is to run all the code in server, not the client. We will be doing this in part 2 of the book.

What is the difference between `let` and `var`?

A variable declared using `let` will cease to exist a program leaves the block in which the variable was declared. This makes `let` very useful for variables that want to use for a short time and then discard. A variable declared using `var` has a longer lifetime. If it is declared in a function body the variable will exist until the function exits. A variable declared as `var` at global level (i.e. outside all functions) is global and will be visible to code in all the functions

in the application. Global variables should be used with caution. They provide convenience (all functions can easily access their content) at the expense of security (all functions can easily access their contents).

What does strict do?

Strict mode changes the behavior of the JavaScript engine so that program constructions which might be dangerous are rejected. One of the things that strict does is stop the automatic creation of global variables when a programmer doesn't specify let or var at the variable creation.

When do I use a constant?

You use a constant when you have a particular value which means something in your code. Using a constant makes it easy to change the value for the entire program. It also reduces the chance of you entering the value incorrectly. Finally, it lets you make a program clearer. Having a constant called `maxAge` in a program, rather than the value 70 makes it very clear what a statement is doing.

Can document elements created by JavaScript have event handlers?

Yes they can. We have actually done this. The best way is to use the `addEventListener` to specify the function to be called when the event occurs.

How does an event handler know which element has triggered an event?

We have done this in two different ways. In the first version of Mine Finder we added a `this` reference to the call of the function that handled the event. This function was specified in the text of a function call bound to the “onClick” attribute added each button element. The function then received the `this` reference (which in this context is set to a reference to the element generating the event) and used it to locate the button that was pressed.

The second way we did this, which is more flexible, used the `addEventListener` method on the new button to add the event handler. When the event handler is called by the browser in response to the event it is passed a reference to an Event object which contains information about the event, including the property `target` which contains a reference to the element that generated the event.

4

Host a website

What you will learn

We now understand how a JavaScript program can interact with the browser and the Document Object Model (DOM) of a web page to create interactive websites. We know that the HTML page that defines a document is only the starting point of the definition of a web site. A JavaScript program can run when a page loads and dynamically create content and connect events to make an active web page. Now we are going to discover how JavaScript can be used to host a web site using node.js. Node.js allows us to run JavaScript programs directly on our computer. In this chapter we are going to discover how a JavaScript program can operate as a web server and generate web sites dynamically. We are also going to take a detailed look at how JavaScript programs can be broken down into modules.

As ever, the Glossary is around to help you with terms you've not seen before.

node.js

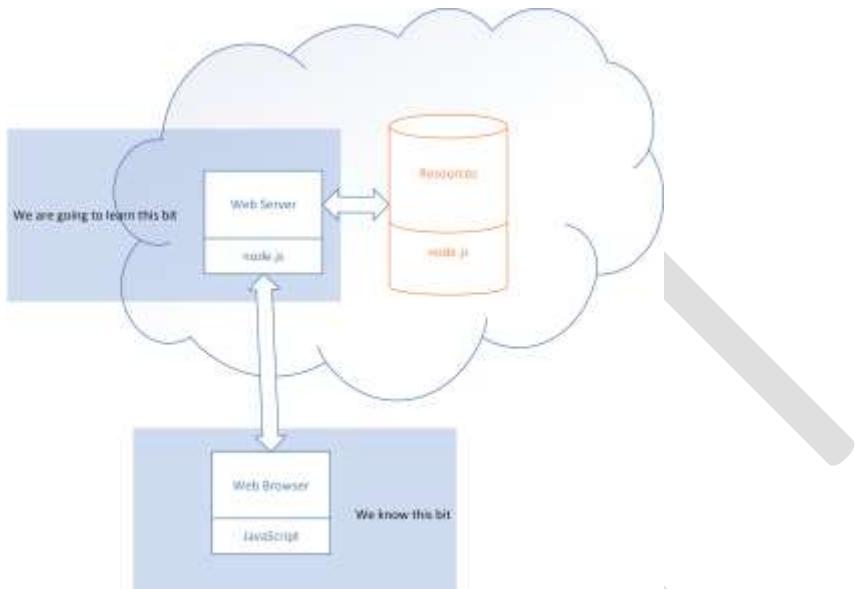


Figure 4.16 Ch-04_Fig_01 Get the role of node.js

Figure 4.1 above shows where we are now in the process of learning how to write applications for the cloud. We can write JavaScript programs that run in the browser and communicate with the user via the elements in the Document Object Model (DOM). Now we are going to learn how to write JavaScript programs that run in the server and respond to requests from the browser. The JavaScript programs that we are going to write will run inside a framework called node.js. Node.js was created by taking the JavaScript component out of a browser and turning it into a free-standing program. I'm going to call it "node" for the rest of this book. The first we need to do is get node running on our computer.

Make Something Happen

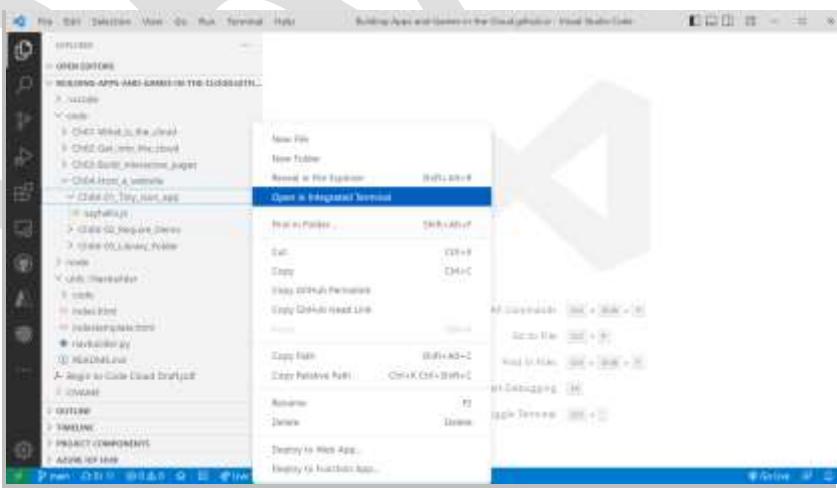
Install node

Before we can use node we need to install it on our machine. The application is free. There are versions for Windows PC, Mac OS and Linux. Open your browser and go to the <https://nodejs.org/en/download/> page.



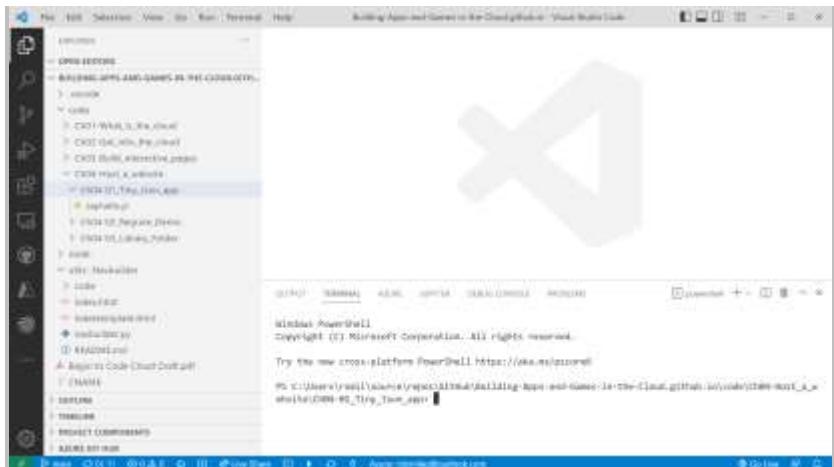
Ch04_inset01_01 Get node.js

Click on the link for the installer for your machine and go through the installation process. Select all the default options that are suggested. Now, let's have a chat with node using the Visual Studio Code Terminal. Start Visual Studio Code and open the GitHub folder for the sample code. Now we want to start a terminal in the folder that contains a JavaScript program we are going to use to test our node installation. Open the explorer view in Visual Studio by clicking the icon at the top of the column on the left. Now look at the code samples and find the folder called **Ch04-01_Tiny_Json_app**. Right click on this folder to open the context menu as shown below.



Ch04_inset01_02 opening terminal

Select **Open in Integrated Terminal** from this menu to start a terminal session in this directory. A new terminal window will open at the bottom right of the page.



Ch04_inset01_03 terminal window

You use this window to send commands to the operating system of your computer. I'm using a Windows PC and so my commands will be performed by a terminal program called Windows PowerShell. If you are using a Mac or a Linux machine you will use the terminal program on that machine.

We tell the terminal what to do by entering text commands. For now, we are going to use the terminal to start node running. Click in the terminal window to make it active and then type the **node** at the terminal command prompt as shown below.



Ch04_inset01_04 starting node

Next press Enter to start the node program running. The node program is now running inside the terminal. Any commands that you enter will be processed by node, not the terminal.



Ch04_inset01_05 running node

When node starts it displays a console prompt. If you don't see the output above, make sure that the install process completed correctly. Enter the sum below and watch what happens.

```
> 2+2+2  
6
```

You should see how eager node is to help as you type in the calculation. It will execute the partial statement and show results even before you've typed in the whole line. You can use this console in the same way as you use the console in the browser developer tools. You can type in a JavaScript statement; it will be obeyed and the result it generates is printed.

We are not going to be using the node console very much. We've just run it now to make sure that node works on our machine. We will be using node to run JavaScript programs that we have written. To exit node type in `.exit` (don't forget the period at the beginning) and press enter. This will stop node running and return you the terminal.

The terminal has been opened in a directory which contains a simple JavaScript source file. We can now use node to run this program. To do this we issue the `node` command and follow it with the name of the JavaScript source file. We want to run a program in the file called `sayhello.js`

```
console.log("We can run this program using node");
```

Above you can see the contents of the program file. It just prints a message on the console. We can run this program by starting the node program and giving it the filename as an argument. Type in "node sayhello" as shown below.

```
Try the 'man' command in a PowerShell: https://aka.ms/powershell  
PS C:\Users\simon\source\repos\GitHub\building-apps-and-games-in-the-cloud\github\src\code\Ch04-Host_a_website\Ch04-01_Tiny_Tiny_app> code  
Welcome to Node.js v16.17.0.  
Type ".help" for more information.  
> 2+2  
4  
> .exit  
PS C:\Users\simon\source\repos\GitHub\building-apps-and-games-in-the-cloud\github\src\code\Ch04-Host_a_website\Ch04-01_Tiny_Tiny_app>
```

Ch04_inset01_06 running sayhello

Now press enter. The terminal will run the `node` program and pass it the string `sayhello` as an argument to the program. The node program will open the file `sayhello.js` and run the JavaScript code in that file.

```
PS C:\Users\simon\source\repos\GitHub\building-apps-and-games-in-the-cloud\github\src\code\Ch04-Host_a_website\Ch04-01_Tiny_Tiny_app> code  
Welcome to Node.js v16.17.0.  
Type ".help" for more information.  
> 2+2+2  
6  
> .exit  
PS C:\Users\simon\source\repos\GitHub\building-apps-and-games-in-the-cloud\github\src\code\Ch04-Host_a_website\Ch04-01_Tiny_Tiny_app> node sayhello  
PS C:\Users\simon\source\repos\GitHub\building-apps-and-games-in-the-cloud\github\src\code\Ch04-Host_a_website\Ch04-01_Tiny_Tiny_app>
```

Ch04_inset01_07 sayhello output

Above is the result of running the `sayhello` program. The message has been displayed on the terminal, the program has ended, and the node program has ended. You can now close the

terminal session by using the **exit** command.

CODE ANALYSIS

Running node

You might have some questions about terminal and node.

Why does mine not work?

There are a few reasons why your program might not run. The most obvious one is that you might have typed the wrong filename. If you give filename of **syHello** the node program will be unable to find such a file and you get an error. The error is not a simple “Hey, you got the filename wrong”. Instead, you get a whole bunch of error reports followed by the message “MODULE_NOT_FOUND”. You can get also get this error if your filename doesn’t have the language extension “.js” which identifies a file as holding JavaScript code. And, most confusingly, you can get this error on some systems if you give a filename of **sayhello**.

On a Windows PC, whether letters in a file name are CAPITALS or lower case is not significant, so if you enter the name **sayhello** the Windows file system will quite happily match this with **sayHello**. Unfortunately, operating systems based on the Unix – which includes those used in Linux and Apple systems don’t do this matching, which means we can issue commands that work on a Windows device but don’t work on others.

There’s another reason that the command might not work, and that is because you are running the terminal program in the wrong directory. Whenever you are using terminal it keeps track of its “current directory”. Whenever a command or a program specifies a filename, the terminal program will look in the current directory for the file. We started Terminal by using the command “Open in Integrated Terminal” at the directory containing our example code. If we had opened the wrong directory the **node** program would be unable to find the **sayHello** program. The **cd** command lets you tell terminal to move to a different directory. You can find out more about the **cd** command in the glossary entry for the terminal.

What happens if the sayhello program contains an infinite loop?

The node program will run a JavaScript program until the JavaScript program ends. If the JavaScript program never ends, node will keep running. This is frequently what we want. If node is hosting web server program written in JavaScript we want the server program to run for ever and so server program will run forever. We can stop a running JavaScript program in node by using keycode CTRL+C (i.e. hold down the control key and press C) in the terminal window.

How does a node program communicate with the user?

The node system will run a JavaScript program but it doesn't provide a document object, so we can't create HTML elements and then change their properties to create a display for the user. Node is intended for tasks such as hosting web sites and node programs do not usually interact with users. However, a node program can use the `console.log` function to send messages. These messages can be very helpful when monitoring applications run by node.

How does the terminal program find the node program?

This is an interesting question. We've just discovered that the Terminal program looks in the current directory to find files. However, the node program isn't in the current directory. And yet the Terminal program can find the node program and run it. How does this work?

The operating system manages a set of **environment variables** which, as the name implies, describe the environment for programs on that computer. One of these variables is called the **path**. The path is a list of "places to look" for things. If I can't find my keys I'll check the front door, the kitchen door, the key hooks, my pocket and finally my right hand. This list of locations is my "path" for finding keys. In the case of my computer the path is places to look for a program to run.

When you enter the command **node** the terminal program searches through all the directories specified in the **path** variable to see if any of them contain a program called node. When it finds the node program it runs it. The installation process for the node application added the location of the directory containing the node program to the path on the computer. You can use the PowerShell command `$Env:path` to display the contents of the path on your Windows PC. You might be surprised by the number of different places there are:

```
C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System  
32\WindowsPowerShell\v1.0\;C:\WINDOWS\System32\OpenSSH\;C:\Program  
Files\Microsoft SQL Server\130\Tools\Binn\;C:\Program Files\Microsoft SQL  
Server\Client SDK\ODBC\170\Tools\Binn\;C:\Program  
Files\Git\cmd;C:\ProgramData\chocolatey\bin;C:\Program Files\dotnet\;  
C:\Program Files\CMake\bin;C:\Program Files\nodejs\;  
C:\Users\rsmil\AppData\Local\Mu\bin;C:\Users\rsmil\AppData\Local\Microsof  
t\WindowsApps;C:\Users\rsmil\AppData\Local\Programs\Microsoft VS  
Code\bin;C:\Users\rsmil\AppData\Local\GitHubDesktop\bin;C:\Users\rsmil\.d  
otnet\tools;C:\Users\rsmil\AppData\Local\Microsoft\WindowsApps;C:\Program  
Files\heroku\bin;C:\Users\rsmil\.dotnet\tools;C:\Users\rsmil\AppData\Roam  
ing\npm
```

Above you can see some of the directories in the `$Env:path` variable on my machine. The directory containing node is in bold.

Programmer's Point

Learn to use the terminal

It is worth spending some time learning how to get the best out of the **terminal**. The tab shortcut is just one of many tricks that you can use. You can find out more about the terminal and how to use it in the Glossary.

JavaScript Heroes: Modules

Node is much more than just a place you can run JavaScript programs. It also allows you to create programs that are made up of modules. Modules are another JavaScript hero you should know about. A module is a package of JavaScript code that you want to reuse. A module can contain functions, variables, and classes. Some of the elements in a module source file can be “exported” from the module for use in other programs. The first implementation of modules was created as part of the node framework. It uses a function called `require` to import items from a module that has exported them. Let’s look at how it works.

Create a module and require it

Let’s consider something we might like to turn into a module. In the last chapter we created a function called `getRandom` to generate random numbers. We used it to pick the position of the mine in the game Mine Finder that we built.

```
function getRandom(min, max) {  
    var range = max - min;  
    var result = Math.floor(Math.random() * (range)) + min;  
    return result;  
}
```

Above you can see the code for `getRandom`. When a program needs a random number in a particular range it can use the function to get one:

```
let spots = getRandom(1,7);
```

The above statement creates a variable called `spots` which is set to the result of a call to `getRandom`. The variable `spots` will contain a value in the range 1 to 6. We could use the value of `spots` to replace dice in a board game. Note that the upper limit of our random number generator is *exclusive*. We will never get 7 returned as the number of spots.

We might want to use `getRandom` in another application that also needs random numbers. We

could just copy the text of the function into the new application, but if we ever find a bug in `getRandom` we would then have to find all the applications where `getRandom` has been used and fix the code in each one. If our programs all used a single shared version, we'd just have to fix the fault in one file and then it would be fixed in all the programs. Modules have other advantages too. A module can be developed independently of the rest of the application, perhaps by a different programmer.

```
function getRandom(min, max) {52
    var range = max - min;
    var result = Math.floor(Math.random() * (range)) + min;
    return result;
}
exports.getRandom = getRandom;53
```

We can make a JavaScript file into a module by adding an `exports` statement that specifies what is being exported. You can see it above. The code above exports just one function; `getRandom`. The name of the function as it is exported is also `getRandom`.

This code could be placed in a source file called `randomModule.js`. A program that wants to use the `getRandom` function can use the `require` function to load the module containing it.

```
const randomModule = require("./randomModule");54
let spots = randomModule.getRandom(1,7);55
```

The two statements above show how a node.js application uses `getRandom`. The variable `randomModule` is declared as a constant and then set to refer the result from the `require` function. The `require` function is supplied with a string giving the file path to the source file `randomModule.js`. The character sequence `."/` in front of the string tells the require function to look in the same directory as the one containing the program.

Make Something Happen

⁵² Create the function to be exported

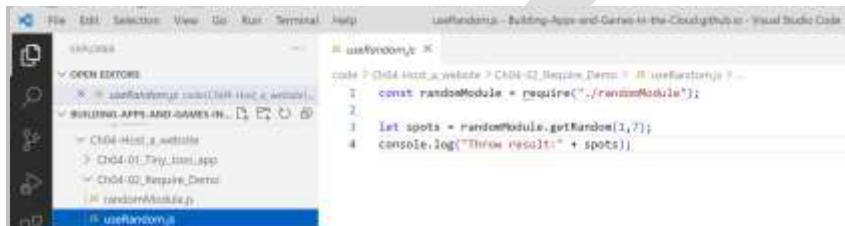
⁵³ Export the function with the name `getRandom`

⁵⁴ Import the function

⁵⁵ Use the function in our program.

Use debug to investigate the require statement

In Chapter 2 in the section **Code Analysis: Calling Functions** we used the debugger in the browser to discover how JavaScript functions are called. Now we are going to use the node.js debugger in Visual Studio Code to investigate how **require** is used to load modules into a program. Start Visual Studio Code and open the GitHub repository for the sample code for this book. Now find the **Ch04-02_Require_Demo** directory and open the file **useRandom.js**:



Ch04_inset03_01 opening the require demo

This program uses **require** to load the **randomModule.js** file. Then it calls **getRandom** from the module. We can use the Visual Studio Code debugger to step through the code in this program. We do this by starting the debugger. Click on the debug icon in the left-hand column. It looks like a run key button with a little bug sitting on it:



Ch04_inset03_02 starting the debugger

The first time you start the debugger it will ask you which debugger to use:



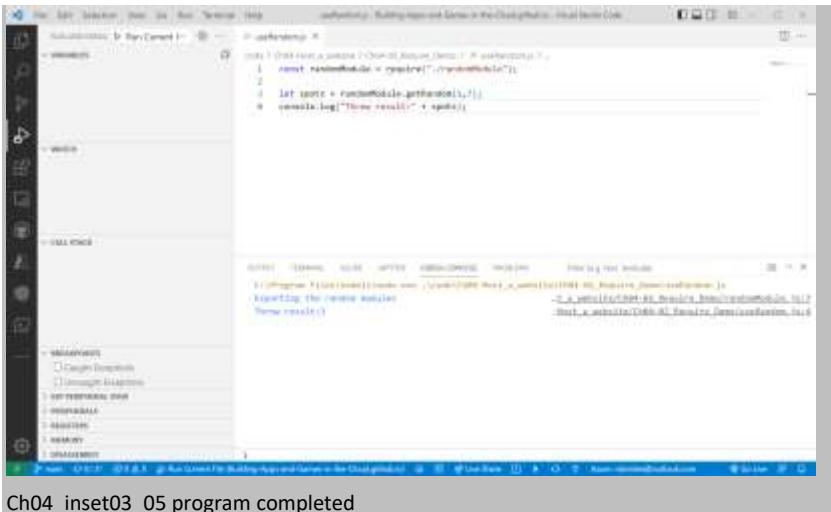
Ch04_inset03_03 selecting the node.js debugger

Select Node.js as suggested. The Run and Debug window is now displayed on the left of the Visual Studio Code window.



Ch04_inset03_04 starting the debugger

Now we can start the program running in the debugger. Press the **Run and Debug** button at the top of the Run and Debug menu to run the program.



Ch04_inset03_05 program completed

You can see above that the program has run and displayed a throw result of 3. You can click the links next to the console log displayed to visit the lines of JavaScript that produced the outputs. This output shows that the program ran correctly, but what we would really like to do is use the debugger to discover how the require process works. We can do this by setting a breakpoint in this program and then stepping through the code.

We've already used breakpoints in the debugger in the browser, they are set in the same way here. Click the left-hand side of the line number. In this debugger a breakpoint is indicated by a red dot in front of the line. Set a breakpoint at the first statement in the program as shown below:



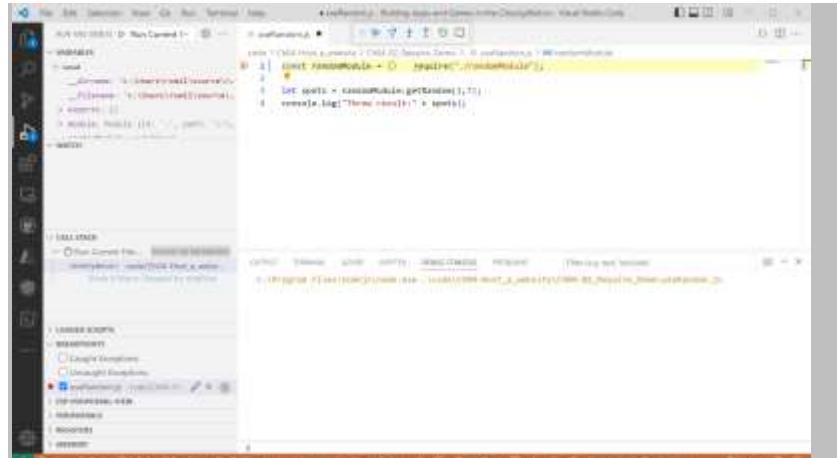
Ch04_inset03_06 setting a breakpoint

When we run the program it will stop at this statement and we can take a look at what it is doing. Now we need to restart the program. Click the green triangle next to the Run Current dropdown at the top of the debugger pane to do this.



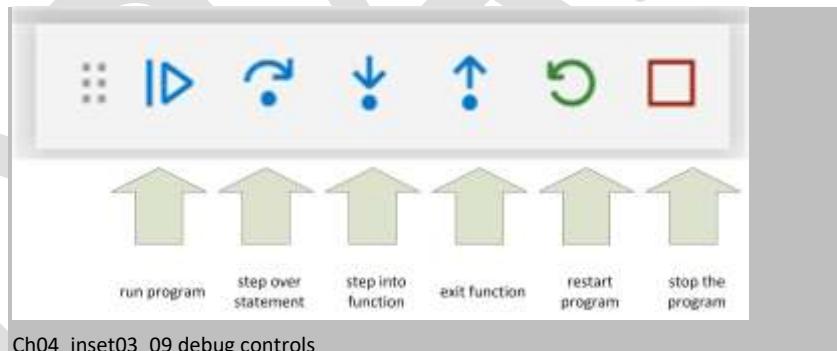
Ch04_inset03_07 rerun the debugger

The node environment will now run the JavaScript program until it hits the breakpoint.



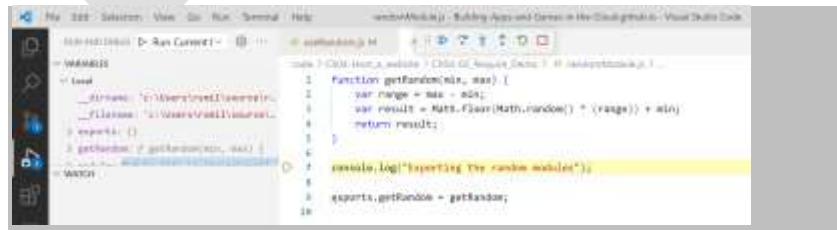
Ch04_inset03_08 hitting the breakpoint

Above you can see how the program stopped at the first statement in `useRandom`. At the top of the window you can see a set of debug controls that look rather like those we saw in the Developer Tools for the browser.



Ch04_inset03_09 debug controls

Press the “step into function” control (the downward pointing arrow) to run the next statement. This will perform the require statement.



Ch04_inset03_10 running a require

When a JavaScript program performs a `require` it executes the JavaScript in the module file that is being required. Above you can see that the module contains a statement that logs a message to the console prior to exporting the `getRandom` function. You can repeatedly press the “step into function” button to watch the program run through. When execution reaches the end of the `randomModule` source file it returns to `useRandom` and calls the `getRandom` function.

It makes sense to store modules in a different place from application code. The node environment is designed to allow this. We can create a directory called `node_modules` and node will search this directory to find required module files. The example **Ch04-03_Library_Folder** contains a `node_modules` directory which contains the module files.

```
const randomModule = require("randomModule");56  
  
let spots = randomModule.getRandom(1,7);57  
console.log("Throw result:" + spots);58
```

The code above shows how a module in the `node_modules` folder is accessed. We don't need to put the `./` prefix to the path to the module file if we store the module file in `node_modules` folder. You can open the programs in the debugger and step through them to see how they work.

Require and Import

The require mechanism works but it does have some disadvantages. A program can use `require` to load a module at any point in a program's execution. You might think that this is a good thing because it is very flexible. However, if you want to manage how external components are used in an application (and this is an important thing to do in large projects) you really should not allow programmers to load modules at any time. It is much more sensible to require all the modules to be loaded at the start a program.

Another issue is that the require mechanism runs *synchronously*. We have seen that when a program uses `require` to load a module it runs the entire contents of the JavaScript file containing the module code. This takes place at the time the require is performed, which will cause a program to pause while modules are loaded. If an application contains multiple requires each must be

⁵⁶ Load the module

⁵⁷ Use `getRandom` from the module

⁵⁸ Print the result to the console

completed before the next can be performed. It is also not possible to use `require` to only load particular elements from a module. The whole module must be scanned each time it is used.

To address these issues JavaScript was given `import` and `export` declarations. These provide the same functionality as `require` but in a slightly different way.

```
function getRandom(min, max) {  
    var range = max - min;  
    var result = Math.floor(Math.random() * (range)) + min;  
    return result;  
}  
  
export {getRandom} ;59
```

Above you can see the JavaScript code for a module that exports the `getRandom` function. Another module can import this by using the `import` declaration:

```
import { getRandom } from "./randomModule.mjs";  
  
let spots = getRandom(1,7);  
console.log("Throw result:" + spots);
```

The code above imports the `getRandom` function from a local module file called `randomModule.mjs`. It then calls the function to generate a `spots` value. There are some important things to remember when you are using `import`.

- The module file and any files that import modules must have the language extension “.mjs” rather than the usual “.js” which means JavaScript program.
- All the imports for a module must be performed at the start of the module.

You can find these example files in the source code folder **Ch04-04_Import_Demo**. If you are creating modules for your own projects I think you should use the `import` mechanism. It is important

⁵⁹ Export the `getRandom` function

⁶⁰ Import the `getRandom` function

to know about `require` because it is very likely that you'll see it used in older projects that you might work on. It is possible to create modules that can be used with either `require` or `import`. All the libraries that we are going to use in node to create our own web servers can be imported or required, we are going to use `import` in all the examples.

Using import in the browser

Up until now we have created our JavaScript applications by embedding code directly in a web page. However, this is not a good solution if you're building large projects. We might also like to use code from modules in JavaScript applications that run in the browser. Let's see how we can create a web page that uses a module.

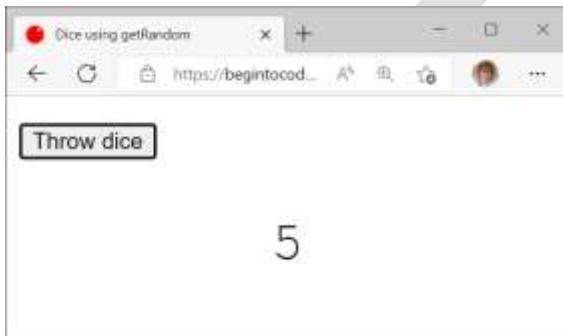


Figure 4.17 Ch-04_Fig_02 Browser dice

Figure 4.2 above shows what we are creating. When the Throw Dice button is clicked a new random value between 1 and 6 is displayed on the page. This page uses the same random number module as we have seen earlier. You can find the files behind the site in the examples in the directory **Ch04-05_Browser_Import**. You can view the web page here: https://begintocodecloud.com/code/Ch04-Host_a_website/Ch04-05_Browser_Import/index.html

```
<!DOCTYPE html>
<html>

<head>
  <title>Dice using getRandom</title>
  <link rel="shortcut icon" type="image/x-icon" href="favicon.ico">
  <link rel="stylesheet" href="styles.css">
</head>

<body>
  <p>
    <button id="diceButton">Throw dice</button>
```

```
</p>
<p id="dicePar" class="dice">*</p>
<script type="module">61
  import {doStartPage} from "./pageCode.mjs";62
  doStartPage();63
</script>
</body>
</html>
```

This the HTML file for the dice web page. It contains a definition for a “Throw dice” button. Our application needs to know when this button is clicked so that it can generate and display new dice throw value. Up until now the definition of a button in the HTML for the web page contained an `onclick` attribute that calls a handler function when the button is clicked:

```
<button onclick="selectFastClock()">Fast Clock</button>
```

This is a button definition from web page for the example **Ch02-04_Time Travel Clock** that we worked on in Chapter 2. It defines the button that is pressed when we want the clock to be five minutes fast. The `onclick` attribute contains the JavaScript code “`selectFastClock();`” that is performed when the button is clicked. It means that when the button is clicked the `selectFastClock` function runs. This works, but it means that the creator of the web page (which contains the button element) must agree with the creator of the JavaScript (which contains the event handler) on the name of the event hander function and how it is called.

A better way to do this is to make the developer in charge of connecting the event handler to the button. Then they can call the handler function whatever they like. All the developer needs to know is the id of the button element in the page. In the HTML for the dice page the button is given an id of “diceButton”.

```
<button id="diceButton">Throw dice</button>
```

⁶¹ Specify that the script is a module

⁶² Import the `doStartPage` function

⁶³ Call the `doStartPage` function

This is the HTML from the dice page. It defines a dice button which is clicked when the user wants a new dice value to be displayed. The element has the id “diceButton”. The binding of a function to the click event is performed in the `doStart` function which is held in a JavaScript module called `pageCode.mjs`. The HTML file imports the `doStartPage` function from this module and then calls the function to start the page running. This is the `pageCode.mjs` source file:

```
import {getRandom} from '/modules/randomModule.mjs';64

function doThrowDice() {65
    let outputElement = document.getElementById("dicePar");
    let spots = getRandom(1,7);
    outputElement.textContent = spots;
}

function doStartPage(){66
    let diceButton = document.getElementById("diceButton");67
    diceButton.addEventListener("click", doThrowDice);68
}

export{doStartPage}; 69
```

This file contains two functions. The first one is `doThrowDice`, which is called to display a new dice value. The second function is `doStartPage` which is called to start the page running. This function connects the `doThrowDice` function to the click event on the button. We have seen `addEventListener` before. We used it to add event listeners to the buttons in the Mine Finder game we created in chapter 3. The `doStartPage` function is exported from the module so that it can be imported and used in the web page.

CODE ANALYSIS

⁶⁴ Import the `getRandom` function

⁶⁵ Function that throws the dice

⁶⁶ Starts the page running

⁶⁷ Find the dice button

⁶⁸ Bind an event handler to it

⁶⁹ Export the `doStartPage` function

Modules in the browser

You might have some questions about this code.

Why must the JavaScript files have the language extension **.mjs**?

JavaScript handles module files differently from “ordinary” files. The `import` declaration only works in a module file. Module files also have `strict` mode enabled by default. We first saw the JavaScript `strict` mode in the Make Something Happen: Investigate let, var and const in chapter 3. It asks JavaScript to perform extra checks to make sure that your program is correct. This means that JavaScript needs to know when it is processing a module file. This difference is indicated by a different language extension. A language extension is added at the end of a filename, preceded by a period (.). The language extension `.js` means “JavaScript program”. The language extension `.mjs` means “JavaScript module”.

If you want to indicate that JavaScript code in an HTML file is a module you use the `type` attribute of the JavaScript element in the HTML file. For a standard JavaScript program the type is `text/javascript` but for a module the type is `module` so that the JavaScript engine in the browser knows it is module code and can use `import`.

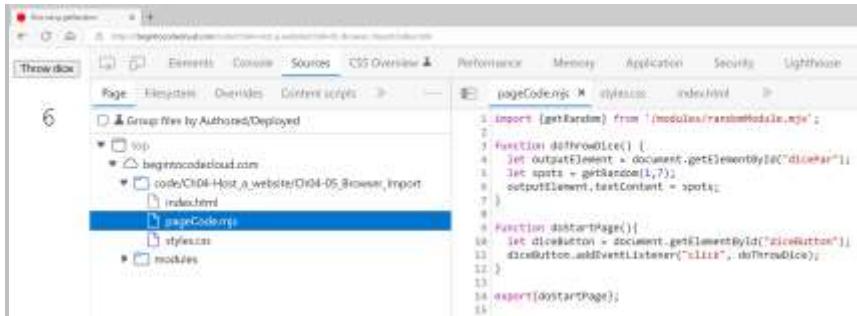
Where is the file **randomModule.mjs** stored?

```
import {getRandom} from './randomModule.mjs';
```

Above you can see a statement that imports `getRandom` into a program. The word `from` is followed by the path to the file that contains the JavaScript code to be imported. The path starts with the sequence `./` which tells JavaScript to look for the `randomModule.mjs` file in the same folder as the program source. This works, but it means that every application has its own copy of the module file.

```
import {getRandom} from '/modules/randomModule.mjs';
```

This is the statement that imports the `getRandom` module for the dice website. Now the path doesn't start with a period (.). Leaving off the leading period tells JavaScript to look in the top directory of the website rather than the directory containing the JavaScript program. At the top of my website I've created a directory called `modules` which stores all my module files. There is a copy of the `randomModule.mjs` file in that directory. This means that all the pages in my website can import from this module.



Ch04_inset04_01 Modules in the browser

Above you can see how this all fits together. The Sources view in the browser Developer Tools shows where all the files are located and you can browse each one.

Can a node application and a browser share the same module files?

Yes, as long as you are careful where you put the shared module files. If you leave the leading period of a file path given in a node application this means “look at the root of the storage device holding the node program”, which is slightly different from the root of a web site.

Can a module export more than one item?

Yes. You just add the items to the list of things that are exported. You can export variables as well as functions.

Can you declare variables in a module file?

Yes you can. Variables that are declared global to the module (i.e. declared outside any function) will be visible to code in that module file only. To understand how this works (and why we are doing it), suppose that we wanted to make a dice that displayed how many times it had been thrown. We would need a variable to keep track of the number of times the dice has been thrown and then update this and display it after every dice throw:

```
var throwCount = 0;

function doThrowDice() {
    let outputElement = document.getElementById("dicePar");
    let spots = getRandom(1,7);
    throwCount = throwCount + 1;
    outputElement.textContent = spots + " " + throwCount;
}
```

Above you can see how we could do this. This code is in the **pageCode.mjs** source file. It uses a modified version of the **doThrowDice**. The **throwCount** variable is used by the **doThrowDice** function which is called each time the “Throw Dice” button is pressed. The function increments the **throwCount** variable and then displays it after the number of spots. You can find this version of the dice in the example **Ch04-06_Throw_Counter**.

This is a neat way of “hiding” variables that you don’t want people to have access to. If the variable is declared inside a module it is only useable by code outside the module if it is explicitly imported.

The dark side of modules

In the next section we’re going to build a working web server using just a few lines of JavaScript and a lot of code imported from modules. However, before we do that, we should look at the “dark side” of modules and mention something you might like to consider when you use them. Let’s start with a look at a special new version of the `getRandom` function that is supposed to return a random number:

```
function getRandom(minimum, maximum) {  
    var range = maximum - minimum;  
    var result = Math.floor(Math.random() * (range)) + minimum;  
  
    let currentDate = new Date();70  
    if(currentDate.getMinutes()<10){71  
        if(result > minimum){72  
            result = result - 1;73  
        }  
    }  
    return result;  
}
```

We’ve used the `getRandom` function to generate values for the Mine Finder program and the dice. However, this version of `getRandom` has an extra special feature. For the first ten minutes of every hour the function subtracts 1 from the result. You might be wondering why you might write code like this. But it means that for ten minutes in every hour I can say “I’ll give you a million pounds if this dice rolls a six” and be sure that I won’t lose any money. Because in that time it is impossible to roll a six. If the above version of `getRandom` ended up in a program used by a casino I could use this knowledge to my advantage. You can find this tampered version of the program

⁷⁰ Get the date

⁷¹ Are we in minute 1?

⁷² Is the result bigger than the minimum?

⁷³ Subtract 1 from the result

in the example **Ch04-07_Tampered_Random**.

When you are using modules you need to be careful that the code inside them doesn't have any nasty extra features like the function above. It is possible that a module will contain faulty code, but it is also possible for a module to contain malicious code. There have even been reports of people copying GitHub repositories and making tampered versions of libraries for unwary developers to use in their applications. Make sure that you are using the "proper" version of a library by checking the activity level on the GitHub site.

Make a web server

In chapter 2, we installed the Live Server Extension in Visual Studio Code. Ever since then we have used this extension to view the web pages that we have created in Visual Studio. Live Server provides a tiny web server that runs on our machine. When the browser asks for a web page the Live Server program finds the file containing the page and sends it back to the browser. Now we are going to create a web server which is powered by JavaScript code running inside node.js. This can send files back to a browser, but it can also generate HTML directly from code.

You might think that hosting our own site would mean that we must put something in the cloud, but this is not the case. We can use node.js to run a web server on our local machine and then connect to it with our browser. Our server will listen on a network **port** for incoming requests. When a message comes in the server will generate an HTML formatted response and send it back. We will give the browser a **localhost** network address for the server address so that it connects to our local computer. The server will use the Hyper Text Transfer Protocol (**HTTP**) to interact with the browser. We will connect event handler functions that will respond when requests arrive.

Once the server code is complete it can be moved into the cloud so that the service we have created can be used by anyone around the world. There are cloud hosting services that can take our JavaScript code and run it for us. There is even an extension for Visual Studio code that can take our site and place it in the cloud for us.

Serving from software

```
import http from 'http';74
```

⁷⁴ Load the http library

```
function handlePageRequest(request, response){75
    response.statusCode = 200; 76
    response.setHeader('Content-Type', 'text/plain'); 77
    response.write('Hello from Simple Server'); 78
    response.end(); 79
}

var server = http.createServer(handlePageRequest);80

console.log("Server running");

server.listen(8080);81
```

The program above hosts a web page that returns a web page containing the text “Hello from Simple Server” when accessed from the web. The web page is hosted on the local machine at port 8080. Let’s use the debugger to step through the code and watch the server build a web response and return it.

Make Something Happen

Use debug to investigate the server

We can use the Visual Studio Code debugger to watch our tiny web server in action. Start Visual Studio Code and open the GitHub repository for the sample code for this book. Now use the Explorer to find the **Ch04-08_Simple_Web_Server** directory and open the file **server.mjs**.

This is the most complicated exercise that we have performed so far. In it you will use two programs, Visual Studio Code and your browser. Visual Studio Code will use node to run a

⁷⁵ Function to deal with a request

⁷⁶ Set the status code for the response

⁷⁷ Set the content type to text

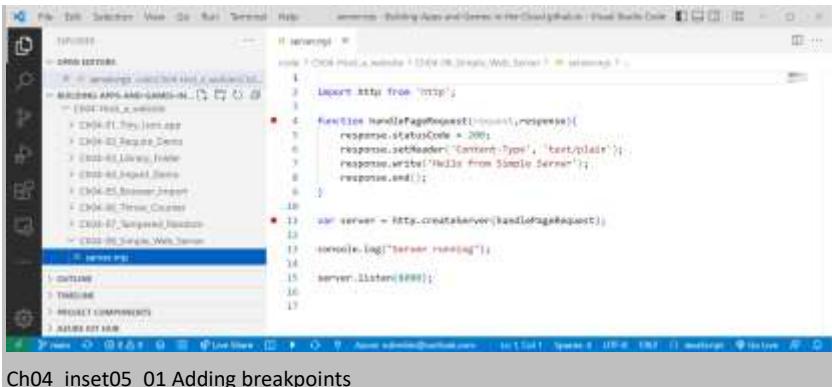
⁷⁸ Add the content

⁷⁹ Send the response

⁸⁰ Create a server

⁸¹ Start the server listening on port 8080

web server written in JavaScript and the browser will visit the server. Make sure that you follow all the steps in the sequence given.

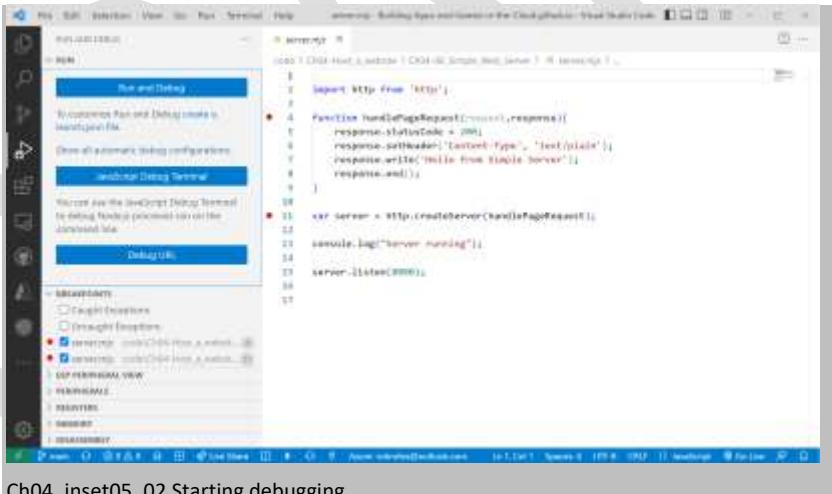


The screenshot shows the Visual Studio Code interface with the file `server.mjs` open. The 'Breakpoints' sidebar on the left is expanded, showing two red circular markers indicating breakpoints have been set at line 4 and line 11. The code editor shows the following JavaScript code:

```
1 import http from 'http';
2
3 function handlePageRequest(request, response) {
4     response.statusCode = 200;
5     response.setHeader('Content-Type', 'text/plain');
6     response.write('Hello from Simple Server');
7     response.end();
8 }
9
10 var server = http.createServer(handlePageRequest);
11
12 console.log("Server running");
13
14 server.listen(8080);
```

Ch04_inset05_01 Adding breakpoints

We used breakpoints and the debugger in the section **Make Something Happen : Use debug to investigate the require statement**. Take a look in there to refresh your knowledge of breakpoints and the debug controls before performing this exercise. You can also use the editor to add breakpoints to a program. Click just before the line numbers on lines 4 and 11 to add breakpoints as shown above. When the program runs it will stop at these two statements. Now press the **Run and Debug** button to open the debug window.



The screenshot shows the Visual Studio Code interface with the file `server.mjs` open. The 'Run and Debug' sidebar on the left is expanded, showing the 'Start Debugging' button highlighted in blue. The code editor shows the same JavaScript code as the previous screenshot. The 'Breakpoints' sidebar on the far left shows two red markers at line 4 and line 11.

Ch04_inset05_02 Starting debugging

Now click the large blue **Run and Debug** button to run program in the file **server.mjs**. This will run the program until it hits a breakpoint:

```
File Edit Select View Insert Run Terminal Help
File Edit Select View Insert Run Terminal Help
src\Ch04_Host_a_Website\Ch04_Host_a_Website\server.js
1 import http from 'http';
2
3 function handlePageRequest(request, response) {
4   response.statusCode = 200;
5   response.setHeader('Content-Type', 'text/plain');
6   response.write('Hello from Simple Server');
7   response.end();
8 }
9
10 var server = http.createServer(handlePageRequest);
11
12 console.log("Server running");
13
14 server.listen(8080);
15
16
17
```

Ch04_inset05_03 Start server breakpoint

Above you can see that the program has hit the breakpoint at line 11. This is the statement that starts the server. You might be wondering why the breakpoint at line 5 wasn't hit. This is because that statement is inside the `handlePageRequest` function, which has not been called yet. The `handlePageRequest` function is passed into the `createServer` function so that the server knows which function to call when a page request is received. Click the step into button in the debug controls (or press F11) to execute that statement the program. The program will now move onto a statement that logs "Server running" on the console.

```
File Edit Select View Insert Run Terminal Help
File Edit Select View Insert Run Terminal Help
src\Ch04_Host_a_Website\Ch04_Host_a_Website\server.js
11 var server = http.createServer(handlePageRequest);
12
13 console.log("Server running");
14
15 server.listen(8080);
16
17
```

Ch04_inset05_04 Console message statement

Click the step into button again to perform this statement. You should see the message appear on the console and execution moves on to the next statement, which starts the server listening by calling the `listen` function.

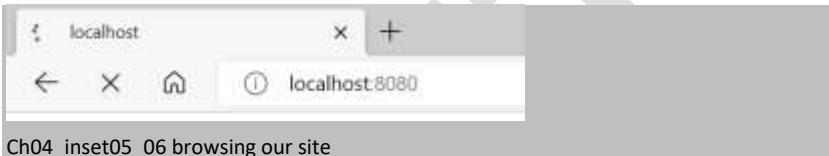
```
File Edit Select View Insert Run Terminal Help
File Edit Select View Insert Run Terminal Help
src\Ch04_Host_a_Website\Ch04_Host_a_Website\server.js
11 var server = http.createServer(handlePageRequest);
12
13 console.log("Server running");
14
15 server.listen(8080);
16
17
```

OUTPUT TERMINAL DEBUG CONSOLE Filter (e.g. test, include)
C:\Program Files\nodejs\node.exe ..\code\Ch04_Host_a_Website\Ch04_Host_a_Website\server.js
Server running

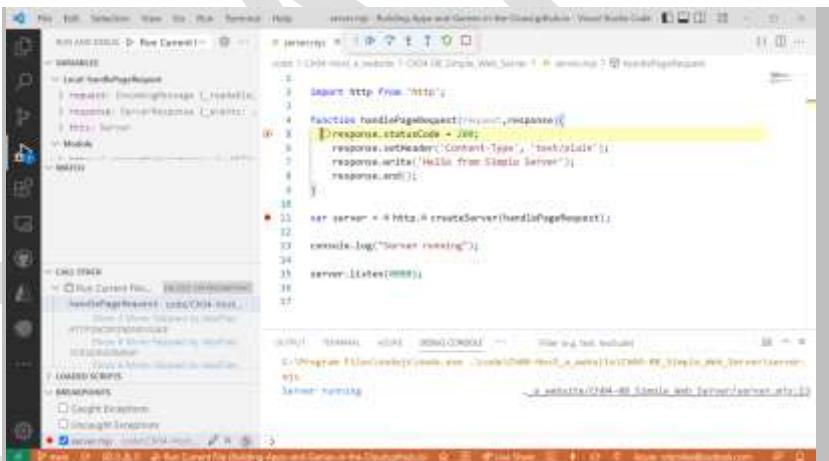
Ch04_inset05_05 Server listen statement

Above you can see the output from the program and the program positioned at line 15 ready to start the server listening for web requests. The call to `listen` is provided with the number of the `port` to listen to. For our demonstration server we will be using port 8080. Press the step button to perform the `listen` function. The listen function is now running and waiting for a web request on port 8080.

We can use our browser to make a web request of the site that our server is hosting. The address we are going to use is **localhost:8080** The first part of the address is the address of the machine (in our case it is our local host) and the second part of the address is the port number (in our case it is 8080 because that is where our server program is listening. Open the Edge browser, type the address into the address bar and press enter to open the site.



You will see that the browser will pause, waiting for the site to arrive from the server. Now, go back to the Visual Studio

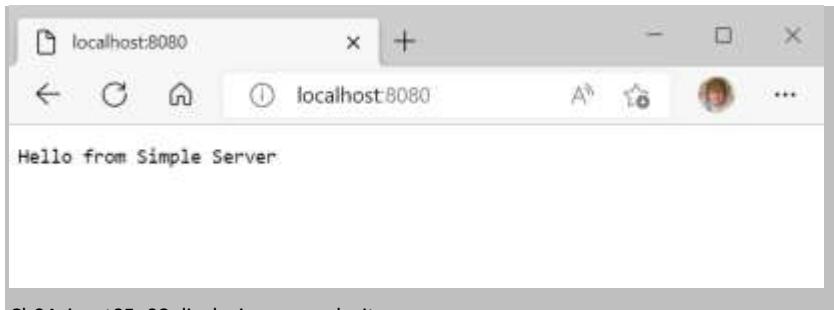


Ch04_inset05_07 hitting the breakpoint in the server

The program has hit the breakpoint in the `handlePageRequest` function. This function is called by the server when a page is requested. The job of the function is to assemble a response and then send it back to the server. We can look at what the function does a little later. For now, it is important that we run the function before the browser times out the web request.

Click the continue button in the debug controls (it's the right pointing blue triangle) to continue the `handlePageRequest` function. You might be expecting the web page to appear in the browser at this point, but it doesn't. Instead, you will find that the breakpoint in the `handlePageRequest` is hit a second time. Click the continue button again. Now you can go

back to the browser and see what has happened:



Ch04_inset05_08 displaying our web site

Above you can see the page produced by our server. This works because the browser it has been told that the content returned by the server is plain text, so it just displays it. We get to how this works in the next section.

Now stop the server (press the red square in the debug controls), edit the text in the call of `response.write` on line 7, run the program again see that the text that is served has changed. When you have finished with the server you can stop it again.

This is a big moment. You now know how both ends of the world wide web work. You've seen how browsers download web pages and display them, and now you know how a program can serve out a web page.

CODE ANALYSIS

Running a server

You might have some questions about what we have just done

What would happen if the server never sent back a response?

The browser sends a request to a web site and then waits for the response. If the response takes too long to arrive the request will time out and the browser tells you that the page is inaccessible.

How does the server build the response to the browser?

```
function handlePageRequest(request,response){  
    response.statusCode = 200;  
    response.setHeader('Content-Type', 'text/plain');  
    response.write('Hello from Simple Server');  
    response.end();  
}
```

When the `handlePageRequest` function is called it is given two parameters. The first parameter, `request`, is a reference to an object that describes the request from the browser. The second parameter, `response`, is a reference that refers to an object that describes the response to be sent to the browser. At the moment we are not using the `request` parameter at all.

The `handlePageRequest` function builds the same response to any request. The first thing the function does is set the `statusCode` property of the `response` to 200. This value is sent back to the browser at the beginning of the response. The value 200 means “all is well, here is the page”. We could use other values to signal error conditions. The value 404 means “page not found”.

The next thing the function does is use the `setHeader` method in the response to set a value in the header that is to be sent to the browser. It sets the value “Content-Type” to the string “text/plain”. The browser uses the value of Content-Type to decide what to do with the incoming data. If the content type was “text/HTML” the browser would build a document object and display that. However, our server just serves out plain text.

The third statement in the `handlePageRequest` function uses the `write` method in the response to write the actual content of the page. In this case it is just a simple message, but this content could be much longer.

The final statement calls the `end` function on the response. This is the point at which the response is assembled and sent back to the browser.

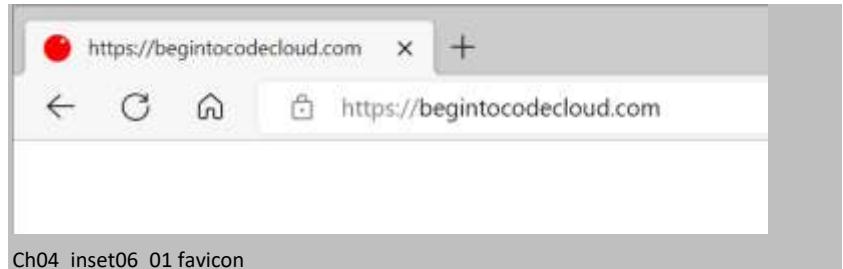
How does the server know which page is being requested by the browser?

The `request` parameter (which we have not used yet) contains a property called `url` which contains the **URL** (uniform resource locator) for the page that has been requested. If the browser is requesting the index page the `url` property is the string “/”. We will use the path in our next server, which will serve out files.

Why did the browser make two requests of the server?

In the **Make Something Happen: Use debug to investigate the server** exercise above we set a breakpoint in the `handlePageRequest` function. The breakpoint is hit when a server requests a page. The breakpoint was hit twice when we tried to load the page into the browser, which means that the browser has asked the server for two responses. Why?

This has to do with the way the web works. Many web pages, including the ones for the sample code for the site, have “favicons” on them. A favicon is the little image that is displayed on the top left-hand corner of the page.



Ch04_inset06_01 favicon

Above you can see the favicon for the **begintocodecloud.com** site. It is a shiny red ball which I think it is quite artistic. When a browser loads a web site it makes two requests. One request is for the favicon image file. The other is for the actual content of the site. Our server sends back the same response to both requests; the text “Hello from Simple Server”. The browser can’t make a text message into a favicon and so it ignores it. If we want our site to have a working favicon we have to create a bitmap file of the correct type and then serve it out when the file is requested.

```
import http from 'http';
import fs from 'fs';

function handlePageRequest(request, response) {
    let url = request.url;

    console.log ("Page request for:" + url);

    if (url == "/favicon.ico") {
        console.log(" Responding with a favicon");
        response.statusCode = 200;
        response.setHeader('Content-Type', 'image/x-icon');
        fs.createReadStream('./favicon.ico').pipe(response);
    }
    else {
        console.log(" Responding with a message");
        response.statusCode = 200;
        response.setHeader('Content-Type', 'text/plain');
        response.write('Hello from Simple Server');
        response.end();
    }
}
```

This version of `handlePageRequest` checks the url of an incoming request. If the request is for a favicon it will open the icon file and send it back to the server. Otherwise, it sends the message “Hello from Simple Server”. You can find this version of the simple server in the folder **Ch04-09_Simple_Web_Server_with_favicon** in the sample code for this chapter. If you use this server you should see that the browser displays a shiny red favicon for the page. We will take a close look at how pages are sent back to server in the next section.

Serving out files

The server we have just created always delivers the same text back to the browser – the string “Hello from Simple Server”. We can turn it into a more useful server by allowing the browser to specify the file that the server is to return. When we ask a browser to show us a particular page the address of the page is expressed as a “universal resource locator” or **url**. This tells the browser where to go and look for a page.



Figure 4.18 Ch-04_Fig_03 url structure

An overview of a url is shown in Figure 4.3 above. The protocol and the host elements tell the browser the address of the computer and how to talk to it. The path specifies the file on the server that is to be read. The port value (if present) specifies the network port on the computer to connect to. If the port elements are missing the browser will try to connect to port 80.

The web server uses the value of the path to find the file that has been requested. The path in Figure 4.3 is for the **index.html** file for the **begintocodecloud.com** website. If you leave the path off the address the server will send the index file automatically. Our server can use the **url** property of a web request to determine what is to be sent back to the browser.

```
import http from 'http';
import fs from 'fs';
import path from 'path';

function handlePageRequest(request, response) {
  let url = request.url;82
  console.log("Page request for:" + url);
```

⁸² Get the url from the response

```

let filePath = '.' + url;83

if (fs.existsSync(filePath)) {84
    console.log("    found file OK")
    response.statusCode = 200;
    let extension = path.extname(url);85
    switch (extension) {86
        case '.html':
            response.setHeader('Content-Type', 'text/HTML');
            break;
        case '.css':
            response.setHeader('Content-Type', 'text/css');
            break;
        case '.ico':
            response.setHeader('Content-Type', 'image/x-icon');
            break;
        case '.mjs':
            response.setHeader('Content-Type', 'text/javascript');
            break;
    }
    let readStream = fs.createReadStream(filePath);87
    readStream.pipe(response);88
}
else {89
    console.log("    file not found")
    response.statusCode = 404;
    response.setHeader('Content-Type', 'text/plain');
    response.write("Cant find file at: " + filePath);
    response.end();
}
}

```

⁸³ Add make the url into a local path

⁸⁴ Check if the file exists

⁸⁵ Get the file extension of the url

⁸⁶ Select the content type

⁸⁷ Create a read stream for the file

⁸⁸ Pipe the stream into the response

⁸⁹ If the file doesn't exist send file not found

```
let server = http.createServer(handlePageRequest);

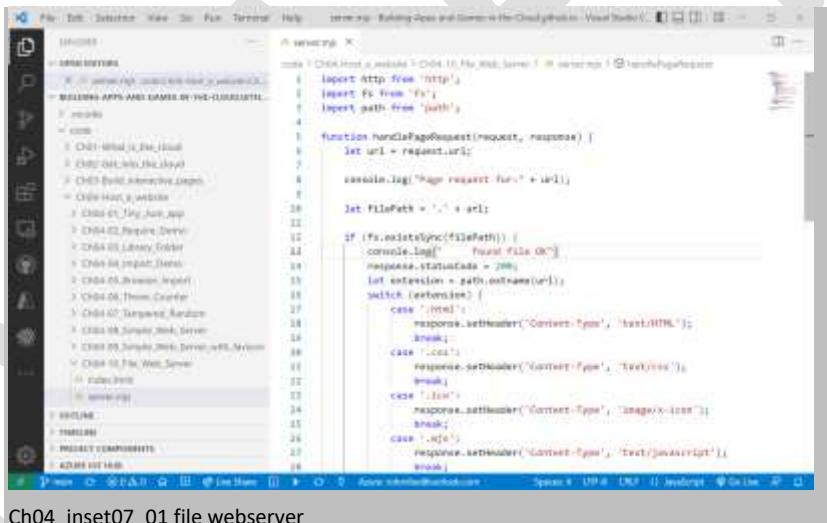
console.log("Server running");

server.listen(8080);
```

Make Something Happen

Using the file server

You can use the above program to view the entire web site for this book. Start Visual Studio Code and open the GitHub repository for the sample code for this book. Now use the Explorer to find the **Ch04-10_File_Web_Server** directory and open the file **server.mjs..**



The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left lists several projects and files, including 'Ch04-10_File_Web_Server'. The main editor area displays the 'server.mjs' file with the following code:

```
const http = require('http');
import fs from 'fs';
import path from 'path';

function handlePageRequest(request, response) {
    let url = request.url;
    console.log(`Usage request: ${url}`);

    let filePath = './' + url;

    if (!fs.existsSync(filePath)) {
        console.log(`  Found File OK`);
        response.statusCode = 200;
        let extension = path.extname(url);
        switch (extension) {
            case '.html':
                response.setHeader('Content-Type', 'text/html');
                break;
            case '.css':
                response.setHeader('Content-Type', 'text/css');
                break;
            case '.js':
                response.setHeader('Content-Type', 'application/javascript');
                break;
            case '.json':
                response.setHeader('Content-Type', 'text/json');
                break;
        }
    } else {
        response.statusCode = 404;
        response.end('File Not Found');
    }
}

http.createServer(handlePageRequest).listen(8080, () => {
    console.log('Server running');
});
```

Ch04_inset07_01 file webserver

Now select the debugger and start the program running.

The screenshot shows a Node.js development environment. The code editor displays a file named `index.js` with the following content:

```
1 import http from 'http';
2 import fs from 'fs';
3 import path from 'path';
4
5 function handleFileRequest(request, response) {
6   let url = request.url;
7
8   console.log("Page request for: " + url);
9 }
```

The terminal window shows the server is running on port 8080, and it lists two files being served: `index.html` and `styles.css`.

Ch04_inset07_02 file server running

Above you can see a debugging session running. I used the browser to open the file <http://localhost:8080/index.html> which is served by this program. The server outputs the name of each file as the browser requests it. The server has sent two files, `index.html` and `styles.css`. If you move to other pages on the site you will see these displayed as well. I find it rather impressive that such a tiny program can act as quite a capable web server.

CODE ANALYSIS

Simple file server

You might have some questions about the server.

How does the server send a file back to the browser?

A node installation includes a few built-in modules. One of these is the `http` module that we are using to host our website. Another library is the `filesystem` module, `fs`, which node programs use to interact with the local file store.

```
let readStream = fs.createReadStream(filePath);
readStream.pipe(response);
```

These statements send a file back to the browser. The first statement uses the `createReadStream` function from the `fs` module to create a read stream connected to the file. The second statement uses the `pipe` function on the read stream to send the file to the web page response. You might need to think of this a bit. A stream is a bunch of data that you might want to send somewhere, like you might have a tank of water you want to use to fill a washbasin. In real life you would use a physical pipe to link the two. In our server we have a

response object that wants to be given some data to send (the wash basin), and a file object that wants to supply that data (the water tank). We use the `pipe` method on the stream object to tell it to send the file to the response object. We don't need to worry about precisely how this works, and the response will automatically end when the file has been received from the stream. If this seems hard to understand, go back to considering what we want to do. We have a thing that has got some data, and a thing that wants to receive some data. The pipe method will perform that transfer using a stream.

What happens if the browser asks for a page that doesn't exist?

The server uses a function from the `fs` module called `existsSync` to check if a requested file exists. This function is part of the file system library. If the file is not found the server responds with a 404 "resource not found" error code.

How does the server know what type of file to send back to the server?

When a server sends a response to a request it must always include Content-Type information so the browser knows what to do with the incoming data. The server works out the type of data to send back by looking at the file extension of the incoming url.

```
let extension = path.extname(url);
```

The statement above uses the `extname` method from the `path` module to get the extension from the url. The extension is the character sequence which is preceded by a period (.) on the end of a file path or url. As an example, the path "index.html" has the extension ".html". The extension specifies the type of data that the file contains. So "index.html" should contain html text that describes a web page. The server uses the extension string to decide what Content-Type to add to the response:

```
switch (extension) {
  case '.html':
    response.setHeader('Content-Type', 'text/HTML');
    break;
}
```

The case construction selects the response type that matches the extension string.

What happens if the browser asks for a file type that doesn't exist?

The server we have just created can handle html, css, mjs and ico file types. If it is asked for a file of a different type (perhaps a JPEG formatted image with a language extension of .jpg) it will not return a Content-Type value to the browser, resulting in the response being ignored. We can expand the range of content types that our server recognizes by creating a lookup table for the content types:

```
let fileTypeDecode = {
  html: "text/HTML",
  css: "text/css",
  ico: "image/x-icon",
```

```
mjs: "text/javascript",
js: "text/javascript",
jpg: "image/jpeg",
jpeg: "image/jpeg",
png: "image/png",
tiff: "image/tiff"
}
```

The code above creates a variable called `fileTypeDecode` which can be used to map language extensions onto content type strings. It allows our browser to handle a range of different image file types.

```
let extension = path.extname(url);
extension = extension.slice(1);
extension = extension.toLowerCase();
let contentType = fileTypeDecode[extension];
```

These four statements get the content type from the url. The first statement creates a variable called `extension` from the url specifying the file that has been requested by the browser. This would create ".html" from a request for "index.html". The second statement removes the leading "." from the extension. It would convert ".html" to "html". The third statement converts the extension to lower case. It would convert "HTML" to "html". The fourth statement gets the file type that matches the extension from the `fileTypeDecode` object. This works because JavaScript allows us to specify the property of an object by using a string. In other words, the statements:

```
let x = fileTypeDecode.html;
```

and

```
let x = fileTypeDecode["html"];
```

- would both set the value of x to be "text/HTML". If we try to use an extension which is not present in the `fileTypeDecode` object the value undefined is returned.

```
let contentType = fileTypeDecode[extension];
if (contentType == undefined) {
    console.log("      invalid content type")
    response.statusCode = 415;
    response.setHeader('Content-Type', 'text/plain');
    response.write("Unsupported media type: " + filePath);
    response.end();
}
else {
    response.setHeader('Content-Type', contentType);
    let readStream = fs.createReadStream(filePath);
    readStream.pipe(response);
}
```

If the content type is not recognized the server will respond with error 415 to indicate this. Otherwise the file is piped out to the response as before. You can find this version of the server in the folder **Ch04-11_Picture_File_Web_Server** in the sample code for this chapter. You can use it to view the picture in the home page for that example. If you wish, you might like to expand the server so that it can deliver audio and video files. You just have to identify the content types for each file type and then add them to the `FileTypeDecode` object.

Active sites

We now know how a JavaScript program running under node.js can serve out both active content (messages from a running program) and file content (the contents of files on the server). Most web applications use a mix of these. Fixed elements of pages will use files and then the program generated content will be inserted as required. It would be useful to have a framework where you could create “templates” which contain the fixed parts of a site and then allow you to inject the program generated content when required. The good news is that you will be learning how to do this in the next chapter. The better news is that you are now well on the way to understanding how the web works in both browser and server.

What you have learned

This has been another busy chapter. We've covered a lot of ground. Here is a recap plus some points to ponder.

- Node.js is a framework that allows JavaScript programs to run outside of the browser. It is a free download for all machines. It doesn't provide a document object model to communicate with the user. Instead, it is controlled via a terminal interface. It provides a console on which you can enter commands to run JavaScript statements. It can also be given a JavaScript program to load and execute.
- The node framework provides support for modules. A file of JavaScript code can contain statements that export data or code elements that can then be introduced into other programs by using the require statement.
- A module file can contain elements that are not exported which can be used internally by that module.
- When elements are fetched by a call to require the node framework will execute all the code in the module source file before exporting the elements. This execution takes place synchronously, i.e. the program performing the require will be paused until the require completes.
- A module source file can contain elements that are not exported. These

are local to the module and are not visible outside it.

- Node applications can be debugged in Visual Studio Code in a manner similar to JavaScript code running in the browser. You can add breakpoints to the code and view the contents of variables.
- The JavaScript language offers an alternative to the require mechanism using the `import` keyword. Modules containing import statements must have the file extension “.mjs” rather than the “.js”.
- It is not possible to use require in JavaScript code that is held in a web page and executed by the browser. However, JavaScript code running in a browser can use the `import` statement. JavaScript code in an HTML file that contains import statements must have the type “module”. It is not possible for JavaScript code embedded in HTML element attributes to access elements in a module. Instead code in a module must obtain a reference to a named element in the HTML file and then act on that directly.
- Whenever you use code that you didn’t write (for example importing a module you have downloaded from the internet) you should make sure that the code doesn’t contain any unwanted behaviors.
- A node.js installation contains several built-in modules. One of these is the `http` module that can be used to create JavaScript program that can act a web server.
- The `http` module contains a `createServer` function that is called to create a web server. The `createServer` function is supplied with a reference to a function that will service incoming page requests from the browser. This function is supplied with two parameters which refer to a request object and a response object. The response object must be populated with page information by the function servicing incoming page requests. The contents of the response object are sent back to the browser making the request.
- A response to a web request contains a `statusCode` property that gives the status of the response. A `statusCode` of 200 means that the page was found correctly.
- A response to a web request contains a `Content-Type` property that the browser will use to decide what to do with the page when it arrives. A type of “text/plain” specifies that the file contains plain text.
- You can add plain text to a web request response by using the `write` method exposed by the request.
- A function servicing incoming requests to an http server also receives a

request parameter describing the request made by the browser. The request parameter contains a [url](#) property that gives the path to the file on the server which is being requested. The server can map this url onto local filestore to locate the file to be sent back to the server.

- The node framework provides modules called [fs](#) and [path](#) that are used to interact with the filesystem on a machine. The fs module contains can make stream objects that are connected to local files. A stream contains a pipe method that can be used to direct the content of a stream into another object. The response object sent to the server in a web request can receive file streams and send them back to the browser.
- When a browser accesses a web site it will also request a favicon.ico file which contains a bitmap that is displayed by the browser.
- A server must ensure that the Content Type element of a reply to a request reflects the content of the file.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions".

What does node.js do?

The node.js framework lets you run JavaScript programs on a computer without using a browser.

What is the difference between synchronous and asynchronous operation?

Synchronous operations are performed "while you wait". In other words, if a program calls a function which performs a task synchronously the program will be paused until the task is complete. Asynchronous operations are performed 'in the background'. A call to a function will start an operation. The completion of the operation will be signaled by a call to another function. Asynchronous operations are harder to organize but they make an application more responsive as it is not paused waiting for things to happen.

When would you use a module?

You should use a module if are writing code that you want to use in several different applications. You can also use a module to share work around. Once you have decided what each module needs to do it can be developed separately. Another reason to use a module is that it gives your code more privacy. Code and variables in a module that are not exported are not visible outside the module. Modules are also useful when testing. For example, we could test a program that uses our random number generator module by creating a random number generator module to produces a fixed sequence of values. It would be tiresome to have to wait for the random number generator to produce a dice throw of 6 to test a game program. Much better to have a "testing dice" that produces the values that you need.

Why do we have require and import?

It turns out that programming languages are continually evolving as people think of new things that they want them to do. It is also the case that the first attempt to solve a problem might not be the best one. Require was developed specifically for use in node.js applications. Import was developed as a language element that built on what Require does.

Can the same module be both required and imported?

Yes it can. We have been importing modules into our node.js applications. We can also use require in node.js applications to bring in the same modules.

Where would you run a web server program?

A web server program accepts page requests from browsers and responds with content for the browser to display. You can run a web server on any computer. In this chapter we have written programs that act as web servers. Web servers for public use are run on machines that have permanent network connections or as processes in the cloud.

Can two applications on one computer share the same port number?

A port is a numbered connection to a program running on a machine. We have used 8080 as the port number behind which our server is running. Once a program has claimed a port number on a machine it is not possible for any other program running on the machine to use that port for connections.

What is the difference between a port and a path?

A program running on a computer can open a network port that can be used by other programs to connect to. Ports are specified by numbers. Port number 80 is traditionally used by web servers. A path is a string of text that specifies a how to traverse a storage system to get to a particular file or location. For example, the path “**code/ Ch04-Host_a_website/ Ch04-10_File_Web_Server/index.html**” tells a program to find the code directory, then look in that for the **Ch04-Host_a_website** and so on down to the file **index.html**.

What happens if a server gets the Content-Type wrong?

A browser adds Content-Type to each response that it sends back to the browser. The browser can then work out what to do with that content. If the server makes a mistake – perhaps sending back a jpeg image with the content type “text/plain” the browser will render the content in the wrong way. If an image is marked as text the browser will show a collection of random looking characters rather than a picture. Remember that a computer has no real understanding of data. We must tell it what is in the file so that it can do the right thing with it.

Is hosting a server on your machine a dangerous thing to do?

It might be. The server that we have made can only serve out the contents of the filetypes that we have specified. In other words, it will serve out a jpeg image but not a spreadsheet or a word document. This means that if someone managed browse a path to any of the other directories on my hard disk they would not be able to look at password or system files. However, they could probably learn a lot about me from the other types of file. You should never host a public (i.e. visible to the outside world) facing web site on your own computer. Instead you should move just the files you want to share onto a separate machine or cloud service and host them from that.

Chapter 5: Build a shared application

What you will learn

In the last chapter we created a JavaScript application running in node.js that could act as a web server. We used the server application in two ways, to serve out the contents of files and to run JavaScript code in response to web requests. We found that we could access our site using a standard browser. In this chapter we'll discover how we can create a server that can host components of an application. Part of our application will run inside the browser, and the rest will run on the server. We'll create services on the server and access them from code running in the browser so that our application is spread across the two platforms. We'll also find out how JavaScript Object Notation (JSON) can be used to transfer the contents of JavaScript variables between server and browser. But first we are going to revisit the game we made in chapter 3 and add

some compelling gameplay elements as we learn more about JavaScript development and debugging.

And don't forget that the Glossary is always out there...

Upgrade Cheese Finder

In chapter 3 we made a simple game called Cheese Finder. Players take it in turns to click on squares in a grid to try and find the squares containing cheese. When a square is clicked it changes a number that gives the from that square to the cheese. Figure 5.1 below shows how the game is played. People seem to quite like it but they would like a few improvements.



Figure 5.19 Ch-05_Fig_01 Cheese Finder Game

Adding some color

Cheese finder works well, but players don't seem to like looking at numbers in the squares. Someone suggests that it might be a good idea to use colors rather than numbers to indicate the distance that a square is from the cheese. This turns out to be quite simple to do. First, we need to make some styles that contain colors for the squares:

```
.cheese,  
.dist1,
```

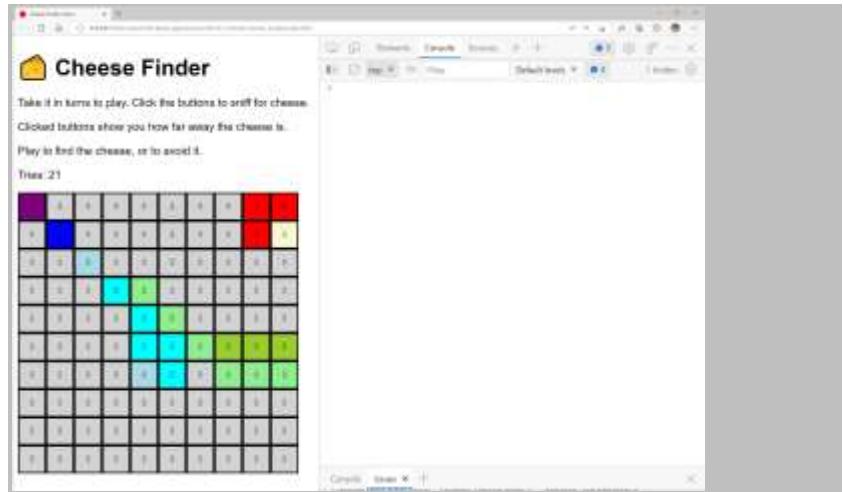
```
.dist2,  
.distFar {  
    font-family: 'Courier New', Courier, monospace;  
    text-align: center;  
    min-width: 3em;  
    min-height: 3em;  
}  
.cheese {  
    background: lightgoldenrodyellow;  
}  
.dist1 {  
    background: red;  
}  
.dist2 {  
    background: orange;  
}  
  
....  
  
.distFar {  
    background: darkgray;  
}
```

I've removed some of the styles to make the above listing shorter. There are actually 10 distance styles called `dist1` to `dist10` in the stylesheet used for the game. All the styles are based on a starting style and then the background color for each is set individually. The next thing we need is a way we can convert distance values into style names. We can use an array for this.

Make Something Happen

Arrays as lookup tables

There are two ways that a JavaScript program can use arrays. One way is to store things, and the other is as a lookup table. The `colourStyles` array is a lookup table. Let's investigate how this works. Start your browser and open the `index.html` file in the `Ch05-01_Colored_Cheese_Finder` example folder for this chapter. This implements a colored version of the Cheese Finder game. Play the game a few times and open the Developer Tools window in your browser. Then open the Console tab in the developer tools:



Ch05_inset01_01 Play colored Cheese Finder

We can now investigate how the array works. In the console we can view the contents of an item just by entering its name. Type:

```
colorStyles
```

and press enter.

```
> colorStyles
< [Object]
```

Ch05_inset01_02 colourStyles array contents

The console shows you the contents of the `colorStyles` array which is used to convert distance values into the names of styles. Items in an array are called elements. Let's take a look at the elements in the array, starting by taking a look at the element at the beginning of the array. We specify the element that we want to look at by giving an index value which tells JavaScript how far down the array to go to get the element that we want. The index is given enclosed in brackets. Type:

```
colorStyles[0]
```

and press enter.

```
> colorStyles[0]
< "cheese"
> |
```

Ch05_inset01_03 colourStyles array element

Arrays in JavaScript are indexed from 0, so cheese is the first element in the array. Let's see what happens when we try to step outside the array. Type:

```
colorStyles[1000]
```

and press enter.

```
> colorStyles[1000]
< undefined
```

Ch05_inset01_04 outside the bounds of an array

There is no element with an index of 1000. Some programming languages would get upset at this point, but JavaScript just returns a value of "undefined". We can also assign values to the elements in an array. In some languages an array is declared as having elements of a particular type. Let's see what happens in JavaScript. Type:

```
colorStyles[0] = 99
```

and press enter. This statement will attempt to put the number 99 into the element at the start of the array.

```
> colorStyles[0]=99
< 99
```

Ch05_inset01_05 Changing an array element

The statement seems to have worked, and the element at the start of the array (with the index 0) is now the number 99. Type:

```
colorStyles
```

and press enter to view the contents of the array.

```
> colorStyles
< [12] [99, 'dist1', 'dist2', 'dist3', 'dist4', 'dist5', 'dist6', 'dist7',
  'dist8', 'dist9', 'dist10', 'distFar']
```

Ch05_inset01_06 Modified colorStyles

Changing the array like this has broken the game program. The cheese will not be detected or displayed correctly. We can fix this by just reloading the page, but not before we've tried something really interesting. Let's try putting a value into an array element that we know is not there. Type:

```
colorStyles[100]='hello world'
```

and press enter.

```
> colorStyles[100]='hello world'
< 'hello world'
```

Ch05_inset01_07 Assigning to a non-existent element

This seems to work. No error has appeared. Type:

```
colorStyles
```

and press enter to view the contents of the array and see what has happened.

```
> colorStyles  
< [101] [99, 'dist1', 'dist2', 'dist3', 'dist4', 'dist5', 'dist6', 'dist7',  
  'dist8', 'dist9', 'dist10', 'distFar', empty * 88, 'hello world']
```

Ch05_inset01_08 Growing an array

The console shows us that the array is now 101 elements long, contains a block of 88 empty elements and has "hello world" on the end.

CODE ANALYSIS

JavaScript arrays

You might have started off thinking that there is nothing special about the way that arrays work in JavaScript. But you might have some questions now.

Question: How do I create an empty array?

```
let arr = [];
```

This will create an empty array call `arr` for use by pirates.

Question: Can I create an array of a particular size?

Some programming languages require you to create an array before you use it. In JavaScript you can't do this. You can `use` the `push` function to add an item on the end of an existing array:

```
arr.push(8);
```

Question: How do I find out the length of an array?

An array has a `length` property that gives the length of the array.

Question: Can I create two-dimensional arrays?

Some languages allow you to create multi-dimensional arrays that can hold grids and layers. In JavaScript an array can only hold a single row of items. If you want two dimensional array (for a grid) you will need to create an array of arrays, which is possible.

Question: What happens when I use an array as a parameter into a function call?

When an array is passed into a function a reference to the array is passed into function. This makes the data passing quicker (it might take a long time to copy an array before giving it to a function). It does mean that if the function changes the contents of the array these changes are made on the only copy of the array itself.

The colorStyles decode array

```
const colorStyles = ["cheese", "dist1", "dist2", "dist3", "dist4", "dist5",
    "dist6", "dist7", "dist8", "dist9", "dist10", "distFar"];
```

The array `colourStyles` is shown above. It contains a list of style names. We can feed an index value into the array to get a style name that represents a particular distance. A distance of 0 will select the style “cheese” whereas a distance of 11 gets the style “distFar”. We can create a function which is fed a distance value and returns the style that should be used for that square:

```
function getStyleForDistance(styles, distance){
    if(distance>=styles.length){90
        distance = styles.length-1;91
    }
    let result = styles[distance];92
    return result;
}
```

The `getStyleForDistance` function is given a list of styles and a distance value and returns the style that matches the distance. It makes sure that a very large value won’t cause an invalid style to be selected. I can use this to set the style for a square.

⁹⁰ Check if the distance fits in the array

⁹¹ Pick the far style if it does

⁹² Look up the name of the style to use

```

function setButtonStyle(button) {
    let x = button.getAttribute("x");93
    let y = button.getAttribute("y");94
    let distance = getCheeseDistance(x, y);95
    button.className = getStyleForDistance(colorStyles, distance);96
}

```

The function `setButtonStyle` is given a reference to a button and then sets the style of that button to match the distance of the button from the cheese. It uses the `getCheeseDistance` function to calculate the distance to the cheese. The `setButtonStyle` function is called in the event handler for the button:

```

function buttonClickedHandler(event) {
    let button = event.target;97

    if (button.className != "empty") {98
        return;
    }

    setButtonStyle(button);99

    if(button.className == "cheese"){100
        alert("Well done! Reload the page to play again");
    }
    else {
        counter++;101
    }
}

```

⁹³ Get the x position of the button

⁹⁴ Get the y position of the button

⁹⁵ Get the distance to the cheese

⁹⁶ Set the button style

⁹⁷ Get the button that was clicked

⁹⁸ If the button is not empty return

⁹⁹ Set the style for the button

¹⁰⁰ If the button has exploded, end the game

¹⁰¹ Update the turn counter

```

        showCounter();102
    }
}

```

The `buttonClickedHandler` function is called whenever the player clicks one of the buttons in the grid. The button that is clicked is located. The `buttonClickedHandler` function knows if a button has already been clicked because an un-clicked button has a `className` of "empty". The function then sets the style for the button. It then checks to see if the style is "cheese". If it is, the game is over and the function displays an alert. If the game is not over (the player has not clicked on the cheese button) the function increments and displays a turn counter and then continues.

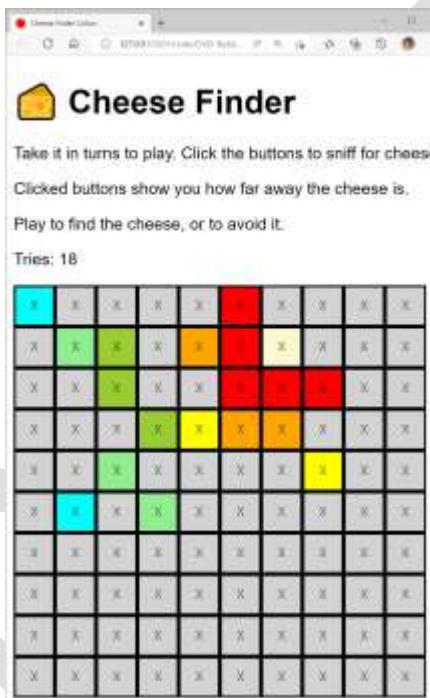


Figure 5.20 Ch-05_Fig_02 Colored Cheese Finder

Figure 5.2 above shows how a game is played on the colored version. You can see the path that I followed to get to the cheese. The cheese square is colored a pale yellow. This version of the game is available in the examples in the folder **Ch05-01_Colored_Cheese_Finder**.

¹⁰² Display the turn counter

Add a game ending

Players like the new colors. Then someone says it would be nice to display all the colored squares when the game ends. This would add nothing to the gameplay, but it sounds like fun, so we agree to do to this. All our program needs to do is work through all the buttons and set their styles. Of course, this means that the program will need an array of buttons to work through:

```
var allButtons = [];
```

The variable `allButtons` is an array that will hold an entry for each button. It is declared outside all the functions as a `var`, so that it can be used by any function in the applications. We add buttons to the array as we create them:

```
for (let y = 0; y < height; y++) {
    for (let x = 0; x < width; x++) {
        let newButton = document.createElement("button");
        newButton.className = "empty";
        newButton.setAttribute("x", x);
        newButton.setAttribute("y", y);
        newButton.addEventListener("click", buttonClickedHandler);
        newButton.textContent = "X";
        container.appendChild(newButton);
        allButtons.push(newButton);103
    }
    let lineBreak = document.createElement("br");
    container.appendChild(lineBreak);
}
```

Above is the nested for loop that creates all the buttons in the game. Each button is now added to the `allButtons` array using the `push` function. Now that we have the array, all we need to do is create a function that uses it to set the style for all the buttons:

¹⁰³ Add the button to the list of all buttons

```
function fillGrid(buttonList){  
    for(let button of allButtons){  
        if(button.className == "empty"){104  
            setButtonStyle(button);  
        }  
    }  
}
```

The `fillGrid` function is supplied with a list of buttons as a parameter. It works through all the buttons in the list using the `for – of` construction. Any buttons with the `className` of “empty” has its style set. The `fillGrid` function is called when the player finds the cheese.

¹⁰⁴ Does this button need filling?

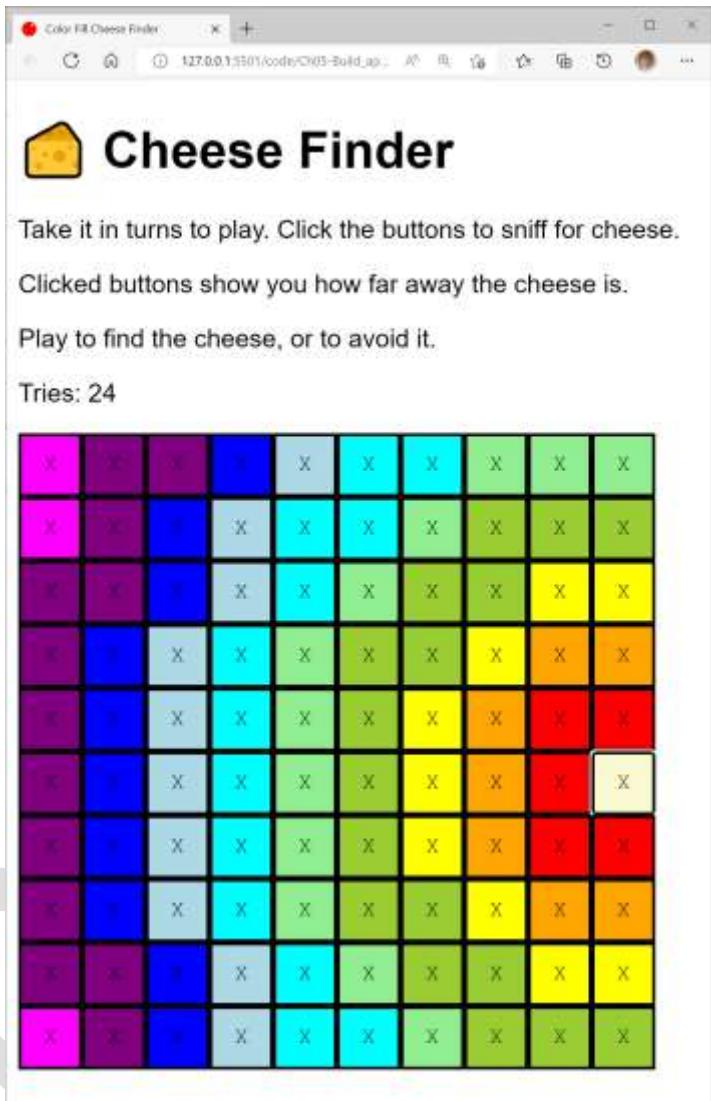


Figure 5.21 Ch-05_Fig_03 Filled Colored Cheese Finder

Figure 5.3 above shows what the end of a game now looks like. You can find this version of the game in the example files for this chapter in the folder **Ch05-02_Color_Fill_Cheese_Finder**.

Add randomness

People now quite like playing the game and they really like the colored display at the end. But fairly soon they discover that the game is actually quite easy to win. Once you have learned the

colors you can find the cheese quite quickly. Someone suggests that it might be more fun if the distance colors are mixed up at the start of the game. Players must now deduce the distance each color represents.

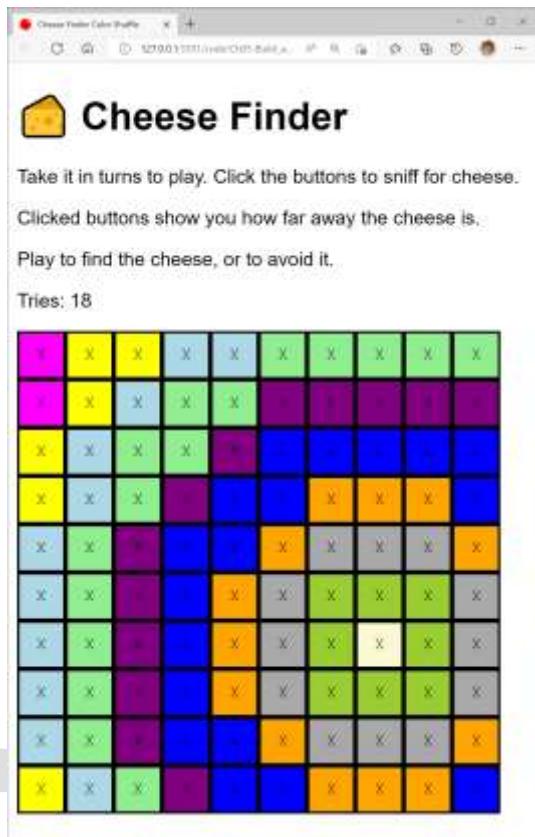


Figure 5.22 Ch-05_Fig_04 Cheese Finder shuffled

Figure 5.4 above shows how the game will work. In this game the color green means a distance of 1 and grey means a distance of two. Each time the game is played the mapping between colors and distance is different. We use a look up table to map distance values onto style names; we need to shuffle the style names before each game.

```
const colorStyles = ["white", "red", "orange", "yellow", "yellowGreen",
    "lightGreen", "cyan", "lightBlue", "blue", "purple",
    "magenta", "darkGray"];
```

Above you can see the `colourStyles` array that is used to convert a distance into a style name. The game must shuffle this list into a random order. We can create a function to shuffle arrays:

```
function shuffle(items){  
    for(let i=0;i<items.length;i++){  
        let swapPos = getRandom(0,items.length);  
        [items[i], items[swapPos]] = [items[swapPos], items[i]];  
    }  
}
```

The `shuffle` function works through an array picking a random position for each item in it. I've used a rather neat way of swapping two items that you might not have seen before. It performs the swap by assigning one array to another, avoiding the use of a temporary variable. This function is used to shuffle the `colorStyles` array:

```
shuffle(colorStyles);
```

The style for each distance will now be different each time the game runs. Note that we can use the `shuffle` function to shuffle any collection, not just names of styles. We will be using `shuffle` later to shuffle something else.

Finding the cheese

In the previous version of the game the cheese style was in the `colorStyles` array at element 0. This meant that as soon as the style of a button became "cheese" it meant that the cheese had been found. In the random version of the game we can't put cheese in the `colorStyles` array as it could get shuffled to any of the distance values. We must modify the `setButtonStyle` function so that it checks to see if the player has found the cheese and set the style to "cheese" if it has.

```
function setButtonStyle(button) {  
    let x = button.getAttribute("x");  
    let y = button.getAttribute("y");  
  
    let distance = getCheeseDistance(x, y);
```

```

if (distance == 0) {105
    button.className = "cheese";106
}
else {107
    button.className = getStyleForDistance(colorStyles, distance);
}

```

The version above checks to see if the button is at the cheese position and sets the style to cheese if it is. If the button is not at the cheese position the function uses the distance and to index the `colorStyles` array. Now, when the game ends it can call `fillGrid` to fill in all the colors.

```

if (button.className == "cheese") {
    fillGrid(allButtons);
}

```

You can find this version of the game in the folder **Ch05-03_Cheese_Finder_Color_Shuffle**. This is quite an interesting game to play.

Add more cheese

I strongly believe that you can improve anything, even cheese, by adding cheese. Some of the game players feel the same way. They think that the game would be even more challenging if there was more than one square containing a cheese. Each time the game starts it will need to set the position of both cheeses and then display the distance of a square to the nearest cheese.

```

var cheeseX = getRandom(0, width);
var cheeseY = getRandom(0, height);

```

¹⁰⁵ Have we found the cheese

¹⁰⁶ Set the style if we have

¹⁰⁷ Otherwise set the style for the distance

This is how the single cheese version of the game works. The two variables give the x and y coordinates of the cheese. A random number generator is used to set the x position of the cheese (the distance across the grid) and the y position of the cheese (the distance down the grid). If we want to handle two cheeses, we can add some extra variables:

```
var cheese1X = getRandom(0, width);
var cheese1Y = getRandom(0, height);
var cheese2X = getRandom(0, width);
var cheese2Y = getRandom(0, height);
```

We have made two new variables to store the position of a second cheese. We could then test for the two cheese locations when calculating the distance to the nearest cheese:

```
function getCheeseDistance(x, y) {
    let d1x = x - cheese1X;
    let d1y = y - cheese1Y;
    let distance1 = Math.round(Math.sqrt((d1x * d1x) + (d1y * d1y)));

    let d2x = x - cheese2X;
    let d2y = y - cheese2Y;
    let distance2 = Math.round(Math.sqrt((d2x * d2x) + (d2y * d2y)));

    let distance;

    if(distance1 < distance2){
        distance = distance1;
    }
    else {
        distance = distance2;
    }

    return distance;
}
```

This version of the `getCheeseDistance` gets the distance to each of the two cheeses and then returns the smallest value. The only other thing that is required is a cheese counter value to keep track of how many cheeses had been found:

```
let gameCheeseCounter = 0;
```

When a cheese is found this counter is increased and the game ends when it reaches two:

```
if (button.className == "cheese") {  
    gameCheeseCounter = gameCheeseCounter + 1;  
    if (gameCheeseCounter == 2) {  
        showCounter();  
        fillGrid(allButtons);  
    }  
}  
else {  
    gameMoveCounter++;  
    showCounter();  
}
```

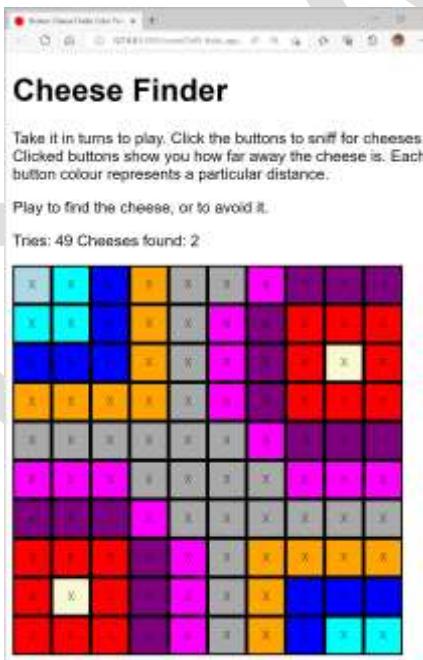


Figure 5.23 Ch-05_Fig_05 Two Cheese Finder

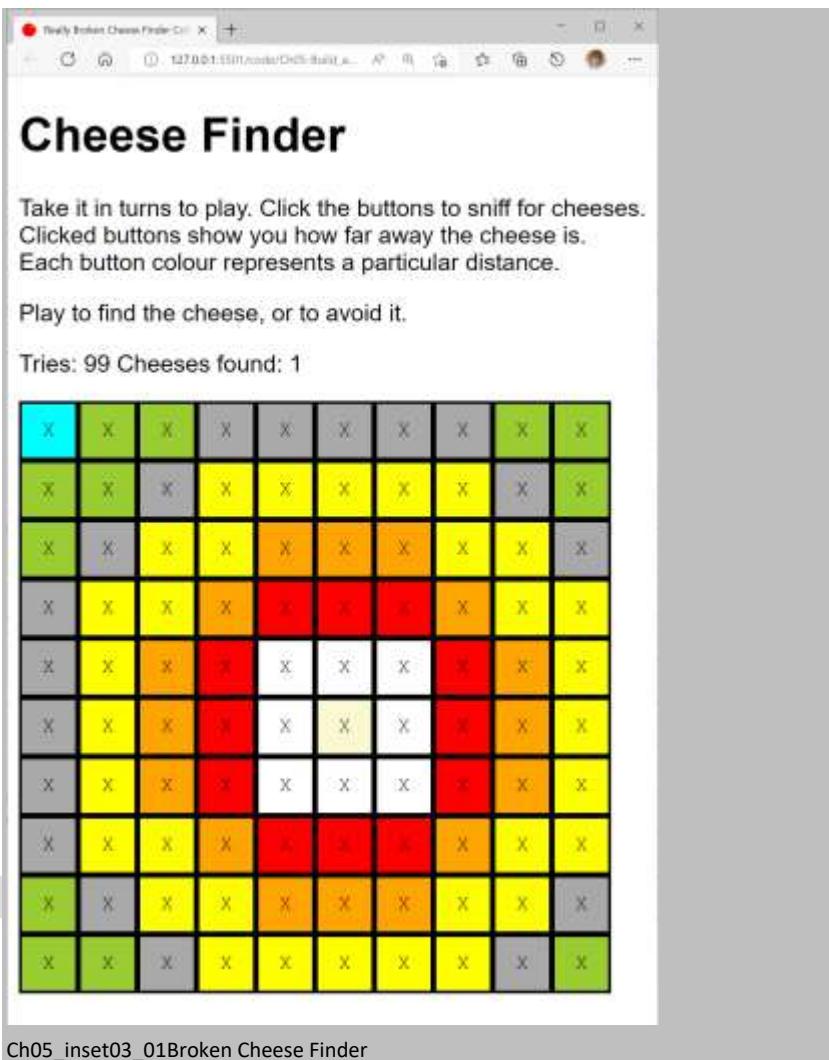
This version of the game seems to work fine. You can find it in the folder **Ch05-04_Broken_Cheese_Finder_2_Cheeses** in the example folders for this chapter. However, the name might not inspire confidence. Every now and then something bad will happen. Let's investigate.

Make Something Happen

Find the bug

There is definitely a bug in the two cheese version of the game. Perhaps you've already spotted the problem. For the rest of us, I've made a version that always goes wrong. Let's take a look at it. Start your browser and open the `index.html` file in the **Ch05-05_Really_Broken_Cheese_Finder_2_Cheeses** example folder for this chapter. If you play this game to completion, you'll notice the problem.

DRAFT



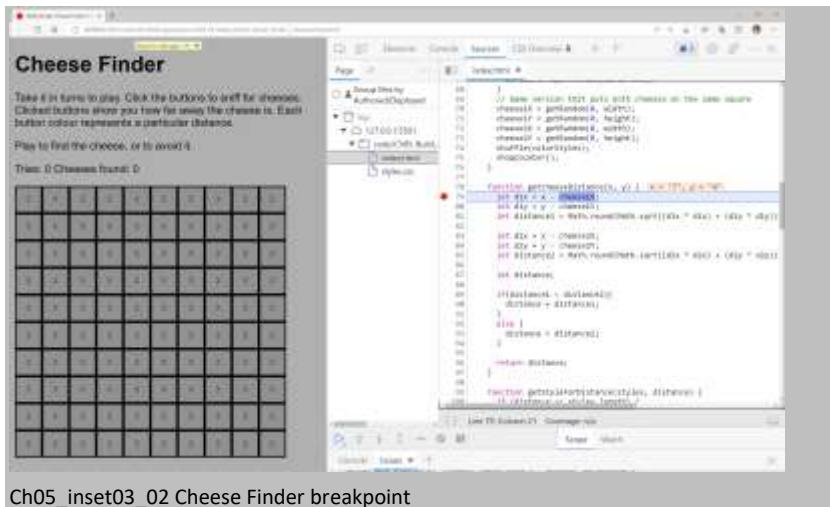
Ch05_inset03_01Broken Cheese Finder

You can see that every square has been clicked, but there is only one cheese on the board. We can investigate what is going on by using the debugger in the Developer Tools. Perform the following steps:

1. Open the Browser Developer tools.
2. Select the Sources tab.
3. Open the index.html file.
4. Find the `getCheeseDistance` function and put a breakpoint on the first statement in the function.

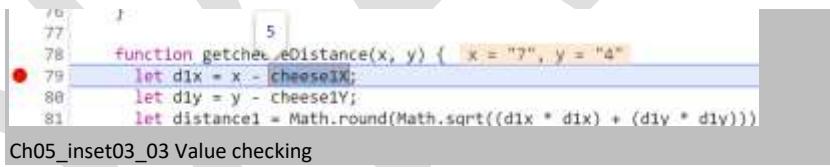
5. Reload the page to start a new game.

6. Click any button in the grid.



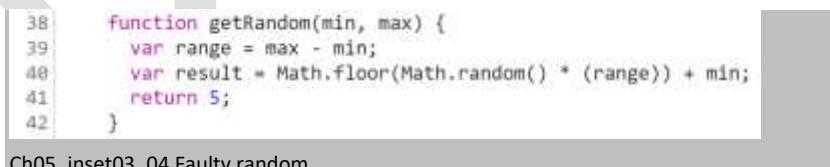
Ch05_inset03_02 Cheese Finder breakpoint

The program has reached the first statement of the `getCheeseDistance` function. If you look closely at the figure above, you will see that I clicked the button located at $x=7$ and $y=4$. Let's take a look at some of the other values in this code by hovering the mouse pointer over the variables in the code:



Ch05_inset03_03 Value checking

Above you can see that the value of `cheese1X` is 5. In fact, if you looked at all the cheese locations you will find that they are all (`cheese1X`, `cheese1Y`, `cheese2X`, `cheese2Y`) 5. This makes our random number generating process look very suspect. Find that code in the source. It is at line 38.



Ch05_inset03_04 Faulty random

The `getRandom` function calculates a random result and then returns a fixed value of 5. This means that the all the position values for the cheese are 5. In other words, both cheeses are at the same location. This will cause our program to break because there is no second cheese, there is only one. In real life the game will put both

pieces of cheese on the same square every now and then, and when it does it will be impossible to complete the game.

This code simulates the situation where both cheeses are placed in the same place on the grid, something which can happen by chance every now and then. We've found out what is wrong – now we need to fix it.

Testing for duplicated locations

We have seen that our two cheese version of the Cheese Finder game has the worst kind of bug; one that only appears every now and then. There is a one in a hundred chance that the second cheese will be placed in the same square as the first cheese. If this happens the game becomes impossible to finish. The fix for this is to check to make sure that the second cheese is not on the same square as the first:

```
cheese1X = getRandom(0, width);
cheese1Y = getRandom(0, height);
do {
    cheese2X = getRandom(0, width);
    cheese2Y = getRandom(0, height);
} while (cheese1X == cheese2X && cheese1Y == cheese2Y);
```

The code above shows how we do this. The position of the second cheese is repeatedly chosen until both values are different from the first cheese. You can find this version of the game in the [Ch05-06_Working_Cheese_Finder_2_Cheeses folder](#) in the example code for this chapter.

Even more cheese

The problem with giving people extra cheese is that they come back asking for even more. The two-cheese version of the game has become so popular that people now want even more cheese. At least three cheeses and probably even more. This is the point that our simple cheese placement code starts to betray its simple origins. To make three cheeses work we must add quite a few more statements to select cheese positions, check for clashes and test for distance. Code to support ten cheeses doesn't bear thinking about. At this point we need to stop and think about the way the program works.

Programmer's Point

Avoid kludges

Our solution for two cheeses works, but it doesn't scale very well when we add more cheeses. Worse still, the solution performance decreases dramatically as the number of cheeses increases. The code repeatedly generates possible cheese locations until it finds one that has not been used. More cheeses mean more chances for the program to pick a location that has been used. It is also not possible to predict exactly how long the program will take to place the cheese. If the program was spectacularly unlucky it could be stuck trying locations for a very long time. One of the aims of JavaScript and node is to make a system that can handle requests as quickly as possible. The last thing it wants is a process that might take a while to complete and could even get stuck.

If you think about it, the upgrade to add the second cheese is a bit of "kludge". A kludge is a solution that works despite the way that it is put together. It will be hard to understand, maintain and expand. One way to make a kludge is by adding something to a working system to adapt it to change in the requirements, which is exactly what we did.

It is important to recognize that a change in application requirements might mean that you must completely change the way that your solution works. This is one of those situations. We should have recognized two things as soon as we were asked to make a two-cheese version. The first is that our one cheese solution was a bad place to start from if we need to handle more than one cheese. The second is that the demand for extra cheese would not stop at two cheeses, we had to make a solution that could handle a very large number.

Make maximum cheese

It turns out that it is not too complicated to create a version of the game that can support a very large number of cheeses, up to the size of the grid. We can use the same technique as we used to create a list of randomized style names to be mapped onto distances for the random colors version of the game. We make a list of all the buttons, shuffle the list, and then work through the elements in the list. Each time the list is shuffled the element at index zero will be different, as will that at index one and so on.

It turns out that the game already contains an array that contains all the buttons in the grid. When the game starts it creates an array called `allButtons` which contains references to all the buttons in the grid. The `fillGrid` function works through this array to set the color style of all the buttons on the grid. If you're not clear on this, look at the section above with the heading "Add a game ending".

```
shuffle(allButtons);
```

The statement above shuffles the `allButtons` array to get a list of buttons in random order. We could say that `allButtons[0]` holds the first cheese in the game, `allButtons[2]` the second, and so on.

The function that checks for the nearest cheese can just work through this array for the number of cheeses that the game contains.

```
function getDistToNearestCheese(x, y) {  
    let result;  
    for (let cheeseNo = 0; cheeseNo < gameNoOfCheeses; cheeseNo = cheeseNo + 1) {  
        let cheeseButton = allButtons[cheeseNo];108  
        let distance = getDistance(cheeseButton, x, y);109  
        if (result == undefined) {110  
            result = distance;111  
        }  
        if (distance < result) {112  
            result = distance;113  
        }  
    }  
    return result;114  
}
```

The function `getDistToNearestCheese` is given an `x` and `y` value and returns the distance to the cheese nearest to that location. It works through the `allButtons` array to get cheese locations, gets the distance of each cheese and returns the smallest one. The variable `gameNoOfCheeses` is set with the number of cheeses being used in the game.

¹⁰⁸ Get this cheese button

¹⁰⁹ Get the distance to this cheese

¹¹⁰ Has result been set yet?

¹¹¹ Set it to the first value

¹¹² Is the distance less than result?

¹¹³ Set result to distance

¹¹⁴ Return the result



Figure 5.24 Ch-05_Fig_06 Four Cheeses

You can find this version of the game in the folder **Ch05-07_Infinite_Cheese_Finder** in the example code for this chapter. The number of cheeses in the game is set in the range 2 to 5 by the following statement which runs when the game starts.

```
gameNoOfCheeses = getRandom(2, 6);
```

This version of Cheese Finder makes a good single player experience, running entirely inside the browser. But now we are going to create a shared version.

Create a shared game

We have spent quite a lot of time creating a fun little cheese finding game. Then someone suggests that it might be fun to make a version that lots of people could play at the same time. Perhaps a room full of Cheese Finders could compete to discover who could find a hidden cheese the fastest. Rather than each game placing the cheese randomly at the start, everyone would start with the same cheese positions, and it would be a race against time to find them all first.

This would be easy enough to do, but there is a snag. As we have seen when using the browser Developer Tools debugger, it is possible for someone to look the code behind a web page and extract information from it. A cunning player could win the big prize simply by pressing F12 and getting the cheese position variable values from the code in the debugger.

This is a problem faced by any browser-based application that wants to keep secrets. We could try to hide the way that our browser code works by a process called **obfuscation**, which takes a program, renames all the variables and generally makes the text hard to understand. However, no matter how clever the obfuscation is, it is still the case that the cheese position is being held in the browser, which makes it vulnerable to attack.

We can keep the cheese position completely secret by storing it on the server that hosts the web page containing the page. The way our game works is that the user clicks a button, the game looks up the style for that button (based on the distance of the button from the nearest cheese) and then sets that style on the page. It then checks to see if the style indicates that a cheese has been found and updates the game state accordingly. With the current version of the game this lookup process takes place in the browser.

For a shared game we are going to make the browser ask the server what the style would be for a particular square. In the section “Make a web server” in chapter 4 we saw how a JavaScript program running under node.js can respond to web queries and serve out messages and files. Now we are going to discover how a program running in a browser can ask questions of a server and update its display accordingly.

Design a conversation

When our server-based Cheese Finder game is running the browser and the server will be exchanging messages. The first thing we need to do is decide the form and meaning of the messages to be exchanged. Then we can work on how the programs will send and receive them. This is an important part of application design. In a way we are creating our own language for a conversation. Let’s describe the conversation in general terms and then look at how we would implement this:

1. Player navigates to the web site to play “Cheese Finder”.
2. The browser loads the web page from the server and starts running a JavaScript program in the web page.
3. The program running in the browser asks the server for some details about the game. It needs to know the width and the height of the grid and the number of cheeses being searched for.
4. The browser builds the web page containing the grid and displays the count of how many cheeses left to be found and the number of turns taken.
5. The player clicks a button on the grid. The browser sends the location of the clicked to the server. The server responds with the style for that location. The browser updates the state of the game on the display and then waits for the next button click.
6. This continues until the game detects that the player has found the last cheese.

It is important check this workflow carefully. A good way to test it is for one person to be the browser and another the server. Then work through playing a game and look at the information that is sent back and forth.



Figure 5.25 Ch-05_Fig_07 Browser and Server Messages

The next thing we need to do is create the code that implements the conversation between the browser and the server. Figure 5.7 above shows what we are going to make. It shows the game being played in the browser and the server providing the cheese management. The code running in the browser will ask the server for the game details when it starts. It will use the details to draw the grid for the game. Then, each time the user clicks on one of the buttons in the grid the browser will send a message asking for the style for that button. The server will respond with a string containing this style. The browser will then update the game.

The game is going to be implemented by two JavaScript programs. One runs on the browser and provides the game user interface. The other runs on the server and manages the cheese position. We will be running both the server and the browser on our computer. Later we can take the server program and put it in the cloud so that anyone can have a go at the game.

When we are looking at code samples in this text it is useful to be reminded of just where the code is going to run. For the rest of this chapter I'm going to adopt a convention that samples of code that run in the browser will have a light yellow border, whereas code samples running on the server will have a light blue border.

Create endpoints

An html endpoint is a web address that a browser can use to get something. My world famous blog (www.robmiles.com) is an endpoint that you can visit if you want to find out about my so-called life. We need to create two endpoints for the Cheese Finder server. The first endpoint will return the game details. The second endpoint will return the style of a particular button on the

grid. The final version of the game that the world will be able to play will be at the address cheesefinder.xyz but now we will be hosting the server on our local machine.

```
let hostAddress = "http://localhost:8080/";
let startUrl = hostAddress + "getstart.json";
let getStyleUrl = hostAddress + "getstyle.json";
```

The JavaScript above runs in the browser and sets the values `startUrl` and `getStyleUrl` which will be used by the browser code as the endpoints. We are going to run a node.js server behind port 8080 on our machine. The browser will use these endpoints to connect to the server. It is important that the server and the browser both agree on the names of these endpoints, otherwise the application will not work. When I move the server into the cloud, I will change the string "http://localhost:8080" to "http://cheesefinder.xyz".

Start the game

If a user wants to play Cheese Finder they will start by opening the web site. The server will send back a page of HTML for the browser to display. Then the browser will build the Document Object Model (DOM) and then run the JavaScript program in the HTML file.

```
<!DOCTYPE html>
<html>

<head>
  <title>Server Cheese Finder</title>
  <link rel="shortcut icon" type="image/x-icon" href="favicon.ico">
  <link rel="stylesheet" href="styles.css">
</head>
<h1>Server Cheese Finder</h1>

<body>

  <p>Take it in turns to play. Click the buttons to sniff for cheeses.
    Clicked buttons show you how far away the cheese is.
    Each button colour represents a particular distance.</p>
  <p>Play to find the cheese, or to avoid it.</p>
```

```

<p id="counterPar"></p>115
<p id="buttonPar"> </p>116

<script type="module">
  import { doPlayGame } from "./client.mjs";117
  doPlayGame();118

</script>
</body>

</html>

```

Above is the html file for the web page sent from the server to the browser. The page contains two paragraph elements, one with the id `counterPar` which is used to display the counters and another with the id `buttonPar` which will hold all the buttons of the grid. The JavaScript code in the file itself is very small. The first statement imports the function `doPlayGame` from the `client.mjs` library and the second statement calls a function called `doPlayGame`. Let's have a look it.

```

function doPlayGame() {
  moveCounter = 0;
  cheesesFound = 0;119
  getFromServer(startUrl, setupGame);120
}

```

This function doesn't do much. It sets the `moveCounter` and the `cheesesFound` variables to 0 and then it calls another function, `getFromServer`. This function will get the game details from the server and then use them to set up the game grid. The first argument sent into the `getFromServer` function is `startUrl`, the endpoint address of the `getstart` service on the server. The second argument is a reference to the `setupGame` function which will set the game up. Let's have a look at the

¹¹⁵ Counter paragraph

¹¹⁶ Button grid paragraph

¹¹⁷ Import the game start function

¹¹⁸ Call the game start function

¹¹⁹ Clear the game counters

¹²⁰ Start the game

`getFromServer` function.

```
function getFromServer(url, handler) {
  fetch(url).then(response => {
    response.text().then(result => {
      handler(result);
    }).catch(error => alert("Bad text: " + error));
  }).catch(error => alert("Bad fetch: " + error));
}
```

The first function, `doPlayGame` looked quite simple. This one, although not much longer, looks horrible. You might be wondering what it does. This function gets a response from a server endpoint url. It uses a JavaScript function called `fetch` which fetches pages from a server. It does the same thing that a browser does when it loads a page from the web, except that rather than taking what it receives and using it to build a page, it passes back what it received to the caller.

Fetching something from the web can take a while and we don't want our game to be held up waiting for the message to come back from the server. To address this the `fetch` function returns a JavaScript `promise` object that represents the fetch operation being performed. We will learn all about promises in the next chapter. Just for now we will have to take this function and just use it. Later we'll discover all about `fetch`, promises and asynchronous code. I'm really looking forwards to that section.

When the `fetch` in our call of `getFromServer` has completed fetching the response from the server it calls the function it was passed, giving that function the response that was received. The function that is going to set up the game is called `setupGame`. Let's take a look at that:

```
function setupGame(gameDetailsJSON) {
  let gameDetails = JSON.parse(gameDetailsJSON);121
  noOfCheeses = gameDetails.noOfCheeses;122
  let container = document.getElementById("buttonPar");
```

¹²¹ Get the game details from the response

¹²² Save the number of cheeses from the details

```
for (let y = 0; y < gameDetails.height; y++) {123
    for (let x = 0; x < gameDetails.width; x++) {
        let newButton = document.createElement("button");
        newButton.className = "empty";
        newButton.setAttribute("x", x);
        newButton.setAttribute("y", y);
        newButton.addEventListener("click", buttonClickedHandler);
        newButton.textContent = "X";
        container.appendChild(newButton);
        allButtons.push(newButton);
    }
    let lineBreak = document.createElement("br");
    container.appendChild(lineBreak);
}
showCounters();
}
```

You've seen most of this code before in the section "Place the buttons" in chapter 4 where we created the first Cheese Finder game. The interesting part of this code is right at the very top. This is where the response from the server is decoded to get the values of the width, height and number of cheeses. This is all done with a single line of JavaScript, using the magic of JavaScript Object Notation or JSON. Let's take a tiny digression to see how that works.

Transfer data with JSON

The program in the browser uses the `startUrl` endpoint to ask the server the question "Can I have the width and height of the screen and the number of cheeses please". It would be useful if there was a way we could encode and decode these values without having to do much work. Well, it turns out that there is. It's called "JavaScript Object Notation" or JSON for short. It lets us take an object and convert it into a string of text. Let's have a play with it.

Make Something Happen

Investigate JSON

JSON is a tremendously powerful part of JavaScript. Let's investigate it now. Start your browser and open the `index.html` file in **Ch05-08_JSON_****Investigation** example folder for this

¹²³ Create the button grid

chapter. Open the Developer Tools and navigate to the console.

In this exercise we will be creating and decoding some JSON strings.
To open Developer Tools in the browser press the F12 key. If your keyboard does not have that key, hold down the CTRL and SHIFT keys and press J.
The console will appear on the right of the page. You may need to confirm that you want to open it.
You can find the steps to follow for the exercise in Chapter 5 of the book in the section "Make Something Happen - Investigate JSON".

Ch05_inset04_01 JSON investigation

JSON works with JavaScript objects. We know that we can create a JavaScript object anywhere in the code (it is one of the things we love about the language). Let's make a tiny object that provides the answer to the first question the browser wants to know; the width and height of the grid and the number of cheeses. Type in the following statement and press Enter:

```
let answer = { width:10,height:10,cheeses:3};
```

This statement creates a variable called `answer` which refers to an object containing `width`, `height` and `cheeses` properties. We can ask the console to show this value to us. Type in the following statement and press Enter:

```
answer
```

The console will now show us the object that `answer` refers to:

```
> let answer = { width:10,height:10,cheeses:3}
< undefined
> answer
< 1: {width: 10, height: 10, cheeses: 3}
>
```

Ch05_inset04_02 answer object

You can see that the properties are inside the object. This display of the object is very similar to the JSON code that represents the object contents. We can use the wonderfully named `stringify` function to turn an object into a JSON string. Type the following statement and press Enter:

```
let jsonString = JSON.stringify(answer)
```

The `stringify` method accepts an object reference and then returns the JSON string that describes the object contents. The statement above creates a variable called `jsonString` that refers to a string that contains the JSON representation of our object. Now we want to take a

look at this string. Type the following statement and press Enter:

```
jsonString
```

The console will now show the JSON string describing the object:

```
> let jsonString = JSON.stringify(answer)
< undefined
> jsonString
< '{"width":10,"height":10,"cheeses":3}'
>
```

Ch05_inset04_03 answer json

You can see that the JSON encoded string looks almost identical to the way that we would declare an object in JavaScript program code. The only difference is that names of the properties are enclosed in double-quote characters. We can convert this string back into an object by using the JSON `parse` function to get this information back from the string. Type the following statement and press Enter:

```
JSON.parse(jsonString)
```

The console will perform the parse and then show the object that was created by the parse function.

```
> JSON.parse(jsonString)
< {width: 10, height: 10, cheeses: 3}
>
```

Ch05_inset04_04 parsed json

The result is an object that holds exactly the values that we need. The JavaScript code in the browser can read the width, height and cheeses properties from the object and use them to set up the page.

CODE ANALYSIS

JSON

JSON is wonderful. But you still might have questions about it.

Question: What kinds of values can I save in a JSON string?

You can write numbers, strings, booleans (true or false) and objects.

Question: Can I save an array of items in JSON?

Yes you can. An array of the required size is created when the JSON is parsed.

Question: Is there a limit to the length of a string that can be produced when you JSON encode an object?

The limit is large. JavaScript strings can be very long.

Question: What happens if a program tries to parse a string that does not contain valid JSON?

The parse function will stop the program and throw an exception if it is given invalid JSON to decode. We will discuss exceptions later in the book.

The game server

We've spent a lot of time on how the browser works. Now it is time to look at what the server is doing. The server needs to serve out the files that make up the web site, including index.html, the code libraries and the stylesheets. The starting point of the server is the one we created at the end of chapter 4. We can use this to serve out the game web site. Then we add the code to handle the `getstart` and `getstyle` endpoints. We'll do this in the `handlePageRequest` function in the server.

```
function handlePageRequest(request, response) {  
  
    let filePath = basePath + request.url;  
  
    if (fs.existsSync(filePath)) {124  
        // If it is a file - return it 125  
    else {  
        // If it is not a file it might be a command  
        console.log("Might have a request");  
        switch (request.url) {  
            case '/getstart.json':126  
                response.statusCode = 200;  
                response.setHeader('Content-Type', 'text/json');  
                let answer = { width:gridWidth,height:gridHeight,  
                    noOfCheeses:cheeses.length};  
                json = JSON.stringify(answer);127  
                response.write(json);  
                response.end();128  
        }  
    }  
}
```

¹²⁴ Is there a file with this path?

¹²⁵ Open the file and send it back

¹²⁶ Case for getstart endpoint

¹²⁷ Create the answer

¹²⁸ Send back the answer

```
        break;
    default:
        console.log("      file not found")
        response.statusCode = 404;
        response.setHeader('Content-Type', 'text/plain');
        response.write("Can't find file at: " + filePath);
        response.end();
    }
}
}
```

The first part of the `handlePageRequest` function serves out files on the server. You can find an explanation of this code at the end of Chapter 4 in the section Serving out files. If the function can't find a file matching the request it uses a switch construction to check if the request is for the endpoint `getstart.json`. The code handling this endpoint assembles the required JSON and sends it back.

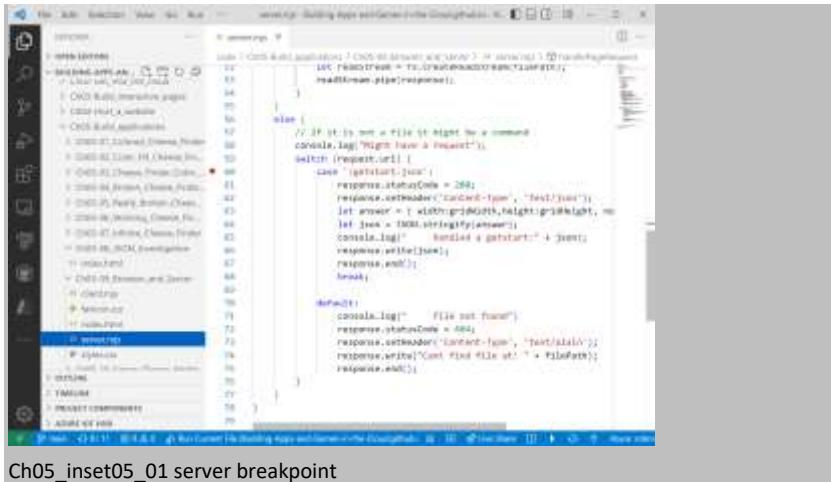
Make Something Happen

Browser and Server

This might be a good time to use our debugging skills to discover how the browser and server work together. We've done this kind of thing before in the Make Something Happen: Use debug to investigate the server in Chapter 4. It's a bit complicated, but totally worth it.

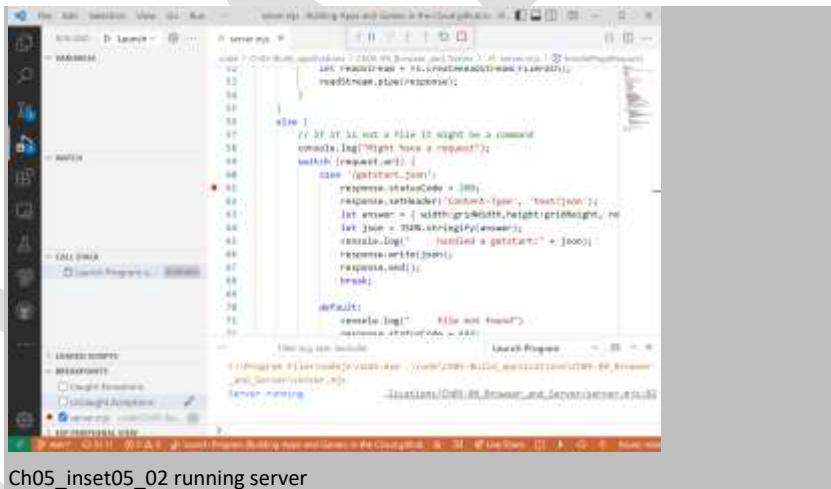
Start Visual Studio Code, open the examples for this chapter and find the **Ch05-09_Browser_and_Server** example folder. We are going to run both the server and the browser at the same time. We'll put breakpoints in the code to watch the two programs interact.

Open the file **server.mjs** in this folder. This contains the code that implements the game server. Now click against line 60 to put in a breakpoint. When the browser uses the `getstart.json` endpoint this breakpoint will be hit.



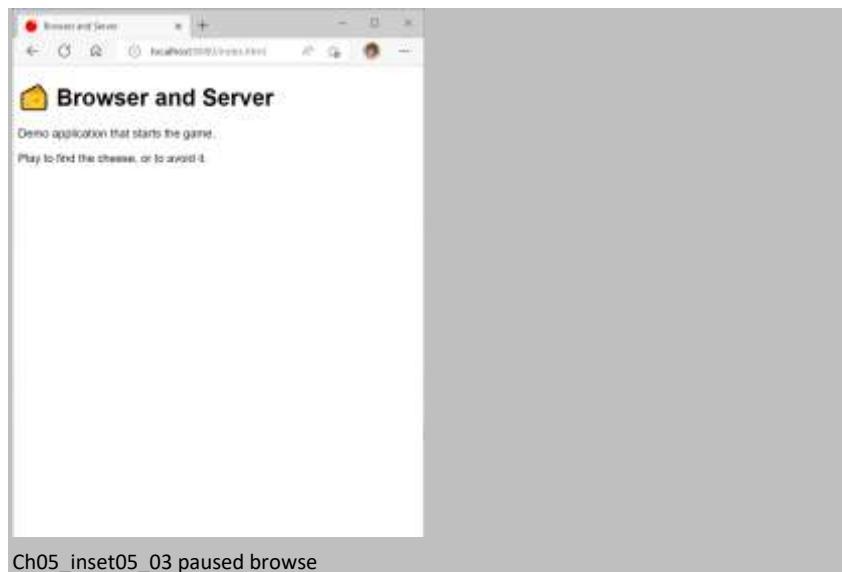
Ch05_inset05_01 server breakpoint

Now press the **Run and Debug** button to open the debug window and start the program running.



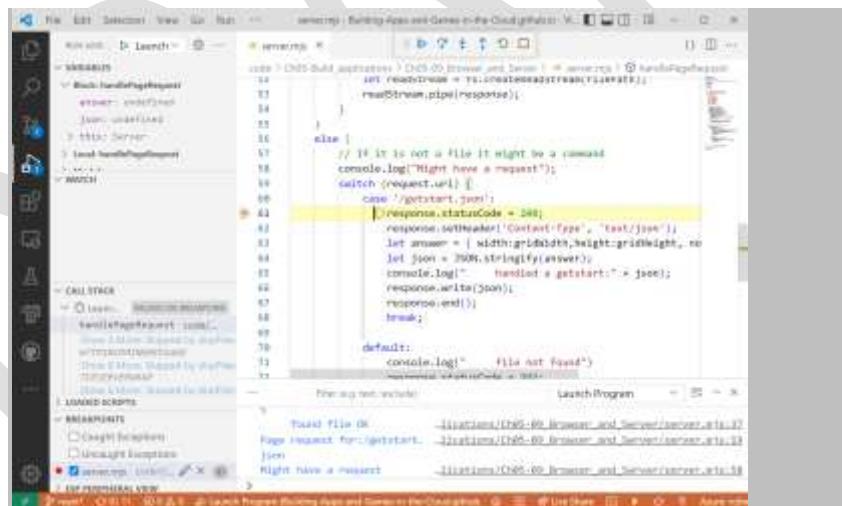
Ch05_inset05_02 running server

The program is now hosting a web server on `http://localhost:8080`. Let's open the browser and load the game website into it. Start the browser and enter the address `http://localhost:8080/index.html` to open the index page of the game. Don't forget to put the `index.html` on the end.



Ch05_inset05_03 paused browse

You will notice that the game board has not appeared in the browser. This is because the game has made a request to the `getstart` endpoint but the server has hit our breakpoint before it could send anything back. Let's take a look at Visual Studio Code window.



Ch05_inset05_04 server breakpoint

You can step through the code after the breakpoint, but you had better be quick before the fetch request from the browser times out. If you click the blue run button in the debug controls (or press F5) the server will continue, and you should see the game board appear in the browser. You might like to use the debugger in the browser Developer Tools to watch the code in the browser pick up the messages from the server and use them to build the pages.

This version of the game doesn't respond to button presses in the grid. We will be doing that in the next section.

Play the game

We've seen the `getstart` endpoint deliver game information to the browser. Now we need to create the `getStyle` endpoint as well. This will be used by the browser code to get the style for a button that has been clicked in the grid. In the original game this task was performed by code running in the browser. Now the distance calculation and style selection takes place in the server. The good news is that since both browser and server are running JavaScript we can take these parts of the solution out of the browser code and drop it into the server code. The bad news is that the browser needs to send the location of the button to the server so that the server can work out the style string to return.

The browser program can use the HTTP query mechanism to pass the values of x and y into the server. Let's look at the code which does this.

```
function setButtonStyle(button) {
  let x = button.getAttribute("x");
  let y = button.getAttribute("y");129
  let checkUrl = getStyleUrl + "?x=" + x + "&y=" + y;130
  getFromServer(checkUrl, result => {131
    button.className = result
    if (button.className == "cheese") {132
      cheesesFound++;
      if (cheesesFound == noOfCheeses) {
        fillGrid(allButtons);133
      }
    }
  });
}
```

¹²⁹ Get the x and y position of the button

¹³⁰ Assemble the query

¹³¹ Get the result from the endpoint

¹³² If we got cheese – check for game end

¹³³ Fill the grid if the game has ended

This is the `setButtonStyle` that function that runs in the browser when the player clicks on a button in the grid. In the browser-based versions of the game this function calculated the distance of the button from the cheese and then used that to set the style of the button. In the server based version the browser creates a query and sends it the `checkUru` endpoint in the server. The query part of the endpoint starts with a ? (query) character and is followed by name-value pairs separated by & characters.

```
http://localhost:8080/getstyle.json?x=2&y=0
```

Above you can see the endpoint address with query information added on the end. This query is requesting the style for the button at location (2,0). Now we need to discover how the server can recover these values and use them. When the server receives this endpoint it must extract the query information from the endpoint. We could write our own code do this but JavaScript provides a `url` library which will do all the work for us.

```
var parsedUrl = url.parse(request.url, true);134
```

The code above uses the `parse` function from the `url` library to create a `parse` object referred to by `parsedUrl`. The call of `parse` has two arguments. The first is the url string that is being parsed. The second argument specifies whether query elements in the url should be parsed. We want them to be parsed (that's why we are doing this) so we set this to `true`. Now we can extract the local path and the queries.

```
case '/getstyle.json':  
    let x = Number(parsedUrl.query.x);  
    let y = Number(parsedUrl.query.y);135  
    response.statusCode = 200;  
    response.setHeader('Content-Type', 'text/json');  
    console.log("Got: (" + x + ", " + y + ")");
```

¹³⁴ Make a url object from the request

¹³⁵ Get the button x and y

```
let styleText = getStyle(x,y);136
let styleObject = {style:styleText};137
let styleJSON = JSON.stringify(styleObject);138
response.write(styleJSON);139
response.end();
break;
```

CODE ANALYSIS

The getstyle.json handler

Above is the code that runs on the server to handle the `getstyle.json` endpoint. You might have some questions about it.

Question: When does this code run?

When a player clicks on one of the buttons in the grid the browser needs to know the style for that button. It gets the x and y location of the button and assembles a web request that is sent to the server. The server identifies the request from the get style endpoint (`getstyle.json`) and so it runs this code to get the style for that button and then send that back as a JSON encoded string.

Question: How does this code get the x and y values for the button location?

```
let x = Number(parsedUrl.query.x);
```

Earlier we created a `parsedUrl` object which contained the web request that was sent to the server. This had a `query` property. The `query` property refers to an object that contains properties for all the elements in the query. Above you can see the statement that extracts the x value from this property. There is a similar statement for the y property.

Question: How does the server determine the style for a grid location?

```
function getStyle(x, y {
```

¹³⁶ Get the style of this button

¹³⁷ Make an object that contains the style value

¹³⁸ Convert the object to a JSON string

¹³⁹ Send the string to the browser

```

let distance = getDistToNearestCheese(x, y);

if (distance == 0) {
    return "cheese";
}

if (distance >= colorStyles.length) {
    distance = colorStyles.length - 1;
}
return colorStyles[distance];
}

```

The style for a grid location is determined by the distance that location is from the cheese. The `getStyle` function is given the x and y location of the button and calls the `getDistanceToNearestCheese` to get the distance to the cheese. It then looks up the required style in the `colorStyles` array. This is the same code as we're using in the browser version of the game.

```

const cheeses = [
    { x: 0, y: 0 },
    { x: 1, y: 1 },
    { x: 2, y: 2 }
];

```

In browser versions of the game the cheese locations were determined randomly. For a shared version of the game all the players must get the same cheese locations. The game presently contains an array of cheese location objects which we can use for testing. A future version of the game will create random cheese locations at regular intervals, perhaps every hour.

Question: What happens if a request doesn't contain query values for x and y?

```
http://localhost:8080/getstyle.json?wally=99
```

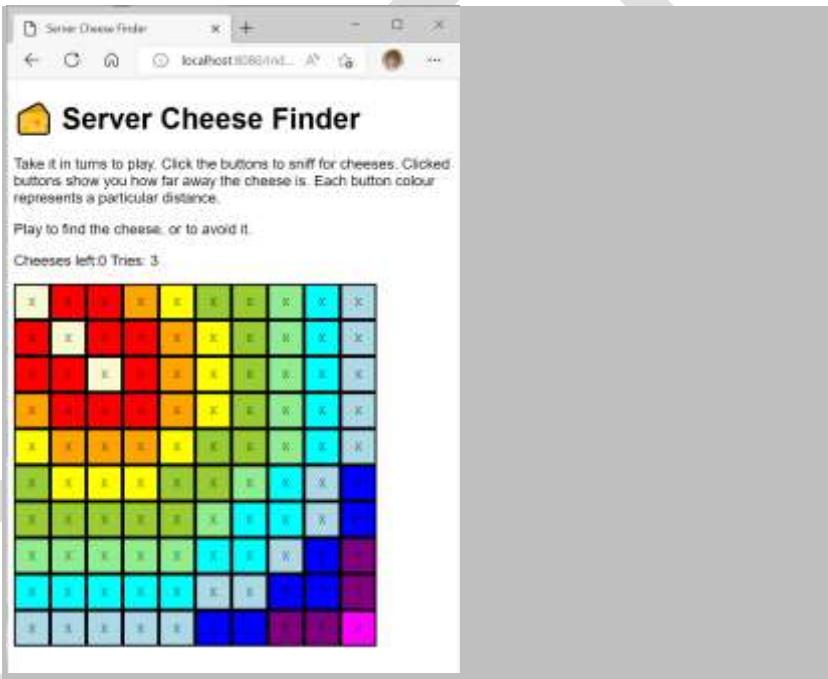
The endpoint above doesn't contain query values for x and y. It just has a value for "wally". The game server would try to decode it, but if a JavaScript program tries to access a property of an object which is not present the result is the value "undefined". If the x and y values were missed off the query the handler would feed the undefined values into the call of `getStyle`. This would end up sending an empty JSON object back to the browser, resulting in the style of the button being set to "undefined". There is no undefined style in the stylesheet, so the button will not have a sensible style.

In this case it looks like nothing bad has happened to the game, but this has been more by luck than judgement. A "battle hardened" version of the server would need to check for the query values and then send back an "error" style to indicate that the grid location could not be determined. The same thing should happen if the browser asks for the style for a grid location which is not present, for example x=99 and y=100.

Make Something Happen

Play Server Cheese Finder

The version of Cheese Finder that we have at the moment is not quite ready for release. We'll have to add some cheese hiding code and tidy a few things up, but the programs as they stand do implement a working version of the game. You can find working versions of the browser and server in the example folder **Ch05-10_Server_Cheese_Finder**. The process of starting and debugging the code is exactly the same as for the previous make something happen.



Ch05_inset06_01 server cheese finder

Above you can see a completed game. Note that the cheeses are always in the same places. You can move them to other squares by editing the code `server.mjs`.

What you have learned

This has been another busy chapter. We've covered a lot of ground. Here is a recap plus some points to ponder.

- Up until now we have used an array to store data in indexed locations. We can also use an array as a lookup table. The index can be used to specify a matching value stored at that position in the array.
- Arrays in JavaScript are indexed from zero. An array is never created as having a particular size, instead it either created from constant data or values are automatically added to the array when values are placed in elements.
- Writing to an indexed value in an array will cause an array to automatically expand to that size. In other words, if you store a value with an index value of 10 in an array which only contains 4 values JavaScript will automatically add the extra values (set to the value undefined) to fill in the gap.
- You can use the function `push` on an array instance to add an element on the end of an array.
- An array exposes a `length` property which gives the number of elements in the array.
- It is not possible to create multi-dimensional arrays in JavaScript. But an array can contain elements that are themselves arrays.
- Arrays are passed into JavaScript functions as references to the array object.
- A program running entirely in the browser is not secure, in that it is possible for a user to look the code and variables in the program. The Developer Tools debugger can even be used to step through the code and view the contents of variables.
- We create secure applications by running part of the application on a server. The client can send web requests to the server which delivers answers. The server the code is not visible to code running outside it.
- When creating an application that runs on the browser and the server it is important to work through the application behaviors and identify the requests to be made by the browser and the contents the replies.
- An endpoint is an html address that can be used by a program to request that the server behind the address perform an action.
- JavaScript provides a `fetch` function which sends a request to a web server and returns the response to the program.
- The JavaScript `fetch` function uses the JavaScript `promise` mechanism to

ensure that a slow fetch operation will not pause the running of a program. A `fetch` operation can be made to run a callback function when the fetch has completed.

- JavaScript Object Notation (JSON) can be used to transfer the data in a JavaScript object. A JSON encoded object is a string of text. The `JSON.stringify` function will act on an object to encode it into string. When an object is encoded numbers and text values are stored directly. A reference is stored by following the reference and JSON encoding the reference that it refers to. The JSON encoded string can contain arrays.
- The `JSON.parse` function will decode an object description string and build an object that contains the values described in the string. If the string does not contain valid JSON the parse function will throw an exception.
- We can watch a browser/server based application run by using the debug tools in the server and browser. The server will serve out the application on the localhost address. When the server code is transferred into the cloud the http address of the server will be changed to the cloud address.
- A url can contain query information which consist of name-value pairs which can be picked up by the server and used to control the behavior of code running on the server.
- If the server receives invalid query information (perhaps a query value is missing or out of range) the program will still run but `undefined` will be returned to the client. It is important that the implantation of a command endpoint deals correctly with invalid endpoints.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions".

How can you tell the type of elements in an array?

Each element in an array will have a type, but an array itself does not have a type. It is just a collection of items.

How do you clear an array?

There is no way to clear an array. However, if you set the array variable to refer to an empty array the previous one will be inaccessible and will eventually be removed by the JavaScript garbage collector.

Can the server send requests to the browser?

There is no way that the server can send messages to the browser. The browser can only

respond to incoming requests. If a program running on the browser wants to respond to an event on the server it must repeatedly query the server until the server responds that the event has occurred.

What is the difference between an endpoint and a web page url?

From a functional point of view there is nothing different about the two, although an endpoint request might have the language extension `json` rather than the `html` extension of a url. It is up to the server examine the incoming address and decide what to send back in response.

Can you use a JSON message to send binary information?

If you want to send binary information (perhaps a graphic or a sound) you will need to encode the binary information as text before sending it.

Can you use a JSON message to send a reference?

No. When stringifying an object JSON will always follow each reference and then insert the JSON text that describes that object. Note that this means that if an object being stringified contains three references to the same object, the JSON string will contain three copies of the object contents.

Can you use a JSON message to send a function?

No. If the object being stringified contains functions they will be ignored.

Can a website from one url access an endpoint on another?

Browsers implement a “same origin policy” that prevents a website running scripts from a website from a different origin. This policy only applies to scripts, it is perfectly OK for a site to load images and stylesheets from a different page. The policy prevents someone making a webpage which runs scripts on another webpage. This would be a very dangerous thing to do, we know how easy it is for a JavaScript program to access data in the Document Object Model. If I could create a web page running on my server that could run scripts on a webpage that you wrote and are running on your server, this would be very bad news for you. So the same origin policy prevents this from happening.

Chapter 6:

Create a shared

experience

What you will learn

Humans enjoy sharing things together. The feeling of being part of a crowd at a football match, or all watching the same TV show together is a great thing. The internet can host shared experiences, from streamed video gameplay to puzzles we can all work on at the same time. In this section we are going to create a version of the Cheese Finder game which is a shared experience. All the players playing the game at any given time will be solving the same grid. To do this we will discover how to synchronize server and browser, use pseudo-random numbers to generate “repeatable” random behavior and spend some time optimizing our code for the cloud. And finally, we will place our shared Cheese Finder in the cloud where anyone can play it.

Sharing gameplay

The first versions of Cheese Finder ran inside the browser on the player’s computer. Every player experience was different because the distance colors and cheese positions were different each time the game ran. These games used the `Math.random()` function from JavaScript to get the random numbers used to position the cheese and select the distance colors. This made for good game play for individuals but running the game engine on a server makes it possible for us to create shared experiences, where everybody gets to play the same game at the same time. Let’s see how we would go about doing this.



Figure 6.26 Ch06_Fig_01 fixed cheese positions

Figure 6.1 shows a completed game of Cheese Finder. You can play this game by running the code in the folder **Ch06-01_Fixed_Cheese_Finder** in the sample code for this chapter. This version provides a shared experience (everybody plays the game with the cheese in the same positions and the same distance colors) but it is not a very good one because the experience is the same every time the game is played.

```
const gameSetup = {  
  
    colorStyles: ["white", "magenta", "red", "lightGreen", "orange",  
                  "yellow", "yellowGreen", "cyan", "lightBlue", "blue",  
                  "purple", "darkGray"],  
    cheeses: [  
        { x: 4, y: 0 },  
        { x: 2, y: 3 },  
        { x: 7, y: 4 }  
    ]  
}
```

The code above shows why you always get the same experience when you play this version of the

game. If you look at the Figure 6.1 you will see that the locations in the code correspond to what the player gets when they play the game. Remember that the origin of the grid (0,0) is the top left-hand corner of the grid. The `gameSetup` variable refers to an object that contains two properties. The `colorStyles` property is an array that contains a particular sequence of style names and the `cheeses` property is an array holding three specific cheese locations. When the game runs these are used to place the cheeses and pick the distance colors. This means that the game is always the same. The first time you play it with your friends you might find it fun to see who can find the cheeses the fastest. But the second time around is much less of a challenge.

What we would like is a way of changing the shared experience at regular intervals, in the same way that a popular word puzzle provides a different word to find every day. Everyone would play with the same cheese positions and distance colors, but these would change at regular intervals.

Create shared gameplay

At the moment the only way to change colors and cheese positions is to edit the contents of the `gameSetup.mjs` file and change the values in the `gameSetup` variable. We could do this, but it would rapidly become very tiresome, particularly as we might like to provide a different grid design every hour. We could create an array of `gameSetup` objects with a setup for each hour of the day.

```
const gameSetups = [  
  { // hour 0  
    colorStyles: ['darkGray', 'purple', 'white', 'blue', 'lightGreen',  
    'red', 'lightBlue', 'yellowGreen', 'yellow', 'cyan', 'magenta', 'orange'],  
    cheeses: [{ x: 4, y: 0 }, { x: 2, y: 3 }, { x: 7, y: 4 }]  
  },  
  ...  
  { // hour 11  
    colorStyles: ['orange', 'cyan', 'darkGray', 'lightBlue', 'blue',  
    'red', 'magenta', 'yellow', 'purple', 'yellowGreen', 'white', 'lightGreen'],  
    cheeses: [{ x: 9, y: 5 }]  
  }  
];
```

The array `gameSetups` in the `server.mjs` contains 12 elements (the listing only shows two) each of which contains a list of colors and a set of cheese positions. When the game starts it uses the current time to decide which map to use.

```
let date = new Date();140
let hour = date.getHours() % 12;141
gameSetup = gameSetups[hour];142
```

This code runs in the server at the start of the code that processes commands from the browser. It gets the setup for the current time. It uses the `Date` object that we first saw when we made a clock in chapter 1. It extracts the hour from the data and then uses the modulus operator (%) to restrict the hour to the range 0 to 11 (I did this because I didn't want to make another 12 game setups). It then looks up the appropriate game setup for the hour in the `gameSetups` array and sets it. This means that players of the game will get a different game setup for each hour of the day.

There is an implementation of this game in the folder **Ch06-02_Hourly_Cheese_Finder**. You can start it and play it if you like. Remember that because this game is hosted on a server you will have to start the server running using the node runtime debugger in Visual Studio Code and then open the `index.html` file on localhost. Use the url <http://localhost:8080/index.html> in your browser. You have done this before in the **Make Something Happen: Browser and Server** exercise in chapter 5. Just follow the same process to start the server and visit the page. Unfortunately, you may find that there is a serious bug with this implementation of the game.

Debug shared gameplay

One of the things you will have to get good at as a programmer is debugging. That's just as well because we now have a bug in our game. Players are reporting that sometimes the cheese moves about while they are playing the game. They don't think that this is fair, and I'm inclined to agree with them. Let's investigate to see if we can find out what is happening.

Make Something Happen

Debug timed gameplay

We can investigate how this code works by debugging it. Use the steps from the **Make Something Happen: Browser and Server** exercise in Chapter 5 to open the `server.mjs` file in the **Ch06-02_Hourly_Cheese_Finder** folder and start the server running. Now open your

¹⁴⁰ Get the current date

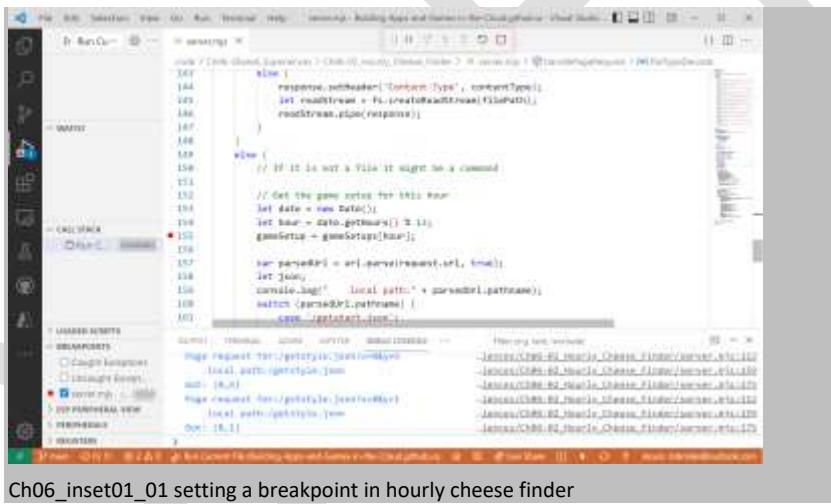
¹⁴¹ Extract the hour value mod 12

¹⁴² Get the game setup for that hour

browser and navigate to <http://localhost:8080/index.html>. The browser will open the page and the game will start. Play a game and watch how it runs. It will probably run perfectly correctly. There is no fault. The reports are wrong.

At this point we could just tell the players that we can't find the fault and that the game is obviously perfectly OK. But we don't. Instead, we ask them for more detail. Are there any special circumstances when they were playing the game? When did they start playing? When did the game go wrong? It turns out that the game seemed to go wrong when the hour changed. Aha! Perhaps when the hour changes the game server switches to a different grid in the middle of the game, causing the cheeses to change position. We can use the debugger to test our theory.

The first thing we do is play a complete Cheese Finder game and make a note of where the cheeses are. The cheeses will be in the same place the next time we play the game if we play it in the same hour. Reload the game and click a few locations to validate that the grid is the same. Now we are going to interrupt the program and move into the next hour by changing the contents of one of the variables.



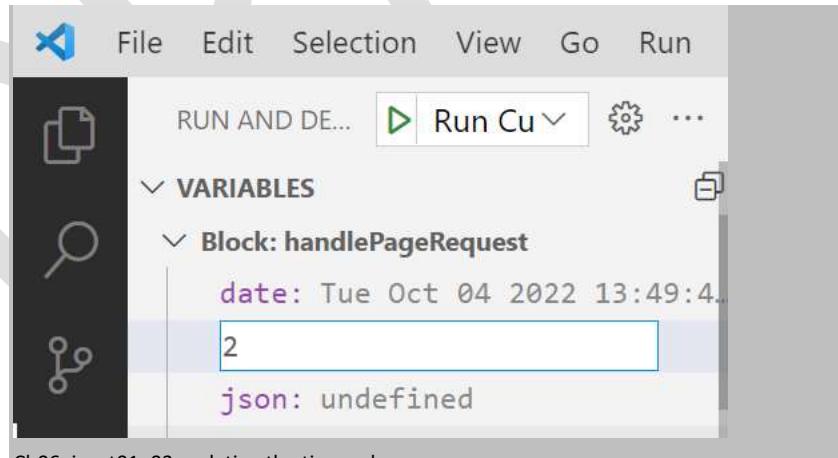
Ch06_inset01_01 setting a breakpoint in hourly cheese finder

One of the great things about the Visual Studio Code debugger is that you can add breakpoints to the program as it is running. Find line 155 and click next to it as shown above to set a breakpoint. Now go back to the browser and click on a square on the grid that you know the color of.

```
private void handleFileRequest(HttpServletRequest request, HttpServletResponse response) throws IOException {
    String filePath = request.getServletContext().getRealPath("/cheese");
    File file = new File(filePath);
    if (file.exists() &amp; file.isFile()) {
        response.setContentType("application/octet-stream");
        response.setHeader("Content-Type", "application/octet-stream");
        response.setContentLength((int) file.length());
        response.getOutputStream().write(new FileInputStream(file).readAllBytes());
        response.getOutputStream().flush();
    } else {
        // If it is not a file it might be a comment
        int date = new Date().getTime();
        int hour = date.getHours() * 60;
        String setup = gameSetup + gameSetup(hour);
        var paramString = url.parse(request.getRequestURI());
        paramString.setQuery("setup=" + setup);
        String paramStringString = paramString.toURL().toString();
        response.sendRedirect(paramStringString);
    }
}
```

Ch06_inset01_02 hitting a breakpoint in hourly cheese finder

The server program will hit the breakpoint you just set. This happens because the browser program has asked the server for the style to use for a square and the server is selecting the `gameSetup` for this hour. Find the `hour` value in the Variables section on the top left of the window. It is in the **Block:handlepagerequest** section as shown above. The hour value is 1. This means that when I took this screenshot it was in the hour between 1 o'clock and 2 o'clock. You will see the hour value of the time when you are running the program. We could wait until the next hour to test our theory about the fault, but we can also change the values inside variables using Visual Studio code. Double click the value for the `hour` and increase it by 1.



~~chose_inset01_03 updating the time value~~

Press enter once you have updated the value. Now click the blue right-pointing button in the debug controls to continue the program. Now go back to the browser and look at the grid. The grid location that you clicked will now have a color, but you should find that it is the

wrong color, because the server is using a different `gameSetup`. The cheese has moved because the hour has changed and the server is now sending out style settings for its new location.

We could have tested our theory by starting a game in one hour, waiting a while and then continuing it in another hour. However, being able to change the values in a variable inside the code made it possible to test the code much more easily. There are lots of other ways you can use debugging to make your life easier. You can create a breakpoint that fires when a certain condition is true or after it has been hit a particular number of times. It is well worth finding out more about these features. Leave Visual Studio code running and the browser open. We will use it to test our fix for the bug.

CODE ANALYSIS

Fault analysis

The story so far: Players play the Cheese Finder game by clicking squares on a grid displayed by their browser. The squares change to a color which represents the distance that location is from the nearest cheese. By working out which color represents which distance a clever player can work out which squares contain the cheese in the smallest number of clicks. The first version of the game ran entirely in the browser. Code running in the browser calculated the color of the squares. We then created a server version of Cheese Finder. When this version is played the locations of grid clicks are sent from the browser into the server which then responds with the color of that square. We've modified the server version to deliver a different player experience every hour by changing to a different grid each hour. Players have complained that the cheese sometimes moves to a different location during play and we have discovered that this is because if the hour changes during a game the cheese location will change as the server moves onto a different game setup. Now we must fix this error.

One of the hardest aspects of debugging is that faults often end up being on a "two for one" deal. Fixing one fault can sometimes create two more. In chapter 5 we saw the dangers of using "kludges" when we design a program. We also need to be careful not to fix our bug with a kludge that might cause other problems. So, let's consider some questions.

Question: Whose fault is the bug?

When things go wrong, we have a natural human tendency to try and look for someone to blame. You should never do this when debugging. I've worked on many projects, and I've discovered that everybody writes faulty code. If you make a big fuss about someone else's faults you can expect them to make a similar fuss about yours. That's not to say you shouldn't discuss how a fault occurred and what you can do to ensure similar ones happen again, it is just that you should do this while recognizing bugs are a consequence of writing code in the same way that we get smoke with fire.

In this case the fault lies with the person who designed how the game works because they didn't think of the situation where the hour changes in the middle of a game. We need to fix that fault and then remember that if we make any time-based behaviors in any new games we must consider what happens when the time changes during gameplay.

Question: How do we fix the fault by adding code to the server?

We can't. This is not a fault that can be fixed at the server end. The server is completely unaware of the history of any browsers requesting style values for particular squares. As we have seen before, one of the fundamental principles of the world wide web is that each transaction (a browser asks for a web page and the server sends it back) is independent of any other. There is no way that our server program can know that a game on a particular browser was started in a previous hour. In the next chapter we will discover how we can add extra code and behaviors into the browser and server code so that the server can know who is using it, but for now we don't have this and so we can't fix the fault in the server.

Question: How do we fix the fault in the code in the browser?

Before we fix the fault we need to consider what we want to happen in this situation. Perhaps it might be nice if a game started in one hour just continued into the next. The browser sends the x and y positions of a grid location to the server when it wants to know the color for that location. We could add an hour value to the request from the that the server can then send back the correct color response for the specified hour. A problem with this solution is that it breaks the "shared experience". At any given time there would be people playing versions from different hours.

A better solution would be for the browser to abandon a game when the hour changes. This adds some interesting jeopardy to the gameplay. You might find people who deliberately wait until the last minute in the hour before starting to play, so that they can complete the game in the nick of time. We will need to change the responses from the server so that they include the hour value for each response. The browser will store the hour value received when the game started and check it against an hour value sent by the server in each response. If a different hour value is received the browser will display an alert and reload the latest game.

Question: Since the browser and the server both have access to the time, why does the server have to send the hour value to the browser?

This is a good question. You might think that the browser program could use the [Date](#) function to get the hour value and use this to decide when a game should be abandoned. However, I don't think this is a very good idea. Most of the time the clocks in the browser and the server would correspond and so the hour would change at the same time on each. However, we can't be completely sure that the hour values will change at *exactly* the same time on the server and the browser. If the dates are obtained independently there is always a chance that an error could arise every now and then. An error which happens every hour is hard to debug. An error which happens every week is really hard to debug. Sending the hour value from the server completely removes the possibility of this error occurring.

If the server sends an hour value to the browser this also makes the application much easier to test. A different hour value from the server should trigger the browser to abandon a game. We can use a breakpoint to change the hour value the server sends to the browser and then make sure the browser reloads the page.

Synchronize the browser and the server

The code we are going to add will keep the browser and the server “in sync”. If the hour changes the server will move into the new hour and the browser must detect this and abandon the game it was playing in the old hour. Each message the server sends to the browser must include an hour value so that the browser can use it to work out if it is still in sync.

When the game is being played the server sends two messages to the browser. One is sent at the start of the game. It tells the browser the size of the grid and the number of cheeses in the grid. The browser uses this information to draw the grid that is used to play the game. It turns out to be very easy to add an hour value to the answer object that is sent back to the browser in response to a “getstart” message:

```
let answer = { width: gridWidth,
               height: gridHeight,
               noOfCheeses: gameSetup.cheeses.length,
               hour: hour };143
```

Above you can see the answer object that is created in the server when the browser asks for information about the game. This will be encoded and sent in a JSON string to the browser. If you are unclear about how this works, look in the section “Create endpoints” in Chapter 5.

```
function setupGame(gameDetailsJSON) {
  let gameDetails = JSON.parse(gameDetailsJSON);
  noOfCheeses = gameDetails.noOfCheeses;
  gameHour = gameDetails.hour;
```

¹⁴³ Send back the hour value when the game was started

```
// reset of setupGame here  
}
```

Above you can see the function in the browser that receives the `answer` object sent from the server. It takes the `noOfCheese` and `hour` properties from the object and stores them in the browser. The `noOfCheese` value is used by the browser so the game can detect when the player has found the last cheese. The `gameHour` value is used to detect when the server becomes out of sync with the browser. When the server sends a style value back to the browser it also includes the hour value:

```
let styleObject = { style: styleText, hour: hour };
```

The statement above shows how the style information is added to the `styleObject` being sent back to the browser. In the previous version of the program the style object just contained the style text. Now it contains the current hour value as well. Now let's see how code in the browser can use this to decide when to restart the game:

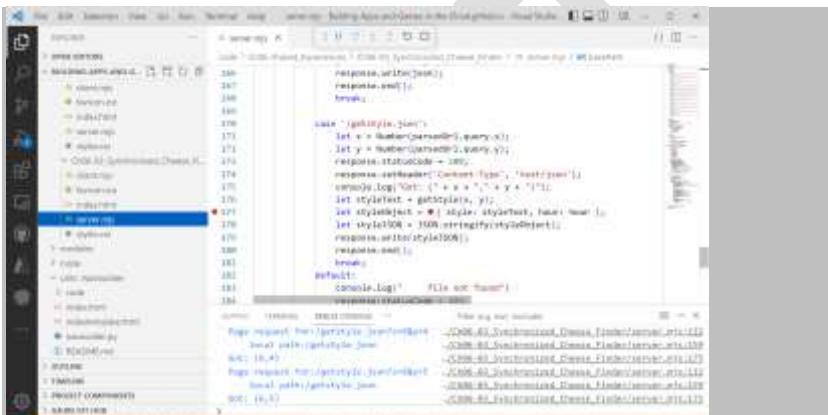
```
let checkDetails = JSON.parse(result);  
if(checkDetails.hour != gameHour){  
    // we have reached the end of the hour  
    // end the game  
    alert("The game in this hour has ended.");  
    location.reload();  
}
```

The statements above are performed when the browser receives a response from the server. If the hour in the received (called `checkDetails`) is different from the one saved when the game started the browser displays an alert to the player and then reloads the page which will restart the game. The JavaScript `location.reload()` function is used to reload the page. You can find a version of the game that synchronizes the server and browser in the example folder **Ch06-03_Synchronized_Cheese_Finder**. We can use the debugger to test it.

Make Something Happen

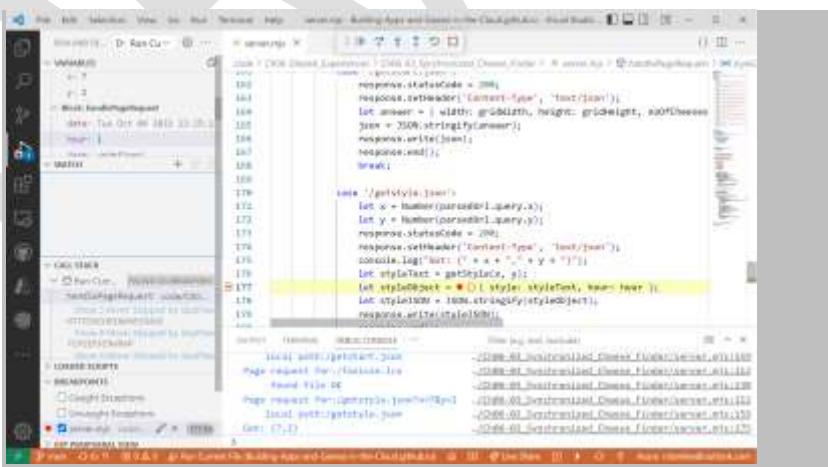
Test synchronized gameplay

We can test whether our fixed version of the game works correctly by going through the same sequence of tests as we used in the previous “Make Something Happen: Debug timed gameplay”. Open the **server.mjs** file in the **Ch06-03_Synchronized_Cheese_Finder** folder and start the server running. Now open your browser and navigate to **http://localhost:8080/index.html**. The browser will open the page and the game will start. Click a few squares and note that they update correctly. Now we are going to do some time travel. We can use the debugger to change the hour value sent from the server back to the browser. This should cause the browser to abandon the game. Set a breakpoint at line 177 in **server.mjs** as shown below. This statement creates the response sent to the browser when the browser asks for the style for a particular square.



Ch06_inset03_01 setting a breakpoint in synchronized cheese finder

Now click on a grid location in the browser. The browser the server for the style for that location and the code in the server will hit the breakpoint.



Ch06_inset03_02 hitting a breakpoint in synchronized cheese finder

Above you can see that the program has hit the breakpoint at the statement that creates the object to be sent back to the browser. We are going to change the value of `hour` to make the browser decide that it is out of sync. Open up the **Block:handlePageRequest** item in **variables** displayed in the top left. This block is where the `hour` variable is described. Double click the `hour` value in this and change the value so that it is one larger. This exactly what we did in the previous Make Something Happen where we were looking for the bug. Now we are making the change to check if we have fixed the bug. Click the blue arrow in the debugging controls to make the program continue. The server will continue running. Now look at the browser window.



The browser is displaying a message indicating that the game has ended. If you click OK the game will be reloaded with the new cheese positions. The fix seems to work.

Make Pseudo-random values

We now have a version of Cheese Finder which we can put on the internet for anyone to use. However, I'm not keen on having to keep updating the grid with new cheese locations and style colors by hand. I would really like a way that the server could create a unique game every hour automatically. It turns out this is possible. We are going to use a technique called "pseudo random numbers". It is used throughout the internet and is the basis of technology that secures network traffic. Let's look at it.

Computers have a real problem with randomness. A computer that behaves in a random way is called "broken". However, programs frequently need random numbers and so we need a way to get randomness out of the machine. It turns out that a computer can make random numbers in the same way I look like I can do lots of things, by faking it. It is hard for a program to make a

random number from nothing, but we can make a function that takes in a number and uses it to make another one that seems to have no connection to the value that was input. It can do this by multiplying the incoming number by a large number, adding another large number and then taking the modulus of the result.

```
function startRand() {  
    randValue = 1234;          // set initial random number  
    randMult = 8121;           // set the multiplier  
    randAdd = 28413;           // set the amount to add  
    randModulus = 134456789;   // set the modulus value  
}
```

Above you can see the function `startRand`. This is called to set the initial values for a pseudo-random number generator. These values are called the *seed* values for the random number generator. These the starting value of `randValue` (which keeps track of our current random number), `randMult` (the number that we multiply the current value by to get the next one), `randAdd` (the number we add) and `randModulus` (the number of the modulus).

```
function pseudoRand() {  
    randValue = ((randMult * randValue) + randAdd) % randModulus;  
    return randValue/randModulus;  
}
```

The function `pseudoRand()` above generates a value in the range 0 (inclusive) to 1 (not inclusive). This is the same range of values that JavaScript `Math.random()` produces which means that it can be used in place of `Math.random` in our programs. The `pseudoRand()` function uses the current value of `randVal` to calculate a new one. The previous random number is multiplied by a large value (`randMult`) has another value added (`randValue`). The statement then applies the modulus value (`randModulus`) produce the next random value. If we divide this value by `randModulus` we will get a value between 0 and 1 (but not including 1 because the modulus operation makes sure that `randValue` will never reach `randModulus`).

CODE ANALYSIS

Making random numbers

You may have some questions about how this function works.

Question: What does the modulus operator do?

This is an important part of the process. Each time we calculate a new random number we multiply the previous value by `randMult`. We need to do something to stop the value of `randValue` getting larger and larger. The modulus operator divides one number by another and returns the remainder of the division. The value 19 modulus 16 is 3. Sixteen goes into 19 once, remainder 3. So, in our program using the modulus value will make sure that the value of `randValue` never goes above 134456789.

Question: Are the `randMult`, `randAdd` and `randModulus` values we are using to generate the random numbers particularly special?

Some combinations of values can cause the successive random values to get “stuck” at particular values or to bounce between them. These values seem to work.

Question: Will the sequence of random numbers ever repeat?

It turns out that the set of starting values above returns to the original `randvalue` of 1234 (the starting value) every 33,614,190 values. In other words, we would have calculate over 33 million random numbers before we saw the sequence repeat. You might like to ponder how I worked out that value of 33,614,190. I’ll give the answer later in this section.

Question: Will the random sequence be different on different computers?

Great question. The JavaScript language specification includes a description of how numeric values are stored and manipulated so the random number sequence should be the same on every computer that is running standard JavaScript.

Make Something Happen

Random dice

The web page in the **Ch06-04_Pseudo_Random_Dice** folder implements a dice using the `pseudoRand` function we have just seen. Let’s have a look at it. Open your browser and navigate to the `index.html` file in the folder **Ch06-04_Pseudo_Random_Dice**

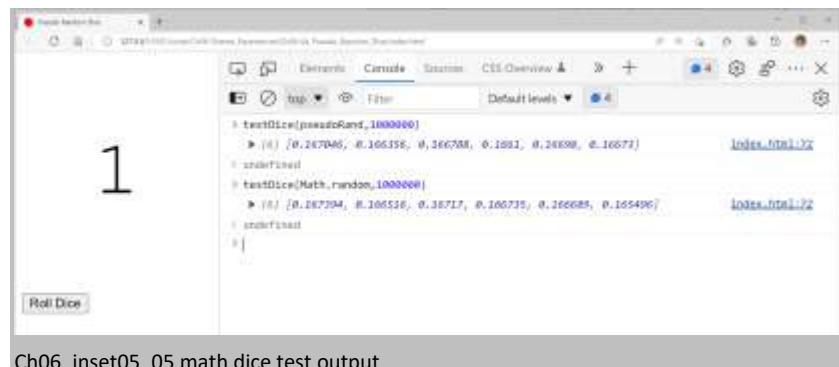


Click the **Roll Dice** button to roll the dice. The dice always rolls a 1 when it starts. In fact, the sequence of values it produces is always the same. If you reload the page in the browser it will take you back to the start of the sequence.

This page has hidden features in the form of a functions we can use to test the randomness of the dice. Open the Developer Tools and switch to the console tab. Now type the statement `testDice(pseudoRand,1000000)` and press enter:



Above you can see the output from this function. It displays an array containing the fraction of the rolls for each of the dice scores, from 1 to 6. For a perfect roll there should be a probability of one sixth (0.166666) for each of the scores. The `testDice` function is supplied with two arguments. The first identifies the random number function to be used. The second give the number of tests to performed. The call above throws the dice a million times using the `pseudoRand` random number function. We can test the performance of the `Math.random` function by using that instead. Type the statement `testDice(Math.random,1000000)` and press Enter:



Ch06_inset05_05 math dice test output

This time the `testDice` function does a million tests using `Math.random` to generate the random numbers. As you can see, the values are very close to the ones produced by our pseudo random function.

CODE ANALYSIS

Test random numbers

You may have some questions about how `textDice` works.

Question: Why do I get different results from the ones you get?

When you test the `pseudoRand` function the program is at a particular position in the random number stream, which affects the scores for each roll. If you perform the test directly after the dice page has loaded, before you click to roll any dice yourself, you will see the same numbers as above.

The `Math.random` function has been designed to provide a totally random value each time it is called, so the results using that function will be different each time it is tested.

Question: How does the `testDice` function work?

Good question. Here's the code:

```
function testDice(testFunction,noOfTests) {
    randFunction = testFunction144
    let totals = [0, 0, 0, 0, 0, 0, 0];145
```

¹⁴⁴ Set the random number function

¹⁴⁵ Create the totals array

```

for (let i = 0; i < noOfTests; i++) {146
  let roll = getRandom(1, 7);147
  totals[roll - 1]++;148
}
let fractions = totals.map((v) => v / noOfTests);149
console.log(fractions);150
}

```

The dice roll page contains a variable called `randFunction` that refers to the random number generator being used by the page. This variable is initially set to `pseudoRand`:

```
let randFunction = pseudoRand;
```

The `testDice` function is supplied with a reference to the function to be used to create the random numbers. The first thing the function does is set the value of `randFunction` to the function provided as a parameter. Then it creates an array to hold the totals for each of the dice scores, from 1 to 6. The initial totals are all 0. Then the test function performs a loop 1,000,000 times.

Each time round the loop the function gets a random value into a variable called `roll`. The value of `roll` is then then used as an index to specify the element in the `totals` array which is to be increased by 1. The function deducts 1 from the roll value because a dice value goes from 1 to 6 and the array is indexed from 0 to 5. As an example, if the roll value returned 4 the element at index 3 in the `totals` array would be increased by 1 (that is what the `++` operator does).

When the loop has finished the program uses the `map` function provided by the `totals` array to make a new array (called `fractions`) which contains each element of `totals` divided by the number of dice rolls. This gives the fraction of the time to divide each of the count values in the array. We can feed `testDice` any function to generate the random values.

```
testDice(()=>0.9,10000)
```

This provides `testDice` with a function literal that returns the value 0.9 every time it is called. This would result in a dice that always rolls a six. The `testDice` function leaves the `randFunction` variable referring to the function it was supplied with, so performing the above test would result in a dice that always returns the value six. You can reset the behavior by

¹⁴⁶ Test loop

¹⁴⁷ Get a dice roll

¹⁴⁸ Update the count

¹⁴⁹ Get the fractions

¹⁵⁰ Log the values

reloading the page.

Question: How did you work out how long it takes for the random sequence to repeat?

```
function getRepeatLoopSize() {  
    startRand();151  
    let firstValue = randValue;152  
    let counter = 1;153  
    while (true) {154  
        pseudoRand();155  
        if (randValue == firstValue) {  
            break;  
        }  
        counter = counter + 1;  
    }  
    console.log("Loop size: " + String(counter));  
}
```

The function `getRepeatLoopSize` starts the random number generator and records the first value in the sequence. It then repeatedly gets random numbers until it sees the first value again. It counts each time around the loop in a variable called `counter` and displays the result at the end.

```
> getRepeatLoopSize()  
Loop size: 33614190  
< undefined  
>
```

Ch06_inset06_01 get repeat loop size

Above you can see the result of calling the function. This took a couple of seconds to return as the program worked through all the random values.

Create a pseudo-random library

```
let randValue;  
let randMult;
```

¹⁵¹ Reset the random number generator

¹⁵² Remember the first random value

¹⁵³ Start a counter

¹⁵⁴ Start a loop

¹⁵⁵ Get the next random number

```

let randAdd;
let randModulus;

function setupRand(settings) {156
    randValue = settings.startValue;
    randMult = settings.randMult;
    randAdd = settings.randAdd;
    randModulus = settings.randModulus
}

function pseudoRand() {157
    randValue = ((randMult * randValue) + randAdd) % randModulus;
    return randValue / randModulus;
}

function getRandom(min, max) {158
    var range = max - min;
    var result = Math.floor(pseudoRand() * range) + min;
    return result;
}

export {setupRand, pseudoRand, getRandom};159

```

The code above is stored in a file called [pseudorandom.mjs](#). This can be imported into any application that wants to use a pseudo-random number sequence. It is the same code that we have already seen, but re-packed as a library. The `setupRand` function accepts a `settings` object which is used to set the initial values of the variables that control the random number generation.

```

import {setupRand, getRandom} from "./pseudorandom.mjs";

let randSettings = {
    startValue:1234,
    randMult:8121,
    randAdd:28413,
    randModulus:134456789

```

¹⁵⁶ Copies the settings into the library

¹⁵⁷ Gets the next number

¹⁵⁸ Gets an integer in a range

¹⁵⁹ Export the functions

```
}
```

```
setupRand(randSettings);
```

The code above shows how we use the library. First we import the two functions from the library that we need to use. The dice program only needs two functions. It uses `setupRand` to set the initial values of and `getRandom` to get random numbers. The configuration values are provided as an object literal. These are the same as were used for the original dice application. You can find a version of the dice application that uses the random library in the folder **Ch06-05_Dice_using_library**.

Generate timed randomness

We now understand the way that computers generate random numbers, but we don't seem to have solved our original problem, which was to find a way that our game can create different cheese positions and style colors every hour. However, now you know the effect of the seed values on the random numbers that are produced you might be able to think of a way that this we could do this. Look at the following function:

```
function getAbsoluteHour(date){  
    let result = (date.getFullYear() * 365 * 24) +  
        (date.getMonth() * 31 * 24) +  
        (date.getDate() * 24) +  
        date.getHours();  
    return result;  
}
```

It works out an “absolute hour value”. This is the approximate number of hours up to the supplied date. It does this by multiplying the year value by the length of a year in hours, adding the month number times the length of a month in hours and so on. We don't mind that this is not an accurate number of hours up to this time, all we want is to be sure that each hour has a different number, because we know that this will lead to a different (but repeatable) sequence of random numbers.

```
// get the date  
let date = new Date();  
  
// get the absolute hour for this date
```

```

let absoluteHour = getAbsoluteHour(date);

// Use the absolute hour to setup the random number generator
let randSettings = {
    startValue: absoluteHour,
    randMult:8121,
    randAdd:28413,
    randModulus:134456789
}

setupRand(randSettings);

```

The code above starts a game running. It gets the date, and then the absolute hour. The absolute hour value is then used as the start value for the random number generator. We will now get a sequence of random numbers specific to the hour that the program runs.

```

let colorStyles;
let cheeseList;
let noOfCheeses;

function setupGame() {
    // set up the initial positions for the game elements
    colorStyles = ["white", "red", "orange", "yellow", "yellowGreen", "lightGreen",
        "cyan", "lightBlue", "blue", "purple", "magenta", "darkGray"];
    shuffle(colorStyles);
    cheeseList = [];
    // build the grid
    for (let y = 0; y < gridHeight; y++) {
        for (let x = 0; x < gridWidth; x++) {
            let square = { x: x, y: y };
            cheeseList.push(square);
        }
    }
    shuffle(cheeseList);
    noOfCheeses = getRandom(2,6);
}

```

The `setupGame` function runs in the server and builds the `colorStyle` list and the `cheeseList` list for the game. It then shuffles the `colorStyles` and then shuffles the `cheeseList` and sets the number of cheeses to be used in the game. It is based on the original setup code for the browser-based

version of Cheese Finder.

Before the server processes an incoming request from a server it gets the date and time and uses it to set up the random number generator before processing the request. This means that the game updates with a new map every hour. You can find this version in the folder **Ch06-06_Time_Synchronized_Cheese_Finder** in the sample code for this chapter. If you repeatedly play the game, you will notice that the grid changes every hour.

Programmer's Point

Pseudo-random technology is very powerful

We can use the magic of pseudo-random sequences in all kinds of ways. Rather than spending time designing a game world by hand we could create a landscape using pseudo-random values to decide where the trees, rivers and mountains go. Lots of games do this. We could send people secret files that can only be decoded by the right pseudo-random sequence. We would just need to give them the keys (the seed values) to decode them. This technology is the basis of secure communications on the internet. When you visit a secure website your browser and the server exchange “keys” which are used in a pseudo-random based encryption process to keep transmitted data secure from eavesdroppers.

One word of warning however. It took my computer just a few seconds to create 33 million values and discover how regularly our random number sequence repeats. This is an important first step in “cracking” the sequence and working out what the seed values are. If you do want to use random numbers in your encryption, make sure that you use the versions provided by JavaScript which are cryptographically secure. These are slower but the sequences that they produce are much harder to “crack”.

Use worldwide time

We can now create a shared experience of the Cheese Finder game which updates the cheese position every hour. The browser will “time out” games that are not completed within the hour that the games were started, and the server uses “pseudo-random” numbers based on the hour value of the time to set the cheese position and distance colors for each game. However, we do have one more problem to deal with and that is the issue of worldwide time.

The current version of the game uses the local time to generate the starting value of the pseudo-random sequence of values that describe the board. This means that if there are multiple versions of the server around the world they will give the players an experience based on their local date and time. This means that I might not have the same shared experience as someone using a different server from mine. I can't talk to a friend in a distant land about how hard the current grid is, because if their local server time is different from mine they won't be using the same game layout.

This is happening because the JavaScript `Date` object provides your program with date information that is correct in your present location. This is normally what you want. But sometimes you write code which needs to use a time which is independent of any location adjustments. UTC (or Coordinated Universal Time) is the time from which all other times in JavaScript are derived. The `Date` object in your machine applies an offset to the UTC time which is determined by the location of the computer running the program.

```
function getAbsoluteHour(date){  
    let result = (date.getUTCFullYear() * 365 * 24) +  
        (date.getUTCMonth() * 31 * 24) +  
        (date.getUTCDate() * 24) +  
        date.getUTCHours();  
    return result;  
}
```

This version of `getAbsoluteHour` uses UTC versions of all the functions that get time values from a `Date` object. It is used in the version of Mine finder in the folder **Ch06-07_World_Synchronized_Cheese_Finder** in the sample folders for this chapter.

Prepare for the cloud

We are nearly ready to take our Cheese Finder game and install it in the cloud for anyone to use. But before we do this we might want to take a look at what we have done and see if there are any more steps we can take to make the application “cloud ready”

Optimize performance

Before we place our solution on a server we might want to look for ways in which we can optimize its performance. Most programs can be written without much need to consider their performance. Modern computers are so powerful that we don't have to hunt for the fastest solution when we write software. Instead, we build applications which are easy to understand and maintain. There are only two situations when I worry much about performance:

- When I (or a user) notice that something seems to be running a bit slowly.
- When I'm paying for the computer time being used.

Cloud providers will give you an amount of free hosting, but at some point you may have to part with some cash for the computer time you use. Any free provision that we can get will have a

limit on use, so it might be a good idea to look at our code and see if we can improve efficiency.

In chapter 5 we talked about “kludges” and how it can be a bad idea to modify an existing solution to a problem if the problem changes. We saw this in the context of adding more cheeses to the Cheese Finder game, but it looks like we have created something rather similar here. The first version of Cheese Finder was written to work in a browser. The player selects a square by clicking on a button. This triggers code that uses the x and y position of the square to calculate the distance to the nearest cheese, selects the style color for that distance and sets that style on the button. This is a good way to structure the code for a single user where a given square was only ever clicked once by the player. But the server is used differently. It will receive lots of requests about the same square. It is wasteful of computer resources to calculate the distance values every we get a request for that square. What the program should do is create a something which can be used to quickly look up a distance and return it to the browser without having to do any calculations at all.

Create a cache

A cache is a copy of data that is made to speed up execution of a program. Hardware caches are used in your computer to speed up access to computer memory and mass storage devices. They are small pieces of very fast memory that hold values the computer is working on. A software cache does something similar. It holds a value to save it being recalculated each time it is needed. When the server sets up a new game it could set the position of the cheeses and then fill a grid with the style values for each x and y coordinate pair. The grid would serve as a cache of the style values. We would feed x and y values into the grid and get back the style for that grid.

Build a two-dimensional array in JavaScript

We have seen how we can use arrays as lookup tables. We already use a lookup table to convert distances into style names. The style lookup is “one-dimensional”. We have a single value that we want to convert into the name of the style to be used for that distance. The lookup table for the grid will have to be two dimensional, converting an x and y coordinate pair into the style value for that square. Some programming languages have support for arrays that have multiple dimensions, but JavaScript does not. We can however create multi-dimensional arrays by creating “arrays of arrays”.

```
// build the grid and cheese list
let grid = [];
let cheeseList = [];
for (let x = 0; x < req.width; x++) {
    let column = [];
    for (let y = 0; y < req.height; y++) {
        let square = { x: x, y: y, style: "empty" };
        // put the square into the cheese list
    }
}
```

```

cheeseList.push(square);
// put the square into the column
column.push(square);
}
// put the column into the grid
grid.push(column);
}

```

The code above creates the grid by using a nested pair of for loops. The outer loop takes us through all the x values in the grid and creates a column array for each x value. The column is filled with all the squares in that column by a loop that goes through all the y values. The column is then added to the grid array. Each square contains three properties, the x position of the square, the y position of the square and the style string for that square.

Each square is also added to a one-dimensional array called `cheeseList`, which is a linear array of squares. We need one-dimensional array of square locations that we can shuffle to create a list of cheese positions. When this code has completed we can access individual squares in the grid by using to index values:

```
grid [9][0].style = "cheese";
```

The above statement would set the style property of the square at the top square right-hand square to “cheese”. The square at the “origin” (i.e. the one with coordinates 0,0) is at the top left of the map. Once we have created our grid, the next thing to do is populate it with style values.

```

shuffle(cheeseList);
noOfCheeses = getRandom(req.minCheeses, req.maxCheeses);
// set the styles for these cheese positions
for (let x = 0; x < req.width; x++) {
    for (let y = 0; y < req.height; y++) {
        grid[x][y].style = getStyle(x, y);
    }
}

```

This is the code that creates the “cache” of style strings. The cheese list is shuffled and then the number of cheeses is determined. Then a second pair of nested for – loops goes through all the grid locations and sets the style string for each. Now, rather than calling `getStyle` to get the style

of a square, the program can just look up the style in the grid.

```
let styleText = game.grid[x][y].style;
```

CODE ANALYSIS

Creating caches

You may have some questions about how the caches are created.

Question: How can the same square object be in two different lists at the same time?

```
let square = { x: x, y: y, style: "empty" };
// put the square into the column
cheeseList.push(square);
// put the square into the cheese list
column.push(square);
```

The code above creates a square object and then adds that object to two lists. This works perfectly because the lists hold references to the object. The square object is not “in” either list. The lists just contain references to objects, not the objects themselves.

You could use the same technique if you wanted to have a list of objects ordered in more than one way, for example a list of books. You might want to work through these in order or author name or in order of book title. You could create two lists of references, one ordered by author and the other ordered by title. Each book object would exist once but it would appear in both lists.

Question: What happens to the old grid when we create a new one?

The code above creates square and grid objects as it builds the cache. This code will run every hour, which means that every hour a new grid is created. What happens to the old grid? Does it remain in memory somewhere taking up space forever? It turns out that we don’t have to worry about this. When the new grid is created the old one is no longer accessible by the program, because the `grid` variable now refers to the new grid, not the old one. JavaScript contains a process called the “garbage collector” which searches for objects which are not referred to by the program code and removes them from memory automatically.

Question: Can an array have more than two dimensions?

Yes. You can write code that creates arrays of arrays of arrays. But I’ve never had to do this when creating an application. If you find yourself thinking “what I need here is a 3D array” you should take a long hard look at how you are structuring your data.

Avoid recalculations

```
// get the date
let date = new Date();

// get the absolute hour for this date
let newabsoluteHour = getAbsoluteHour(date);

if (newabsoluteHour != absoluteHour) {

    // Set up the new game

    // update the absoluteHour value
    absoluteHour = newabsoluteHour;
}
```

The original version of the server set up the game for every request. However, it only needs to do this once in every hour. Once the game has been set up there is no need to set it up again. The code above will reduce the load on our server significantly. Now the game is only set up once an hour, not every time a request is received. It works in the same way as the code in the browser which is checks to see if the hour has changed and reloads the page if it has.

CODE ANALYSIS

Avoiding recalculations

You may have some questions about avoiding recalculations.

Question: What happens the very first time that the above program runs?

We need to force the server to set up the game the very first time that a request is received. We can do this by setting the initial value of `absoluteHour` to something which will force an update.

```
let absoluteHour = 0;
```

The value of `absoluteHour` will never be 0, so the very first time that a request is received a game setup will be triggered.

Question: What happens if a web request comes in while the server is updating the game setup?

This can never happen. The node.js system is single threaded, which means that web requests are queued up and processed one at a time.

Question: Why don't we care about performance on the browser?

As long as the application gives the user a good experience there is no need to spend time trying to make it go faster. However, we should make sure we have tested the application on low performance machines to make sure it works on them. I use a cheap laptop and a Raspberry Pi computer to test the performance of my applications. If it runs well on those it will be OK for most people.

Question: Could we improve efficiency even more?

Yes we could. The present version creates a new grid every hour. We could make the `getGame` reuse the existing grid rather than make a new one each time. There are other tricks we could use to make `getGame` even faster. However, this function is only used once an hour, so it doesn't seem worth the effort to do this.

Improve structure

This is not really going to improve performance, but it will make the program easier to manage and maintain. Whenever you write a program you need to be mindful that people other than you might have to work with it. That means that you should make the code easy to understand and try to reduce the number of ways in which the code could be inadvertently damaged. We can do this by changing the structure of our code slightly.

Put the game engine in a library file

At the moment the functions and variables that manage the gameplay are in the `server.mjs` file. We can extract these from the game and put them in a library of their own. We could create a library called `game.mjs` which exports the behaviors and variables that are needed by the game. This is a good idea because someone looking at the code now knows exactly where to look for code that manages gameplay. It also makes the application more flexible.

```
export {setupGame, grid, noOfCheeses};
```

The statement above shows the items that are exported from the engine. The `setupGame` function is called to set up the game and the `grid` and `noOfCheeses` variables give the styles for the squares and the number of cheeses in the game.

One big advantage of putting the game engine in a separate library is that it can be incorporated into other applications. We can use `game.mjs` to create a browser-based version of the Cheese

Finder game. The sample folder **Ch06-08_Optimized_Cheese_Finder** contains a browser based version of the game alongside the server based one. You can play this version on your computer by opening the [local.html](#) file in your browser. The JavaScript code that uses the [game.mjs](#) file to implement a local version of the game can be found in the file [local.mjs](#).

Use object literals in function calls

In chapter 1 when we discussed functions, we discovered that arguments to JavaScript function calls (the things you provide when the function is called) are matched by position with the parameters in the function (the things that the function works on when it runs). If you get the order of your arguments wrong when you call a function you can expect it to do strange and wonderful things which you won't want. We noted that one way to resolve any ambiguity about arguments to functions is to use object literals.

```
let gameRequest = {  
    width: gridWidth,  
    height: gridHeight,  
    colorStyles: ["white", "red", "orange", "yellow",  
        "yellowGreen", "lightGreen", "cyan",  
        "lightBlue", "blue", "purple", "magenta", "darkGray"],  
    minCheeses: 1,  
    maxCheeses: 6,  
    startValue: newAbsoluteHour,  
    randMult: 8121,  
    randAdd: 28413,  
    randModulus: 134456789  
}
```

Above you can see the definition of an object literal that contains all the information needed to create a new game. This is passed as an argument into the [getGame](#) function which builds a new game description and return it.

```
setupGame (gameRequest);
```

This statement calls the [setupGame](#) function to set up a game. The [gameRequest](#) object contains everything that the [setupGame](#) function needs to know as named items.

Buy a domain name

The domain name is the first part of the address that you enter to access a site on the web. You will be familiar with popular domains such as microsoft.com, apple.com and robmiles.com. When you create a web application hosted in the cloud the domain name will contain the name of your hosting provider. If hosted on Azure the cheesefinder game could have the url cheesefinder.azurewebsites.net. This will work fine, but it would be better if the address on the web had a bit more personality. It would be nice if the game could have the url cheesefinder.com. However, that web address turns out to be rather expensive. Fortunately, other domain names are available at very low prices. A service such as namecheap.com allows you to search for and register your own domain name. You will have to pay an annual fee for the registration and hosting of the name, but this need not be too expensive. I have managed to obtain cheesefinder.xyz domain and will be setting this as the address of the game.

Put your name on it

```
// Cheese Finder server by Rob Miles October 2022
// Version 1.0
// If you move the server to a different location you will
// have to update the base path string to reflect the new location

// Rob Miles www.robmiles.com
```

If you have made something which you are proud of you should put your name on it. Great artists always signed their paintings, so there is nothing wrong with you putting your name in your code. At the very least you should provide a way that users can find you. A good way to do this is to create an email account (and maybe even a web page) specifically for your creation.

Deploy an application

Up until now we have been using our computer to simulate the server. Both the server and the browser programs have been running on one machine. But now we want to put the server part of the game, plus all the other files it needs, into a cloud-based host. To do this we will have to make some modifications to the files so that they will run correctly in their new home. We will also need to create a file that describes the application.

package.json

You might think that programming was just about writing code and solving problems. While you do get to do a lot of this, you also have to do a fair bit of organization and management. This is because any non-trivial solution will have components that need to be managed. A very large part of the wonderfulness of JavaScript comes from the node package manager (NPM) system that we will explore in detail in the next chapter. This makes it easy to make applications out of existing elements. The package manager can also make sure that your system uses the latest (or particular) versions of components and can manage different applications configurations you might like to use for development, testing and deployment.

A key component of the package management process is a file that contains the package settings for any given application. This file has the name `package.json` and it should be placed in the folder along with the files that comprise your application. In the next chapter we'll discover tools that can create this file for you automatically, but for the Cheese Finder application we can create one by hand. The `package.json` file does not contain a list of all the files in the project. It is assumed that all the files in the project folder are part of it. In other words, we don't have to include anything mention of files such as `pseudorandom.mjs` and `index.html`.

```
{  
  "name": "cheesefinder",160  
  "version": "1.0.0",161  
  "description": "A cheese finding game",162  
  "main": "server.mjs",163  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1",164  
    "start": "node server.mjs"165  
  },  
  "engines": {166  
    "node": ">=7.6.0",  
    "npm": ">=4.1.2"  
}
```

¹⁶⁰ Name of the application

¹⁶¹ Version number

¹⁶² Description

¹⁶³ File that contains the body of our application

¹⁶⁴ No tests

¹⁶⁵ Command to start the server

¹⁶⁶ Versions of node and npm to use

```
},
"author": "RobMiles",
"license": "ISC",
"dependencies": {167},
},
"devDependencies": {},168
"repository": {169
  "type": "git",
  "url": "https://github.com/Building-Apps-and-Games-in-the-Cloud.github.io"
},
"homepage": "https://begintocodecloud.com/"170
}
```

Above you can see a [package.json](#) that describes the cheese finder application. The most important elements are the ones that specify the name of the file that is started to run the application. The application we want to run is the server application, which is in the file [server.mjs](#). If we were using features of a specific version of JavaScript or wanted to make sure that our application is not broken by an “upgraded” version, we could specify the versions to use in the “engines” property. If we put this file into the folder containing our application it will be picked up automatically by node (and other programs) when required.

Set the port for the server

The internet uses port numbers to identify program connections to external clients. A program can listen behind a particular port on a machine. When we are running servers on our local machine we can use whatever port numbers that we like. The Live Server Extension we installed Visual Studio Code in Chapter 2 (see the section “Install the Live Server Extension”) creates a web-server that listens on port 5050. When we created our own web server in Chapter 4 (see the section “Make a web server”) we created one that listens on port 8080. When we use the browser to access one of the locally hosted files we add the port number to the url (unified resource locator).

¹⁶⁷ No dependencies

¹⁶⁸ No dev dependencies

¹⁶⁹ Tell people where to find the source

¹⁷⁰ Provide a homepage link

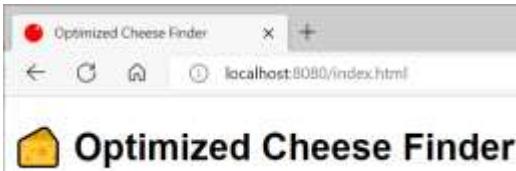


Figure 6.27 Ch06_Fig_02 Local address and port

Figure 6.2 above shows how this works. It shows that we have used our browser to open a web site hosted on our local machine by a process running a server program which is listening on port 8080. The Azure hosting that I am using for Cheese Finder has the url "[chessefinder.azureweb-sites.net](https://cheesefinder.azurewebsites.net)". This was set up when I created the application.

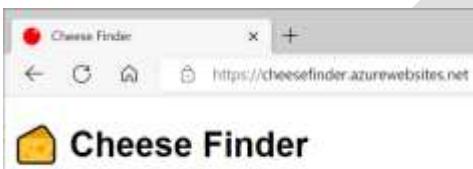


Figure 6.28 Ch06_Fig_03 Web address and port

Figure 6.3 shows this site being accessed. The browser will be using port 80 to access this server because that is the default port for HTML requests. It is important that the server we deploy into the cloud listens on port 80. Ideally, we'd like our server to listen on port 80 when it is running in the cloud and on port 8080 when it is running on the local host.

```
const port = process.env.PORT || 8080  
  
server.listen(port);
```

The two statements above make this work. The first statement creates a constant called `port`. This is set to the value of the `PORT` property of the process environment variables (which will be set by the cloud service) or the value 8080 if this `PORT` property has not been set. The `||` operator in this context means “here is a default value to use if the given value is undefined, NaN or null”. So, if there is no `PORT` property (or even any process object) the value of `port` is set to 8080. The Azure infrastructure sets an appropriate environment variable before it runs the server program. We need to add this code at the point where the server starts to listen. The environment variables are very useful, we will be doing more with them later.

Set the server path

```
let hostAddress = "http://localhost:8080/";
```

The client program (the one that runs in the browser to display the Cheese Finder game to the player) sends requests to the server for game setup information (grid size and number of cheeses) and square style (color style string for a square at a given x and y location). The variable `hostAddress` above is declared in the `client.mjs` file and gives the first part of the address for any request. This is presently set to be `localhost`. This is because while we have been testing, we have been running both the client and the server on our machine, so the address of the server is the local host running the client program. When we move our application into the cloud, we will need to change this address so that it refers to the location in the cloud where the application is hosted. When a browser connects to that site it will trigger the node.js application that implements the server.

```
let hostAddress = "https://cheesefinder.azurewebsites.net";
```

The above statement is the updated host address I need to put in the `client.mjs` file so that web requests are sent to the server in the cloud. When you move an application of your own into the cloud you will have to update the address to point to the location where your application is hosted.

Set the local file path

We have been running different versions of the Cheese Finder server from the sample program folders that we copied from GitHub in chapter 1. Each folder holds a set of files for use by that example. Each version of the server contains a file path which is set to refer to the files that it uses. This path is held in `server.mjs` in a variable called `basePath`. This path is needed because of the way files are located when Visual Studio Code runs a program in the local debugger.

```
const basePath = "./code/Ch06-Shared_Experiences/Ch06-08_Optimized_Cheese_Finder/";
```

Above you can see `basePath` value for one of the example applications. The string in `basePath` is added to the path of the file that has been requested so that the correct files are loaded. When the program is running in the cloud we must change this path to direct the server to the local

folder.

```
const basePath = "./";
```

This will cause the server to look in the local folder for the files, which is what we want. When we make larger applications with lots of resources we will put them in separate folders to make them easier to manage.

Make Something Happen

Create an Azure App service

This is a very exciting “Make Something Happen”. This is where you put something into the cloud for others to find and use. The code you want to deploy to the cloud should be held in a GitHub repository on your machine. You can test the application by using the local settings as we have been doing, and then make the modifications described above to make it ready for deployment.

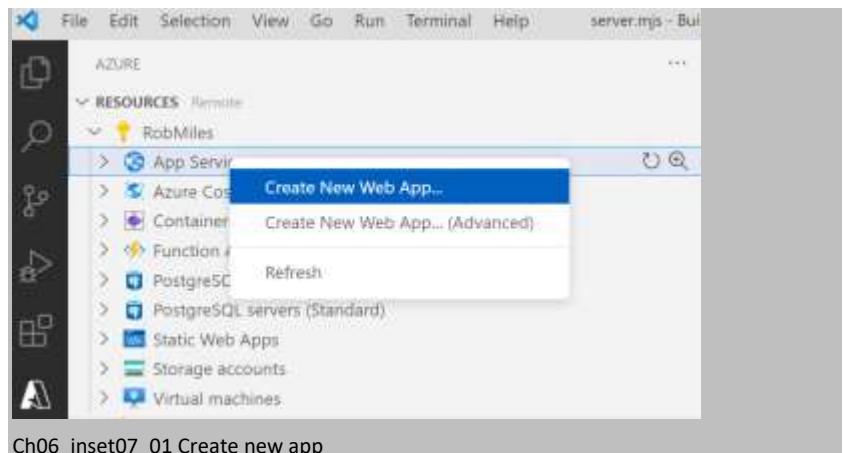
If you don’t have an application to deploy and just want to practice the deployment process you can find a deployable version of Cheese Finder in the folder **Ch06-09_Azure_Deployment** in the sample code for this chapter.

Sign up for Azure at <https://azure.microsoft.com/>. You will need a Microsoft Account to do this.

Open Visual Studio Code and install the Azure App Service extension. Follow the process we used to install Live Server in chapter 2, but search for **Azure App Service**. When the extension starts you will be asked to sign in to your Azure Account. Do this.

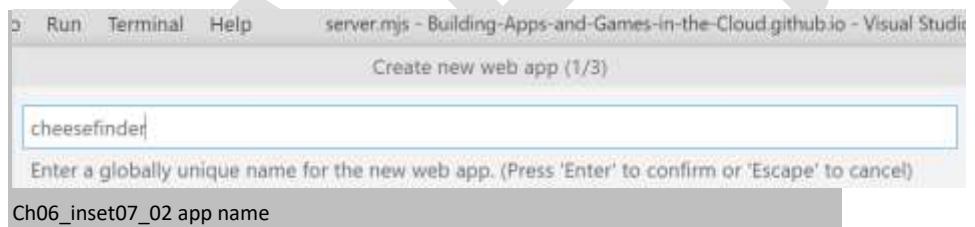
Your application should be in a GitHub repository and you should have this repository open before you begin deployment. If you want to use the sample project, just create an empty repository and then copy all the source files from the example folder into that repository.

Open the Azure extension and click on the arrow next to the Resources item and then open the pulldown for your account. Right click the App Service item. Select **Create New Web App..** from the menu that appears.



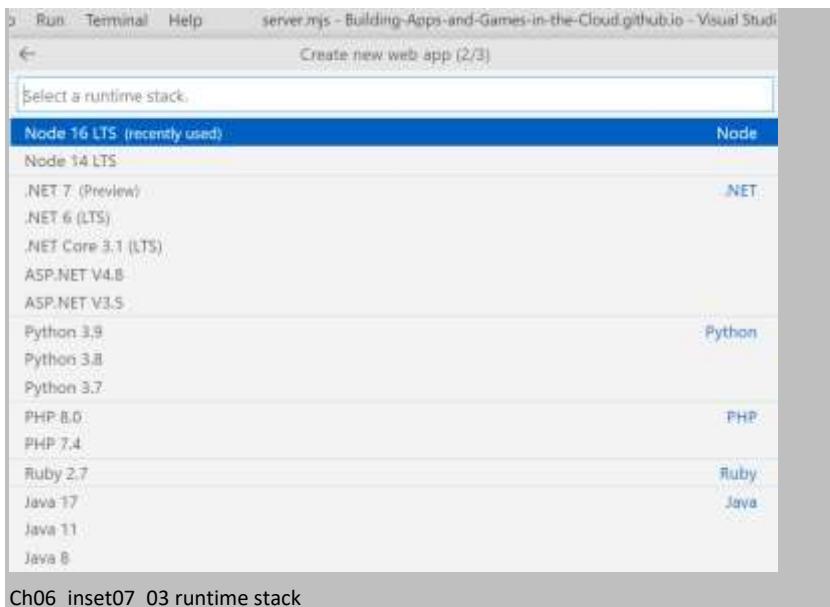
Ch06_inset07_01 Create new app

You will now be asked for the name of the app you are creating. This is the name that will be used in the url that will identify the app on the internet. I called mine cheesefinder:

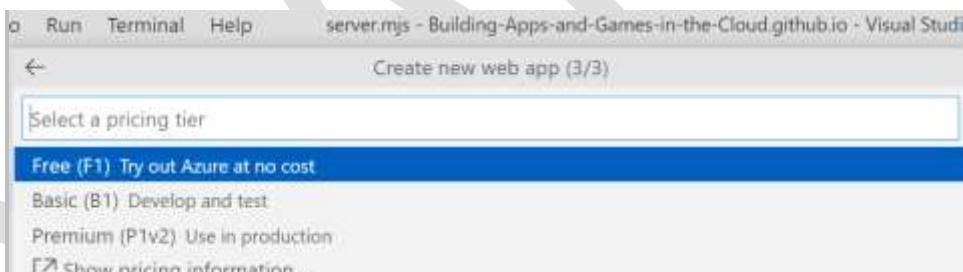


Ch06_inset07_02 app name

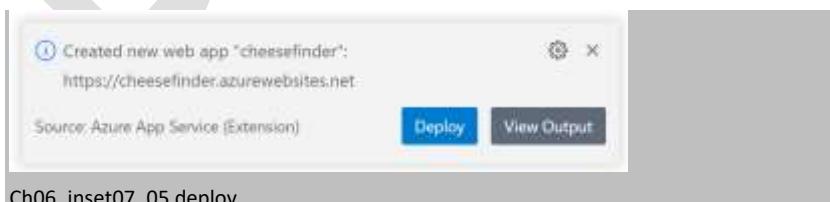
Next you need to select the runtime stack for the app. We are using Node, so select the latest version (16).



Finally



Now Azure wants to know the pricing option for your service. Select Free (F1). You will get a limited amount of service, but it will be enough to get started. The Azure extension will work for a while building your application. Then you will get the option to deploy to the applications.



Note that the dialog shows the url for the application that will be created. Press the Deploy

button to deploy your site to the cloud. Once the deployment has completed you should be able to view your application



Ch06_inset07_06 portal

Once the application has been deployed you should be able to connect to it using your browser. If you go to portal.azure.com you can open the dashboard for your new application and watch as requests come through.

Note: This is a tough “Make it Happen” and the precise steps you need to perform to complete it may change over time. Check the book website for updated instructions. And don’t be afraid to let us know if the steps you can see printed above go out of date.

What you have learned

This chapter has been super-busy. We have learned lots of things. We’ve done some programming, some debugging, some optimization and some deployment. And we’ve learned a lot about how random numbers run the net. Here’s the recap plus some points to ponder.

- We can create applications that give a shared experience to all those who access them. The site can give a different experience to visitors by using the date and time to determine what is served by the site.
- If the content served by a site can change over time an application must have a means of determining when the content being viewed in the browser is out of date. This can be achieved using timestamps that are sent along with content. The browser can check these against the current time and act when content is found to be out of date.
- When debugging a program it is possible to set a breakpoint while the program itself is running. This can be done within the debugger which is part

of Visual Studio Code and with the debugger built into the browser Developer Tools.

- When debugging code it is useful to be able to change the contents of variables in a program to test theories about the location of faults in the program.
- Faults in a program can arise when an existing design is modified to work in a different context. A new context may give rise to fault conditions or considerations which the original program was not designed to handle.
- A computer cannot generate a truly random value, but it can perform a combination of multiplication, addition and modulus to create a “new” random number from an existing one. Repeating this process on the “new” number generates a “pseudo-random” stream of values. Different initial values (or seed values) will result in different sequences of values, but each sequence is repeatable with the same seeds.
- Pseudo-random number sequences can be used in games to generate realistic looking artefacts and behaviors. Their repeatability means that they can also be used for encryption and decryption. The JavaScript libraries include functions that can be used to create “encryption quality” random number streams which have successive values which cannot be easily analyzed and predicted.
- Pseudo-random numbers can be combined with time inputs to create random sequence that are unique to a particular time.
- Under normal circumstances it is not necessary to optimize the performance of your programs. However, if code is going to be called repeatedly (as it would be in a web server responding to requests from clients) and you are paying for the computer time to run the program (as you do in a web server running on a cloud service provider) you should consider some optimization.
- One way to optimize a server is to create a pre-calculated cache of values which can be sent back in response to requests.
- JavaScript does not provide support for two-dimensional arrays. However, these can be created in the form of an “array of arrays”.
- Placing components of an application into library files makes the code clearer and raises the possibility of code reuse.
- Applications built to run within node.js can be given a package.json file which describe the application, how to run different application configuration and any dependencies that the application may have on both run-

time systems and software libraries that the application uses.

- Visual Studio Code provides an Azure App Service plugin which can be used to deploy into the cloud the contents of a node.js application described by a package.json file.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions".

What do we mean by a “shared experience”?

Before computers and the internet a shared experience would be provided by a football match or a “must watch” TV show. Today a community can all get the same experience at the same time by visiting a website which provides changing content that they all interact with.

How can we make a client that automatically determines when the content it is displaying is out of date?

In the current version of Cheese Finder the browser checks to see if the game is out of date (i.e. the server has moved into the next hour) when the player clicks a square. The response from the server includes an hour value which the browser can check. One of the defining characteristics of the World Wide Web is that it is not possible for the server to spontaneously send a message to the browser. So there is no way that the server could tell a particular browser that the game had expired.

The only way a browser could detect that the content is out of date is by repeatedly asking the server “is the content still in date?”. When we wrote the clock program, we saw that it is possible to cause trigger events at regular intervals in a JavaScript program running inside the browser. We used the events to update a clock display, but they could be used to trigger a request to the server to check the hour value. We could use this to add a feature to Cheese Finder where the browser automatically detects when a game has timed out, rather than having to wait for the player to click a square and trigger an action on the server.

Could a computer learn how to play Cheese Finder by watching players of the game?

This is a very interesting question. To answer it we must consider what information goes into the server. The server receives requests when users click buttons on the game grid. The browser sends the x and y values of a particular square and the server sends back the color style for that square. The server doesn't know which player each request is from, so it can't work out what any given player is doing. However, it can determine which are the most “popular” squares on the grid. It is possible that the server might be able to combine the knowledge of square popularity and the actual cheese positions to get some broad ideas of strategy, but it would not be possible to learn more than that. However, if we changed the game so that the browser sent the server a user ID with each request that would make a huge difference. The server could then observe individual players and form strategies based

on what they do.

How do we use pseudo-random numbers to encode data?

The key to this is the exclusive-or logical operator. You have probably heard of the **and** logical operator (output a true if both inputs are true) and the **or** operator (output a true if either or both inputs are true). The **exclusive-or** operator outputs a true if either input is true but not if both inputs are true). You could say that exclusive-or outputs a true if the inputs are different.

The interesting thing about exclusive or is that if you apply the operation twice you will get back to the value you started with. What do I mean by this? Let's look at some JavaScript. The JavaScript operator to perform exclusive or is `^`. If we apply this between two values we will get a result which is the exclusive or of all the bits in the values.

```
99^45  
78
```

If we exclusive or the value 99 with the value 45 we get the value 78. The exclusive or operation combines the bits used to store the values 99 and 45 using the exclusive or operation and generates 78 a result.

```
78^45  
99
```

The interesting thing about exclusive or is that if I now exclusive or the value 78 with 45 I get back to the value 99 again. Now, consider that 99 is the number I want to keep secret, and 45 is my encryption key. I can send the value 78 as a public message because only someone who has the key can convert it back into the actual value. What I really would like is a repeatable stream of numbers I can use to encrypt successive values in a block of data, and that is what pseudo random numbers give me. I can send you a block of data over a public channel and I only need to use a super secure connection to send you the seed values for the pseudo-random stream that will be used to decrypt it.

How do we make pseudo-random numbers harder to “crack”?

We have seen that we can use multiplication, addition and modulus to create a new pseudo-random value based on the previous one. However, someone with a lot of computer power could look at the sequence of values and work out the seed values by trying lots of combinations.

To counter this we could generate two streams and then combine successive value in some way to produce the output. This would require more computing effort to create the random sequence, and more complex seed values, but it would make it much harder to crack. JavaScript provides the `random` function that generates “general purpose” random numbers and it provides a `crypto` version which provides sequences which are harder to “crack”.

Do pseudo random numbers have anything to do with crypto currencies?

Crypto currencies assign value to the process of solving mathematical puzzles based around pseudo-random sequences. For example, I could give you a sequence of 10000 numbers and ask you to work out the random number seed values that were used to create it. When you find the seeds, we could give that solution some value in our economy, and move on to the next problem to calculate ourselves some more money. The “puzzles” set are much more complex than simple pseudo-random sequences but the underlying principle is the same.

How do you get “proper” random numbers?

We now know that most random numbers generated by a computer are made from a stream of pseudo-random values. But the Math.random function provided by JavaScript seems to be random each time we use it. How does this work. The Math.random function uses something that is “random enough” as a starting value. This might be the time in microseconds (a value that will be different each time the program runs). For “professional” quality randomness you have to use additional hardware, perhaps a component that produces a “noisy” electrical signal that can be measured to get a truly random value.

When should you use a cache?

The aim of a cache is to reduce the amount of time a program spends calculating values that it uses a lot as it runs. When deciding when to add a cache you have to consider how often the value is recalculated, how much memory it would take to store the cache, and how much time it would take to get values from the cache. You should also consider the cost of your time needed to write the extra code that will perform the caching.

What does the default operator do?

The default operator (||) allows us to specify a value which is to be used if a given value is not valid. We used it to select a port value for our server.

```
const port = process.env.PORT || 8080
```

Node.js provides a `process` object which provides information about the currently running process to our application. The `process` object contains an `env` property which is a list of environment variable which are used to send setting information into our program. The external environment can set an “environment variable” if it wants to tell our program something. The external environment might want to tell our server the network port on which the server is to run. It can do this by adding a `PORT` environment variable. On the other hand, the external environment might not add a `PORT` value, in which case the server will need have a default to value to use.

The `||` operator makes it easy to create a default value. You could also use it in a function if you want to assign default values to function parameters which have not been provided or are invalid. Default values can be useful, but you need to make sure that a default value is

appropriate for every kind of invalid input.

Chapter 7:

Design an

application

What you will learn

We can now create applications that use the browser and own server to deliver a user experience. The browser downloads HTML pages from the server and runs JavaScript programs in those pages which interact with services that the server provides. Our first versions of the Cheese Finder game ran entirely in the browser. This made their operation vulnerable as it is possible to view the execution of a JavaScript program on the browser using the Developer Tools. We improved the security of Cheese Finder by moving some functions into the server and creating a protocol of requests and responses that allow code in the browser to display the game to the player. We've seen that part of the process of developing an application that works in this way involves designing the interactions between server and browser and deciding where tasks should be performed.

In this chapter we are going to design an application from the ground up. We are going to start with an idea, make sure the idea is ethically sound and then create the workflow that will be followed when running the application. Then we will move on to create the underlying data structures that the application will need as it runs. This will provide the perfect platform for building

the application itself, which we will do in the next chapter. On the way we will learn another JavaScript hero, the class and discover how to add style to our HTML pages.

As we move into content which we are explaining as we go the need for the Glossary might be dropping. But don't forget that it is always there.

The Tiny Survey application

We are going to create a voting application called "Tiny Survey". When a bunch of people decide to do something the first problem they have is deciding what to do. The decisions could range from pizza toppings, movies to see or perhaps even the person to marry. We can help this process along by creating an application to manage a voting process. We will use the application to pick the best of several possible options. Anyone can create a tiny survey for a particular topic on our web application, and then people can vote for the option that they want. Once someone has voted they get to see the state of the votes and after the last person has had their vote you can then use the result to decide what to do. Or perhaps to decide to have another vote about something else.

Ethics, Privacy and Security

Now that we have decided what tiny survey will do, we should consider whether we should do it and the implications of building it. This is not necessarily a programming consideration, but I think it is a necessary one. Just because something can be built does not necessarily mean that it is a good idea to build it. We need to consider the ethical aspects (can this application be used to make people unhappy?) the privacy aspects (can this application be used to violate the privacy of its users?) and security (can this application be provided in a secure way?). Let's take each of these in turn.

Ethics

Ethics is all about "right" or "wrong". You might think that a programming book is a strange place to be having a discussion about what is right or wrong, but I think this is a worthwhile exercise. Companies take ethical stances on what they do and will examine their products and business process from this perspective.

The tiny survey application does not appear to raise any ethical issues. It is simply a way that a group of people can agree on a preferred option. It doesn't encourage bad behavior or steer the users in any particular direction. It could be used for a group of people to decide the most appalling insult for a rival football team, but they could use piece of paper or email for that too.

However, we might consider adding an "Option Recommendation" feature that offers survey options based on the first three that were entered when a survey is being built. The feature would

look through all the submitted surveys and find matches. This would mean that once I had entered the first three types of pizza for my robspizza survey the program would suggest “vegetable supreme” as one of the options because other lists containing pizza names contained this one as well. I might find this useful (I’d forgotten about that type of pizza) but is it ethical?

This is an interesting question. It might be useful, but it could also be dangerous, particularly if the feature ranks options and recommends the most popular ones. Now the options take on a life of their own. Nasty people could “game” the application by creating lots of lists containing options that they want other people to see. At this point I don’t think we have an ethical application anymore.

I might be over thinking things a bit here – I am a programmer after all – but I think an “ethical check” of applications you are about to build is a good idea. And, like other risks to a project that should be monitored, you should regularly review a project during development to check if any new ethical issues have arisen.

Privacy

At this point I’m not considering issues around the dangers of people stealing data from our application, that is for the security section. Instead, I’m thinking about how someone providing the application could compromise the privacy of the users. What do I mean by this? Well, after a couple of uses the tiny survey system could end up knowing my five favorite pizzas and five favorite movies. The application could track its users so that the application knows who created a survey and whether a given person has responded to a particular survey. This tracking would need to use the local storage of my browser which means that the survey can only work out that my browser likes pepperoni pizza and Clueless the movie, not that it is Rob Miles that has those preferences. And if I clear out my browser local storage or use a multiple browsers I can’t be tracked like this. However, gathering information in this way could be very useful. Next time I make a tiny survey it might pop up a recommendation for a movie or a pizza topping that other people with my preferences also liked. Or it could find a matching video game and tell me about that. The fact that lots of people who like pepperoni pizza also like Clueless may also have value in itself. Once some data has been collected it can be used in many ways.

Someone using tiny survey has no way of knowing whether the information they enter is being used in these ways, they just have to trust that the application is respecting their privacy. There are data protection codes of practice that require an application to ask the user for informed consent if data is to be used for anything other than the purpose than it was entered. In our case tiny survey is not going to use responses anywhere else so there is no need to ask for consent. However, you might want to think about these issues next time you fill in a survey on your favorite social media platform. They already have your consent to do anything they like with your surveys and the responses they receive.

Security

The “tiny survey” application doesn’t take any steps to keep the responses in a survey private.

Once someone has the name of a survey they can access it, pick favorites and see the current scores. This might be a problem if a group of people use the application to pick the best password, but anyone who does that probably deserves all the problems they get.

However, when considering security you also have to consider the vulnerability of the system to attack. The application will expose endpoints that are used to enter lists of responses, vote on options and read back results. These endpoints are used by the application in the browser as it runs. But what if someone creates a program that talks directly to these endpoints? The program could use the endpoints to create surveys, search for surveys by repeatedly asking for different survey names and then randomly vote on them. It could create spurious surveys containing skewed data to try and convince people that ham and pineapple is the best pizza topping by giving it thousands of votes.

These problems could be addressed by requiring users to login before they could use the application. However, this would make it harder to use and raise a whole new set of problems in respect of user management. In the next chapter we will discover how to implement logins on an application.

Happy Endings

I've always considered programming as "The science of the happy ending". If you do everything right, you end up with a happy user running a program that does what they want and keeps them safe. Considering ethics, privacy and security before you create any of an application you are about to build is a very good idea to get you to that happy place. Now we can move on to consider how our application is going to work

Application workflow

The absolute worst thing we could do right now is start writing JavaScript. Before we write any code we must decide how the application is used. We have a text description of what Tiny Survey is going to do. But we need more detail. We need to create a *workflow* that sets out the steps that will be performed when a survey is created. We use workflows all the time. When we follow a recipe to bake a cake we are implementing a workflow. Workflows have a particular sequence (we need to turn the oven on before we put the cake in) and conditional elements (we need to have all the ingredients for our cake before we start). A good way to design a workflow for the tiny survey application is to make a prototype version. This will have all the pages that will be present in the finished version, but the pages will not be functional, they will just set out what the user will see when using the application. We can work through each page in turn in the same order that they would be used; starting with the index page.

Index page

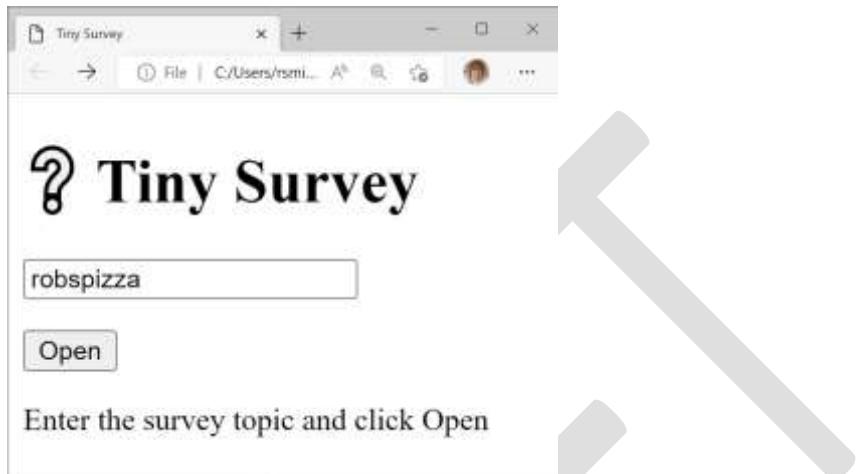


Figure 7.29 Ch-07_Fig_01 Tiny_Survey_start_page

Figure 7.1 above shows a prototype index page for the tiny survey application. The user can either enter the topic of an existing survey (to select their preferred option) or the name of a survey that doesn't exist (to create a new survey). Let's follow the workflow to make a new survey first. I'm buying everyone pizza and so I want to know what toppings people prefer. I'm going to create a survey topic of [robspizza](#). When I click **Open** the page will change to one where I can enter the toppings I want people to choose between.

```
<!DOCTYPE html>
<html>

<head>
  <title>Tiny Survey</title>
</head>

<body>
  <h1>? Tiny Survey</h1>171
  <p>
    <input type="text" spellcheck="false" value="robspizza">
  </p>
  <p>
```

¹⁷¹ Question emoji in the title

```
<button onclick="doEnterOptions();">Open</button>
</p>
<p>Enter the survey topic and click Open</p>

<script>
    function doEnterOptions() {172
        window.open("enteroptions.html", "_self");
    }
</script>
</body>

</html>
```

Above you can see the HTML for the index page. The content is very simple with a question mark emoji as the application logo. At the bottom of the page you can see the function `doEnterOptions` which is called when the Open button is clicked. This uses the `window.open` function to open a page called `enteroptions.html`. Let's look at that page next.

¹⁷² Open button event handler

Enter options

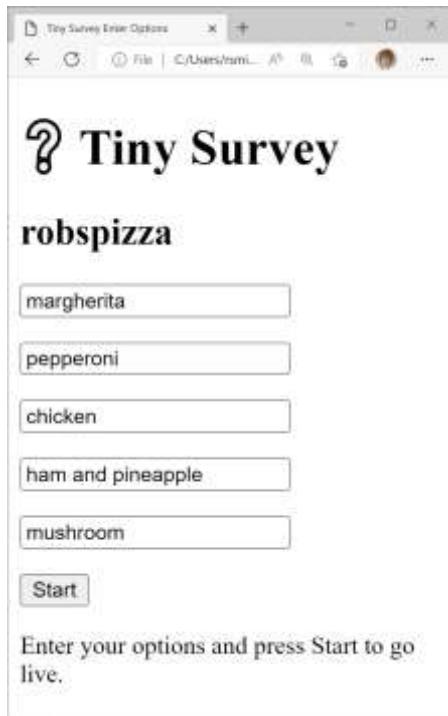


Figure 7.30 Ch-07_Fig_02 Tiny_Survey_enter_options

The [enteroptions.html](#) page displays five inputs that are used to set the options for the survey. The demonstration survey is all about pizza toppings. Figure 7.2 above shows that I've typed in the name of the toppings that I want everyone to choose from. When I click **Start** the survey will go live and offer me a chance to vote.

```
<!DOCTYPE html>
<html>

<head>
  <title>Tiny Survey Enter Options</title>173
</head>

<body>
```

¹⁷³ Enter options title

```

<h1>&#10068; Tiny Survey</h1>
<h2>robspizza</h2>174
<p><input value="margherita"></p>175
<p><input value="pepperoni"></p>
<p><input value="chicken"></p>
<p><input value="ham and pineapple"></p>
<p><input value="mushroom"></p>
<p>
    <button onclick="doStartSurvey()">Start</button>
</p>
<p>
    Enter your options and press Start to go live.
</p>

<script>
    function doStartSurvey () {176
        window.open("seleoption.html", "_self");
    }
</script>
</body>

</html>

```

Above is the content of the [enteroptions.html](#) page. This contains five HTML input elements which have been pre-set with the demonstration pizza topping names. We will add the working HTML content once we have decided we are OK with the workflow through the application. When the Start button is clicked a page called [seleoption.html](#) is displayed. Let's look at that.

¹⁷⁴ Survey topic

¹⁷⁵ Pre-set inputs

¹⁷⁶ Start button event handler

Select option

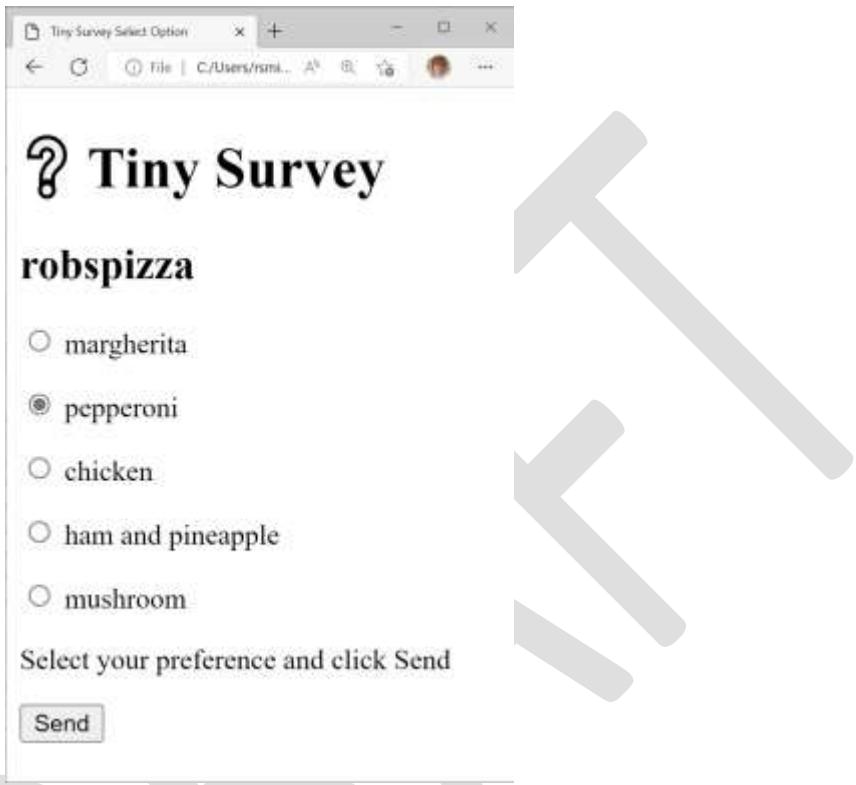


Figure 7.31 Ch-07_Fig_03 Selection_page

Figure 7.3 above shows the selection page. The `selectoption.html` page holds five “radio button” inputs. They are grouped so that only one can be selected at any time. Users can click on the button to select the option they want. I’m a big fan of pepperoni. When I click the **Send** button my selection will be saved and the count for that option updated.

```
<!DOCTYPE html>
<html>

<head>
    <title>Tiny Survey Select Option</title>177
</head>
```

¹⁷⁷ Select option title

```
<body>
  <h1>&#10068; Tiny Survey</h1>
  <h2>robspizza</h2>
  <p>
    <input type="radio" name="selections" id="option1">178
    <label for="option1">margherita</label>179
  </p>
  <p>
    <input type="radio" name="selections" id="option2">
    <label for="option2">pepperoni</label>
  </p>
  <p>
    <input type="radio" name="selections" id="option3">
    <label for="option3">chicken</label>
  </p>
  <p>
    <input type="radio" name="selections" id="option4">
    <label for="option4">ham and pineapple</label>
  </p>
  <p>
    <input type="radio" name="selections" id="option5">
    <label for="option5">mushroom</label>
  </p>
  <p>
    Select your preference and click Send
  </p>
  <p>
    <button onclick="doSendSelection()">Send</button>
  </p>
  <script>
    function doSendSelection() {180
      window.open("displayresults.html", "_self");
    }

  </script>
</body>
```

¹⁷⁸ Radio button

¹⁷⁹ Label for the button containing the option

¹⁸⁰ Event handler for Send button

```
</html>
```

The listing above shows the [selectoption.html](#) page. It uses HTML labels which are associated with the radio buttons that are used to get the option selection. Radio buttons that have the same name attribute are part of a group. Only one button can be pressed at any time. The user of the survey will select an option and then click Send. This runs the [doSendSelection](#) function which loads the [displayresults.html](#) page.

Display results

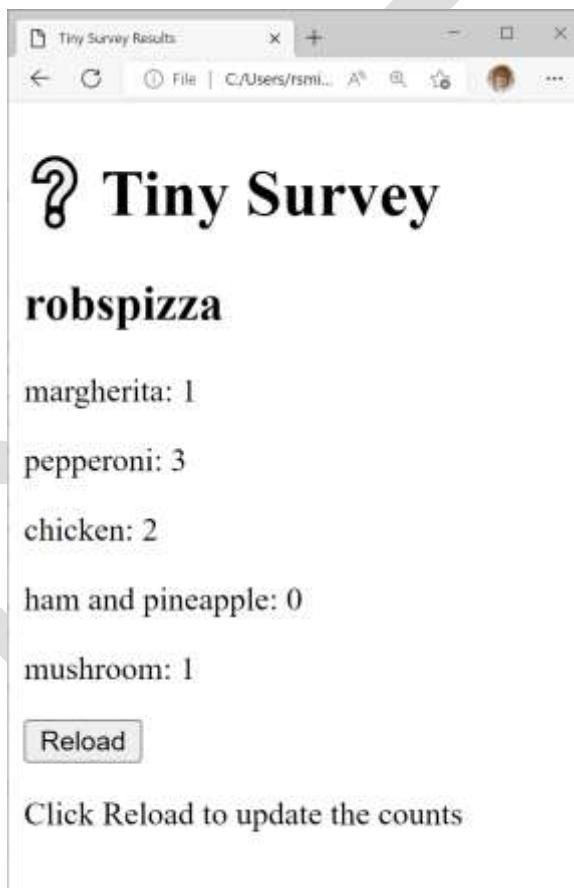


Figure 7.32 Ch-07_Fig_04 Results_page

Figure 7.4 above shows the results page. At the moment my personal favorite seems to be

getting the most votes, which is nice. The user can click Reload to reload see new count values. This version of the survey doesn't automatically update the display.

```
<!DOCTYPE html>
<html>

<head>
    <title>Tiny Survey Results</title>
</head>

<body>
    <h1>&#10068; Tiny Survey</h1>
    <h2>robspizza</h2>
    <p>margherita: 1</p>
    <p>pepperoni: 3</p>
    <p>chicken: 2 </p>
    <p>ham and pineapple: 0</p>
    <p>mushroom: 1</p>
    <p>
        <button>Reload</button>
    </p>
    <p>
        Click Reload to update the counts
    </p>
</div>
</body>

</html>
```

This is the HTML that shows the results display. The results values are all fixed strings of text. The Reload button does nothing when it is pressed.

Extra workflows

We have just worked through the workflow for creating a survey, entering the options, selecting an option and then displaying the results. The workflow will be slightly different if the user enters the name of an existing survey. In that case they will be taken directly to the Select option page, followed by the show results page.

Make Something Happen

Play with the Tiny Survey pages

Now that we have our prototype we can work through it so that we understand what we are trying to build and also make sure that there aren't any ambiguities or unknowns. Start your browser and open the `index.html` file in the **Ch07-01_Tiny_Survey_Prototype** example folder for this chapter. You can work through the screens above to examine how the application would be used.

CODE ANALYSIS

Tiny Survey

This is not really a code analysis because we haven't got any code yet. But it does seem rather complicated and you may have some questions about it.

Question: Does the prototype need to simulate opening an existing survey?

The current prototype shows the workflow for creating a new survey (called `robspizza`) which is all about pizza toppings. It doesn't show the situation where a user enters the name of a survey that someone else has created and selects an option. You could do this by creating two `index.html` files, one called `indexNew.html` and the other `indexSel.html` which show the two different workflows. Software designers talk about "user stories" where they describe a scenario for a workflow in which a user comes to the system with a particular need.

Question: Is it possible for someone to vote more than once?

Yes. This is to keep the workflow simple. In later chapters we will discover how an application running on a server can use "cookies" (small amounts of data stored by the browser) to track the actions of users and prevent this.

Question: Is it possible to view the count values without selecting an option?

If the user opens an existing survey they are taken to the option selection page for the survey, not the results page, so they have to vote to see the count values. Tiny Survey is only designed for quick "which movie shall we watch" surveys in which everyone in the room picks their option and the survey ends.

If we were making the Tiny Survey application for a customer, and they really wanted the application to allow users to view the counts without selecting an option we could allow users to click Send on the select option page without clicking any radio button or add a radio button to the select option page which is labeled "No selection". If the application used cookies (see above) it could record in the browser which surveys a user has voted for and take them straight to the results page if they select a survey in which they have already

voted.

Programmer's Point

Make good use of sample pages

Temporary pages like these are a great way to design a workflow. If you are working for a customer you can show them exactly how the application will used. Simple HTML files like the ones we have used are very quick to make, and once you have made a few you can copy and reuse them for each new application you design. Try very hard not to put any actual working code in your sample pages, just populate them with sample data. However, I think it is worth spending a few minutes picking a logo on each page. Emojis are a great source for simple “place-holder” images that you can replace with better ones later.

If you don't want to use HTML for your workflow designs you can draw out the different screens on sticky notes, put them on a whiteboard and then draw lines between them to show the workflow. If you do this, remember to take a photograph of your finished design at the end.

Application data storage

In the discussion up to now we've talked about the data elements in the Tiny Survey application in general terms. Now that we have an idea of the workflow of the application we can identify what data should be stored and how it should be structured. This is a great point to introduce a new JavaScript hero who can help with this.

JavaScript hero: classes

We know there are some data items, for example **option**, **survey** and **count** that we will need in the tiny survey application but we haven't really considered how we will structure them. Now we are going to take a first look at a JavaScript hero which makes this much easier. Many other programming languages use “classes” as a way of structuring their solutions. You may have heard the languages C# and Java being described as “class based”. This means that you can't create a program in those languages without creating classes to enclose all the components in the solution. JavaScript is not like that. We've managed to create quite a few useful applications without using any classes at all. JavaScript provides classes and supports useful class-based features such as **inheritance** but you are not forced to use them. However, the Tiny Survey application would benefit from some classes, so lets take a look at them and how they work.

Classes vs objects

The first thing we need to do is work out the difference between classes and objects. Up until now the data in our applications has been made up of individual values. There has not been much of a need to “lump” things together. And when we have need to lump things together, we have used an object literal. As an example of an object literal in action, consider what we did in chapter 1 in the Programmer’s Point “Make good use of object literals”:

```
displayPersonDetails({address:"House of Rob", name:"Rob Miles"});
```

The statement above creates an object literal as a parameter to the `displayPersonDetails` function. It allows us to send multiple data items into a function without any possibility of the items being mixed up. The function will get something which contains name and address information. The object literal that is created contains properties called `address` and `name`. The `displayPersonDetails` function gets these properties and uses them in the display it generates. Using a literal object in this way makes it less likely that the wrong information will be sent into the function.

There is a kind of “contract” between the function and the caller in that the object supplied will contain name and address properties. If one of them properties is missed out (perhaps we forgot to add the `address` property) or the name mis-spelled (perhaps we called it `Address`) the code in the function will see that property as `undefined`. It would be very useful if we could create something that defines the required properties for a person details object so that someone using it could be sure that it has all the required ones. We can do this with a class.

```
class PersonDetails {  
    constructor(name,address){  
        this.name = name;  
        this.address = address;  
    }  
}
```

The class `PersonDetails` is defined above. The important thing to note about the definition is that it doesn’t make a data store that can hold `PersonDetails` information. Instead, it tells JavaScript how to make an object which is an instance of the `PersonDetails` class. Every class has a constructor method that runs when a new instance of that class is created. The constructor creates properties that are part of the class: in this case the name and the address.

```
let rob = new PersonDetails("Rob", "Rob's house")
```

The statement above creates a new `PersonDetails` object which is referred to by the variable `rob`. This object is called an **instance** of the `PersonDetails` class. The object has `name` and `address` properties which are set by the constructor. The keyword `new` tells JavaScript to make a new instance of the class. The arguments to the call are passed into the constructor function for the class.

Make Something Happen

Investigate classes

Let's take a look at how we can create instances of classes and work with them. Start your browser and open the `index.html` file in the **Ch07-02_Classes_Investigation** example folder for this chapter. Open the browser Developer Tools and select the console view.

```
class PersonDetails {  
  
    constructor(newName, newAddress) {  
        this.name = newName;  
        this.address = newAddress;  
        console.log("I just made a PersonDetails object:" + this.name +  
                  " :" + this.address);  
    }  
}
```

The JavaScript code in this sample file contains a definition of a `PersonDetails` class. It is exactly the same as the definition we saw earlier, but there is an extra statement in the constructor method that logs a message to the console each time that a `PersonDetails` instance is created. Type:

```
let rob = new PersonDetails("Rob", "Rob's house")
```

and press enter:

```
> let rob = new PersonDetails("Rob", "Rob's house")  
I just made a PersonDetails object:Rob :Rob's house  
Ch07_inset04_01 create PersonDetails instance
```

When a program makes a new instance of a class the constructor method for the `PersonDetails` class gets control and displays a message. Properties are assigned to values inside the new object by using the `this` keyword. Let's take some time to consider what `this` means.

A keyword is a word in JavaScript that has special meaning. We've seen a few. The words `if` and `function` are keywords. These words have special meaning in the language. The word `this` is another keyword. Humans use this all the time. It's a shorthand that means "the thing we are talking about". My wife will occasionally pick up a T shirt I have just got out of the wardrobe and ask "You're not going to wear this?". In this context the word `this` means "this ugly shirt you insist on wearing". In this case of JavaScript the keyword `this` is a shorthand for "a reference to the object in whose context this code is running". In the constructor of a class a `this` reference refers to the object being created.

```
this.address= newAddress;
```

The statement above adds an `address` property to the object that `this` refers to (which is the object being set up by the constructor).

The confusing thing about `this` in JavaScript is that it refers to different things according to the context in which it is being used. We will discuss these contexts when we get to them, or you can read the whole story of `this` in the glossary.

When the constructor has finished it has created a new instance of the class which will contain `name` and `address` properties that were set by the constructor. Leave the console window open, you will be using it in the next section.

CODE ANALYSIS

Class Construction

You may have some questions about the class construction process.

Question: What does `new` do?

The `new` keyword starts the creation of a new instance of a class. The `new` is followed by the name of a class. When a running JavaScript program encounters `new` it then looks for the class that has been specified and then copies the arguments that follow the class name into a call of the constructor.

```
let rob = new PersonDetails("Rob","Rob's house")
```

In the case of the `new` above, JavaScript will look for the `PersonDetails` class definition, create an empty object and then call the `PersonDetails` constructor method passing it "Rob" as the first argument and "Rob's house" as the second. The constructor will then fill in any properties of that object and it will then be returned. In the case of the statement above the variable `rob` would be made to refer to the new object. If a program tries to create a new instance of a class that has not been defined the program will stop with an error.

Question: Do we ever call the constructor method in a class?

We never call the constructor ourselves. It is called when a `new` is performed.

Question: What does `this` mean again?

Rather than think about what `this` means, start by thinking about the problem it is being used to solve. The `PersonDetails` constructor is making a new instance of the class which is being given `name` and `address` properties. For the `PersonDetails` constructor to work it will need a reference to the new being created. The reference is provided in the form of a keyword called `this`.

Question: What is the difference between a function and a method?

A function is a block of code with a name that exists outside of any object. We've created quite a few functions as we've been learning JavaScript. A method is a block of code with a name that is declared inside an object and allows the object to perform behaviors. At the moment the only method we've created has had the special name "constructor" and runs when an instance of the object is created. We can add other methods to a class, we will be doing this later in the text.

Question: What is the difference between a class and an object?

The class is the recipe. The object is the cake. In other words, the class provides the instructions that tell JavaScript how to make the object, and the object is made from the class.

Class Advantages

At the moment you might be wondering why you would use a class. You can put an object literal directly into your code any time you want one. Whereas with a class you must make the class definition and then write the constructor function. One answer is that using classes allows you to make your programs more reliable. Code in a constructor can validate incoming properties and only create an instance if they are correct.

Make Something Happen

Valid objects

You should already be in the `index.html` file in the **Ch07-02_Classes_**Investigation example folder. If not, browse to this file, open it with the browser and then open the console the Developer Tools view. Now type in the following:

```
let x = new PersonDetails(1/"fred")
```

This is completely legal JavaScript which will run perfectly successfully. It is also a very bad thing to do. The result of dividing a number by a string is the JavaScript value `NaN`. The result of leaving off an argument to a function call (the above construction of `PersonDetails` does

not have an address value) is a parameter set to `undefined` when the function runs. So we are asking the `PersonDetails` constructor to create an object which is a `PersonDetails` instance with a name of `NaN` and an address of `undefined`. Press enter to run the statement:

```
> let x = new PersonDetails(1/"fred")
I just made a PersonDetails object:NaN :undefined
```

Ch07_inset06_01 create a broken PersonDetails instance.png

The constructor has made a `PersonDetails` object with a name of `NaN` and an address of `undefined`. If the program subsequently uses this object it will produce spurious results. If you've ever seen a web page display the messages "NaN" or "undefined" you might understand why this happens now.

What we would like is a way that we can ensure that a `PersonDetails` object always has valid content. It turns out that I've done this. I've made a class that has taken a course in self defence. It won't let you create an invalid person. I've called it `ValidPersonDetails`. Let's see if we can break that. Type in the following to test it:

```
let y = new ValidPersonDetails(1/"fred")
```

This is trying to make a new `ValidPersonDetails` instance using the same invalid arguments as before: the name is not a number and the address is undefined. Press Enter to see what happens this time.

```
> let y = new ValidPersonDetails(1/"fred")
○ > Uncaught [2] ['Invalid name', 'Invalid address']
```

Ch07_inset06_02 ValidPersonDetails construction error

This time the program seems to have stopped. The constructor for the `ValidPersonDetails` class has thrown an **exception**. We will learn more about exceptions later in the text. An exception is an object which describes something bad that has just happened. A program can throw an exception at any point in the code. When an exception is thrown JavaScript transfers the program execution to a piece of code designed to catch the exception and handle it. In our case the exception information is an array containing the messages "Invalid name" and "Invalid address" which describes what has gone wrong. If there is no code in place to catch the exception and deal with it the program stops. We will look at exceptions later in the text (although of course they are described in the glossary). Our program will be stopped if it ever tries to create an invalid `PersonDetails` instance.

You might like to try creating more `ValidPersonDetails` objects. You will discover that it is impossible to create a `ValidPersonDetails` object with anything other than two strings for name and address.

```
> let z = new ValidPersonDetails("Rob", "Rob's House")
I just made a ValidPersonDetails object:Rob :Rob's House
```

Ch07_inset06_03 ValidPersonDetails successful construction

Above you can see proof that it is possible to create an instance of the `ValidPersonDetails` as

long as we provide the correct kind of arguments. Let's look at how this works.

Create valid objects

We've seen that a crucial difference between creating a literal object (by just creating the object in the code) and creating a new instance of a class (by using the `new` keyword) is that our code gets control during the construction process. Code in the constructor for an object can enforce rules that will ensure that the object contains valid content. Let's see how this works.

```
class ValidPersonDetails {  
  
    constructor(name, address) {  
        let error = "";181  
        if (typeof (name) == 'string') {182  
            this.name = name;183  
        }  
        else {184  
            error = "Invalid name. ";185  
        }  
        if (typeof (address) == 'string') {186  
            this.address = address;187  
        }  
        else {188  
            error = error + "Invalid address. ";  
        }  
  
        if (error != "") {189  
    }
```

¹⁸¹ Create an empty error string

¹⁸² Make sure that the name is a string

¹⁸³ If it is, create the property

¹⁸⁴ If the name isn't a string...

¹⁸⁵ ..add an error message

¹⁸⁶ Make sure the address is a string

¹⁸⁷ Set the address property if it is valid

¹⁸⁸ Add to the error string if the address is invalid

¹⁸⁹ If the error string is not empty something failed

```
        throw error;190
    }
    console.log("I just made a ValidPersonDetails object:" + name + " :" + address);
}
}
```

CODE ANALYSIS

Constructor validation

We've just seen a constructor that can make sure an object is only created with valid content. You might have some questions about how it does this.

Question: What does `typeof` do?

The `typeof` operator acts on a JavaScript variable and returns a string containing the type of that variable. We've seen that all variables have a particular type, and that type is inferred when a variable is created.

```
let age = 99;
console.log(typeof age);
```

The above pair of statements would display the message "number" on the console because the type of the variable is `number`. If you assign the value 99 (or any expression that evaluates to a number) to a variable JavaScript will set the type of variable to `number`.

```
let name = "Rob";
console.log(typeof name);
```

The following statements would output the message "string", since JavaScript has worked out that `name` contains a string. The constructor for `ValidPersonDetails` uses `typeof` to make sure that `name` and the `address` parameters are both strings.

```
let rob = new PersonDetails("Rob", "Rob's house");
console.log(typeof name);
```

The code above would output the message "PersonDetails" because the `rob` reference has been made to refer to an object of that type. This means that we could make a function that would only work on particular types of parameter. It could check the type of an incoming

¹⁹⁰ Stop the function and throw an exception

object and only accept certain types.

Question: What does the `throw` operation throw in our constructor?

As the constructor runs it creates a tiny “error report” about the construction process in the variable `error`. The variable `error` starts off as an empty array. If the constructor detects that the name is not a valid string it adds “Invalid name.” to the error array. It adds another message if the address is not valid. If everything works correctly the constructor the `error` variable will contain an empty array at the end of the construction process. If `error` contains anything else, the constructor decides that the construction process has failed and throws the `error` array as an exception description.

Question: Could the constructor enforce other rules?

Yes it could. At the moment the `ValidPersonDetails` is quite happy to create an object with an empty string for the name or the address. You might want it to reject empty strings for these items. It could even test to make sure that the name was only made up of alphabetic characters or had a minimum length, and that the address contains a valid zip code. All these tests would take place when the object was created and make it much more difficult to make an invalid person.

Question: Is there anything to stop a program assigning an invalid value to a property of an existing `ValidPersonDetails` object?

No. The construction gets control when the object is created and can stop bad things happening then, but once the object has been created there is nothing to stop changes to the properties in the object.

```
let rob = new ValidPersonDetails("Rob", "Rob's house");
rob.name = 1/"fred";
```

The above two statements would result in a `ValidPersonDetails` object with a `name` property with the value of `NaN`. Unlike some languages, JavaScript has no way of making object properties private. They are always vulnerable to modification in this way.

Question: Could we use a literal object to provide the values for a constructor?

This is a very good idea. We have already seen how we can use literal objects to reduce the chance of mistakes when calling functions. It makes sense to do this with the constructor method too:

```
let rob = new PerfectPersonDetails({address:"House of Rob",
                                    name:"Rob Miles"});
```

The statement above creates an instance of the `PerfectPersonDetails` class. The constructor for the class looks like this:

```
class PerfectPersonDetails {
```

```
constructor(newValue) {
    let error = [];
    if (typeof newValue.name == 'string') {
        this.name = newValue.name;
    }
    else {
        error.push("Invalid name");
    }
    if (typeof newValue.address == 'string') {
        this.address = newValue.address;
    }
    else {
        error.push("Invalid address");
    }
    if (error.length != 0) {
        throw error;
    }
    console.log("I just made a PerfectPersonDetails object:" +
                newValue.name + " : " + newValue.address);
}
```

The constructor has a single parameter called `newValue` which is an object containing all the setup information required to create a new `PerfectPerson`. This class is provided in the [Ch07-02_Classes_Invitation](#) example so you can create an instance yourself if you wish.

Programmer's Point

Catch errors “before they happen” with TypeScript

You might find the heading for this programmer’s point confusing. Surely, catching errors before they happen would involve using a time machine? It turns out that there are two types of error handling. The code we are adding to the `ValidPersonDetails` constructor is reacting to an error when it occurs during program execution. However, a better idea might be to have a programming language which allows you to specify the type of an element in a program and then reject code where the element is being used incorrectly before the program even runs. In part 3 of this book we will discover TypeScript which is an extension to the JavaScript language which allows us to just this.

If you want to refresh your understanding of object literals, take a look at the Programmer’s Point “Make good use of object literals” in chapter 1.

Classes for Tiny Survey

Now that we know how to use classes we can create some to hold the data items in the Tiny Survey application. There are three classes that we need, which we can call `Survey`, `Option` and `Surveys`. We work out what properties they should hold by examining the workflow we have created.

The Option class

```
class Option{  
    constructor(newValue) {  
        this.text = newValue.text;  
        this.count = newValue.count;  
    }  
}
```

We can start by creating the class for the option object. The code above declares the [Option](#) class and shows the constructor method for the class. The constructor accepts an object that contains the initial values for the [Option](#), the text of the option (for example “pepperoni”) and the count (which will usually be 0).

```
let option1 = new Option({ text: "pepperoni", count:0 });
```

The statement below creates a variable called [option1](#) which refers to a new [Option](#) instance with “pepperoni” as the option text and a count of 0.

Option methods

Up until now all the classes that we have created have only contained the constructor method. However, we can add our own methods to a class which can allow the class to do things for us. Putting methods in a class allow us to make our code slightly safer. For example, consider the effect of the code below:

```
option1.Count = option1.count + 1
```

This code was intended to increment the [count](#) value for an option. Unfortunately, it won’t work. You might like to have to look hard to find the mistake. However, if you look carefully, you will see that the identifier [count](#) has been mis-spelt. One version has [Count](#), with an upper-case C at the start. This is not going to cause the program to fail. JavaScript will just add a new property, called “Count”, to the object referred to by [option1](#) and continue happily on its way. You, however, will not be happy because the count value is not going up when it should do. A better

solution is to create a method in the `Option` class which increments the count for us. We could also add a method that returns the count value.

```
class Option{  
    constructor(text){  
        this.text = text;  
        this.count = 0;  
    }  
  
    incrementCount(){  
        this.count = this.count+1;  
    }  
  
    getCount(){  
        return this.count;  
    }  
  
    getText(){  
        return this.text;  
    }  
}
```

A program can now call the `incrementCount` method on an option to get the option to make the count value for that option 1 larger. It can also use the `getCount` function if it needs to know the value of the count. I've also added a method called `getText` which returns the text of the option.

You might think I have made a mistake by missing the word `function` of the front of the method declarations above. However, this is not the case (as if). When creating a method in a class you start with the method name, there is nothing in front of it.

Programmer's Point

Use methods to improve resilience

Using methods like `incrementCount` and `getCount` in the `Option` class make the operation of the class more secure. If someone tries to call `incrementcount` to increment the counter the program will fail at run time because `count` has been spelt incorrectly. But at least you won't have to spend ages trying to work out why a counter is not updating when it should.

Providing an `incrementCount` method rather than expecting people to increment the `counter` variable inside an `Option` object also improves security slightly. If the only thing the application is supposed to do with the `count` value is make it larger should not be possible to do anything else with it. Some programming languages can mark class members as *private* which means they can only be used in methods running inside the class. It would be wonderful if we could make the `count` data member private to the `Option` class and prevent external access to it. Unfortunately, JavaScript doesn't support this, so any code outside an `Option` object can access and change any member of the class, including `count`. This means that using methods such as `incrementCount` don't bring as much extra security as they should, but they are still worth doing because they make the code clearer. It is never a good idea to expect users of a class to directly access the properties in the class, which is why there is now a `getText` method to get the text of the option.

The Survey class

Each survey will have a topic (what the survey is about) and a list of options. The constructor for the survey is given the topic for the survey and a list of options.

```
constructor(newValue) {  
    this.topic = newValue.topic;191  
    this.options = [];192  
    newValue.options.forEach(optionValues => {193  
        let newOption = new Option(optionValues);194  
        this.options.push(newOption);195  
    });  
}
```

The `Survey` constructor above makes a new `Survey` instance from the values that are supplied to it. The parameter `newValue` refers to an object that contains the values for the new `Survey`.

```
let newSurveyValues = {  
    topic: "robspizza",
```

¹⁹¹ Set the topic name

¹⁹² Make an option array

¹⁹³ Work through the supplied option texts

¹⁹⁴ Make a new option

¹⁹⁵ Add the option to the list

```

options: [
  { text: "margherita" },
  { text: "pepperoni" },
  { text: "chicken" },
  { text: "ham and pineapple" },
  { text: "mushroom" },
]
};

let pizzaSurvey = new Survey(newSurveyValues);

```

The code above creates a literal object referred to by `pizzaSurvey` that contains all the initial settings for a `Survey` object. It then uses this object to initialize a new instance of the `Survey` class referred to by the variable `pizzaSurvey`. The `newSurveyValues` object has all the properties of a `Survey` and these will be copied into the new `Survey` instance by the constructor.

CODE ANALYSIS

Survey Constructor

You might have some questions about what this constructor does.

Question: What does `forEach` do?

A JavaScript array object provides a `forEach` method that can be used to work through all the elements of the array. `forEach` is supplied with a function which is called for each element in the array. In the case of the Survey constructor the program must work through all the supplied option values and create an option for each one.

```

newValue.options.forEach(optionDetails => {
  let newOption = new Option(optionDetails);
  this.options.push(newOption);
});

```

The first parameter to the function called by `forEach` is the element of the array that is currently being processed. In the above code this parameter has been called `optionDetails` and contains the details for the new option being created. The details are fed into the `Option` constructor to make a new `Option` instance which is then pushed onto the list of options that the constructor is building.

Question: Why do we allow the `Option` constructor to set the count value for a new `Option` instance?

Since all options should have an initial count of 0 (the option has been picked 0 times) you might be wondering why the constructor for the [Option](#) accepts a count value. This is because we might want to create an [Option](#) from a string of JSON which has been sent to us. In that case we wouldn't want to set the count to 0, we want set the count to whatever was in the object we received.

Survey methods

One of the fundamental principles of object-oriented programming (that sounded very posh – didn't it) is that you shouldn't need to know how an object works internally to make use of it. When the Tiny Survey application is using the [Survey](#) object it should not know (or care) how the [Survey](#) object works or what is inside it. The [Survey](#) object should provide methods that can be called to do exactly what the application needs, and nothing more. Once a [Survey](#) has been created there are only three things that it needs to be able to do for an application. It needs to increment the counter for a particular option, get the option texts and their names for display when an option is being picked and get the option texts and their count values to display the results. Let's look at each of these methods in turn.

```
incrementCount(text){  
  let option = this.options.find(196  
    item=>item.text == text); 197  
  if(option != undefined){ 198  
    option.incrementCount(); 199  
  }  
}
```

The [incrementCount](#) method increments the [count](#) for an option. We give it the text of the option (perhaps “pepperoni”) and it finds the option with that text. If the option is defined it then calls [incrementCount](#) on that option to make the count 1 for that option 1 bigger. If you are not sure how the finding process works, we will be taking a more detailed look at [find](#) in the next section.

¹⁹⁶ Find the option

¹⁹⁷ Match the option name

¹⁹⁸ Is the option defined?

¹⁹⁹ Increment the counter if it is

```

getOptions(){
  let options = [];200
  this.options.forEach(option=>{201
    let optionInfo = { text: option.text };202
    options.push(optionInfo);203
  });
  let result = {topic:this.topic, options:options};
  return result;
}

```

The `getOptions` method returns an object that contains the topic of the survey and a list of objects that contain the option texts (for example `{topic:"robspizza",options: [{text:"pepperoni"}, {text:"chicken"}]}` for a really tiny survey). This list is used to build the display of radio buttons that the user will use to select their preferred option.

```

getCounts(){
  let options = [];204
  this.options.forEach(option=>{205
    let countInfo = { text: option.text,
      count: option.getCount() };206
    options.push(countInfo);207
  });
  let result = {topic:this.topic, options:options};208
}

```

²⁰⁰ Make an empty option array

²⁰¹ Work through all the options

²⁰² Make an object that contains the option text

²⁰³ Add the option to the list

²⁰⁴ Make an empty option array

²⁰⁵ Work through the options in the survey

²⁰⁶ Build an object containing the count and the option text

²⁰⁷ Add the object to the options

²⁰⁸ Make a result object

```
    return result;209  
}
```

The `getCounts` method returns an object that contains the topic of the survey and a list of objects that contain the option text and the count for each option (for example `{topic:"robspizza",options:[{text:"pepperoni",count:1}, {text:"chicken",count:0}]}`).

It uses `ForEach` loop that works through all the options in the survey and builds an object containing the option text and the count value for each. It then creates a `result` object which contains the topic name for the survey and this option list. The list is used to build the page that shows the counts for each item.

Make Something Happen

Tiny Survey classes

Open the `index.html` file in the **Ch07-03-Tiny_Survey_Classes** example folder using your browser. Open the console the Developer Tools view. The page contains the Tiny Survey classes and a function called `makeSampleSurvey` function which creates a sample survey.

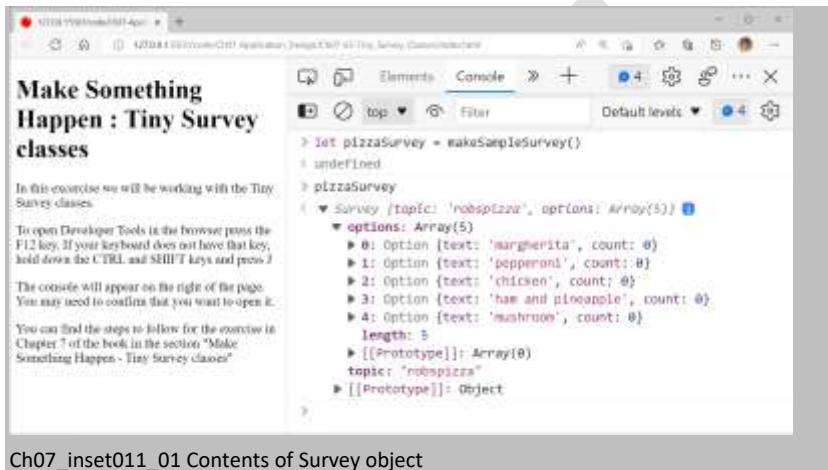
```
function makeSampleSurvey() {  
  
    let newSurveyValues = {  
        topic: "robspizza",  
        options: [  
            { text: "margherita",count:0 },  
            { text: "pepperoni", count:0 },  
            { text: "chicken", count:0 },  
            { text: "ham and pineapple", count:0 },  
            { text: "mushroom", count:0 },  
        ]  
    };  
  
    let result = new Survey(newSurveyValues);  
    return result;  
}
```

The function makes an array of options and a topic name value to create a Survey instance. Type in the following to use this function to make a sample survey.

²⁰⁹ Return the result

```
let pizzaSurvey = makeSampleSurvey()
```

The variable `pizzaSurvey` now refers to an instance of the `Survey` class. We can use the console to view the contents of the survey. Type `pizzaSurvey` into the command prompt and press Enter. The contents of the object will be displayed. Expand the items in the object to see the entire contents.



Ch07_inset011_01 Contents of Survey object

Above you can see the contents of the class. Now let's get it to do some work for us. Let's add a vote for pepperoni. Type the command and press enter. This will increase the value of the count for the option with the name item2.

```
pizzaSurvey.incrementCount('pepperoni')
```

We can check that this works by asking the survey object to give us an object that contains the results for the survey. Type the command below and press enter.

```
let counts = pizzaSurvey.getCounts()
```

The `counts` variable now refers to an object which contains the results information for the survey. We can take a look at the content of the `counts` object. Type `counts` and press Enter. Then expand the contents of the object.

The screenshot shows a browser window with developer tools open. The console tab is selected. The code being run is:

```
> pizzaSurvey.incrementCount('pepperoni')
undefined
> let counts = pizzaSurvey.getCounts()
undefined
> counts
<--> [
  * (topic: 'robspizza', options: Array(5))
    * options: Array(5)
      0: {text: 'margherita', count: 0}
      1: {text: 'pepperoni', count: 1}
      2: {text: 'chicken', count: 0}
      3: {text: 'ham and pineapple', count: 0}
      4: {text: 'mushroom', count: 0}
    length: 5
  [[Prototype]]: Array(0)
  topic: "robspizza"
  [[Prototype]]: Object
```

Ch07_inset011_02 Contents of counts object

You can add some more votes for other topics and take a look at what the `getOptions` method returns. Leave your browser open, we will be using this sample page in the next “Make Something Happen”.

CODE ANALYSIS

Tiny Survey classes

The Tiny Survey classes work well. But you might have some questions.

Question: What is the difference between an instance of the `Option` class and a JavaScript object that just happens to contain `text`, and `count` properties?

This is a brilliant question. From a data property perspective there is no difference at all. JavaScript does not make a distinction between an `Option` instance and any other object with matching properties. JavaScript uses what is called “duck typing”, from the saying “If it walks like a duck and talks like a duck – it is a duck”. So any part of an application that could work with the data in an `Option` instance would work also with an object which contains a `count` and a `text` property. However, only an instance of the `Option` class would contain the `getCount`, `getText` methods.

Question: If you turn an instance of a class into JSON string, do the methods get encoded into the string as well?

This is a good question. The answer is no. A JSON string only ever contains the data

properties in an object, not the methods. However, we could feed the JSON string into the `Option` constructor to make a “proper” `Option` with all the methods. We could use this to send options from one place to another. Perhaps we want to send our `pizzaSurvey` somewhere. We can do this:

```
let pizzaSurveyJSONString = JSON.stringify(pizzaSurvey);
```

This creates a string called `pizzaSurveyJSONString` which contains a JSON description of the survey contents. We can send that string to someone else and they can create an object from it using `JSON.parse`:

```
let receivedObject = JSON.parse(pizzaSurveyJSONString);
```

Now we can turn the `receivedObject` into a `Survey` just by feeding it into the `Survey` constructor.

```
let receivedSurvey = new Survey(receivedObject);
```

This `receivedObject` will have all the data properties from the original Survey object and so the `receivedSurvey` will have those set by the constructor.

The Surveys class

The application needs to store active surveys. We can create a class called `Surveys` which will do this. There are lots of ways that we could store the survey values but to start with we are going to use a simple array. We save a survey by adding it to the array. We find a survey by searching the array for one with a matching topic.

```
class Surveys {
  constructor() {
    this.surveys = [];
  }

  saveSurvey(survey) {
    this.surveys.push(survey);
  }

  getSurveyByTopic(topic) {
    return this.surveys.find(element => element.topic == topic);
  }
}
```

The complete `Surveys` class is above. The constructor method for a survey creates the array that holds the surveys. The `saveSurvey` method pushes a received survey value onto the survey list. The `getSurveyByTopic` method returns the survey with the matching topic name.

Make Something Happen

The Surveys class

The `index.html` file in the **Ch07-03-Tiny_Survey_Classes** example folder also contains the `Surveys` class for us to look at. We can start by making a new pizza survey:

```
let pizzaSurvey = makeSampleSurvey()
```

Type the statement above and press Enter. The variable `pizzaSurvey` now refers to an instance of the `Survey` class. We now need to create a survey store:

```
let store = new Surveys()
```

Type the statement above and press Enter. The variable `store` now refers to a `Surveys` instance. Now we can store our survey in the store:

```
store.saveSurvey(pizzaSurvey)
```

Type the statement above and press Enter. We now have a survey in the store. We can take a look at the store contents.

```
store
```

Type the statement above to get the console to display the `store` object.

```
> let pizzaSurvey = makeSampleSurvey()
< undefined
> let store = new Surveys()
< undefined
> store.saveSurvey(pizzaSurvey)
< undefined
> store
< < Surveys {surveys: Array(1)} >
  * surveys: Array(1)
    > 0: Survey {topic: 'robspizza', options: Array(5)}
      length: 1
      > [[Prototype]]: Array(0)
    > [[Prototype]]: Object
>
```

Ch07_inset013_01 Contents of store

Now we can use the `getSurveyByTopic` function to get a survey out of the store. There is only one survey in there, and it has the topic "robspizza".

```
let loadedSurvey = store.getSurveyByTopic("robspizza")
```

Type the statement above to make `loadedSurvey` refer to the survey with the topic "robspizza". Now we can view the content of `loadedSurvey`. Type in the name of the variable and press enter to take a look at what it contains.

```
loadedSurvey
```

```
> let loadedSurvey = store.getSurveyByTopic("robspizza")
< undefined
> loadedSurvey
< < Survey {topic: 'robspizza', options: Array(5)} >
  * options: Array(5)
    > 0: Option {text: 'margherita', count: 0}
    > 1: Option {text: 'pepperoni', count: 0}
    > 2: Option {text: 'chicken', count: 0}
    > 3: Option {text: 'ham and pineapple', count: 0}
    > 4: Option {text: 'mushroom', count: 0}
    length: 5
    > [[Prototype]]: Array(0)
    topic: "robspizza"
  > [[Prototype]]: Object
```

Ch07_inset013_02 Loaded survey

Above you can see that `loadedSurvey` contains the same values as `saveSurvey`. The correct

survey has been located.

CODE ANALYSIS

The Surveys class

The Surveys class is very small. But you still might have some questions about it. .

Question: How does `findSurvey` work?

The `findSurvey` method is given the topic of the survey (for example “robspizza”) and it then goes and finds the survey with that topic.

It uses the `find` method provided by the `surveys` array. All arrays provide a `find` method. You give the `find` method a function that accepts a parameter (which will be an element of the array) and returns `true` when the required element has been found. In the case of `findSurvey` the element is found if it has a `topic` that matches the one supplied as a parameter, so the function provided to `find` tests for this.

Question: What does `findSurveysByTopic` return?

The method returns a reference to the survey with the matching topic. Changes to this survey will be reflected in the survey “in” the store because they are both actually the same survey object.

Question: What happens if the topic name is not matched?

If we ask `findSurveyByTopic` to find a survey that doesn’t exist it will return the value `undefined`. Any program using `findSurveyByTopic` must check for `undefined` in case the survey was not found.

Question: Why have we made a `Surveys` class? The `Surveys` class is very small. Each function in the class is contains single statement. Would it not be easier just to put these statements in the Tiny Survey application code and use them directly?

Creating a `Surveys` class separates the survey storage from the application. It makes it very easy for us to change from using an array to using a database. We would just have to change the `find` and `save` methods in the class and not change the application at all.

Question: How are old surveys removed?

At the moment they aren’t. The `surveys` array just keeps getting larger as more surveys are pushed onto it. However, we could modify the `saveSurvey` method to set an upper limit on the size of the `surveys` array and remove the oldest one each time a new survey was added beyond that limit.

Coming next...

We now have a workflow and the data storage for our Tiny Survey application. In the next chapter we will create the working web pages and the code to make Tiny Survey work.

What you have learned

This has been another busy chapter. We've covered a lot of ground. Here is a recap plus some points to ponder.

- Companies that create applications and regulating bodies have ethical standards against which new applications will be judged before release. It is important that issues of ethics, privacy and security are considered during the design process of a new application.
- Before you start writing code or building web pages for a new application you need to create the “workflow” for it. The workflow expresses the items the user will see, the order they are presented and how the user will move from one item to the next. Workflow can be designed initially on paper. It is a good idea to create a prototype application from web pages that just contain fixed text. These are very useful when documenting the steps through the workflow.
- Once the workflow of an application has been designed you can move on to consider the data storage which will be required by the application. Initial discussions can talk about the storage items in general terms (for example survey) but these can be refined to consider precisely what is held in each item.
- The JavaScript class defines an object which has a constructor method that can create specific object properties set at values determined by those supplied to the constructor as parameters. The parameters are best sent into the constructor as a literal object holding properties with values which are transferred into the new object being created.
- The JavaScript keyword `this`, when used in a constructor method means “a reference to the object currently being created”.
- A class constructor can validate initial values supplied as parameters but the only way that the constructor can indicate invalid parameter values is by throwing an exception.
- An instance of a class is created using the keyword `new`. This will invoke

the constructor method for the class. The constructor is given parameters which contain values to be used to initialize the instance.

- A JavaScript class can contain methods as well as properties. A method is equivalent to a function but it exists within a class. Within the code for a method the keyword this means “a reference to the instance within which this method is running”.
- Code external to a class instance should not have to manipulate the contents of the instance to change the data in it. Instead, the instance should provide methods that can be called to perform the specific action required. As an example; the Option class in TinySurvey provides a method that can be used to increment the count value. External code should not access the internal count property directly.
- Arrays provide a `forEach` function which can be made to perform a given function on every element of the array.
- JavaScript doesn't regard objects in terms of their particular type. Any object with a particular set of properties is interchangeable with any other object with the same set of properties.

Converting an instance of a class into a JSON string does not encode the particular type of the instance or any of the method properties into the resulting string.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions".

How much effort should you put into ethical, privacy and security issues when starting an application?

It is always a useful exercise to consider these things at the start of a development. It is often the case that simple changes to the behavior of an application can make it more ethical. The reverse is also true. Attempts to monetize an application can convert it from ethical to unethical. Ethical considerations need to be tracked during a project in terms of the effects of changes to the application and also observations of what people are doing with it.

Are there any other issues to consider other than ethics when deciding to build an application?

Other issues worth of consideration are business model – how you will make money from the application and who owns what – if you are building something as a team, how any profits would be shared out and who can claim ownership for different parts of the application.

Do you need a workflow diagram if you are only creating a single page application?

The Tiny Survey application is spread across several pages. However, you can also create single page applications. In this case, rather than documenting the navigation between pages you should document the workflow between different states of page a user can occupy as the user works with the application.

What is the difference between a class and an object?

A class sets out what behaviors the object should have and provides a mechanism (the constructor) of setting initial values in the object. You can think of a class as the recipe (how to bake a cake) and an instance of a class as the result of following the recipe (a cake).

Can a class have multiple constructors?

The constructor method in a class is used to set up initial values held in the class. It might be useful to have multiple constructors so that an instance could be created from a JSON encoded string or some other means. JavaScript does not allow this. There can only be one constructor method in a class. However, you could make a constructor that examines the type of the parameter coming into the constructor and acts appropriately.

How can a program detect the class of a particular object?

The `typeof` function will return the type of an object, but this does not work with classes. An instance of a class always has the type of "Object". However, the `instanceof` operator can be used to determine if a reference to an object refers to an object of a particular type.

```
if (pizzaSurvey instanceof Survey) console.log("A Survey")
```

The above statement shows how `instanceof` is used. The message would be logged if the reference `pizzaSurvey` refers to a `Survey` object.

Can classes be exported from a library?

Yes they can. When we create the Tiny Survey application in the next chapter we will use a `SurveyStore.mjs` library file that exports the `Survey`, `Option` and `Surveys` classes for use by the application.

Chapter 8: Build an application

What you will learn

In the last chapter we designed an application for tiny surveys. We set out the pages that the user will interact with and the workflow followed as the application is used. Then we designed the data storage to underpin the application and implemented a set of classes that provide the storage behaviors needed by the application.

In this chapter we are going to build the application. We are going to learn how add style to our pages and how the Express framework makes it easy to create web applications. We'll also create our first project using Node Package Manager and discover how to create and configure applications that use multiple libraries. Along the way we'll learn some new tricks with git to help us manage changes to our code. And at the end we will have our working survey ready for deployment into the cloud.

This is the point in this section where I remind you about the glossary. I'd hate to disappoint you, so don't forget about the glossary.

Put on the style with Bootstrap

Before we start to create the pages to be used in Tiny Survey we should think about how they are going to be styled. The prototype pages that we have created so far have used the standard HTML styles for the elements. This will work, but it doesn't give a very good user experience. In previous applications we have added our own stylesheets to set the styles for the elements on the page. We first saw stylesheets in chapter 2 when we used them to set the size of the text displayed by our JavaScript clock. See the Code Analysis section "Document objects and JavaScript" to find out what we did. Ever since then, whenever we have created a web page we have created a stylesheet file to go with it. The stylesheet file contains definitions of styles which are applied to the elements in the HTML files. The header for the document contains a reference to a local file called `style.css`.

```
<link rel="stylesheet" href="styles.css">
```

The `style.css` file contains the definition of classes which give display settings for the elements in our page. Creating good stylesheets that work on multiple devices is hard work that has already been done by lots of people who are very good at it. Some of those people are at Bootstrap (<https://getbootstrap.com/>). They have made stylesheets that we can use in our applications. The stylesheets are very powerful and include dynamic behaviors that make pages adjust for different sized displays. We are not going to use all these features in Tiny Survey, but they are well worth exploring. We can use a Bootstrap stylesheet by setting it as the stylesheet that for a document:

```
<link rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/css/bootstrap.min.css"
      integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
      crossorigin="anonymous">
```

This does the same job as the previous link statement; it sets a stylesheet for a document. However, the stylesheet is now downloaded from the internet. The `integrity` property is used by the browser to validate the stylesheet file that it receives. We can use this stylesheet to format the index page for the Tiny Survey application.

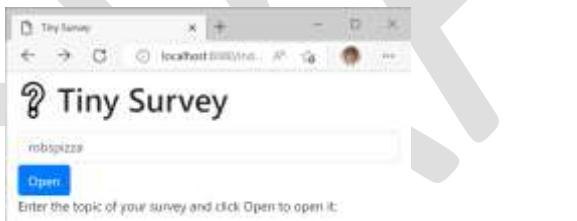


Figure 8.33 Ch-08_Fig_01 Tiny_Survey_bootstrap

Figure 8.1 above shows the index page for Tiny Survey. The user will enter the name of the topic for the survey. We are going to select the best pizza topping got me, so the topic has been called “robspizza”. This page is styled using the Bootstrap stylesheet. Let’s take a look at the HTML behind the page.

```
<!DOCTYPE html>
<html>
<head>
```

```

<title>Tiny Survey</title>
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<link rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/css/bootstrap.min.css"
      integrity="sha384-  

gg0yR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUhcWr7x9JvoRxT2MZw1T"  

crossorigin="anonymous">
</head>

<body>
<div class="container">
  <h1 class="mb-3 mt-2">Tiny Survey</h1>
  <p>
    <input type="text" class="form-control" id="topic" spellcheck="false">
  </p>
  <p>
    <button class="btn btn-primary mt-1" onclick="doEnterOptions()>Open</button>
  </p>
  <p>
    Enter the topic of your survey and click Open to open it.
  </p>
</div>
<script>
  function doEnterOptions() {
    window.open("enteroptions.html", "_self");
  }
</script>
</body>
</html>

```

CODE ANALYSIS

Using Bootstrap in html

You can find the Bootstrap version of Tiny Survey in the folder **Ch08_01_Bootstrap_Tiny_Survey**. All the application pages are now styled using bootstrap. You may have some questions about the index file.

Question: What does the line starting `<meta` do?

This creates a viewport which allows the Bootstrap styles to scale the page appropriately for

different devices.

Question: What does `div` do?

Sometimes you want to lump some page content together so that you can apply things to it. The `div` encloses all the content on the page, which we want to assign style with the class `container`. The `container` class is provided by Bootstrap and allows us to group together related items on the page.

Question: What does `❔` mean?

You can find this in the heading for the page. It is the code for an emoji which displays a white question mark. This is a low-cost way of getting a simple logo for the application.

Question: What does `class="mb-3 mt-2"` mean?

These are class names which contain layout instructions for Bootstrap. `mb-3` sets the bottom margin and `mt-2` sets the top margin. You can use these to vertically position items on the page.

Question: What does `spellcheck="false"` mean?

```
<input id="topic" spellcheck="false">
```

Topic names chosen by the user might not match words in the dictionary. I've called my topic "robspizza". If the input is not meant to be correctly spelt you can add this property to tell the browser not to complain if words in the input are not in the dictionary.

Question: What does `window.open` do?

When the user presses the Open button we want the application to move to another window. For our demonstration application it will move to the page where the survey options will be entered. The page that will do this is called `enteroptions.html`.

```
window.open("enteroptions.html", "_self");
```

The `window.open` function tells the browser to open the specified url. The second parameter, `_self` tells the browser to open the url in the existing window.

Getting started with Express

We created the "Cheese Finder" application by writing client JavaScript that ran in the browser and server JavaScript that run in the server. The client JavaScript built the HTML pages that the player sees. The server JavaScript ran the game itself. The server JavaScript also served out the web pages that were used by the browser to build the page. We could use the same technique to build Tiny Survey. A JavaScript program running in the browser could build each of four pages

depending on what part of the application the user was interacting with. However, it turns out that there is a framework we can use that makes the construction of multi-page applications much easier. It is called Express and you can find it at <http://expressjs.com/>. Before we can start using Express we have to look into just how libraries are incorporated into node.js applications.

Express and Node Package Manager

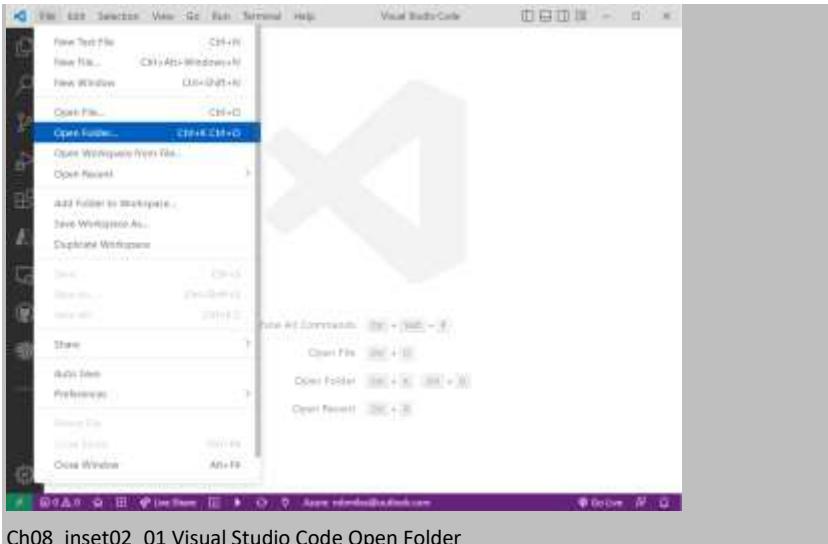
In chapter 6 we used a [package.json](#) file to describe the elements of the Cheese Finder application that we wanted to deploy into the cloud. The Cheese Finder program didn't depend on any other code to run. Everything it used was part of JavaScript, the node.js framework or a library file we created for the application (for example [pseudorandom.mjs](#)). The Tiny Survey application will depend on many external libraries. The code in these libraries will need to be added to our application and something will have to make sure that they are all kept up to date. The Node Package Manager will do this for us.

The [npm](#) (Node Package Manager) program is supplied as part of a node.js installation and talks to servers which are managed by npm (<https://www.npmjs.com/>). These servers host the library files and the [npm](#) program reads the [package.json](#) file to discover which library files a project needs. The [npm](#) program manages the contents of [package.json](#), starting with creating a new application. Lets see how we do this.

Make Something Happen

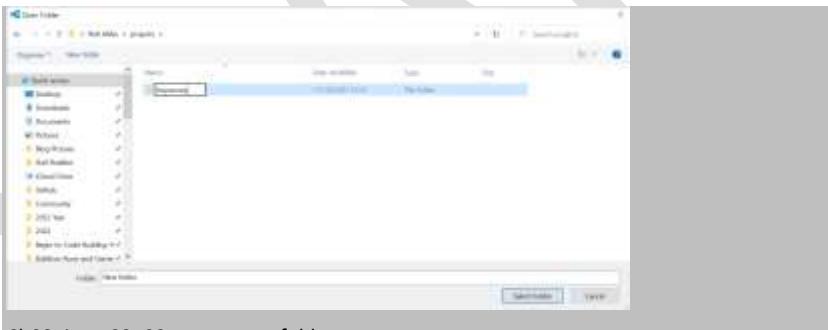
Create the Tiny Survey project

This is a different "Make Something Happen". Rather than use pre-built code as a starting point we are going to start with a completely blank folder and then create an application in it. Later we will add some pre-built components. We will use the integrated terminal in Visual Studio Code to do this. In Chapter 1 we installed the [git](#) program on our computer. We have not used it from the terminal before, but in this exercise we are going to use [git](#) to make a new repository on our computer and then add the files we need to get started.



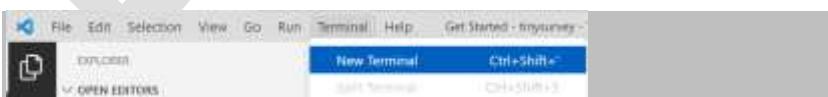
Ch08_inset02_01 Visual Studio Code Open Folder

The first thing to do is to start Visual Studio Code running. Then close any folders that might be open in Visual Studio and then select the **Open Folder...** option from the **File** menu. This will open the folder browser.



Ch08_inset02_02 create new folder

Click **New Folder** in the Open Folder window to make a new folder. You can put the folder anywhere on your machine. I have a folder called **projects** in my home folder where I keep the projects I'm working on. Create a folder called **tinySurvey** and then click **Select Folder** to select it in Visual Studio.



Ch08_inset02_03 Visual Studio Code New Terminal

Now that we have a folder we can use the terminal interface in Visual Studio Code to set it up. Select **New Terminal** from the **Terminal** menu. The terminal window will open at the bottom of the Visual Studio Code window.



```
OUTPUT TERMINAL AZURE Jupyter DEBUG CONSOLE ⓘ
PS C:\Users\rsmil\projects\tinysurvey> git init
```

Ch08_inset02_04 Create Git repository

The first thing we are going to do is create a git repository for this folder. This will track the changes in our files. We use the [git](#) program to do this.

```
git init
```

Type the command as shown above and press Enter.



```
OUTPUT TERMINAL AZURE Jupyter DEBUG CONSOLE ⓘ
PS C:\Users\rsmil\projects\tinysurvey> git init
Initialized empty Git repository in C:/Users/rsmil/projects/tinysurvey/.git/
PS C:\Users\rsmil\projects\tinysurvey>
```

Ch08_inset02_05 View Git repository

The command will confirm that it has completed. Now that we have a git repository we can start adding files to our application. The Node Package Manager (npm) program will do this for us. We will be asked a series of questions and the answers will be used to build a [package.json](#) file for the project.

```
npm init
```

Type the command as shown above and press Enter.

```
PS C:\Users\rsmil\projects\tinysurvey> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible
defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (tinysurvey)
version: (1.0.0)
description: Tiny Survey application
entry point: (index.js) tinysurvey.mjs
test command:
git repository:
keywords:
author: Rob Miles
license: (ISC)
About to write to C:\Users\rsmil\projects\tinysurvey\package.json:
```

```
{  
  "name": "tinysurvey",  
  "version": "1.0.0",  
  "description": "Tiny Survey application",  
  "main": "tinysurvey.mjs",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "author": "Rob Miles",  
  "license": "ISC"  
}
```

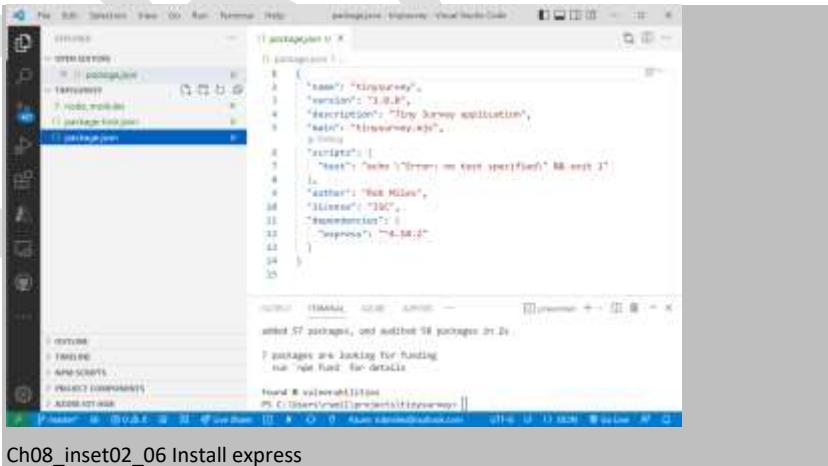
Is this OK? (yes)

```
PS C:\Users\rsmil\projects\tinysurvey>
```

For most of the questions you can just press enter. The answers that you give are shown above in **bold**. You only have to give the entry point (the name of the file which will be run to start the project) and the author name. Now we have a `package.json` file for our application. We can look at it (in fact it is shown above) but we usually let `npm` look after the contents of the file. The next step is to add the Express library to our project. The `npm install` command will do this for us. The install command is followed by the name of the package to be installed.

```
npm install express
```

Type the command as shown above and press Enter. The `npm` program looks for Express on the server and discovers that it uses 57 other packages. These are all copied into a folder called `node_modules` which is added to the application along with two package files.



Ch08_inset02_06 Install express

Open the `package.json` file in the Visual Studio Editor so you can see the changes. As you can see above, there is now an entry in the dependencies section to indicate that this application uses Express, and needs a version which is newer than 4.18.2 The only thing we are missing is

the **tinsurvey.mjs** file which contains the JavaScript which will run the application.

Click on the “new file” icon to the right of the TINY SURVEY item in the Explorer, and then give the new file the name **tinsurvey.mjs**

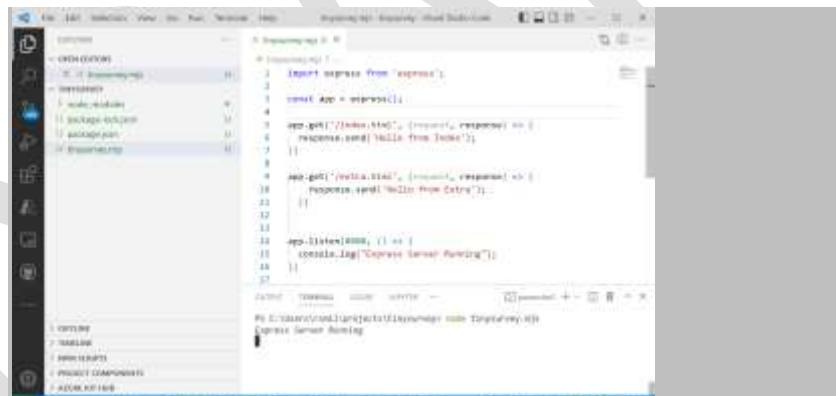


Ch08_inset02_07 Create tinsurvey.mjs

This will create an empty file. You can find the contents of the **tinsurvey.mjs** file we are going to use in the folder **Ch08-02_Express_Hello_World** in the sample programs for this chapter. Find the file on your machine and copy the contents of it into the **tinsurvey.mjs** file into yours. Save **tinsurvey.mjs**. Now we can use node to run it.

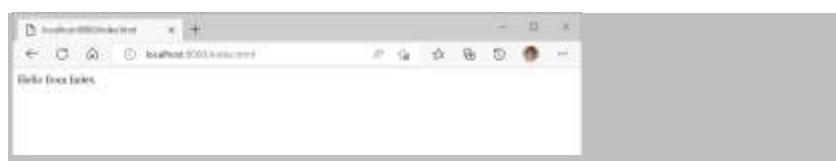
```
node tinsurvey.mjs
```

Open a terminal window, type the command above and press enter to start your program running.



Ch08_inset02_08 Express Server Running

The Express framework is hosting a web server running on port 8080. Let’s use the browser to see what it is serving (although the listing above might contain some strong hints). Open your browser and navigate to **localhost:8080/index.html**



Ch08_inset02_09 Hello from index

Above you can see the response from the index page. You might like to find out what happens when you visit **localhost:8080/extra.html**



Ch08_inset02_10 Hello from extra

You may be wondering what happens if the browser tries to access a page which is not present. Try to access **localhost:8080/fred.html**.



Ch08_inset02_11 Access fred.html

The Express server will generate the correct response. There is no route for **fred.html**. We can add as many end points as we wish to the Express project. The code that behind each endpoint sends out a simple response, but we can also run code to generate page content, of which more later.

We can debug our application by using the debugger to start the **tinysurvey.mjs** program in the same way we have been debugging the Cheese Finder code. However, to do this we must first stop the node.js application by pressing **CTRL+C** in the terminal window. Stop the application now but leave Visual Studio Code running so that you can use it in the next exercise.

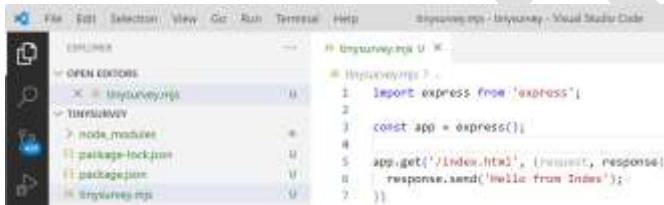
Express routes

```
app.get('/index.html', (request, response) => {
  response.send('Hello from Index');
})
```

The code above deals with a **get** request for the **index.html** page. A browser generates a **get** request when it wants something from the server. The **get** method in Express defines a what is called a *route*. This is a mapping between an endpoint and a JavaScript function that will deal with it. The first argument to **get** is a string containing the endpoint that the route is handling. In this case it is the page **index.html**. The second argument to **get** is a function which accepts two

arguments, one called `request` and the other `response`. We could have declared a function called `handlePageRequest` (as we did in Chapter 4 in the section “Serving from software”) to deal with this route. Instead, we have created an anonymous arrow function which calls the `send` method on the `response` parameter to send a reply to the browser. We can create as many routes as we like to deal with different endpoints. But before we do this, we will save this working demonstration.

Manage versions with git

A screenshot of the Visual Studio Code interface. The left sidebar shows the Explorer with files like 'index.html', 'node_modules', 'package-lock.json', 'package.json', and 'surveyapp.js'. The main editor area shows a file named 'index.js' with the following code:

```
import express from 'express';
const app = express();
app.get('/index.html', (request, response) => {
  response.send('Hello from Index');
});
```

The top status bar indicates 'SurveyApp - 1 untracked file'.

Figure 8.34 Ch-08_Fig_02 Git Changes

Racing drivers and programmers have one thing in common. They both hate going backwards. When you work on code one of the worries is that you might break your application and be unable to get it back to how it was before. The `git` program provides a solution to this problem. We currently have a working application, so we should probably commit these changes to `git`, ready for the next part of the development. Figure 8.2 above shows the Source Control item on the lefthand side. The number 428 on the control means that there have been 428 changes to files in the repository since it was created. Visual Studio Code is using the `git` repository that we created to track changes to the files in the application folder and installing Express and its dependencies creates a lot of new files.

Use `gitignore`

We could just commit all the changes to `git` as you can see in Figure 8.2. This would add 428 files to the repository for our application. However, we didn't create most of the files. They were created when `npm` installed `Express`. The `package.json` file contains a list of dependencies which set out the libraries that are needed by this project. There is no need to keep all the `Express` files in our repository because anyone wanting to build our application could look in the `package.json` file, find the dependencies and then get all the required files. It would be useful if there was a way of saying to `git` “Don't bother putting these files like these in the repository because they are from a library”.

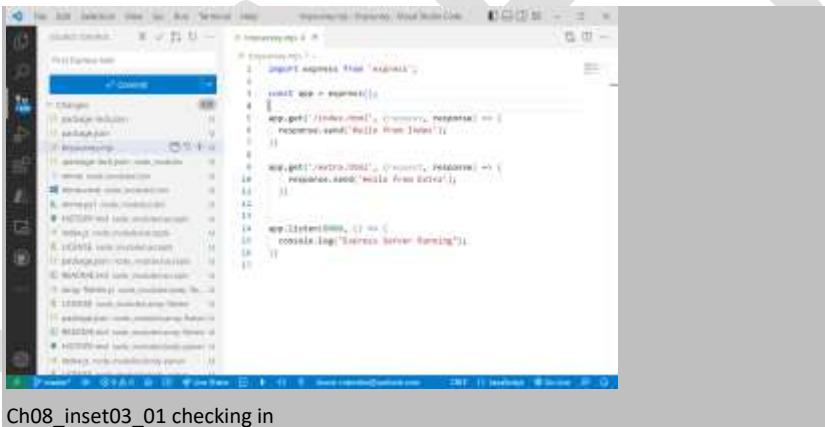
It turns out that there is. It is called `gitignore`. You can add a file called `gitignore` to your repository. This file contains a list of patterns matching files and folders that should not be included in the repository. How you create a `gitignore` file is bit beyond the scope of this book, but you don't

need to because GitHub has a list of [gitignore](#) files you can use for any project which you can find here: <https://github.com/github/gitignore>. We are using the [gitignore](#) file for node projects. Let's see how all this works.

Make Something Happen

Check in your code

We saw that when we used [git init](#) to create the repository git replied with the message "Initialized empty git repository". The repository is a folder managed by git which holds all the versions of your files. A commit takes a snapshot of the files in your application that have changed and stores it in the repository. If you click the Source Control icon on the left you can see all the files that have changed. Type a summary of this commit (I've put "First Express test"). You use this comment to identify this version so it is a good idea to add some useful detail.



The screenshot shows the Visual Studio Code interface. On the left is the Source Control sidebar showing a single file named 'package.json'. The main editor area contains a file named 'server.js' with the following code:

```
1 // Import express
2 import express from 'express';
3
4 const app = express();
5
6 app.get('/index.html', (request, response) => {
7   response.sendFile('index.html');
8 })
9
10 app.listen(3001, () => {
11   console.log('Server is running');
12 })
```

Ch08_inset03_01 checking in

If you click the [Commit](#) button the new files are copied into the repository. Don't press it just yet though, because we want to reduce that value of 428 changes to something more manageable by adding a [gitignore](#) file. You can find the file in the folder **Ch08_03_Node_gitignore**. Copy it into your application folder. Now go back and look at Source Control view.

```
src/main/resources
```

Ch08_inset03_02 Added gitignore

The only files that will now be added to the repository are the ones that we have created. Plus the `gitignore` file. Press the **Commit** button to check them in. When it has completed you will see a screen like the one below. The number of changes since the last commit is now zero.

Commit message: Initial commit

Ch08_inset03_03 Commit complete

The “Publish Branch” button allows you to publish this repository on GitHub. If you click it you will be asked to login with your GitHub username to begin the publish process. You don’t need to use GitHub just to get the file tracking however, you can keep your repositories local. We can see the tracking in action by making a change to one of our files. Return to the Explorer view and make a change to the `tinysurvey.mjs` file (I’ve added endpoint for `extra1.html`). Save the file.

The screenshot shows the Visual Studio Code interface with the 'tinyserver.mjs' file open in the editor. The status bar at the bottom indicates a 'Changes' count of 1. The code itself is a simple Express.js application.

```
1 // tinyserver.mjs
2 import express from 'express';
3 const app = express();
4
5 app.get('/index.html', (request, response) => {
6   response.send('Hello from Index');
7 }
8
9 app.get('/extra.html', (request, response) => {
10   response.send('Hello from Extra');
11 }
12
13 app.listen(8080, () => {
14   console.log('Express Server Running');
15 }
16
```

Ch08_inset03_04 file modification

The change counter above the Source Control button is now showing one change to the committed repository. Click on the Source Control button to open the source control view and then click [tinyserver.mjs](#) to view the changes that were made to this file.

The screenshot shows the Visual Studio Code interface with the Source Control view open. It displays a diff between the 'tinyserver.mjs' file and its previous version. The changes are highlighted in green and red, indicating additions and deletions respectively.

```
1 // tinyserver.mjs
2 import express from 'express';
3 const app = express();
4
5 app.get('/index.html', (request, response) => {
6   response.send('Hello from Index');
7 }
8
9 app.get('/extra.html', (request, response) => {
10   response.send('Hello from Extra');
11 }
12
13 app.listen(8080, () => {
14   console.log('Express Server Running');
15 }
16
```

Ch08_inset03_05 Source control changes

Visual Studio Code shows you a view of both files, the original and the changed version so you can see what you have been doing. If you press the undo arrow next to the filename you can revert the file back to its original version.

Programmer's Point

You should use Git during code writing

I have found Git very useful when writing a program, not just when using it to recover from mistakes. You can use the Source Control view to discover what changes you have

made to files, so that it is much easier to retrace your steps and find out what you have been doing. Try to get into the habit of committing your changes before you start working on your application.

Use page templates with ejs

At the start of chapter 7 we created a set of web pages showing how the Tiny Survey application will be used. It would be great if we could use these pages to start making our application. Well, it turns out that we can. Ejs (<https://ejs.co/>) is a piece of “application-level middleware” that can be used with the Express framework. Wow. That sounds seriously impressive. Tonight, you can tell your family (or the cat) that you spent the day installing “application-level middleware” and I’m sure they will be very impressed.

What it really means is that you’ve told the Express application to use the ejs engine to generate the pages to be sent to the browser. Middleware is software that works with an application and does something useful. There are several different types of middleware. “Application level” middleware is used within an application to perform a particular task. In this case the task is to render an html source page to provide the text to be sent to the browser. We will see some other forms of middleware later. We must install the `ejs` engine before we can use it, which we can do using the node package manager (npm) command in the terminal:

```
npm install ejs
```

We give this command at the terminal and another 200 or so files are added into our project to support `ejs`. Now we can tell the Express application to use this engine by adding the following statement to `tinyserver.mjs`:

```
app.set('view-engine', 'ejs');
```

The call of the `app.set` function is given two arguments. The first is a string which specifies the middleware being used for (in this case it is being used as the view-engine). The second argument is the name of the library that is providing the function. In this case it is the `ejs` library we have just installed.

The application can now call the `render` function on the `response` object we received from route and tell it to render the required eps file:

```
app.get('/index.html', (request, response) => {
  response.render('index.ejs');
})
```

The `render` action has been added to the route for `index.html`. This means that when a browser client requests the page at `index.htm` it will now be sent the contents of `index.ejs`. By convention the `eps` files are stored in a folder called `views`. We can have as many `eps` pages in the `views` folder as we need for a particular application.

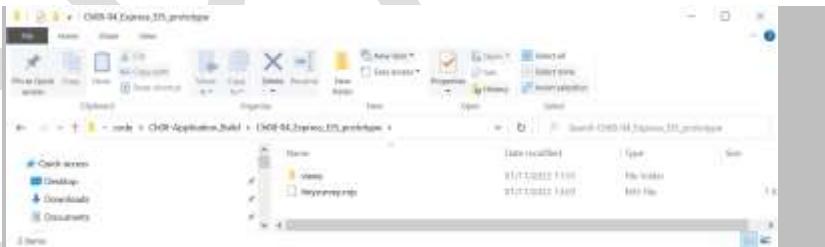
Make Something Happen

Use `eps` template files

We are going to make an Express application that displays the prototype web pages we created earlier. The first thing we need to do is install the `ejs` library.

```
npm install ejs
```

Enter the command above into the terminal window of Visual Studio Code and press Enter. Watch as `ejs` is installed in your project. Now you need to add some items to your project. You can find these extra items in the **Ch08-04_Express_EJS_Prototype** folder in the sample files for this chapter.



Ch08_inset04_01 extra files

You'll need an updated `tinysurvey.mjs` file and the contents of the `views` folder. The `eps` middleware looks in the `views` folder for any templates that are used in your application. Once you have copied the files into place, use Visual Studio Code to open your application and open the file `tinysurvey.mjs`.

The screenshot shows the Visual Studio Code interface with the file 'tinysurvey.mjs' open in the editor. The code is a Node.js application using Express.js. It defines a server that handles requests for 'index.html', 'enteroptions.html', 'selectoption.html', and 'displayresults.html', rendering corresponding EJS templates. It also listens on port 3001.

```
import express from 'express';
const app = express();
const port = 3001;
app.set('view-engine', 'ejs');
app.get('/index.html', (request, response) => {
  response.render('index.ejs');
});
app.get('/enteroptions.html', (request, response) => {
  response.render('enteroptions.ejs');
});
app.get('/selectoption.html', (request, response) => {
  response.render('selectoption.ejs');
});
app.get('/displayresults.html', (request, response) => {
  response.render('displayresults.ejs');
});
app.listen(port, () => {
  console.log("Server running");
});
```

Ch08_inset04_02 the tinysurvey.mjs file for eps

Your Visual Studio Code window should look like the above. You can see from the Explorer window at the top left that the views folder contains four files; [build.ejs](#), [entry.ejs](#), [index.ejs](#) and [results.ejs](#). These are the template files that will be used by the application. If you look inside them you'll see that they contain exactly the same text as the original html files. Click on [index.ejs](#) to take a look.

The screenshot shows the Visual Studio Code interface with the file 'index.ejs' open in the editor. The code is an EJS template for 'index.html'. It includes a DOCTYPE declaration, an HTML tag, a head section with a title, a meta viewport tag, and a link to a Bootstrap CSS file. The body section contains a container div with a class of 'mb-3 mt-2', a text input field with an id of 'topic', a button with a class of 'btn btn-primary', and a script block defining a function 'doEnterOptions' that opens 'enteroptions.html' in a new tab.

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Tiny Survey</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.5.2/dist/css/bootstrap.min.css" integrity="sha384-gH23oQ2mforq+90D404VrDjhSQyf+v3+皆の間で、彼は常に笑顔を保っていた。>">
</head>
<body>
  <div class="container">
    <h1 class="mb-3 mt-2" id="TopicInput">Tiny Survey</h1>
    <p>
      <input type="text" class="form-control" id="topic" spellcheck="false" value="What's your favorite color?">
    </p>
    <p>
      <button class="btn btn-primary mt-1" onclick="doEnterOptions()>">Enter Options</button>
    </p>
    <p>
      Enter the topic of your survey and click Open to open it.
    </p>
  </div>
</body>
<script>
  function doEnterOptions() {
    window.open("enteroptions.html", "_self");
  }
</script>
```

Ch08_inset04_03 index.ejs file

Now we have our Express powered version of the application we can start it running. Open a terminal and type the following command to start the application running.

```
node tinysurvey.mjs
```

If you open your browser and view the page at the <http://localhost:8080/index.html> location you will see the same example pages as before, but now they are being hosted by Express. We can use these template .ejs pages as the basis of the ones for the finished application.

Get the example application

We now have a set of page template files and an Express application that we can use to navigate between the pages. Now we are going to discover how each page works. We will do this by examining a complete implementation of [TinySurvey](#). The application is stored on GitHub in the repository <https://github.com/Building-Apps-and-Games-in-the-Cloud/TinySurvey>. We can use the git tool in the terminal to clone this onto our machine.

Make Something Happen

Get the example application using git

We are going to use the `git` command line to clone the example application. You can use the terminal in Visual Studio code. Start the terminal and navigate to a folder where you want to put the clone of the survey application.

```
git clone https://github.com/Building-Apps-and-Games-in-the-Cloud/TinySurvey
```

Ch08_inset05_01 clone command

Now enter the above command to clone the sample application. This will create a new folder called [TinySurvey](#) which contains all the files for the application. Note that it will not copy all the library files. You will need to install those yourself. Navigate into the [TinySurvey](#) folder that was created by git and enter the following command:

```
npm install
```

The `npm` program will open the `package.json` file and load all the dependencies for the application. Now you can run the program by using the `node` command:

```
node tinysurvey.mjs
```

Enter the command above and press Enter. Open your browser and navigate to **localhost:8080/index.html** and you will find a working Tiny Survey application. To stop the server hold down the CTRL key and press C.

You can open the [TinySurvey](#) application using the Open Folder command in Visual Studio Code. Any changes that you make to the application on your machine will not affect the version stored on GitHub.

The index page

We now have an application which contains working versions of each page plus the JavaScript program that navigates between them. Now we are going to investigate how each of the pages works. We are going to follow the same sequence as we used when designing the application. We are going to start with the index file which serves as the “home page”. The page takes the survey topic from the user and either creates a new survey with that topic (if the topic does not exist) or allow the user to select options for the survey (if the topic does exist).

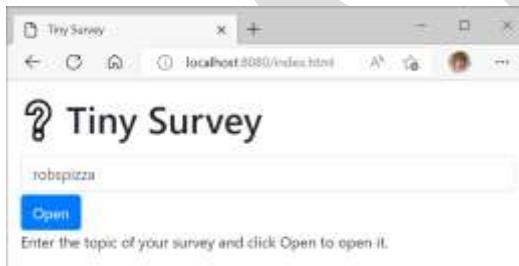


Figure 8.35 Ch-08_Fig_03 home page

Figure 8.3 shows the home page for the application. The HTML for this page is held in the [index.ejs](#) template file in the [views](#) folder.

```
<!DOCTYPE html>
<html>

<head>
  <title>Tiny Survey</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet"
    href="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/css/bootstrap.min.css"
    integrity="sha384-"
    ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUohcWr7x9JvoRxT2M Zw1T"
    crossorigin="anonymous">
```

```

</head>

<body>
  <div class="container">
    <h1 class="mb-3 mt-3">#10068; Tiny Survey</h1>
    <form action="/gotopic" method="POST">210
      <input type="text" name="topic" required class="form-control"
        spellcheck="false">
      <button type="submit" class="btn btn-primary mt-1">Open</button>
    </form>
    <p>
      Enter the topic of your survey and click Open to open it.
    </p>
  </div>
</body>

</html>

```

Above you can see the HTML for the index file. The Express server sends this file back to the browser when the browser asks for the endpoint “index.html”.

```

app.get('/index.html', (request, response) => {
  response.render('index.ejs');
})

```

Above is the Express route that deals with a request for the index.html page. The contents of `index.ejs` is rendered in response to the request. Now let’s look at what `index.ejs` does. The topic for the survey (`robspizza`) is entered into the page and sent to the server when the user clicks the Open button. We have already seen one way that a page can send data to the web server; we used the HTTP query mechanism in the Cheese Finder game to send the X and Y coordinates of a grid square to the game engine when the button in that square was clicked. Look at the section “Play the game” in Chapter 5 for details of how this works. We could use the same query mechanism to send the topic information to the application, but when we are reading input from the user the best way to do this is by using HTTP post.

²¹⁰ Input form

Post data from a form

A browser asks for content from a server using the `get` command. It sends content to the server using a `post` command. We specify a `post` action by creating a form in the web page. A form can contain multiple input elements, so you could read the name, address and age of a user in a single form. In the case of the index page for tiny survey we just need one input, the topic of the survey.

```
<form action="/gottopic" method="POST">
  <input type="text" id="topic" name="topic" required
    spellcheck="false">
  <button type="submit">Open</button>
</form>
```

Above you can see the HTML in `index.ejs` that defines a form containing a text input field and a button containing the text “Open” which will submit the form. The `action` attribute of the form specifies the server endpoint that will be used when the form is submitted. The form above sends the result to the `gottopic` endpoint. The `method` attribute is set to `POST` because that is what we will be using to transfer the data.

We first saw the `input` element in chapter 3 in the section “The html input element” when we needed to read an offset value for our time travel clock. In that application the input was read by JavaScript running inside the browser. In the code above the input is going to be transmitted to the server using the `post` action. The input element has both an `id` and a `name` attribute, both of which are set to the string “topic”. The `id` attribute is used within the HTML document to refer to the element. The `name` property gives the name this input will have when it is sent to the server. It makes sense for both these attributes to be set to the same string.

We have used buttons in our HTML pages before. This button has a `type` attribute which is set to “submit”. When the button is pressed the form is submitted by the browser. To do this the browser assembles a message containing the contents of the input fields encoded as a JSON string and sends it to the server in a `post` command. Now lets see how code in the server application can receive that input and do something with it.

Receive input from a post

We now know how to create an HTML page that can post a form of data to the server. Now we need to find out how we can write code in the server that responds to the post and receives the input from it. To do this we are going to use another piece of middleware. This middleware is part of the `Express` framework itself, so we don’t need to install anything new. We just need to

turn the middleware on. This piece of middleware takes incoming post requests, decodes the JSON in them and converts this into an object which can be used by the code dealing with the post message.

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

We've seen `app.use` before, it is how we told Express to use the `ejs` framework to render the pages to be sent to the browser. The statement above is enabling and configuring an Express component called `urlencoded`. This takes in post messages and converts them into objects. It is configured using an object literal that contains setting values. The only setting that we are using is "`extended:false`". The `urlencoded` middleware can do lots of clever things such as decode compressed data in a post. We don't want any of that and so we turn off the extended options. Once have added this middleware to the `Express` application something magical happens. The `request` object in our route now has a `body` property that contains the items in the form that was submitted.

```
// Got the survey topic
app.post('/gottopic', (request, response) => {211
  let surveyTopic = request.body.topic;212

  let survey = surveys.getSurveyByTopic(surveyTopic)213
  if (survey == undefined) {214
    // need to make a new survey
    response.render('enteroptions.ejs',215
      { topicName: request.body.topic, numberOfOptions: 5 });
  }
  else {216
    // enter scores on an existing survey
  }
})
```

²¹¹ Route for a post to `gottopic`

²¹² Get the survey topic from the post

²¹³ Find the survey for this topic

²¹⁴ If there is no survey – make one

²¹⁵ Select the page to get options

²¹⁶ This survey exists – select option

```
let surveyOptions = survey.getOptions();217  
response.render('selectoption.ejs', surveyOptions);218  
}  
});
```

If you look back at the HTML page that set up the Form element you will see that the action for this form was specified as `/gottopic`. Above is the post route in our server code that responds to this action. This is the point in the workflow where the server decides whether the user is making a new survey or selecting an option on an existing one. The code above uses the survey storage classes that we created in Chapter 7. The `Surveys` class provides a method called `getSurveyByTopic` which is given a survey topic (in our case “robspizza”) and will return a reference to that survey if it exists in the survey store. If the survey is not found the `getSurveyByTopic` method returns the value `undefined`. If the survey is found the application should allow the user to enter their options by sending them to the `selectoption.html` page. If the survey is not found the application will create a new survey with that name by sending the user to the `enteroptions.html` page.

Make Something Happen

Handle a post

You may have found the above explanation a bit confusing. Seeing the system in action will make it a lot clearer. The next few “Make Something Happen” exercises will use the completed application, so open that in Visual Studio Code. Now open the `tinysurvey.mjs` file.

²¹⁷ Get the options for display from this survey

²¹⁸ Select the page to get a selection

The screenshot shows the Visual Studio Code interface with the 'Survey' project open. The left sidebar displays the project structure with files like `trivsurvey.js`, `index.ejs`, and `surveys.js`. The main editor area shows the `trivsurvey.js` file containing Node.js code for a survey application using Express and MongoDB.

```
const app = express();
// Create an empty survey store
let surveys = new Surveys();
const port = 8080;
// Select ejs middleware
app.set('view-engine', 'ejs');
// Select the middleware to decode incoming posts
app.use(express.urlencoded({ extended: false }));
// Home page
app.get('/index.html', (request, response) => {
  response.render('index.ejs');
});
// Get the survey topic
app.post('/getTopic', (request, response) => {
  let surveyTopic = request.body.topic;
  let survey = surveys.getSurveyByTopic(surveyTopic);
  if (survey == undefined) {
    // need to make a new survey
    response.render('emptyTopic.ejs',
      { topic: surveyTopic, numberOfOptions: 3 });
  } else {
    // enter scores on an existing survey
  }
});
```

Ch08_inset06_01 tinysurvey.mjs application breakpoints

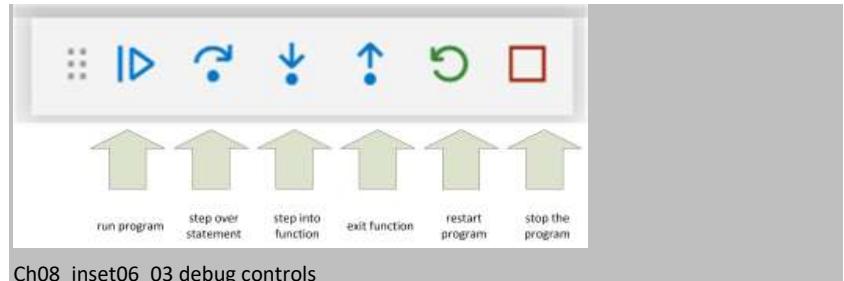
The **tinysurvey.mjs** file controls the application. Add a pair of breakpoints at lines 20 and 25 as shown above. One will be hit when `index.html` is requested by the browser. The other will be hit when the user posts the topic for the survey. Select the **Run and Debug** option from the debug menu and start the application. Open your browser and navigate to `localhost:8080/index.html`. You will notice that no page appears. Look back at Visual Studio to find out why.

```
18 // Home page
19 app.get('/index.html', (request, response) => {
20   response.render('index.ejs');
21 });


```

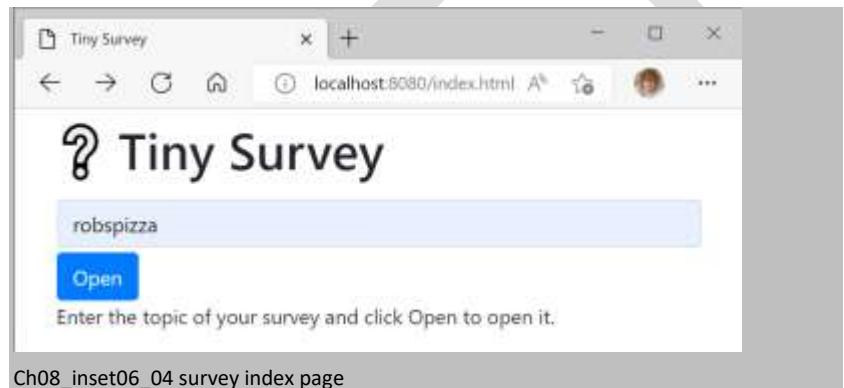
Ch08_inset06_02 index render breakpoint

The program has stopped at a breakpoint in `tinysurvey.mjs`. The browser has asked for the `index.html` file and the server is running the Express route for that endpoint. The breakpoint is on the statement that will open the `index.ejs` file and send it back to the browser.



Ch08_inset06_03 debug controls

Click the **run program** button in the debug controls to continue the program which will call the `render` function and serve out the page. Now look back at the browser and you will see that the index page has appeared.



Ch08_inset06_04 survey index page

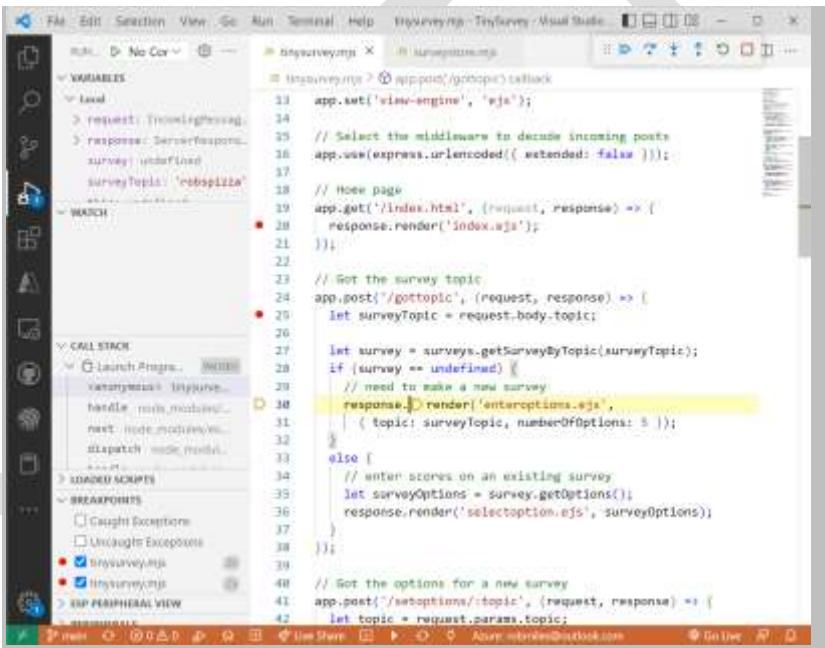
Enter the topic of the survey and click **Open**. The browser will appear to get stuck again. Move back to Visual Studio Code to investigate.

```
23 // Got the survey topic
24 app.post('/gottopic', (request, response) => {
25   let surveyTopic = request.body.topic;
26
27   let survey = surveys.getSurveyByTopic(surveyTopic);
28   if (survey == undefined) {
29     // need to make a new survey
30     response.render('enteroptions.ejs',
31       { topic: surveyTopic, numberofOptions: 5 });
32   }
33   else {
34     // enter scores on an existing survey
35     let surveyOptions = survey.getOptions();
36     response.render('selectoption.ejs', surveyOptions);
37   }
38 );
```

Ch08_inset06_05 got topic breakpoint

The application has stopped at a breakpoint in the function that services the `gottopic` route. The first thing that the function does is get the survey topic from the `body` of the request. The browser put this information into the post message. Click the **function** button to perform the statement. If you hover the mouse pointer over the `surveyTopic` variable you will see that it contains the topic that you entered into the index page. This is how code in the application gets values from a form in a web page.

Continue to click the **step into function** and step through the code. The `getSurveyByTopic` method in the survey storage is called to get a survey with the given topic. There is no survey with this topic (the program has just started) and so the `getSurveyByTopic` function will return undefined. This will cause the application to make a new survey.



Ch08_inset06_06 enter options statement

Above you can see that the program has reached statement 30. If you look in the top left hand corner in the **VARIABLES** area you will see that the variable `surveyTopic` holds the value "robspizza", which is what was entered onto the form. The server is about to render `the enteroptions.ejs` page which will build a form to read the 5 survey options from the user. Press the **run program** button in the debug controls continue running the server. Leave everything as it is for the next Make Something Happen.

We are following this path through the program because there is not an existing survey with the topic "robspizza". If he survey exists the `selectoption.ejs` page will be rendered.

If you are having problems with this exercise there are two things that are important. The

first is that if you start the program using the command `node tinysurvey.mjs` in the command line the program does not run within the debugger and so no breakpoints will be hit. The second thing is that you must have the `tinysurvey.mjs` file as the selected file in the Visual Studio Code editor before you use Run and Debug to run it. If you have another file selected, you will not get the correct run options.

This is a big step towards making the Tiny Survey application. In fact, this is a huge step in making any web-based application. You can now create a form to capture any information that you want to a user to enter and then process the data in a server-based application. If you wanted to capture more details from your user you just have to add more input elements to the form and give them `name` attributes to identify them. They will appear as properties of the `body` object which is a property of the request object passed to the function that handles the route. In the next section we will discover how the `enteroptions.ejs` page works.

Enter the survey options

We are implementing the pages behind our Tiny Survey application. We've created an index page which receives the topic for the survey. The index page contains a form which holds one text input that was used to set the topic for the survey. When the user presses the Open button the contents of that form was submitted to a post route called `settopic`. We've just seen how this works, and how the topic entered by the user can be picked out of the post from the browser. Now we need to generate a page for the user to enter the five items that we are choosing between. We have seen that this is done by a page called `enteroptions.ejs`. Let's take a look at what this produces.

The enteroptions page

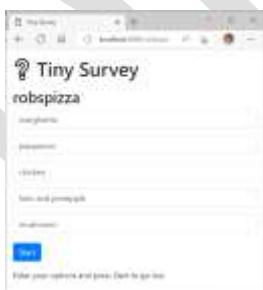


Figure 8.36 Ch-08_Fig_04 Enter Options page

Figure 8.4 above shows the enter options page generated from the `enteroptions.ejs` template. The user has entered the five options and when the `Start` button is clicked the survey will go live for users to select their preferences. The HTML that the browser should receive looks like this:

```

<h1>&#10068; Tiny Survey</h1>219
<h2>robspizza</h2>220
<form action="setoptions/robspizza" method="POST">221
  <input type="text" value="" name="option1">222
  <input type="text" value="" name="option2">
  <input type="text" value="" name="option3">
  <input type="text" value="" name="option4">
  <input type="text" value="" name="option5">
  <button type="submit">Start</button>223
</form>
<p>Enter your five options and press Start to go live.</p>

```

We could create an html file that contained the above code and it would work, but only for a survey called “robspizza” which contained five items. If I wanted a survey called “Robs Movies” I would need to generate a different page. In a perfect world we could generate the different pages automatically from the [enteroptions.eps](#) page. The great news is that we are going to do exactly that.

Generate pages using an eps template

This next part might hurt your head a little. It certainly did mine when I found out about it. But the powers you once you know how to do this are awesome. Before we look at the contents of the file [enteroptions.ejs](#) I want to remind you of the problem we are trying to solve. We want to make a page that looks like Figure 8.4 for our application. The page must contain the topic of the survey (robspizza) and input elements for five options. We have received the survey topic from the post to the [settopic](#) route and now we need to produce a page that contains input elements for the names of the options in the survey. Let’s look at the JavaScript that does this.

²¹⁹ Page heading

²²⁰ Name of topic

²²¹ Form for items

²²² Input for an item

²²³ Start button

```

    response.render('enteroptions.ejs', 224
    { topic: surveyTopic, 225
      numberofOptions: 5 });
}

```

My favorite way of solving a problem is to get someone else to do it. That is what this code does. The function render has two arguments. The first is the name of the template file (in this case enteroptions.js). The second argument is an object literal that contains two properties, the **topic** of the survey (which is loaded from the post that we received from the user) and the number of options (in this case 5). The `enteroptions.ejs` template uses these values to make an HTML page. Let's look at how it does this.

```

<!DOCTYPE html>
<html>

<head>
  <title>Tiny Survey Enter Options</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet"
  href="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/css/bootstrap.min.css"
  integrity="sha384-
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
</head>

<body>
  <div class="container">
    <h1 class="mb-3 mt-2">&#10068; Tiny Survey</h1>
    <h2>
      <%= topic %>226
    </h2>
    <form action="/setoptions/<%=topic%>" method="POST">
      <% for(let i=1;i<=numberofOptions;i++) { %>227
        <p>

```

²²⁴ Render build.ejs

²²⁵ Values to send to build.ejs

²²⁶ Display the topic in the heading

²²⁷ Loop to create options

```

<input type="text" required class="form-control"
spellcheck="false" value="" name="option<%=i %>">228
</p>
<% } %>229
<p>
    <button type="submit" class="btn btn-primary mt-1">Start</button>
</p>
</form>
<p>
    Enter your options and press Start to go live.
</p>
</div>
</body>

</html>

```

This is the [enteroptions.eps](#) page. At first glance it looks like ordinary HTML. But if you look closer you will find something interesting. There are pieces of JavaScript code in there too. This code runs as the HTML is being generated. The first time we see this is when the topic is being included in the page.

```

<h2 class="mb-2 mt-2">
<%= topic %>
</h2>

```

The html above puts the value of `topic` into the page text. This value was passed into the page when the render method was called. The value to be displayed is enclosed in `<%=` and `%>` delimiters which mean “include the value of this expression in the content of the page at this point”.

This is how we get the `robspizza` text into the heading on the page.

```

<% for(let i=1;i<=numberOfOptions;i++) { %>230
<p>

```

²²⁸ Counter in option name

²²⁹ End of loop

²³⁰ Loop to count through the options

```

<input type="text" required class="form-control" spellcheck="false"
       value="" name="option<%=i %>">231
</p>
<% } %>232

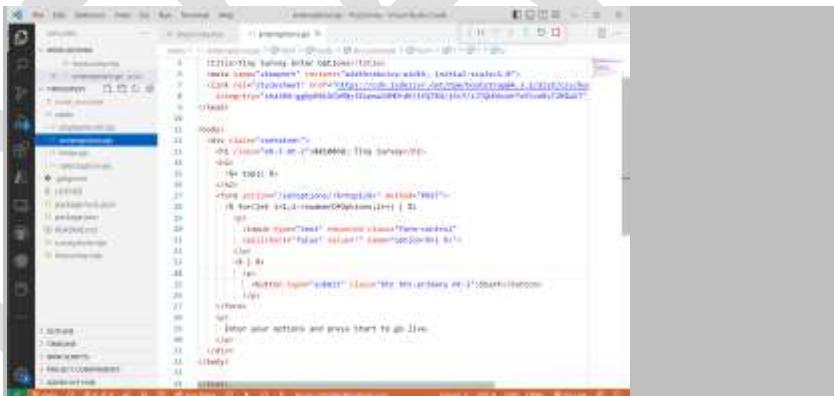
```

This loop runs when the page is being rendered. It creates as many input boxes as are required for the survey. Each input box will have a different name based on the word `option` followed by a count. The JavaScript statements are enclosed in `<%` and `%>` delimiters. The value of `numberOfOptions` was passed into the page when `render` was called in the same way as the page received the value of `topic`.

Make Something Happen

Rendering enterptions

Being able to run JavaScript code inside a page as it is being generated is very useful. But it takes a little while to get used to seeing HTML and JavaScript quite so close. So let's watch a page being rendered. We can continue working through the example application. Go back to Visual Studio code, clear the breakpoints in `tinysurvey.mjs` and open the file `enterptions.ejs` in the `views` folder.



Ch08_inset09_01 enterptions page

This page generates the input options. We can see how it works by making a change to the page and seeing the effect on the output. The page uses a JavaScript for loop to generate the

²³¹ Create the option name using i

²³² End of the loop

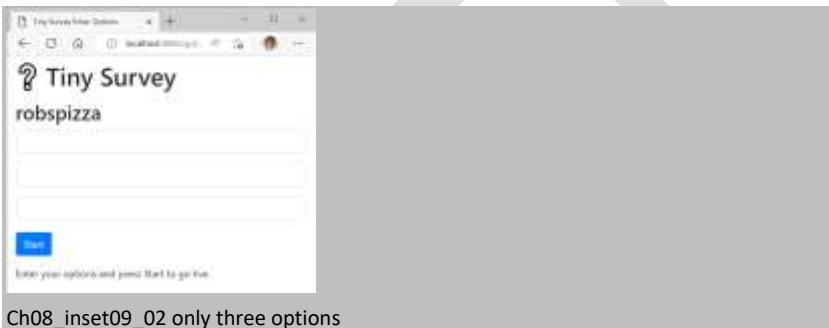
input elements on the page. The for loop is on line 18 of template.

```
<% for(let i=1;i<=number0fOptions;i++) { %>
```

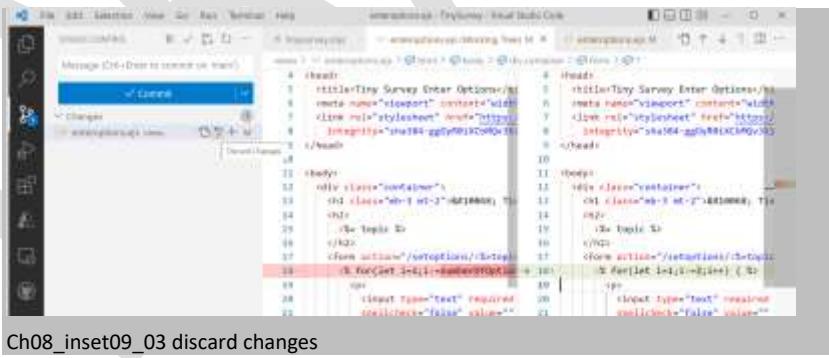
The loop is controlled by the value in `numberOfOptions`. We could change the number of options by changing this value. Edit the code so that instead of using the value in `numberOfOptions` the loop now uses the value 3:

```
<% for(let i=1;i<=3;i++) { %>
```

Save your updated version of `enterOptions.ejs` and run the application again. Enter a topic for the survey and you will see the options page has only three options.



Our change to the loop limit has affected the amount of HTML in the page. We don't want the number of options to be only three, so we should put `enterOptions.ejs` back to the original file. We can use the Source Control in Visual Studio Code to do this. Click the Source Control button in the left-hand button strip.



Source Control will show you that one file has changed: `enteroptions.ejs`. Click on the filename to open the file and you can see the changes that have been made. Now click the left-hand pointing arrow next to the filename to select **Discard Changes**. You will be asked if you want to discard the changes. If you do the changes will be removed. Changes that have been discarded in this way cannot be recovered. But the command can be useful if you've made a terrible mess of editing a file and you want to get back to how things were before.

CODE ANALYSIS

Rendering in Express

You may have some questions what we have just seen.

Question: What happens if the JavaScript in `enteroptions.ejs` contains errors?

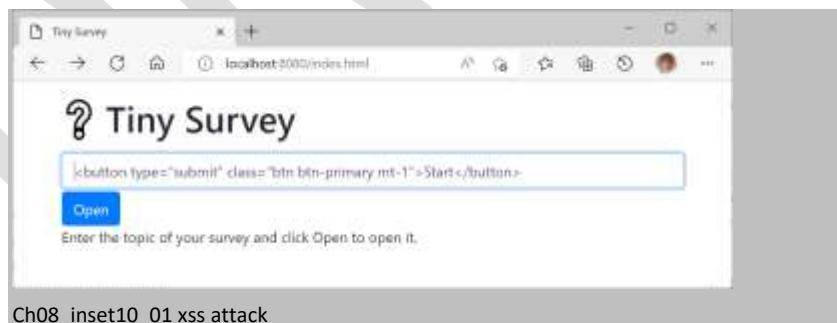
If you make a mistake in a JavaScript program you will get an error when it runs. The same is true with JavaScript in a template. The error will be displayed in the browser output and in Visual Studio Code. It will tell you the statement which caused the error.

Question: Can we use JavaScript code like this to validate input values entered into the web page?

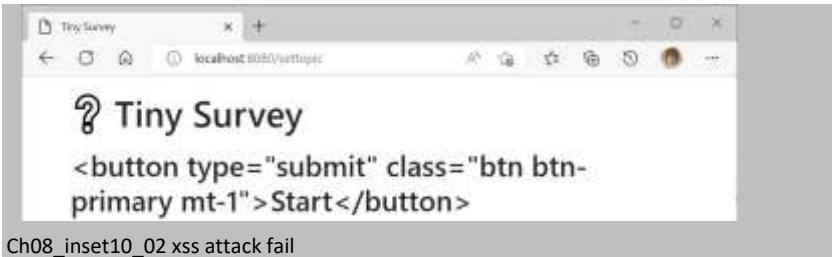
This is an interesting question. We've seen situations where we've put JavaScript into a web page to respond to inputs entered onto the web page. Can we do this here? It turns out the answer is no and understanding why not goes to the heart of understanding what these templates do. If you think about it; the JavaScript code in an ejs file only gets to run as the HTML text is generated. If you look at the HTML produced by the build page you won't find any JavaScript code in there. The code in the template did its job when it made the page. None of it is ever visible in the output. However, you could put JavaScript code into the eps page that is generated. This could validate inputs.

Question: Can a user of Tiny Survey upset the application by typing in an HTML command as the topic name?

Another good question. There's a web site hacking technique called Cross Site Scripting (XSS) where you try to upset an application by typing HTML (or other) commands. Consider the following:



The user of the survey is trying to be sneaky. They've noticed that the `enteroptions.eps` file just includes the topic text directly into the HTML it generates, and they fancy making another button appear on the page. The good news is that this won't work for our application.



Ch08_inset10_02 xss attack fail

The survey has a strange title, but we don't get extra buttons appearing on our page. This is because the `<%=` operation in eps "HTML escapes" any text before it displays it. In other words, characters that would be interpreted as commands by the browser are converted into escaped versions:

```
&lt;button type=&#34;submit&#34; class=&#34;btn btn-primary mt-1&#34;&gt;Start&lt;/button&gt;
```

Above you can see the text that is placed in the web page. You can see that the `<` character which would normally start an HTML element definition has been converted into `<`, as have all the other dangerous characters. In this case the eps framework has taken care of issues with this form of attack, but you should treat every user input as potentially hostile and make sure that it is "sanitized".

Use named route parameters

We now have a good part of the application working. A user can enter a topic and five options which are sent back to the server to create a survey. We do have another problem to address though because of the way that the server doesn't remember the identity of any web request.

The server needs to know it is receiving options for the "robspizza" survey. How can we tell it this? The answer turns out to be quite simple. We can add the topic name to the url that will receive the post when the options are submitted.

```
<form action="setoptions/<%=topic%>" method="POST">
```

Above you can see the definition of the form that contains the options for the survey. It contains an action attribute which specifies the endpoint where the form will be submitted. The code above adds the topic name to the end of this entry point. It gets the topic name from the `topic` value supplied to the template. This means that the form will be sent to "setoptions/robspizza". We can now modify the route so that the topic name can be picked up.

```
app.post('/setoptions/:topic', (request, response) => {
  let topicName = request.params.topic;
  // handle the rest of the response
})
```

Above you can see how this works. The name of the route is now '[/setoptions/:topic](#)'. This is called a "named route parameter". When the incoming post arrives the Express framework splits the `topic` element off endpoint name and copies it into a `request.params.topic` variable. The JavaScript dealing with the incoming post can now use this to build a survey object which will be held on the server.

Build a survey data object

We are working through the workflow of our Tiny Survey, viewing each step as we go. We are at the point where the user has entered the options for the survey in a form produced by the page [enteroptions.eps](#) and clicked the Start button to post the form back to the server. The server has received the form, along with a named route parameter that gives the topic for the survey. Now we need an object to hold the survey. The object will be built from the topic name and the five option names.

```
app.post('/setoptions/:topic', (request, response) => {
  let topicName = request.params.topic;233
  let options = [];234
  let optionNo = 1;235
  do {236
    // construct the option name
    let optionName = "option" + optionNo;237
    // fetch the text for this option from the request body
    let optionText = request.body[optionName];238
```

²³³ Get the topic of the survey

²³⁴ Create an options array

²³⁵ Start counting options at 1

²³⁶ Start of option loop

²³⁷ Add the option number to the word "option"

²³⁸ Get the option property from the body

```

// If there is no text - no more options
if (optionText == undefined) {239
    break;
}
// Make an option object
let option = new Option({ text: optionText, count: 0 });
// Store it in the array of options
options.push(option);
// Move on to the next option
optionNo++;
} while (true);

// Build a survey object
let survey = new Survey({ topic: topic, options: options });

// save it
surveys.saveSurvey(survey);

// Render the survey page
let surveyOptions = survey.getOptions();
response.render('selectoption.ejs', { surveyOptions });
}

```

This is the most complicated piece of code in the application. The inputs to the code are the `topic` (which we got from the named route) and a `request.body` object that contains the options for our survey in the form of properties called `option1`, `option2`, `option3`, `option4` and `option5`. The code above contains a `do – while` loop which works through the properties in `request.body` starting with a property called `option1` and going on to try to find a property called `option6` which is undefined (we only have five options) and causes the loop to exit. The code was written this way because I didn't want to fix the number of options at 5 in the program. Experience with people changing their minds about the number of cheeses in cheese finder has left me thinking that it is useful to be flexible about some parts of your program. Each time round the loop a new option object is created by the following statement:

```
let option = new Option({ text: optionText, count: 0 });
```

²³⁹ Exit if no option for this name

The `option` object is initialized with an object containing the text of the option (in my case margherita, pepperoni etc) and the count of responses for that option (starting at 0). Each object is added to a list of options:

```
options.push(option);
```

We created the `Option` class in chapter 7 to hold the text and the count of an option in the survey. We import this class, along with `Survey` and `Surveys` at the start of the `tinysurvey.mjs` file. The list of options is used to create a survey object which also contains the topic name. This is performed after the loop has finished.

```
let survey = new Survey({ topic: topic, options: options });
```

Once we have created the survey we can store it.

```
surveys.saveSurvey(survey);
```

Make Something Happen

Build a survey data object

Now let's have a look at the process of building a data object. Put a breakpoint at line 42 in the `tinysurvey.mjs` file, start the program running and enter a new survey topic followed by the survey option. When you click Start in the enteroptions page the program will hit the breakpoint.

```
if (request.method === 'POST') {
  if (request.url === '/createSurvey') {
    // need to make a new survey
    response.render('newSurvey.ejs', {
      topics: surveyTopics, numOptions: 5
    });
  } else {
    // user comes in on an existing survey
    let surveyOptions = survey.getOptions();
    response.render('selectTopic.ejs', {surveyOptions});
  }
}

// Get the options for a new survey
app.post('/newSurvey/:topic', (request, response) => {
  let topic = request.params.topic;
  let options = [];
  let optionName = '';
  do {
    // construct the option name
    let optionName = `option${options.length}`;
    // fetch the text for this option from the request body
    let optionText = request.body[optionName];
    // If there is no text - no more options
    if (optionText === undefined) {
      break;
    }
    options.push({
      name: optionName,
      text: optionText
    });
  } while (true);
  response.render('newSurvey.ejs', {topics: surveyTopics, options: options});
});
```

The breakpoint will be hit after you have entered the survey options and clicked the Start button. The Variables view on the top left show the values that are being used to create the option. You can step through the code and watch as the values are fetched from the form data and used to build a `Survey` object which is then stored in `surveys`. When you have finished stepping through the code you can remove the breakpoint and press the run program button in the debug controls to resume program operation.

Build a select option page

We now have our survey. And we know how we can create a web application that can accept information from the user and store it in an application running in the cloud. The next thing we need to make is a page where the users select their preferred option by clicking a radio button.

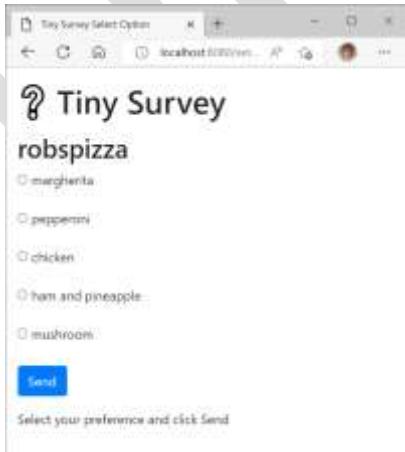


Figure 8.37 Ch-08_Fig_05 Option voting page

Figure 8.5 above shows how the page looks. This page is displayed during the creation of a survey and if a user selects a topic of a survey which already exists. The page will be created from another template:

```
let surveyOptions = survey.getOptions();
response.render('selectoption.ejs', surveyOptions);
```

The statement above calls the `render` function to render the page from the `selectoption.ejs` template file. The `render` function is passed a `surveyOptions` object which contains the data needed to build the page. The `surveyOptions` object contains two things, the topic for the survey and a list of the option texts to be displayed.

```
let surveyOptions = survey.getOptions();
```

The `tinysurvey` application can ask a survey to give it the survey options by calling the `getOptions` method.

```
getOptions() {
  let options = [];240
  this.options.forEach(option => {241
    let optionInfo = { text: option.text };242
    options.push(optionInfo);243
  });
  let result = { topic: this.topic, options: options };244
}
```

²⁴⁰ Make an empty options array

²⁴¹ Work through all the options

²⁴² Make an object that contains the option text

²⁴³ Add the object to the array

²⁴⁴ Create the result object

```
    return result;245  
}
```

Above you can see the `getOptions` method from the `Survey` class. It creates an object which contains just the option information for use in the `selectoption` page.

```
<!DOCTYPE html>  
<html>  
  
<head>  
  <title>Tiny Survey Select Option</title>  
  <meta name="viewport" content="width=device-width, initial-scale=1.0">  
  <link rel="stylesheet"  
  href="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/css/bootstrap.min.css"  
        integrity="sha384-gg0yR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"  
        crossorigin="anonymous">  
</head>  
  
<body>  
  <div class="container">  
    <h1 class="mb-3 mt-3">#10068; Tiny Survey</h1>  
    <h2>  
      <%=topic%>246  
    </h2>  
    <form action="/recordselection/<%=topic%>" method="POST">  
      <% options.forEach(option=> { %>247  
        <p>  
          <input type="radio" name="selections" id="<%=option.text%>"  
                value="<%=option.text%>">248  
          <label for="<%=option.text%>">  
            <%=option.text%>249  
        </p>  
      <% } %>  
    </form>  
  </div>  
</body>
```

²⁴⁵ Return the result object

²⁴⁶ Display the topic of the survey

²⁴⁷ Loop through the options

²⁴⁸ Radio button

²⁴⁹ Add the option text to the radio button

```

        </label>
    </p>
<% }) %>
    <p>
        <button type="submit" class="btn btn-primary mt-1">Send</button>
    </p>
</form>
<p>
    Select your preference and click Send
</p>
</div>
</body>

</html>

```

The `selectoption.ejs` template is like the `enteroptions.ejs` template that we used to enter the survey options, except that rather than making a list of input boxes (for users to enter the options) it makes radio buttons (for users to vote). If you compare this with the `selectoption.ejs` template there are some differences. The `selectoption` template uses a `for` loop to count through option numbers and create an option name for each one. The code in the `selectoption.js` template works through each option in the `options` list of the survey details using a `foreach` loop. When the `Send` button is clicked the form is set to the `recordselection` endpoint in the server. The endpoint has the `topic` of the survey appended so that the `post` handler in the server can use a named route to get the name of the survey the results are for.

```

<form action="/recordselection/robspizza" method="POST">250
    <p>
        <input type="radio" name="selections" id="margherita" value="margherita">
        <label for="margherita">
            margherita
        </label>
    </p>
    <p>
        <input type="radio" name="selections" id="pepperoni" value="pepperoni">
        <label for="pepperoni">
            pepperoni
        </label>
    </p>

```

²⁵⁰ Named route using the topic

```

<p>
  <input type="radio" name="selections" id="chicken" value="chicken">
  <label for="chicken">
    chicken
  </label>
</p>
<p>
  <button type="submit" class="btn btn-primary mt-1">Send</button>251
</p>
</form>

```

The code above shows the form that is created for the pizza survey. The browser groups the radio buttons together because they all have the same name attribute. When one of the buttons is selected all the others are cleared. When the `Send` button is clicked the browser sends the value of the radio button. If the user clicks `Send` with the chicken button selected the form will post the value “chicken” to the application. The form will be posted to the `/recordselection/robspizza` endpoint.

CODE ANALYSIS

Rendering options

You may have some questions what we have just seen.

Question: What just happened?

You may be finding this hard to understand. The key to understanding what is happening is to go back and think about what the code has to do. The application needs to create a web page which contains a radio button for each of the survey options. The `selectoption.ejs` template contains the HTML definition of one radio button along with a JavaScript loop that produces a radio button for each option. To do this the `selectoption.ejs` template needs to know the topic of the survey (which it will display and use as a named endpoint for the response) and the text of each of the options. This information is supplied to the template in the form of an object that contains a `topic` property and an `options` property.

This object is generated by calling the `getOptions` method on a `Survey` instance. This creates an object which contains the information to be passed into the `selectoptions.ejs` page. As we discover how more of the application pages work we are going to see a pattern where the application creates a lump of information which is passed into the page to tell it what to do.

²⁵¹ `Send` button

When we look at the results page (which we will see next) we will see the same pattern at work.

Question: Why don't we just pass the survey object into the `enteroptions` page?

An instance of a `Survey` object contains all the data that is needed by the `selectoption.ejs` template to build the page. But the code sends the result of a call of `getOptions` into the `selectoption.ejs` page instead. Why not pass the survey? This is because if I give something a reference to a survey object they can do all kinds of things with it, including change the counts for options or even the topic name. But if I pass a copy of the data that is required, there is no way this can happen. This is what I call *defensive programming*.

Question: Is it OK to use text entered by the user as attribute names in the HTML?

If you look at the code above you will notice that the radio button called chicken has name and id attributes which are also "chicken". As far as HTML is concerned these are just strings of text. There is no need for them to obey any particular rules for variable names in programs.

Question: What would happen if two options had the same text?

The user might create two options for "pepperoni". What would happen? You might like to try this. The program will work OK, but votes for the second version of the option will be counted as being for the first. A later version of the application could check for repeated option text and produce an error.

When the user clicks `Send` in the `selectoption` page the form posts a result value to the `recordselection` endpoint in the application. This endpoint needs to increment the counter of the selected option and then display the results page.

Record survey responses

```
// Got the selections for a survey
app.post('/recordselection/:topic', (request, response) => {
  let topic = request.params.topic;252
  let survey = surveys.getSurveyByTopic(topic);253
  if (survey == undefined) {254
```

²⁵² Get the topic out of the request

²⁵³ Find the survey with this topic

²⁵⁴ Check if the survey exists

```

        response.status(404).send('<h1>Survey not found</h1>');255
    }
} else {
    let optionSelected = request.body.selections;256
    survey.incrementCount(optionSelected);257
    let results = survey.getCounts();258
    response.render('displayresults.ejs', results);259
}
);

```

Above is the post route in the server that records a selection. It finds the survey option which has been selected, adds one to the count for that option and renders the results page. If the survey is not found (there is no survey matching the supplied topic name) the code generates a 404 (page not found) error response. Otherwise, it updates the count and then uses the [results.ejs](#) template to render the results. We can step through this code and watch it run.

Make Something Happen

Record a response

Now let's have a look at the process of recording a response. Put a breakpoint at line 75 in the [tinysurvey.mjs](#) file, start the program running and enter a new survey topic followed by the survey options. Then vote for your favorite pizza topping. When you click Send in the [selectionoption](#) page the program will hit the breakpoint in the Express handler for that route.

²⁵⁵ Display an error if the survey is not found

²⁵⁶ Get the selection

²⁵⁷ Increment the count for the selection

²⁵⁸ Get the survey counts

²⁵⁹ Display the counts

```

    // set the selection for a survey
    app.post('/recordselection/:topic', function(req, res) {
      let topic = req.params.topic;
      let survey = surveys.getSurvey(topic);
      if (survey === undefined) {
        res.status(404).send('404');
      } else {
        let optionSelected = req.query.options[0];
        survey.incrementCount(optionSelected);
        let results = survey.getCounts();
        res.render('displayresults.ejs', { results });
      }
    });
  }
}

```

Ch08_inset13_01 record response route

You can repeatedly click “Step into function” button in the debug controls to watch the code get the topic of the survey, find that survey, get the selected option from the response and then increment the counter for that response. The program running above has been stepped through to line 84 and is showing the contents of the results object which is being sent to the [displayresults](#) page.

Render the results

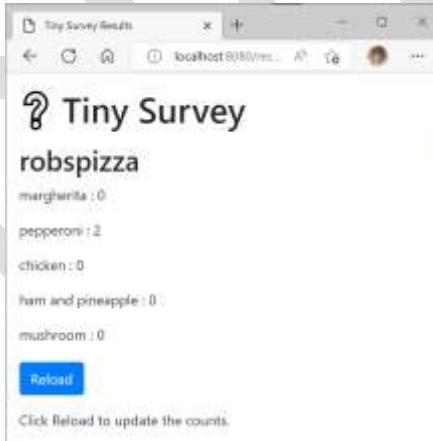


Figure 8.38 Ch-08_Fig_06 Results page

The final page that we need is the one that renders the counts for each option. The [displayresults.ejs](#) template does this. You can see the output that we want in Figure 8.6. The render function that uses the template is supplied with the results object from which the counts will be

obtained.

```
response.render('displayresults.ejs', results);
```

You should be noticing by now that all the pages that deal with the survey contents are very similar. They contain a loop which works through the options and outputs HTML elements that contain element properties. In this case the properties that are used are `topic` of the survey and the `text` and `count` properties of each option.

```
<!DOCTYPE html>
<html>

<head>
  <title>Tiny Survey Results</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet"
    href="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/css/bootstrap.min.css"
    integrity="sha384-gg0yR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
    crossorigin="anonymous">
</head>
</head>

<body>
  <div class="container">
    <h1 class="mb-3 mt-3">Tiny Survey</h1>
    <h2 class="mb-10 mt-2">
      <%=topic%>260
    </h2>
    <% options.forEach(option=> { %>261
      <p>
        <%= option.text%> : <%= option.count %>262
      </p>
    <% }) %>
```

²⁶⁰ Topic of the survey

²⁶¹ Loop for each option

²⁶² Option text and count value

```

<p>
    <button class="btn btn-primary mt-1"
    onclick="window.location.href='/displayresults/<%=topic%>'> Reload
    </button>
</p>
</form>
<p>
    Click Reload to update the counts.
</p>
</div>
</body>

</html>

```

The page also contains a button that can be used to request a refresh of the count display. When this button is pressed it must open an endpoint that will re-render the results page.

```
<button onclick="window.location.href='/displayresults/<%=topic%>'>
    Reload </button>
```

The Reload button above opens the `displayresults` endpoint when it is clicked. This location is supplied with the topic name so that the server can use a named route to determine which results to display.

```
// Get the results for a survey
app.get('/displayresults/:topic', (request, response) => {
    let topic = request.params.topic;263
    let survey = surveys.getSurveyByTopic(topic);264
    if (survey == undefined) {
        response.status(404).send('<h1>Survey not found</h1>');
    }
    else {
```

²⁶³ Get the topic

²⁶⁴ Find the survey

```
let results = survey.getCounts();265  
response.render('displayresults.ejs', results);266  
}  
});
```

The code that handles the `displayresults` endpoint is shown above. It is similar to the code that deals with the `recordselection` endpoint, except that it doesn't update any counts.

What you have learned

In this chapter we have created a fully function server-based application. We now know how data is passed between the browser and the server and how to use Express to generate webpages which are customized by code running inside them. Whenever you use a web page from now on you can start to see how it is made to work. The Express framework is only one of many that are used to create web applications, but you now understand a lot of the fundamental principles that underpin how it works. Here's the usual recap plus some points to ponder.

- You can use the Bootstrap stylesheets to improve the appearance of your web pages. A page that uses Bootstrap just contains a reference to a stylesheet file which is served by Bootstrap rather than a local file.
- The node package manager (npm) program is provided as part of a node.js installation. The company behind npm hosts a library of resources which can be incorporated into node applications.
- The resource used by an application are itemized in the dependencies section of the `package.json` file that describes the application. The `npm` program can be controlled via the terminal. The `npm init` command creates an empty `package.json` file and `npm install` is used to install resources. If npm installs a resource that needs other resources to work, these are automatically installed as well.
- You can create a git repository using the terminal command `git init` (as long as you have installed git on your computer). Git is integrated into Visual Studio Code. You can use the Source Control window to view changed files and commit changes. You can add a `gitignore` file to a project to provide patterns to identify files which are not to be committed into a

²⁶⁵ Get the counts

²⁶⁶ Display them

repository. These are files which are part of libraries and not created specifically for the application.

- The Express framework can be used to create an application server that runs under [node.js](#). The server code assigns functions to Express “routes” which equate to endpoints that the browser hits. An express route receives a request object describing the request from the server and populates a reply object with items to be sent back to the browser.
- Frameworks such as Express can host *middleware* elements that add functionality to an application. One such framework is ejs (Embedded JavaScript templates). This allows Express to render HTML files which contain active JavaScript components that can generate HTML elements to be incorporated into the page being generated.
- A FORM element in an HTML page contains INPUT elements into which the user of the page can enter data values. The FORM can also contain a button to trigger a submit action by the form. If the submit method is set to POST the form browser will send an HTTP POST to the server to deliver the values entered into the form. The server can use the urlencoded middleware to decode the information in the post and add it to the request object to be processed by the route handler for the post.
- We can use the debugger in Visual Studio Code to watch an application run and view the contents of variables.
- Within an ejs template file the delimiters `<%` and `%>` mark the start and end of JavaScript code sections. If a template contains a JavaScript loop the elements of HTML inside the loop are output into the page each time the loop goes round. The delimiters `<%=` and `%>` are used to enclose JavaScript expressions. When the page is rendered the value of the expression is placed at that point in the page.
- The call to ejs to render a template can include a reference to an object which contains values to be used by the JavaScript running inside the template page.
- A post request from a web page can add a named route parameter to the end of the request. This allows the browser to send information to the server. The route running in the server can then extract this name. We used this in Tiny Survey so that a route could contain the topic of the survey.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions".

Do the large stylesheet files loaded from Bootstrap increase web traffic and slow down web sites?

Bootstrap stylesheets improve the appearance of web pages and also allow them to adapt to the device they are being displayed on. The stylesheet files are quite large, but a browser will automatically cache them on the host computer (i.e. keep local copies) so that using them has very little impact on page performance and network use.

What is the difference between node.js and node package manager?

Node.js is the framework which runs JavaScript code on a machine. Node package manager is a separate program which manages the packages in an application. It uses [the package.json](#) file for the application to keep track of the resources used by a particular application.

How does middleware work?

When the application starts running it binds the middleware in by calling an [app.use](#) function to specify the point in the process where the middleware is to be used and the middleware to import.

What is the difference between a route and an endpoint?

The endpoint is specified by a url. It specifies the address of the server and a path to the resource provided by the server, for example <http://localhost:8080/index.html> the server on port 8080 of the local machine and the path is index.html. The route is a mapping by Express of a path (for example index.html) to the JavaScript function that will run to deal with the request and generate a response.

What is the difference between a GET and a POST?

The HyperText Transfer Protocol (HTTP) describes commands that can be sent from the browser to the server. The GET command means “please send me this page please”. The POST command means “here is a chunk of data”. A GET is sent when the user goes to a page. A POST is sent when the user submits a form.

The Cheese Finder application generated a web page by adding elements to the Document Object Model. The Tiny Survey generates web pages using ejs templates. Which is better?

When we created the Cheese Finder application we needed to make 100 buttons (one for each square on the board). We did this by running JavaScript inside the browser which created HTML button elements and added them to the Document Object Model (DOM) that the browser uses to manage the contents of the page that is being displayed to the user. The buttons were not downloaded from the server. The Tiny Survey application works differently. It uses JavaScript running on the server inside the ejs page templates to generate HTML elements in the page being sent to the browser. This is an important distinction. For Cheese Finder the work to make the page is done by the browser, but for Tiny Survey the

work is done by the server. So, which is best? Using the browser (Cheese Finder) might reduce the loading on the server. Using the server (Tiny Survey) means that you can send the page to a browser that doesn't support JavaScript (although most do). It is up to the designers of the application to choose their approach. There are other frameworks which share the work between browser and server in different ways.

Chapter 9: Turn professional

What you will learn

You might be wondering what separates a student or hobbyist program from a "professional" one. The only thing that makes "professional" code different is the fact that someone has paid money for it. This changes the relationship between the user and the author. The user is now a customer and has a right to expect a certain level of quality in the product they have purchased. The problem is that, from the outside, it is very hard to tell if that quality is present. It is only when the software fails, runs slowly, or proves to be impossible to modify that the lack of quality shows up.

In this chapter you will learn techniques to make your applications more worthy of being paid for. You'll discover how to break applications into individual modules that are documented, tested, and contain error handling and logging. Along the way we'll meet a new JavaScript hero in the form of the Exception and discover how to use cookies to retain the state of an application in the browser. It's going to be fun.

This is the point in this section where I remind you about the glossary. I'd hate to disappoint you, so don't forget about the glossary.

Modular code

One way to make our code more “professional” is by breaking it into separate modules. The idea behind this is that modules can be individually tested and interchanged without affecting others. The Tiny Survey application is somewhat modular, in that the code that manages survey storage is in a separate file from the main program, but we can improve this by creating a helper that implements a well-defined interface between the application and its data storage needs.

```
import { Survey, Surveys } from './surveystore.mjs'

class SurveyManager{

    constructor(){267
        this.surveys = new Surveys();
    }

    storeSurvey(newValue){268
        let survey = new Survey(newValue);
        this.surveys.saveSurvey(survey);
    }

    incrementCount(incDetails){269
        let topic = incDetails.topic;
        let option = incDetails.option;
        let survey = this.surveys.getSurveyByTopic(topic);
        survey.incrementCount(option);
    }

    surveyExists(topic){270
        return this.surveys.getSurveyByTopic(topic) != undefined;
    }

    getCounts(topic){271

```

²⁶⁷ Constructor for the controller

²⁶⁸ Store a survey

²⁶⁹ Increment an option count

²⁷⁰ Check if a survey exists

²⁷¹ Get the count values for a survey

```

        let survey = this.surveys.getSurveyByTopic(topic);
        return survey.getCounts();
    }

    getOptions(topic){272
        let survey = this.surveys.getSurveyByTopic(topic);
        return survey.getOptions();
    }
}

export { SurveyManager } ;

```

The `SurveyManager` class above provides all the survey storage functions for the Tiny Survey application. It uses the `SurveyStore` classes to store the data. The `SurveyManager` class adds some new functionality. It has a `surveyExists` method that tests for the existence of a survey with a particular topic. The Tiny Surveys application imports this class, creates an instance and then uses that for all its survey storage needs.

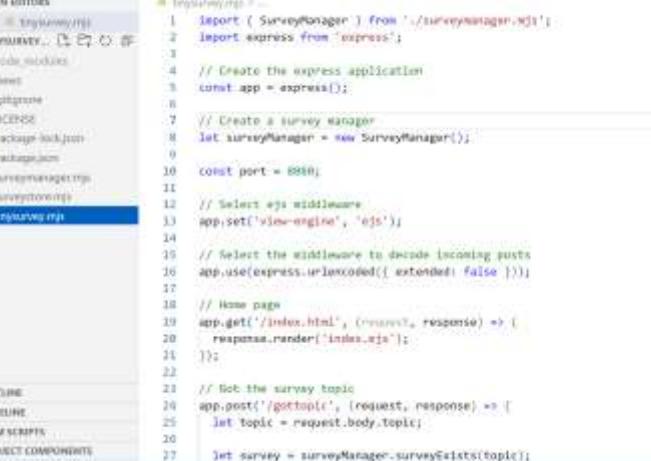
```
// Create a survey manager
let surveyManager = new SurveyManager();
```

Make Something Happen

Use the SurveyManager class

The starting point for this Make Something Happen is a version of Tiny Survey you can download from GitHub here: <https://github.com/Building-Apps-and-Games-in-the-Cloud/TinySurveyManager> Clone this repository onto your machine and open it with Visual Studio Code. If you are unsure how to get the sample application, use the sequence described in the section **Get the example application** in Chapter 8. Just change the address of the repository that you clone. Open the application in Visual Studio Code and then open the `tinySurvey.mjs` source file.

²⁷² Get the options for a survey



The screenshot shows the Microsoft Visual Studio Code interface with the following details:

- File Explorer:** On the left, it shows the project structure:
 - OPEN EDITORS: surveyManager.js
 - surveyManager.js
 - surveyManager
 - index.js
 - node_modules
 - client
 - gitignore
 - package.json
 - package-lock.json
 - README.md
 - surveyManager
 - surveys
- Code Editor:** The main area displays the content of surveyManager.js:

```
1 import { SurveyManager } from './SurveyManager';
2 import express from 'express';
3
4 // Create the express application
5 const app = express();
6
7 // Create a survey manager
8 let surveyManager = new SurveyManager();
9
10 const port = 8080;
11
12 // Select ejs as the view engine
13 app.set('view-engine', 'ejs');
14
15 // Select the middleware to decode incoming posts
16 app.use(express.urlencoded({ extended: false }));
17
18 // Home page
19 app.get('/index.html', (request, response) => {
20   response.render('index.ejs');
21 });
22
23 // Get the survey topic
24 app.post('/getTopic', (request, response) => {
25   let topic = request.body.topic;
26
27   let survey = surveyManager.surveyExists(topic);
28
29   if (survey)
30     response.send(survey);
31   else
32     response.send('Survey does not exist');
33 });
34
35 // Set the survey topic
36 app.put('/setTopic', (request, response) => {
37   let topic = request.body.topic;
38
39   let survey = surveyManager.surveyExists(topic);
40
41   if (survey)
42     surveyManager.setSurvey(topic);
43   else
44     surveyManager.createSurvey(topic);
45
46   response.send(`Survey ${topic} has been ${topic === surveyManager.surveyExists(topic) ? 'updated' : 'created'}`);
47 });
48
49 // Delete the survey topic
50 app.delete('/deleteTopic', (request, response) => {
51   let topic = request.body.topic;
52
53   let survey = surveyManager.surveyExists(topic);
54
55   if (survey)
56     surveyManager.deleteSurvey(topic);
57   else
58     response.send(`Survey ${topic} does not exist`);
59
60   response.send(`Survey ${topic} has been deleted`);
61 });
62
63 // Get all surveys
64 app.get('/surveys', (request, response) => {
65   let surveys = surveyManager.getAllSurveys();
66
67   response.send(surveys);
68 });
69
70 // Get survey by ID
71 app.get('/surveys/:id', (request, response) => {
72   let id = request.params.id;
73
74   let survey = surveyManager.getSurvey(id);
75
76   if (survey)
77     response.send(survey);
78   else
79     response.send(`Survey ${id} does not exist`);
80
81   response.send(`Survey ${id} has been found`);
82 });
83
84 // Create a new survey
85 app.post('/createSurvey', (request, response) => {
86   let survey = request.body.survey;
87
88   surveyManager.createSurvey(survey);
89
90   response.send(`Survey ${survey.name} has been created`);
91 });
92
93 // Update an existing survey
94 app.put('/updateSurvey', (request, response) => {
95   let survey = request.body.survey;
96
97   surveyManager.updateSurvey(survey);
98
99   response.send(`Survey ${survey.name} has been updated`);
100 });
101
102 // Delete a survey
103 app.delete('/deleteSurvey', (request, response) => {
104   let survey = request.body.survey;
105
106   surveyManager.deleteSurvey(survey);
107
108   response.send(`Survey ${survey.name} has been deleted`);
109 });
110
111 // Get survey by name
112 app.get('/surveysByName/:name', (request, response) => {
113   let name = request.params.name;
114
115   let survey = surveyManager.getSurveyByName(name);
116
117   if (survey)
118     response.send(survey);
119   else
120     response.send(`Survey ${name} does not exist`);
121
122   response.send(`Survey ${name} has been found`);
123 });
124
125 // Get survey by category
126 app.get('/surveysByCategory/:category', (request, response) => {
127   let category = request.params.category;
128
129   let surveys = surveyManager.getSurveysByCategory(category);
130
131   response.send(surveys);
132 });
133
134 // Get survey by question
135 app.get('/surveysByQuestion/:question', (request, response) => {
136   let question = request.params.question;
137
138   let surveys = surveyManager.getSurveysByQuestion(question);
139
140   response.send(surveys);
141 });
142
143 // Get survey by answer
144 app.get('/surveysByAnswer/:answer', (request, response) => {
145   let answer = request.params.answer;
146
147   let surveys = surveyManager.getSurveysByAnswer(answer);
148
149   response.send(surveys);
150 });
151
152 // Get survey by date
153 app.get('/surveysByDate/:date', (request, response) => {
154   let date = request.params.date;
155
156   let surveys = surveyManager.getSurveysByDate(date);
157
158   response.send(surveys);
159 });
160
161 // Get survey by status
162 app.get('/surveysByStatus/:status', (request, response) => {
163   let status = request.params.status;
164
165   let surveys = surveyManager.getSurveysByStatus(status);
166
167   response.send(surveys);
168 });
169
170 // Get survey by rating
171 app.get('/surveysByRating/:rating', (request, response) => {
172   let rating = request.params.rating;
173
174   let surveys = surveyManager.getSurveysByRating(rating);
175
176   response.send(surveys);
177 });
178
179 // Get survey by user
180 app.get('/surveysByUser/:user', (request, response) => {
181   let user = request.params.user;
182
183   let surveys = surveyManager.getSurveysByUser(user);
184
185   response.send(surveys);
186 });
187
188 // Get survey by location
189 app.get('/surveysByLocation/:location', (request, response) => {
190   let location = request.params.location;
191
192   let surveys = surveyManager.getSurveysByLocation(location);
193
194   response.send(surveys);
195 });
196
197 // Get survey by language
198 app.get('/surveysByLanguage/:language', (request, response) => {
199   let language = request.params.language;
200
201   let surveys = surveyManager.getSurveysByLanguage(language);
202
203   response.send(surveys);
204 });
205
206 // Get survey by platform
207 app.get('/surveysByPlatform/:platform', (request, response) => {
208   let platform = request.params.platform;
209
210   let surveys = surveyManager.getSurveysByPlatform(platform);
211
212   response.send(surveys);
213 });
214
215 // Get survey by device
216 app.get('/surveysByDevice/:device', (request, response) => {
217   let device = request.params.device;
218
219   let surveys = surveyManager.getSurveysByDevice(device);
220
221   response.send(surveys);
222 });
223
224 // Get survey by browser
225 app.get('/surveysByBrowser/:browser', (request, response) => {
226   let browser = request.params.browser;
227
228   let surveys = surveyManager.getSurveysByBrowser(browser);
229
230   response.send(surveys);
231 });
232
233 // Get survey by operating system
234 app.get('/surveysByOs/:os', (request, response) => {
235   let os = request.params.os;
236
237   let surveys = surveyManager.getSurveysByOs(os);
238
239   response.send(surveys);
240 });
241
242 // Get survey by device type
243 app.get('/surveysByDeviceType/:type', (request, response) => {
244   let type = request.params.type;
245
246   let surveys = surveyManager.getSurveysByDeviceType(type);
247
248   response.send(surveys);
249 });
250
251 // Get survey by device model
252 app.get('/surveysByDeviceModel/:model', (request, response) => {
253   let model = request.params.model;
254
255   let surveys = surveyManager.getSurveysByDeviceModel(model);
256
257   response.send(surveys);
258 });
259
260 // Get survey by device brand
261 app.get('/surveysByDeviceBrand/:brand', (request, response) => {
262   let brand = request.params.brand;
263
264   let surveys = surveyManager.getSurveysByDeviceBrand(brand);
265
266   response.send(surveys);
267 });
268
269 // Get survey by device OS
270 app.get('/surveysByDeviceOs/:os', (request, response) => {
271   let os = request.params.os;
272
273   let surveys = surveyManager.getSurveysByDeviceOs(os);
274
275   response.send(surveys);
276 });
277
278 // Get survey by device browser
279 app.get('/surveysByDeviceBrowser/:browser', (request, response) => {
280   let browser = request.params.browser;
281
282   let surveys = surveyManager.getSurveysByDeviceBrowser(browser);
283
284   response.send(surveys);
285 });
286
287 // Get survey by device language
288 app.get('/surveysByDeviceLanguage/:language', (request, response) => {
289   let language = request.params.language;
290
291   let surveys = surveyManager.getSurveysByDeviceLanguage(language);
292
293   response.send(surveys);
294 });
295
296 // Get survey by device platform
297 app.get('/surveysByDevicePlatform/:platform', (request, response) => {
298   let platform = request.params.platform;
299
300   let surveys = surveyManager.getSurveysByDevicePlatform(platform);
301
302   response.send(surveys);
303 });
304
305 // Get survey by device location
306 app.get('/surveysByDeviceLocation/:location', (request, response) => {
307   let location = request.params.location;
308
309   let surveys = surveyManager.getSurveysByDeviceLocation(location);
310
311   response.send(surveys);
312 });
313
314 // Get survey by device model
315 app.get('/surveysByDeviceModel/:model', (request, response) => {
316   let model = request.params.model;
317
318   let surveys = surveyManager.getSurveysByDeviceModel(model);
319
320   response.send(surveys);
321 });
322
323 // Get survey by device brand
324 app.get('/surveysByDeviceBrand/:brand', (request, response) => {
325   let brand = request.params.brand;
326
327   let surveys = surveyManager.getSurveysByDeviceBrand(brand);
328
329   response.send(surveys);
330 });
331
332 // Get survey by device OS
333 app.get('/surveysByDeviceOs/:os', (request, response) => {
334   let os = request.params.os;
335
336   let surveys = surveyManager.getSurveysByDeviceOs(os);
337
338   response.send(surveys);
339 });
340
341 // Get survey by device browser
342 app.get('/surveysByDeviceBrowser/:browser', (request, response) => {
343   let browser = request.params.browser;
344
345   let surveys = surveyManager.getSurveysByDeviceBrowser(browser);
346
347   response.send(surveys);
348 });
349
350 // Get survey by device language
351 app.get('/surveysByDeviceLanguage/:language', (request, response) => {
352   let language = request.params.language;
353
354   let surveys = surveyManager.getSurveysByDeviceLanguage(language);
355
356   response.send(surveys);
357 });
358
359 // Get survey by device platform
360 app.get('/surveysByDevicePlatform/:platform', (request, response) => {
361   let platform = request.params.platform;
362
363   let surveys = surveyManager.getSurveysByDevicePlatform(platform);
364
365   response.send(surveys);
366 });
367
368 // Get survey by device location
369 app.get('/surveysByDeviceLocation/:location', (request, response) => {
370   let location = request.params.location;
371
372   let surveys = surveyManager.getSurveysByDeviceLocation(location);
373
374   response.send(surveys);
375 });
376
377 // Get survey by device model
378 app.get('/surveysByDeviceModel/:model', (request, response) => {
379   let model = request.params.model;
380
381   let surveys = surveyManager.getSurveysByDeviceModel(model);
382
383   response.send(surveys);
384 });
385
386 // Get survey by device brand
387 app.get('/surveysByDeviceBrand/:brand', (request, response) => {
388   let brand = request.params.brand;
389
390   let surveys = surveyManager.getSurveysByDeviceBrand(brand);
391
392   response.send(surveys);
393 });
394
395 // Get survey by device OS
396 app.get('/surveysByDeviceOs/:os', (request, response) => {
397   let os = request.params.os;
398
399   let surveys = surveyManager.getSurveysByDeviceOs(os);
400
401   response.send(surveys);
402 });
403
404 // Get survey by device browser
405 app.get('/surveysByDeviceBrowser/:browser', (request, response) => {
406   let browser = request.params.browser;
407
408   let surveys = surveyManager.getSurveysByDeviceBrowser(browser);
409
410   response.send(surveys);
411 });
412
413 // Get survey by device language
414 app.get('/surveysByDeviceLanguage/:language', (request, response) => {
415   let language = request.params.language;
416
417   let surveys = surveyManager.getSurveysByDeviceLanguage(language);
418
419   response.send(surveys);
420 });
421
422 // Get survey by device platform
423 app.get('/surveysByDevicePlatform/:platform', (request, response) => {
424   let platform = request.params.platform;
425
426   let surveys = surveyManager.getSurveysByDevicePlatform(platform);
427
428   response.send(surveys);
429 });
430
431 // Get survey by device location
432 app.get('/surveysByDeviceLocation/:location', (request, response) => {
433   let location = request.params.location;
434
435   let surveys = surveyManager.getSurveysByDeviceLocation(location);
436
437   response.send(surveys);
438 });
439
440 // Get survey by device model
441 app.get('/surveysByDeviceModel/:model', (request, response) => {
442   let model = request.params.model;
443
444   let surveys = surveyManager.getSurveysByDeviceModel(model);
445
446   response.send(surveys);
447 });
448
449 // Get survey by device brand
450 app.get('/surveysByDeviceBrand/:brand', (request, response) => {
451   let brand = request.params.brand;
452
453   let surveys = surveyManager.getSurveysByDeviceBrand(brand);
454
455   response.send(surveys);
456 });
457
458 // Get survey by device OS
459 app.get('/surveysByDeviceOs/:os', (request, response) => {
460   let os = request.params.os;
461
462   let surveys = surveyManager.getSurveysByDeviceOs(os);
463
464   response.send(surveys);
465 });
466
467 // Get survey by device browser
468 app.get('/surveysByDeviceBrowser/:browser', (request, response) => {
469   let browser = request.params.browser;
470
471   let surveys = surveyManager.getSurveysByDeviceBrowser(browser);
472
473   response.send(surveys);
474 });
475
476 // Get survey by device language
477 app.get('/surveysByDeviceLanguage/:language', (request, response) => {
478   let language = request.params.language;
479
480   let surveys = surveyManager.getSurveysByDeviceLanguage(language);
481
482   response.send(surveys);
483 });
484
485 // Get survey by device platform
486 app.get('/surveysByDevicePlatform/:platform', (request, response) => {
487   let platform = request.params.platform;
488
489   let surveys = surveyManager.getSurveysByDevicePlatform(platform);
490
491   response.send(surveys);
492 });
493
494 // Get survey by device location
495 app.get('/surveysByDeviceLocation/:location', (request, response) => {
496   let location = request.params.location;
497
498   let surveys = surveyManager.getSurveysByDeviceLocation(location);
499
500   response.send(surveys);
501 });
502
503 // Get survey by device model
504 app.get('/surveysByDeviceModel/:model', (request, response) => {
505   let model = request.params.model;
506
507   let surveys = surveyManager.getSurveysByDeviceModel(model);
508
509   response.send(surveys);
510 });
511
512 // Get survey by device brand
513 app.get('/surveysByDeviceBrand/:brand', (request, response) => {
514   let brand = request.params.brand;
515
516   let surveys = surveyManager.getSurveysByDeviceBrand(brand);
517
518   response.send(surveys);
519 });
520
521 // Get survey by device OS
522 app.get('/surveysByDeviceOs/:os', (request, response) => {
523   let os = request.params.os;
524
525   let surveys = surveyManager.getSurveysByDeviceOs(os);
526
527   response.send(surveys);
528 });
529
530 // Get survey by device browser
531 app.get('/surveysByDeviceBrowser/:browser', (request, response) => {
532   let browser = request.params.browser;
533
534   let surveys = surveyManager.getSurveysByDeviceBrowser(browser);
535
536   response.send(surveys);
537 });
538
539 // Get survey by device language
540 app.get('/surveysByDeviceLanguage/:language', (request, response) => {
541   let language = request.params.language;
542
543   let surveys = surveyManager.getSurveysByDeviceLanguage(language);
544
545   response.send(surveys);
546 });
547
548 // Get survey by device platform
549 app.get('/surveysByDevicePlatform/:platform', (request, response) => {
550   let platform = request.params.platform;
551
552   let surveys = surveyManager.getSurveysByDevicePlatform(platform);
553
554   response.send(surveys);
555 });
556
557 // Get survey by device location
558 app.get('/surveysByDeviceLocation/:location', (request, response) => {
559   let location = request.params.location;
560
561   let surveys = surveyManager.getSurveysByDeviceLocation(location);
562
563   response.send(surveys);
564 });
565
566 // Get survey by device model
567 app.get('/surveysByDeviceModel/:model', (request, response) => {
568   let model = request.params.model;
569
570   let surveys = surveyManager.getSurveysByDeviceModel(model);
571
572   response.send(surveys);
573 });
574
575 // Get survey by device brand
576 app.get('/surveysByDeviceBrand/:brand', (request, response) => {
577   let brand = request.params.brand;
578
579   let surveys = surveyManager.getSurveysByDeviceBrand(brand);
580
581   response.send(surveys);
582 });
583
584 // Get survey by device OS
585 app.get('/surveysByDeviceOs/:os', (request, response) => {
586   let os = request.params.os;
587
588   let surveys = surveyManager.getSurveysByDeviceOs(os);
589
590   response.send(surveys);
591 });
592
593 // Get survey by device browser
594 app.get('/surveysByDeviceBrowser/:browser', (request, response) => {
595   let browser = request.params.browser;
596
597   let surveys = surveyManager.getSurveysByDeviceBrowser(browser);
598
599   response.send(surveys);
600 });
601
602 // Get survey by device language
603 app.get('/surveysByDeviceLanguage/:language', (request, response) => {
604   let language = request.params.language;
605
606   let surveys = surveyManager.getSurveysByDeviceLanguage(language);
607
608   response.send(surveys);
609 });
610
611 // Get survey by device platform
612 app.get('/surveysByDevicePlatform/:platform', (request, response) => {
613   let platform = request.params.platform;
614
615   let surveys = surveyManager.getSurveysByDevicePlatform(platform);
616
617   response.send(surveys);
618 });
619
620 // Get survey by device location
621 app.get('/surveysByDeviceLocation/:location', (request, response) => {
622   let location = request.params.location;
623
624   let surveys = surveyManager.getSurveysByDeviceLocation(location);
625
626   response.send(surveys);
627 });
628
629 // Get survey by device model
630 app.get('/surveysByDeviceModel/:model', (request, response) => {
631   let model = request.params.model;
632
633   let surveys = surveyManager.getSurveysByDeviceModel(model);
634
635   response.send(surveys);
636 });
637
638 // Get survey by device brand
639 app.get('/surveysByDeviceBrand/:brand', (request, response) => {
640   let brand = request.params.brand;
641
642   let surveys = surveyManager.getSurveysByDeviceBrand(brand);
643
644   response.send(surveys);
645 });
646
647 // Get survey by device OS
648 app.get('/surveysByDeviceOs/:os', (request, response) => {
649   let os = request.params.os;
650
651   let surveys = surveyManager.getSurveysByDeviceOs(os);
652
653   response.send(surveys);
654 });
655
656 // Get survey by device browser
657 app.get('/surveysByDeviceBrowser/:browser', (request, response) => {
658   let browser = request.params.browser;
659
660   let surveys = surveyManager.getSurveysByDeviceBrowser(browser);
661
662   response.send(surveys);
663 });
664
665 // Get survey by device language
666 app.get('/surveysByDeviceLanguage/:language', (request, response) => {
667   let language = request.params.language;
668
669   let surveys = surveyManager.getSurveysByDeviceLanguage(language);
670
671   response.send(surveys);
672 });
673
674 // Get survey by device platform
675 app.get('/surveysByDevicePlatform/:platform', (request, response) => {
676   let platform = request.params.platform;
677
678   let surveys = surveyManager.getSurveysByDevicePlatform(platform);
679
680   response.send(surveys);
681 });
682
683 // Get survey by device location
684 app.get('/surveysByDeviceLocation/:location', (request, response) => {
685   let location = request.params.location;
686
687   let surveys = surveyManager.getSurveysByDeviceLocation(location);
688
689   response.send(surveys);
690 });
691
692 // Get survey by device model
693 app.get('/surveysByDeviceModel/:model', (request, response) => {
694   let model = request.params.model;
695
696   let surveys = surveyManager.getSurveysByDeviceModel(model);
697
698   response.send(surveys);
699 });
700
701 // Get survey by device brand
702 app.get('/surveysByDeviceBrand/:brand', (request, response) => {
703   let brand = request.params.brand;
704
705   let surveys = surveyManager.getSurveysByDeviceBrand(brand);
706
707   response.send(surveys);
708 });
709
710 // Get survey by device OS
711 app.get('/surveysByDeviceOs/:os', (request, response) => {
712   let os = request.params.os;
713
714   let surveys = surveyManager.getSurveysByDeviceOs(os);
715
716   response.send(surveys);
717 });
718
719 // Get survey by device browser
720 app.get('/surveysByDeviceBrowser/:browser', (request, response) => {
721   let browser = request.params.browser;
722
723   let surveys = surveyManager.getSurveysByDeviceBrowser(browser);
724
725   response.send(surveys);
726 });
727
728 // Get survey by device language
729 app.get('/surveysByDeviceLanguage/:language', (request, response) => {
730   let language = request.params.language;
731
732   let surveys = surveyManager.getSurveysByDeviceLanguage(language);
733
734   response.send(surveys);
735 });
736
737 // Get survey by device platform
738 app.get('/surveysByDevicePlatform/:platform', (request, response) => {
739   let platform = request.params.platform;
740
741   let surveys = surveyManager.getSurveysByDevicePlatform(platform);
742
743   response.send(surveys);
744 });
745
746 // Get survey by device location
747 app.get('/surveysByDeviceLocation/:location', (request, response) => {
748   let location = request.params.location;
749
750   let surveys = surveyManager.getSurveysByDeviceLocation(location);
751
752   response.send(surveys);
753 });
754
755 // Get survey by device model
756 app.get('/surveysByDeviceModel/:model', (request, response) => {
757   let model = request.params.model;
758
759   let surveys = surveyManager.getSurveysByDeviceModel(model);
760
761   response.send(surveys);
762 });
763
764 // Get survey by device brand
765 app.get('/surveysByDeviceBrand/:brand', (request, response) => {
766   let brand = request.params.brand;
767
768   let surveys = surveyManager.getSurveysByDeviceBrand(brand);
769
770   response.send(surveys);
771 });
772
773 // Get survey by device OS
774 app.get('/surveysByDeviceOs/:os', (request, response) => {
775   let os = request.params.os;
776
777   let surveys = surveyManager.getSurveysByDeviceOs(os);
778
779   response.send(surveys);
780 });
781
782 // Get survey by device browser
783 app.get('/surveysByDeviceBrowser/:browser', (request, response) => {
784   let browser = request.params.browser;
785
786   let surveys = surveyManager.getSurveysByDeviceBrowser(browser);
787
788   response.send(surveys);
789 });
790
791 // Get survey by device language
792 app.get('/surveysByDeviceLanguage/:language', (request, response) => {
793   let language = request.params.language;
794
795   let surveys = surveyManager.getSurveysByDeviceLanguage(language);
796
797   response.send(surveys);
798 });
799
800 // Get survey by device platform
801 app.get('/surveysByDevicePlatform/:platform', (request, response) => {
802   let platform = request.params.platform;
803
804   let surveys = surveyManager.getSurveysByDevicePlatform(platform);
805
806   response.send(surveys);
807 });
808
809 // Get survey by device location
810 app.get('/surveysByDeviceLocation/:location', (request, response) => {
811   let location = request.params.location;
812
813   let surveys = surveyManager.getSurveysByDeviceLocation(location);
814
815   response.send(surveys);
816 });
817
818 // Get survey by device model
819 app.get('/surveysByDeviceModel/:model', (request, response) => {
820   let model = request.params.model;
821
822   let surveys = surveyManager.getSurveysByDeviceModel(model);
823
824   response.send(surveys);
825 });
826
827 // Get survey by device brand
828 app.get('/surveysByDeviceBrand/:brand', (request, response) => {
829   let brand = request.params.brand;
830
831   let surveys = surveyManager.getSurveysByDeviceBrand(brand);
832
833   response.send(surveys);
834 });
835
836 // Get survey by device OS
837 app.get('/surveysByDeviceOs/:os', (request, response) => {
838   let os = request.params.os;
839
840   let surveys = surveyManager.getSurveysByDeviceOs(os);
841
842   response.send(surveys);
843 });
844
845 // Get survey by device browser
846 app.get('/surveysByDeviceBrowser/:browser', (request, response) => {
847   let browser = request.params.browser;
848
849   let surveys = surveyManager.getSurveysByDeviceBrowser(browser);
850
851   response.send(surveys);
852 });
853
854 // Get survey by device language
855 app.get('/surveysByDeviceLanguage/:language', (request, response) => {
856   let language = request.params.language;
857
858   let surveys = surveyManager.getSurveysByDeviceLanguage(language);
859
860   response.send(surveys);
861 });
862
863 // Get survey by device platform
864 app.get('/surveysByDevicePlatform/:platform', (request, response) => {
865   let platform = request.params.platform;
866
867   let surveys = surveyManager.getSurveysByDevicePlatform(platform);
868
869   response.send(surveys);
870 });
871
872 // Get survey by device location
873 app.get('/surveysByDeviceLocation/:location', (request, response) => {
874   let location = request.params.location;
875
876   let surveys = surveyManager.getSurveysByDeviceLocation(location);
877
878   response.send(surveys);
879 });
880
881 // Get survey by device model
882 app.get('/surveysByDeviceModel/:model', (request, response) => {
883   let model = request.params.model;
884
885   let surveys = surveyManager.getSurveysByDeviceModel(model);
886
887   response.send(surveys);
888 });
889
890 // Get survey by device brand
891 app.get('/surveysByDeviceBrand/:brand', (request, response) => {
892   let brand = request.params.brand;
893
894   let surveys = surveyManager.getSurveysByDeviceBrand(brand);
895
896   response.send(surveys);
897 });
898
899 // Get survey by device OS
900 app.get('/surveysByDeviceOs/:os', (request, response) => {
901   let os = request.params.os;
902
903   let surveys = surveyManager.getSurveysByDeviceOs(os);
904
905   response.send(surveys);
906 });
907
908 // Get survey by device browser
909 app.get('/surveysByDeviceBrowser/:browser', (request, response) => {
910   let browser = request.params.browser;
911
912   let surveys = surveyManager.getSurveysByDeviceBrowser(browser);
913
914   response.send(surveys);
915 });
916
917 // Get survey by device language
918 app.get('/surveysByDeviceLanguage/:language', (request, response) => {
919   let language = request.params.language;
920
921   let surveys = surveyManager.getSurveysByDeviceLanguage(language);
922
923   response.send(surveys);
924 });
925
926 // Get survey by device platform
927 app.get('/surveysByDevicePlatform/:platform', (request, response) => {
928   let platform = request.params.platform;
929
930   let surveys = surveyManager.getSurveysByDevicePlatform(platform);
931
932   response.send(surveys);
933 });
934
935 // Get survey by device location
936 app.get('/surveysByDeviceLocation/:location', (request, response) => {
937   let location = request.params.location;
938
939   let surveys = surveyManager.getSurveysByDeviceLocation(location);
940
941   response.send(surveys);
942 });
943
944 // Get survey by device model
945 app.get('/surveysByDeviceModel/:model', (request, response) => {
946   let model = request.params.model;
947
948   let surveys = surveyManager.getSurveysByDeviceModel(model);
949
950   response.send(surveys);
951 });
952
953 // Get survey by device brand
954 app.get('/surveysByDeviceBrand/:brand', (request, response) => {
955   let brand = request.params.brand;
956
957   let surveys = surveyManager.getSurveysByDeviceBrand(brand);
958
959   response.send(surveys);
960 });
961
962 // Get survey by device OS
963 app.get('/surveysByDeviceOs/:os', (request, response) => {
964   let os = request.params.os;
965
966   let surveys = surveyManager.getSurveysByDeviceOs(os);
967
968   response.send(surveys);
969 });
970
971 // Get survey by device browser
972 app.get('/surveysByDeviceBrowser/:browser', (request, response) => {
973   let browser = request.params.browser;
974
975   let surveys = surveyManager.getSurveysByDeviceBrowser(browser);
976
977   response.send(surveys);
978 });
979
980 // Get survey by device language
981 app.get('/surveysByDeviceLanguage/:language', (request, response) => {
982   let language = request.params.language;
983
984   let surveys = surveyManager.getSurveysByDeviceLanguage(language);
985
986   response.send(surveys);
987 });
988
989 // Get survey by device platform
990 app.get('/surveysByDevicePlatform/:platform', (request, response) => {
991   let platform = request.params.platform;
992
993   let surveys = surveyManager.getSurveysByDevicePlatform(platform);
994
995   response.send(surveys);
996 });
997
998 // Get survey by device location
999 app.get('/surveysByDeviceLocation/:location', (request, response) => {
1000   let location = request.params.location;
1001
1002   let surveys = surveyManager.getSurveysByDeviceLocation(location);
1003
1004   response.send(surveys);
1005 });
1006
1007 // Get survey by device model
1008 app.get('/surveysByDeviceModel/:model', (request, response) => {
1009   let model = request.params.model;
1010
1011   let surveys = surveyManager.getSurveysByDeviceModel(model);
1012
1013   response.send(surveys);
1014 });
1015
1016 // Get survey by device brand
1017 app.get('/surveysByDeviceBrand/:brand', (request, response) => {
1018   let brand = request.params.brand;
1019
1020   let surveys = surveyManager.getSurveysByDeviceBrand(brand);
1021
1022   response.send(surveys);
1023 });
1024
1025 // Get survey by device OS
1026 app.get('/surveysByDeviceOs/:os', (request, response) => {
1027   let os = request.params.os;
1028
1029   let surveys = surveyManager.getSurveysByDeviceOs(os);
1030
1031   response.send(surveys);
1032 });
1033
1034 // Get survey by device browser
1035 app.get('/surveysByDeviceBrowser/:browser', (request, response) => {
1036   let browser = request.params.browser;
1037
1038   let surveys = surveyManager.getSurveysByDeviceBrowser(browser);
1039
1040   response.send(surveys);
1041 });
1042
1043 // Get survey by device language
1044 app.get('/surveysByDeviceLanguage/:language', (request, response) => {
1045   let language = request.params.language;
1046
1047   let surveys = surveyManager.getSurveysByDeviceLanguage(language);
1048
1049   response.send(surveys);
1050 });
1051
1052 // Get survey by device platform
1053 app.get('/surveysByDevicePlatform/:platform', (request, response) => {
1054   let platform = request.params.platform;
1055
1056   let surveys = surveyManager.getSurveysByDevicePlatform(platform);
1057
1058   response.send(surveys);
1059 });
1060
1061 // Get survey by device location
1062 app.get('/surveysByDeviceLocation/:location', (request, response) => {
1063   let location = request.params.location;
1064
1065   let surveys = surveyManager.getSurveysByDeviceLocation(location);
1066
1067   response.send(surveys);
1068 });
1069
1070 // Get survey by device model
1071 app.get('/surveysByDeviceModel/:model', (request, response) => {
1072   let model = request.params.model;
1073
1074   let surveys = surveyManager.getSurveysByDeviceModel(model);
1075
1076   response.send(surveys);
1077 });
1078
1079 // Get survey by device brand
1080 app.get('/surveysByDeviceBrand/:brand', (request, response) => {
1081   let brand = request.params.brand;
1082
1083   let surveys = surveyManager.getSurveysByDeviceBrand(brand);
1084
1085   response.send(surveys);
1086 });
1087
1088 // Get survey by device OS
1089 app.get('/surveysByDeviceOs/:os', (request, response) => {
1090   let os = request.params.os;
1091
1092   let surveys = surveyManager.getSurveysByDeviceOs(os);
1093
1094   response.send(surveys);
1095 });
1096
1097 // Get survey by device browser
1098 app.get('/surveysByDeviceBrowser/:browser', (request, response) => {
1099   let browser = request.params.browser;
1100
1101   let surveys = surveyManager.getSurveysByDeviceBrowser(browser);
1102
1103   response.send(surveys);
1104 });
1105
1106 // Get survey by device language
1107 app.get('/surveysByDeviceLanguage/:language', (request, response) => {
1108   let language = request.params.language;
1109
1110   let surveys = surveyManager.getSurveysByDeviceLanguage(language);
1111
1112   response.send(surveys);
1113 });
1114
1115 // Get survey by device platform
1116 app.get('/surveysByDevicePlatform/:platform', (request, response) => {
1117   let platform = request.params.platform;
1118
1119   let surveys = surveyManager.getSurveysByDevicePlatform(platform);
1120
1121   response.send(surveys);
1122 });
1123
1124 // Get survey by device location
1125 app.get('/surveysByDeviceLocation/:location', (request, response) => {
1126   let location = request.params.location;
1127
1128   let surveys = surveyManager.getSurveysByDeviceLocation(location);
1129
1130   response.send(surveys);
1131 });
1132
1133 // Get survey by device model
1134 app.get('/surveysByDeviceModel/:model', (request, response) => {
1135   let model = request.params.model;
1136
1137   let surveys = surveyManager.getSurveysByDeviceModel(model);
1138
1139   response.send(surveys);
1140 });
1141
1142 // Get survey by device brand
1143 app.get('/surveysByDeviceBrand/:brand', (request, response) => {
1144   let brand = request.params.brand;
1145
1146   let surveys = surveyManager.getSurveysByDeviceBrand(brand);
1147
1148   response.send(surveys);
1149 });
1150
1151 // Get survey by device OS
1152 app.get('/surveysByDeviceOs/:os', (request, response) => {
1153   let os = request.params.os;
1154
1155   let surveys = surveyManager.getSurveysByDeviceOs(os);
1156
1157   response.send(surveys);
1158 });
1159
1160 // Get survey by device browser
1161 app.get('/surveysByDeviceBrowser/:browser', (request, response) => {
1162   let browser = request.params.browser;
1163
1164   let surveys = surveyManager.getSurveysByDeviceBrowser(browser);
1165
1166   response.send(surveys);
1167 });
1168
1169 // Get survey by device language
1170 app.get('/surveysByDeviceLanguage/:language', (request, response) => {
1171   let language = request.params.language;
1172
1173   let surveys = surveyManager.getSurveysByDeviceLanguage(language);
1174
1175   response.send(surveys);
1176 });
1177
1178 // Get survey by device platform
1179 app.get('/surveysByDevicePlatform/:platform', (request, response) => {
1180   let platform = request.params.platform;
1181
1182   let surveys = surveyManager.getSurveysByDevicePlatform(platform);
1183
1184   response.send(surveys);
1185 });
1186
1187 // Get survey by device location
1188 app.get('/surveysByDeviceLocation/:location', (request, response) => {
1189   let location = request.params.location;
1190
1191   let surveys = surveyManager.getSurveysByDeviceLocation(location);
1192
1193   response.send(surveys);
1194 });
1195
1196 // Get survey by device model
1197 app.get('/surveysByDeviceModel/:model', (request, response) => {
1198   let model = request.params.model;
1199
1200   let surveys = surveyManager.getSurveysByDeviceModel(model);
1201
1202   response.send(surveys);
1203 });
1204
1205 // Get survey by device brand
1206 app.get('/surveysByDeviceBrand/:brand', (request, response) => {
1207   let brand = request.params.brand;
1208
1209   let surveys = surveyManager.getSurveysByDeviceBrand(brand);
1210
1211   response.send(surveys);
1212 });
1213
1214 // Get survey by device OS
1215 app.get('/surveysByDeviceOs/:os', (request, response) => {
1216   let os = request.params.os;
1217
1218   let surveys = surveyManager.getSurveysByDeviceOs(os);
1219
1220   response.send(surveys);
1221 });
1222
1223 // Get survey by device browser
1224 app.get('/surveysByDeviceBrowser/:browser', (request, response) => {
1225   let browser = request.params.browser;
1226
1227   let surveys = surveyManager.getSurveysByDeviceBrowser(browser);
1228
1229   response.send(surveys);
1230 });
1231
1232 // Get survey by device language
1233 app.get('/surveysByDeviceLanguage/:language', (request, response) => {
1234   let language = request.params.language;
1235
1236   let surveys = surveyManager.getSurveysByDeviceLanguage(language);
1237
1238   response.send(surveys);
1239 });
1240
1241 // Get survey by device platform
1242 app.get('/surveysByDevicePlatform/:platform', (request, response) => {
1243   let platform = request.params.platform;
1244
1245   let surveys = surveyManager.getSurveysByDevicePlatform(platform);
1246
1247   response.send(surveys);
1248 });
1249
1250 // Get survey by device location
1251 app.get('/surveysByDeviceLocation/:location', (request, response) => {
1252   let location = request.params.location
```

© 2013 Pearson Education, Inc.

At the top of the file at line 5 is the statement that creates and initializes the `SurveyManager` instance. At line 27 you can see the statement that checks whether a survey with a particular topic exists. If you run the application and use the browser to navigate to `localhost:8080/index.html` and enter a survey you'll find it works in the same way as before. You can add breakpoints to watch as the different methods are called. Leave the application open so that we can use it to investigate comments in the next section.

CODE ANALYSIS

Using a Manager Class

You may have some questions about how the manager class is used in the Tiny Survey application.

Question: What is the difference between using a manager class and just having storage classes?

The previous version of Tiny Survey used a set of classes ([Option](#), [Survey](#) and [Surveys](#)) to store survey information. This works fine. Why are we making another class which duplicates what they are doing? The answer is that we want to separate the Tiny Survey application code from that which is used to store the data. This will make it possible to change the underlying data storage without changing any part of Tiny Survey.

Question: Why does the manager class have a `surveyExists` method?

The `Surveys` class in the survey storage provides a method called `getSurveyByTopic` to find a survey with a particular topic. However, an application using the `SurveyManager` class does have access to the `Surveys` storage class. The Tiny Survey application needs to know whether a survey exists, and it can now use this method to determine this.

Question: Doesn't using a manager class slow the application down?

The previous version of Tiny Survey called the storage methods directly. This version calls a method in the manager class which then calls the storage methods. This will introduce a tiny extra delay but this will not affect performance in any great way. The advantages of having a manager class are worth the performance penalty.

Comments/Documentation

Proper code documentation adds a lot to your code. You might have been wondering why I've not mentioned comments just yet. This is because I think that up until now, we haven't had much need for them. If you use sensible variable names and simple code structure you don't need many comments. Otherwise, you find yourself writing statements like this:

```
totalPrice = salesPrice + handlingPrice + tax; // work out the total price
```

I think the above comment is not really needed. However, when we create a manager class, we must have comments and documentation. People using the class will need to be told what each method does and how to use it. There are two kinds of comments you can add to JavaScript programs:

```
// single line comment

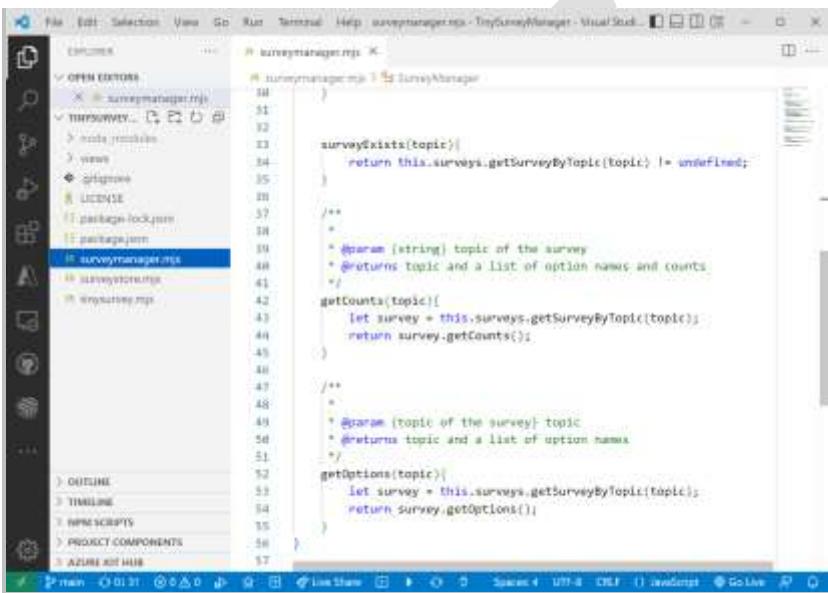
/* Multi-line comments
   that can extend over several
   lines
*/
```

We could just sprinkle these kinds of comments over the `SurveyManager` class, but it turns out that there is a much better way of adding comments which are properly useful.

Make Something Happen

Working with comments

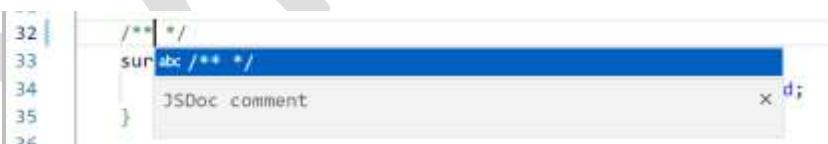
The starting point for this Make Something Happen is a version of Tiny Survey you can download from GitHub here: <https://github.com/Building-Apps-and-Games-in-the-Cloud/TinySurveyManager> You should already have this open from the previous exercise. Open the file `surveymanager.mjs` and move to line 32.



```
File Edit Selection View Go Run Terminal Help surveymanager.mjs - TinySurveyManager - Visual Studio ... × surveymanager.mjs 32 surveymanager.mjs
30
31
32
33     surveyExists(topic){}
34         return this.surveys.getSurveyByTopic(topic) != undefined;
35     }
36
37 /**
38 * @param {string} topic of the survey
39 * @returns topic and a list of option names and counts
40 */
41 getCounts(topic){
42     let survey = this.surveys.getSurveyByTopic(topic);
43     return survey.getCounts();
44 }
45
46 /**
47 * @param {topic of the survey} topic
48 * @returns topic and a list of option names
49 */
50 getOptions(topic){
51     let survey = this.surveys.getSurveyByTopic(topic);
52     return survey.getOptions();
53 }
```

Ch09_inset03_01 Comments in the surveymanager class

The `getCounts` and `getOptions` methods have formatted comments above them. We are going to add similar comments to the `surveyExists` method. Visual Studio code will do a lot of the work for this. Use tab to move the cursor into line 32 so that it is directly above the `surveyExists` declaration. Now type `/**` (forward slash followed by two asterisks).



Ch09_inset03_02 Inserting a JSDoc comment

A dialog will pop up inviting you to enter a JSDoc comment. Press Enter.

```
31
32
33
34     * @param {string} topic
35     * @returns {boolean}
36   /
37   surveyExists(topic){
38     return this.surveys.getSurveyByTopic(topic) != undefined;
39 }
```

Ch09_inset03_03 Editing a JSDoc comment

You can now enter the comment text. The item between the braces where the cursor is presently located is where you enter the type of the parameter. You can also enter descriptions of the method, the topic parameter and the return value.

```
31
32
33  /**
34   * Checks if a survey exists
35   * @param {string} topic topic of the survey
36   * @returns true if the survey exists in the storage
37   */
38   surveyExists(topic){
39     return this.surveys.getSurveyByTopic(topic) != undefined;
}
```

Ch09_inset03_04 Completed comment text

Above you can see the completed comment text. The great thing about comments entered like this is that Visual Studio Code can use them to provide interactive help. To see this in action, open the **tinysurvey.mjs** file in the editor and move to line 27. Now rest the mouse cursor over the identifier **surveyExists**.

```
18 // Home page
19 app.get('/index.html', (request, response) => {
20   response.render('index.e
21 });
22
23 // Got the survey topic
24 app.post('/gottopic', (req
25   let topic = request.body
26
27   let survey = surveyManager.surveyExists(topic);
```

Ch09_inset03_05 Help display in Visual Studio Code

Visual Studio will display a pop-up window containing the comment information you have just entered. You will get similar displays as you are editing code. This is a very good way to make “self-documenting” programs. The JSDoc comment format can be used with JSDoc3, a documentation generator which can produce websites for you. You can find out more here: <https://jsdoc.app/>

Error checking

The current implementation of the `SurveyManager` class has no error checking at all. The methods work on the basis that they will be provided with the correct information to perform their tasks. If I'm writing a small program for my own amusement this is fine, but if I want to incorporate the code into a program I'm going to sell, or I want to make it available for others to use there should be some error checking in the code.

```
/**  
 * Stores a survey  
 * @param {Object} newValue topic string and option list  
 */  
storeSurvey(newValue){  
    let survey = new Survey(newValue);  
    this.surveys.saveSurvey(survey);  
}
```

As an example of what I mean by missing error handling, consider the method above. It stores a new survey value. The `newValue` parameter to `storeSurvey` is an object that describes a survey. The `goodSurveyValues` variable below refers to an object which contains all the information required to make a survey and could be passed into `storeSurvey`. However, the `storeSurvey` method doesn't check that the values it receives are valid.

```
let goodSurveyValues = {  
    topic: "robspizza",  
    options: [  
        { text: "margherita", count: 0 },  
        { text: "pepperoni", count: 0 },  
        { text: "chicken", count: 0 },  
        { text: "ham and pineapple", count: 0 },  
        { text: "mushroom", count: 0 }  
    ]  
};
```

The `badSurveyValues` variable below describes a survey which has quite a few things wrong with it. The `topic` property is mis-spelled `Topic`, one of the options has a count value with is a string and another option has no count value at all. However, the current `storeSurvey` method in the

`SurveyManager` class would accept this and try to store it as a survey. We really should not be selling an application that allows things like this.

```
let badSurveyValues = {
    Topic:"bad survey",
    options: [
        { text: "margherita", count: "hello" },
        { text: 99, count: 0 },
        { text: "chicken"}
    ]
};
```

The code below uses the JavaScript `in` operator to test that a new value contains `topic` and `options` properties. If these properties are missing an error description is added to the `errors` string. This string will be checked at the end of the `storeSurvey` method. If it is empty the survey can be stored. Otherwise, the `errors` string will contain a description of what is wrong with the incoming survey.

```
let errors = "";
if (!("topic" in newValue)) {
    errors = errors + "Missing topic property in storeSurvey\n";
}
if (!("options" in newValue)) {
    errors = errors + "Missing options property in storeSurvey\n";
}
```

Error handling

Detecting errors is one thing but deciding what to do when they happen is another. There are two kinds of errors that can happen when an application is running. There are ones that are expected (the user might enter an invalid username or password) and there are errors that shouldn't happen (the program might try to store an invalid survey).

The first kind of error should be built into the workflow of the application. If the user enters an invalid username or password into a login page the application should display an appropriate message and invite the user to try again. This workflow might be expanded to only allow the user 5 attempts before an account is locked. This depends on the importance of the application (and how much security effort the customer is prepared to pay for). When you create the workflow of

an application you must look at each step and then consider what the user could do wrong. Then you add extra steps to handle this. These kinds of errors are routine errors that are to be expected.

The second kind of error is caused by a fault in the software or a failure of an external service. The workflow should ensure that a user will always enter valid surveys. These errors are outside the normal workflow of the application. These errors can best be handled by an *exception*. As the name implies, an exception is an exceptional event. Let's have a look at how exceptions are dealt with in JavaScript.

JavaScript Heroes: Exceptions

A good superhero always has a good escape plan. The same is true of an application. If the application is in a situation where it would not be meaningful for it to continue it should throw an exception object to indicate this. In the case of making an attempt to store a survey with invalid contents it is very important that this action not continue. JavaScript can abandon an action by throwing an exception. The code below is at the end of the `storeSurvey` method. It checks to see if the `errors` variable contains any messages. If it does the `errors` variable is thrown as an exception.

```
if (errors != "") {  
    throw errors;  
}
```

An exception abandons the current flow of execution and diverts it into a piece of code intended to handle the exception. Below you can see how this works. The code creates a new `SurveyManager` instance and uses it to store the survey defined in the variable `goodSurveyValues`. The code to do this is in a block following the `try` keyword. When this code runs the survey is stored correctly.

```
let mgr = new SurveyManager();  
  
try {  
    mgr.storeSurvey(goodSurveyValues);  
}  
catch(error){  
    console.log("Survey store failed:" + error);  
}
```

If we replace the `goodSurveyValues` variable with `badSurveyValues` we are trying to store an invalid

survey. The `storeSurvey` method will throw an exception and execution will transfer into the `catch` block which will log the error on the console. The argument to the `catch` is the object that was thrown, which in this case is the string containing the error report.

Make Something Happen

Exploring Exceptions

Let's take a look at how exceptions work. Start your browser and open the `index.html` file in the **Ch09-01-Exploring_Exceptions** example folder for this chapter. Open the browser Developer Tools and select the console view. The JavaScript code in the page contains the survey management classes. We can work with them in the console. Let's start by creating a `SurveyManager` instance. Type in the following command:

```
let mgr = new SurveyManager()
```

and press enter. A new `SurveyManager` is created and the variable `mgr` is set to refer to it.

```
> let mgr = new SurveyManager()  
< undefined  
>
```

Ch09_inset04_01 create a survey manager

Now we have our survey manager we can use it to store a survey. The web page code contains definitions of the `goodSurveyValues` and `badSurveyValues` variables. We can try to store those in the survey. Type in the following:

```
mgr.storeSurvey(goodSurveyValues)
```

and press enter. Storing a good survey works just fine.

```
> mgr.storeSurvey(goodSurveyValues)  
< undefined  
>
```

Ch09_inset04_02 store a good survey

Now let's store a bad survey and see what happens. Type in the following:

```
mgr.storeSurvey(badSurveyValues)
```

and press enter. This time the `storeSurvey` method will throw an exception.

```
> mgr.storeSurvey(badSurveyValues)
➊ > Uncaught Missing topic property in storeSurvey
  Count not a number in option margherita in storeSurvey
  Missing count in option chicken in storeSurvey
```

Ch09_inset04_03 store a bad survey

The `storeSurvey` call was made outside a `try` block and so the JavaScript engine displays an error along with the uncaught exception value. If we enclose the statement in a `try` block we can get control when the exception is thrown. Type in the following:

```
try { mgr.storeSurvey(badSurveyValues); } catch(error)
{console.log("Ooops:"+error)}
```

and press enter. This statement contains a `try` block and a `catch` block. If code inside the `try` block throws an exception the code inside the `catch` block will execute.

```
> try { mgr.storeSurvey(badSurveyValues); } catch(error)
  {console.log("Ooops:"+error)}
  Ooops:Missing topic property in storeSurvey
  Count not a number in option margherita in storeSurvey
  Missing count in option chicken in storeSurvey
< undefined
>
```

Ch09_inset04_04 catch an exception

This time no errors are displayed as the exception has been caught. You can experiment with the error handling in the `SurveyManager` class by trying to store different objects of your own as surveys. You can also take a look at the source of the site and discover just how much code you need to add to perform error testing.

Exceptions and finally

There is one last thing that you need to know about exceptions. And that is the aptly named “finally” block. You can add this to an exception construction so that you can define code that is guaranteed to run, whatever happens inside the `try – catch` blocks. The code below tries to store a survey. If the survey storage throws an exception the `catch` block runs to deal with this. The `finally` block contains code that will run whatever happens inside the `try – catch`. If the `catch` block throws another exception or even returns from an enclosing function or method, the `finally` code still runs. The `finally` block is where we can put code that will release resources that may have been obtained in the `try` block.

```
try {
  mgr.storeSurvey(surveyValues);
}
catch(error){
  console.log("Survey store failed:" + error);
```

```
}
```

```
finally{
    console.log("This message is always displayed");
}
```

CODE ANALYSIS

Using Exceptions

You may have some questions about exceptions.

Question: How much code can you put in the `try` block?

The `try` block can contain as much code as you like.

Question: Can exception handlers be nested?

Yes. You can put a `try – catch` construction in code already running inside a `try-catch` construction. When an exception is thrown JavaScript will look for the “nearest” `catch` and use that.

Question: Does the exception handler “know” where it was thrown?

No. You can of course put information into the block of data that is thrown that will tell code in the `catch` block where the failure occurred.

Question: Is it possible to go back to the code where the exception was thrown after the exception has been handled?

No. When the exception is thrown the execution transfers to the `catch` block (if any) and can never return.

Question: What happens if code in the `catch` block throws an exception?

If code in the `catch` block throws an exception JavaScript will loop for the “nearest” `catch` block (if there is one) and run that.

Question: Does every `try – catch` construction need a `finally` block?

No. You only need a `finally` block if code in the `try` block allocates resources (for example connects to a database or opens a file) which you want to release when the code has finished. This reduces the chances of resources being “stuck” allocated to something which will never give them back.

Question: How do we use try-catch in an application like Tiny Survey?

Good question. We could enclose the code which handles each Express route in a try – catch and then render an error page in the catch block if the code throws an exception. The code below shows how this works. It is very unlikely that rendering the index page would throw an exception, but if it does this would now cause the error to be logged and the error page to be displayed.

```
app.get('/index.html', (request, response) => {
  try {
    response.render('index.ejs');
  }
  catch(error) {
    console.log(error);
    response.render('error.ejs');
  }
});
```

Testing

Code that is to be sold should be tested first. And by testing I don't mean running through the program a couple of times to see what happens. I'm not sure what that is called, perhaps "playing with it". Proper testing is a managed process which is part of the project. For tests to be any use they must be repeatable and, as much as possible, automatic. We can test the Tiny Survey application itself by writing a set of survey creation steps and then working through them. Software companies employ testers to do this, and there are also automated tools that can be used to provide simulated inputs and track the outputs. This is testing at the application level, but there is also a need for testing at the module level. We've just added a whole bunch of error handling to the [SurveyManager](#) class. It would be useful to be able to test these error handlers automatically. There are several tools available that we can add to our application to perform testing. We are going to use one called jest. You can find out more about jest here: <https://jestjs.io/>

Installing jest

We add jest to our application in the same way as we have added everything else so far, using node package manager:

```
npm install --save-dev jest
```

Once jest has been installed, we need to configure the application to use it for testing. This involves modifying the [package.json](#) file and adding a [jest.config.mjs](#) file to our application. We have to go through this complication because the standard version of jest doesn't work with the [.mjs](#) source files we are using in our application. There's no need to worry about this however, as the sample application for this chapter already has these settings. You can find out more about [.mjs](#)

files in the section [Require and Import](#) in chapter 4.

Writing tests

Once we have jest installed, we now need to write some tests for our code. The jest framework provides functions we can use to perform individual tests. It will then run these tests and report on the results. We can start by creating a little helper function for our tests.

```
function surveyTest(newValue) {  
  let manager = new SurveyManager();273  
  manager.storeSurvey(newValue);274  
}
```

The function `surveyTest` accepts an object parameter called `newValue` that defines a survey. It creates a new `SurveyManager` and uses it to store the survey. If the `newValue` contains an invalid survey description the `storeSurvey` method will throw an exception. The test must check that the correct exception is thrown and that the exception contains the correct message. The survey description below has no options, so it should be rejected by `storeSurvey`.

```
let missingOptions = {  
  topic: "robspizza"  
};
```

We can create a test for this behavior using the jest `test` function as shown below. The first argument to the call is a string describing the test. The second argument is the function to be called to run the test. The test below uses the `expect` method, which is given a function to test, in this case `surveyTest` with an argument of `missingOptions`. After the call of `expect` we see a condition to match, in this case a `toThrow` element which is given the text of the exception that will be thrown if the test is successful.

```
test('Missing options', () => {  
  expect(() =>
```

²⁷³ Create a manager

²⁷⁴ Use the manager to store the survey

```
    surveyTest(missingOptions)).toThrow("Missing options in storeSurvey\n");
});
```

The nice thing about jest is that if you read the code aloud it provides a very good description of what the test is going to do. “Expect the `surveyTest` with missing options to throw ‘Missing options in storeSurvey’”. Some tests use a method to perform them. To test the increment count behavior the test must create a survey, add one to the count for an option and then check that the correct option value is now 1. The `incrementTest` function below does this. The function returns an empty string if it works, or an error message if it fails.

```
function incrementTest() {
  let manager = new SurveyManager();
  let error = "";
  manager.storeSurvey(newSurveyValues);
  manager.incrementCount({ topic:"robspizza", option:"pepperoni"});
  let surveyOptions = manager.getCounts("robspizza");

  newSurveyValues.options.forEach(option => {
    let testOption = surveyOptions.options.find(item => item.text == option.text);
    if (testOption == undefined) {
      error = error + "option missing\n";
    }
    else {
      if (testOption.text == "pepperoni") {
        if (testOption.count != 1) {
          error = error + "count increment failed";
        }
      }
      else {
        if (testOption.count != 0) {
          error = error + "incremented wrong count";
        }
      }
    }
  });
  return error;
}
```

We can use the `incrementTest` function to perform a test by using a different matching function on the result of calling the test method, as shown in the code below.

```
test('Increment a count', () => {
  expect(incrementTest()).toBe("");
});
```

The `toBe` match enables a test to check for a particular result value, rather than an exception. The jest framework contains lots of different match functions. The test programs are stored in a file that has the word `test` in the filename. The tests are run from the terminal using the `npm test` command. The `npm` program opens `package.json`, finds the test command (in our case to run jest) and runs it.

Make Something Happen

Running tests

The starting point for this Make Something Happen is a version of Tiny Survey you can download from GitHub here: <https://github.com/Building-Apps-and-Games-in-the-Cloud/TinySurveyTest>. The repository contains the application with error handling and a set of tests. Clone this repository onto your machine and open it with Visual Studio Code. If you are unsure how to get the sample application, use the sequence described in the section **Get the example application** in Chapter 8. Just change the address of the repository that you clone. Open the application in Visual Studio Code and then open the `surveymanager.test.mjs` source file.



The test file contains several tests for the `SurveyManager` class. We want to run these from the terminal, but we want to run them in the debugger, so press the debugger icon in the left hand menu strip (it's the triangle with a bug on it) and then click the JavaScript Debug

Terminal button.



This terminal lets you run `npm` and node applications in the debugger from the command line. We should do this because we might want to step through the tests. Go into the terminal window and type the following command and press enter:

```
npm test
```

This will start the debugger which will load `npm` and run the test command. This starts jest running. Jest will search for JavaScript files with the word test in their name. These files will be opened and the tests in them will be run. At the end you will see a list of passed tests.



In the screenshot above I've increased the size of the terminal window so you can see more of the output from the tests. There are 8 tests at the moment. You might like to add some breakpoints to the tests and then re-run them so that you can watch them being performed.

Tests are hard work but very useful

The test file for the [SurveyManager](#) class is around 200 lines long. It took a little while to write, but it was totally worth the effort. If I make any changes to the way that the storage works (or even implement a database powered storage system as we are going to in the next chapter) I will be able to test it without doing anything other than running these tests. If you have got as bored typing in pizza toppings to test Tiny Survey as I have, you will appreciate that automatic tests are much easier. There is a lot more to testing, and the jest framework, than has been described here, so it is worth looking at what you can do with it.

Personally, I quite like writing tests. You don't need to worry too much about efficiency since tests don't run very often. It's OK to block copy lumps of code to make new tests (that's how I wrote most of the above) and you get to think about how you can attack systems and make them break by creating cunning tests. Which can be fun. Testing is also a really good way of making sure that the functions you have created are fit for purpose. There is even a development technique where you write the tests first and then fill in the code until all the tests are passed.

Many programmers don't like testing much and so they really appreciate someone who is good at it so testing might be a good way to make a name for yourself.

Logging

The final feature I think is worth mentioning here in the context of "professional" development is logging. When something bad happens to your application you should record this for future reference. In its simplest form the log could just be a message written using `console.log`. You might like to add the date that the event occurred. When your application is hosted in the cloud you can view the output from the log messages. One thing you should not do is make any of your logging output visible to the user. You might think it would be useful to enable the user to see detailed error reports of your program on the error page for your application. However, anyone trying to attack your site would find detailed descriptions of what has gone wrong very useful. You should restrict visible error reports to very small amounts of information.

Professional Coding

You don't have to do all the things above when you write a program. Sometimes I write code just to see whether other people like what it does. In that case it would be far too much effort to use the techniques above. However, if everyone liked my application and were willing to pay for it, I'd then create a version which was much more professional. This approach is used a lot in the

software industry. A “quick and dirty” prototype is created first and then a second version is built which addresses all the factors described above.

Store application status with cookies

In chapter 8 we built a Tiny Survey application. The application works well. But there is room for improvement. One issue is that people can enter multiple responses to a survey. We could use the Tiny Survey application to choose a pizza topping. However, there is nothing to stop someone particularly keen on a ham and pineapple topping from repeatedly reloading the voting page and adding more and more votes for that topping. Tiny Survey should only allow one response per user.

One way to achieve this would be to use browser local storage. We first saw this in Chapter 3 in the section “Storing data on the local machine” where we used local storage to hold the setting values for the time-travel clock. We could store a list in local storage of the survey topics a user has completed. Each time the user votes the survey topic could be added to the list. JavaScript running in the web page could use the list to stop the user from re-submitting multiple votes for the same topic. This could be made to work but the code to do this would need to run in the browser. In the current version of Tiny Survey all the code for our application runs in the node.js server and it would be simpler to keep it that way. Fortunately, there is a way that a node.js server application can store data in the browser. It is called the *cookie*.

The origins of the name “cookie” are unclear. Around the time that the cookie was invented software engineers were working on what they called “magic cookies” to send data from one place to another, so the name might have come from there. It might also be that the message passing aspect of a cookie came from fortune cookies which are hollow biscuits containing paper messages. You can think of a cookie as a lump of data which a server can give to a browser as part of a response to a web request. The browser will give the cookie back to the server when the browser next requests a page from that server.

Cookies in Tiny Survey

We could use cookies to keep track of the surveys a user has voted in. When a user votes on a survey the response from the server will include a survey list cookie that is stored by the browser. When the user visits the survey web address the survey list cookie will be sent to the server as part of the `get` request.

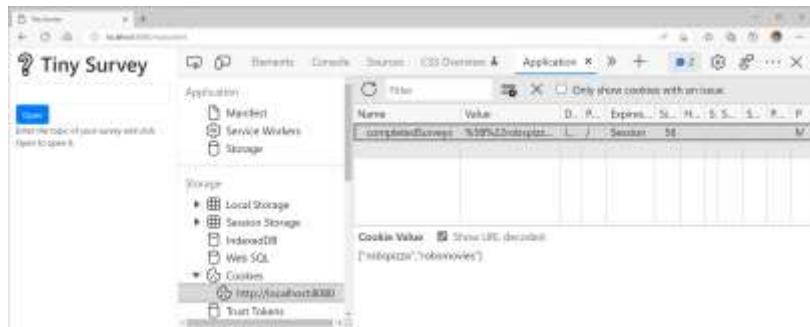


Figure 9.39 Ch-09_Fig_01 Tiny_Survey_cookies

Figure 9.1 above shows the cookie storage in a version of Tiny Survey which uses a cookie to keep track of survey responses. You can view this information by opening the Developer Tools in your browser and going to the Application tab. You can see that there is a cookie called `completedSurveys` and the cookie string contains a JSON encoded array giving the topics of two surveys; “rob-pizza” and “robsmovies”. The user has voted in these. Code in the server uses this information to determine which surveys the user is allowed to vote in. Each web address has its own area of cookie storage. The survey in Figure 9.1 is being hosted on the local computer with the address <http://localhost:8080>. You can find out more about viewing local storage in the **Make It Happen: Software Sleuthing** section in Chapter 3.

Cookie middleware

When an application is using cookies, the server adds cookie data to the response sent back to the browser and responds to cookie values received as part of `get` and `post` messages from the browser. Code to implement this can be found in the cookie-parser middleware. This needs to be installed in an application by using node package manager ([npm](#)).

```
npm install cookie-parser
```

Once the cookie parser library has been installed it can be imported to the application and added to the middleware used by the application.

```
import cookieParser from 'cookie-parser';

app.use(cookieParser());
```

The cookie-parser middleware adds a `cookies` property to the web request object and a `cookie` property to the reply object.

Use cookies in Tiny Survey

Now that we know what a cookie is we can use it in Tiny Survey. We want to stop a user being able to vote more than once. When the user opens a survey, we need to check to see if the user has already voted in the survey and take them straight to the results page of the survey if they have. Figure 9.2 below shows the first page in the survey workflow. The user clicks the Open button having entered the topic of the survey they want to interact with.

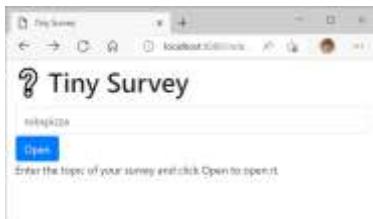


Figure 9.40 Ch-09_Fig_02 Survey start

Clicking Open causes the page to send a post to the `gottopic` route in the Express application running Tiny Survey. Below you can see the first part of the function that deals with this post. The code checks to see if the survey storage already contains a survey with this topic. If it does the code then checks to see if `completedSurveys` list contains the topic name. If it does it means that the user has already completed this survey, and so the user is directed to the results page. If not, the user is directed to the `enterOptions` page to make their choice.

```
app.post('/gottopic', (request, response) => {

  let topic = request.body.topic;

  if (surveyManager.surveyExists(topic)) {
    // Need to check if the survey has already been filled in
    // by this user
    if (request.cookies.completedSurveys) {
      // Got a completed surveys cookie
      // Parse it into a list of completed surveys
      let completedSurveys = JSON.parse(request.cookies.completedSurveys);
      // Look for the current topic in the list
      if (completedSurveys.includes(topic)) {
        // This survey has already been filled in using this browser
        // Just display the results
        let results = surveyManager.getCounts(topic);
      }
    }
  }
})
```

```

        response.render('displayresults.ejs', results);
    }
}
else {
    // enter scores on an existing survey
    let surveyOptions = surveyManager.getOptions(topic);
    response.render('selectoption.ejs', surveyOptions);
}
}

```

Now that we have dealt with the situation where the survey exists, it is time to handle the scenario where the user enters a topic for a survey that doesn't exist. In this situation the user should be directed to the survey entry page so that they can create a new survey. I thought this code would be quite simple to write, but it turned out to be more complicated than I thought. This was because of the possibility that a survey might not exist in the survey store, but it might have an entry in the `completedSurveys` cookie. This happens if the user fills in a survey and then restarts the Tiny Survey application, clearing the survey store. If the user re-enters a survey with the same topic they will be unable to vote in the new survey because of the entry in `completedSurveys`. The code below deletes a survey topic from the cookie if the survey topic is not found in the storage.

```

app.post('/gotopic', (request, response) => {

    let topic = request.body.topic;

    if (surveyManager.surveyExists(topic)) {
        // Code to handle an existing survey
    } else {
        // There is no existing survey - need to make a new one
        // Might need to delete the topic from the completed surveys
        if (request.cookies.completedSurveys) {
            // Get the cookie value and parse it
            let completedSurveys = JSON.parse(request.cookies.completedSurveys);
            // Check if the topic is in the completed ones
            if (completedSurveys.includes(topic)) {
                // Delete the topic from the completedSurveys array
                let topicIndex = completedSurveys.indexOf(topic);
                completedSurveys.splice(topicIndex, 1);
                // Update the stored cookie
                let completedSurveysJSON = JSON.stringify(completedSurveys);
                response.cookie("completedSurveys", completedSurveysJSON);
            }
        }
    }
}

```

```

        }
    }
    // need to make a new survey
    response.render('enteroptions.ejs',
      { topic: topic, numberOfOptions: 5 });
}
);

```

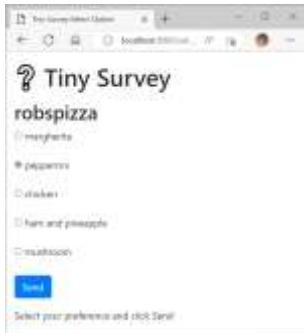


Figure 9.41 Ch-09_Fig_03 Survey select option

Figure 9.3 above shows the survey page which is used to enter a vote. Clicking Send will post the selection to the `recordselection` route. The code in this route needs to update the count and add the survey topic to the list of completed surveys stored in the `completedSurveys` cookie. The incoming cookie contains a JSON string describing the array holding the list of survey topics. The code converts the JSON string into a JavaScript object (the list of survey topics). It then checks to see if the list contains the current survey topic. If the topic is in the list the user is directed to the `displayresults` page. This stops the user from adding more votes by refreshing the option selection page in their browser. If the `completedSurveys` list doesn't contain the current survey topic the code records the vote, adds the survey topic to the list of completed surveys and then sets the cookie in the response to contain the updated survey list. The code below does this. There are plenty of comments and it repays careful study.

```

// Got the selections for a survey
app.post('/recordselection/:topic', (request, response) => {
  let topic = request.params.topic;

  if (!surveyManager.surveyExists(topic)) {
    // This is an error - display survey not found
    response.status(404).send('<h1>Survey not found</h1>');
  }
  else {

```

```
// Start with an empty completed survey list
let completedSurveys = [];
if (request.cookies.completedSurveys) {
    // Got a completed surveys cookie
    completedSurveys = JSON.parse(request.cookies.completedSurveys);
}
// Look for the current topic in completedSurveys
if (completedSurveys.includes(topic) == false) {
    // This survey has not been filled in at this browser
    // Get the text of the selected option
    let optionSelected = request.body.selections;
    // Build an increment description
    let incDetails = { topic: topic, option: optionSelected };
    // Increment the count
    surveyManager.incrementCount(incDetails);
    // Add the topic to the completed surveys
    completedSurveys.push(topic);
    // Make a JSON string for storage
    let completedSurveysJSON = JSON.stringify(completedSurveys);
    // store the cookie
    response.cookie("completedSurveys", completedSurveysJSON);
}
let results = surveyManager.getCounts(topic);
response.render('displayresults.ejs', results);
}
});
```

Programmer's Point

Sometimes you just have to write messy code

The code to handle different combinations of survey storage and cookie state might look a bit messy to you. It would be lovely if there was a really neat way of implementing this logic with just a couple of lines of JavaScript. Unfortunately there isn't. Sometimes, particularly when creating user interfaces, you just end up having to write a sequence of steps to make sure that every unlikely eventuality is properly handled. This is one situation where lots of comments do make a huge difference.

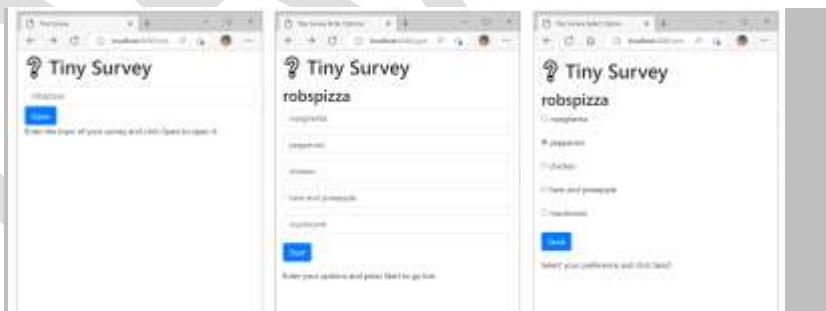
Make Something Happen

Investigate cookies in Tiny Survey

The starting point for this Make Something Happen is a version of Tiny Survey you can download from GitHub here: <https://github.com/Building-Apps-and-Games-in-the-Cloud/TinySurveyCookies> Clone this repository onto your machine and open it with Visual Studio Code. If you are unsure how to get the sample application, use the sequence described in the section **Get the example application** in Chapter 8. Just change the address of the repository that you clone.

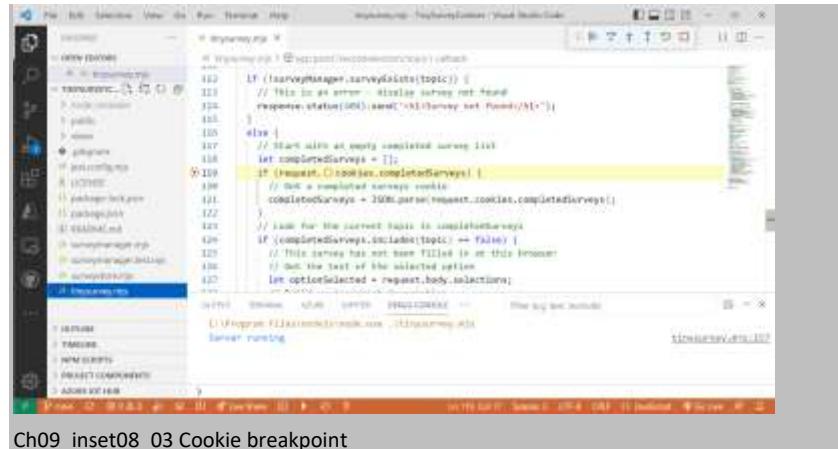
Ch09_inset08_01 Cookie handling breakpoint

Set a breakpoint at line 119 as shown above. Now open your browser and navigate to **localhost:8080/index.html** where you will find a working Tiny Survey application. Enter a new survey, add the survey options and then vote as shown below.



Ch09_inset08_02 Fill in a survey

When you click **Send** the browser will pause because the server hits the breakpoint at line 102. The server is running the function for the `recordselection` route. Click [step into function](#) in the **debug controls** to work through the code.

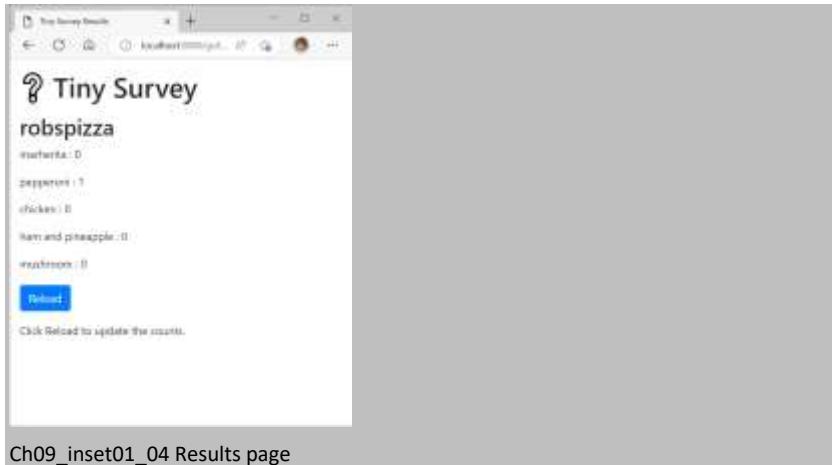


Ch09_inset08_03 Cookie breakpoint

The first time that the browser visits the survey page there will be no cookies, so the code will start with an empty `completedSurveys` list. If you continue to step through the code you will see the counts updated and following three statements are performed:

```
// Add the topic to the completed surveys
completedSurveys.push(topic);
// Make a JSON string for storage
let completedSurveysJSON = JSON.stringify(completedSurveys);
// store the cookie
response.cookie("completedSurveys", completedSurveysJSON);
```

The first statement adds the survey topic to the `completedSurveys` list. The second statement creates a JSON string that describes the list and the third stores this in the cookies for this site with the name `completedSurveys`. Step through these statements to watch them work and then press the continue button in the debug controls to restart the server. Now go back to the browser and open a new tab at the address **localhost:8080/index.html** Now enter the same survey topic (in my case `robspizza`) and click open. You will now be directed straight to the results page for the survey.



Ch09_inset01_04 Results page

If you want to perform further investigations, you could put a breakpoint in the server at line 30. Then reload the index page for the survey, enter the topic and watch as the server gets the cookie and checks for a survey topic. You can also use the Developer Tools in the browser to view the cookie that was created, as we saw in Figure 9.1

CODE ANALYSIS

Using Cookies

You may have some questions about cookies.

Question: How long do cookies live?

Normally cookies are deleted when the browser program is closed. If you want cookies to hang around longer you can give them a maximum age or an expiry date when you create them.

```
response.cookie("completedSurveys", // cookie name  
                completedSurveysJSON, // cookie string  
                {maxAge:1000*60}); // age of 60 seconds
```

The above cookie would only be stored for 60 seconds. The `maxAge` option specifies the maximum age of the cookie in milliseconds.

```
response.cookie("completedSurveys", // cookie name  
                completedSurveysJSON, // cookie string  
                { expire: 24*60*60*1000 + Date.now()}); // this time tomorrow
```

The above cookie would expire a day after it was created. The `expire` option lets you specify a date when the cookie will be removed. The `expire` value is a date expressed in milliseconds.

A user could opt to remove all the cookies in their browser at any time, so it is important not to store application critical data in a cookie.

Question: Can I stop people viewing the contents of my cookies?

No. The best way to keep cookie contents secret is to encrypt the cookie string before it is sent to the browser. We will be doing this in the next chapter when we store session data in cookies.

Question: Can I stop anyone tampering with my cookies?

No. A browser could send any cookie value back to the server. However, you can “sign” the value in a cookie so that any tampering would be detected. To do this you have to give the cookie middleware a “secret” when it is created:

```
app.use(cookieParser("encryptString"));
```

The `cookieParser` above uses the string “`encryptString`” as the secret. Now, when you store a cookie you must ask for it to be signed by adding an option:

```
response.cookie("completedSurveys", // cookie name  
    completedSurveysJSON, // cookie string  
    {signed:true}); // sign the cookie
```

When the cookie is created the cookie contents are combined with the secret string to create a validation string which is appended to the cookie. The same process is repeated with received cookie values. If the validation strings don’t match the cookie is not valid and will be ignored. To get your incoming signed cookies processed the program uses the `signedCookies` property of the request to get the stored cookie:

```
let completedSurveysJSON = request.signedCookies.completedSurveys;
```

Note that it is a very bad policy to put strings such as “`encryptString`” directly into your source code as we have done above. Later we will discover how to separate secret information from the program code.

Question: Do all browsers share the same cookies?

No. If you visit a site using Chrome and again using Edge you will discover that the cookies stores are different.

Question: Can I manually delete cookies from my browser?

Yes. You can use the Application tab in the browser developer tools. This makes it possible for a determined person to vote more than once in Tiny Survey. They would have to delete the cookie after each vote.

Question: Can the Tiny Survey application delete cookies?

Yes. The `clearCookie` function can be called on a response to clear that cookie in the browser. The function accepts the name of the cookie as a string.

```
response.clearCookie("completedSurveys");
```

The above statement removes the `completedSurveys` cookie.

Question: Which is the best cookie?

My preference is for chocolate chip, but I also like oatmeal ones a lot.

Programmer's Point

You should tell people you are using cookies

The Tiny Survey application only uses cookies to keep track of the surveys that a user has taken part in. It doesn't really have any privacy implications. Even so, it is considered polite to let the user know that you are using cookies and perhaps even offering them the option to not use the application if they don't want to.

What you have learned

In this chapter we have examined aspects of software construction from a professional perspective. We've learned how we can break an application into separate modules to make it easier to work with. We've also seen how to create well-defined interfaces between modules and how we can generate interactive documentation by adding JSDoc comments to the code. We've also learned the importance of error checking and how an application can use exceptions to indicate error conditions. We've worked with a test framework and built a set of unit tests for a module. Finally we've taken a look at cookies and discovered how they can be used to allow an application to retain state information in the browser. Here's a recap plus some points to ponder.

- JavaScript applications can be broken down into modules that can be individually written and tested. A module can contain a class which contains methods which implement the behaviors the module provides.
- JavaScript code can contain comments in the JSDoc format. These can be created in Visual Studio Code by entering the sequence `/**` on the line above the item to be documented. If the item is a function or method Visual Studio Code will create template comment to be filled in. These comments

can be used to create documentation web pages and are also used by the Visual Studio Code editor to display information as code is being entered.

- JavaScript behaviors should contain error checking. User errors should be handled by the workflow of an application. Other faults that are detected can be handled by JavaScript exceptions.
- A program can use the JavaScript `throw` to throw an object during program execution. The `throw` ends the current sequence of execution and transfers execution to a `catch` block or ends the sequence if no `catch` block is present. The object can describe the event that caused the `throw`. For exceptions to be caught the code must be running inside a `try` block which should be followed by a `catch` block which contains code to run when the exception is thrown. Exceptions can be nested; code in a `try` block can contain other `try – catch` constructions. When an exception is thrown the catch code that runs is that “nearest” to the `throw`. It is impossible to return to code after an exception has been thrown.
- A `try – catch` construction can be followed by a `finally` block which contains code that runs irrespective of what happens in the `try – catch` construction. A `finally` block can be used to release resources which were allocated in the `try` block.
- The jest framework can be used to manage automated testing of JavaScript code. A test file can contain a large number of tests that should each deliver a particular result. Jest provides a range of matching mechanisms to check test outputs. It will automatically run a number of tests and indicate which have passed.
- Applications should log output if they detect errors while running, but they should not make detailed error reports to publicly visible on the site they are hosting.
- A cookie is a small piece of data which an application can send to the browser as part of a response to a web request. The browser will return the cookie content as part of future requests to the same site address.
- A cookie contains name-string pairs. An application can store data values in a cookie by encoding the value as a JSON string.
- Cookies for a site are visible in browser using the Developer Tools Application view of the site. They can also be deleted using this interface.
- A browser will delete cookies when the browser program is closed. Cookies can be given a lifetime or an expiry date if they are to be held for longer. An application can also delete cookies.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions".

Should we always try to write "professional" code?

No. But you should be aware of situations where it might be a good idea to take on at least a few of the professional considerations described here.

How many functions should a module provide?

A module should be concerned with one purpose. You could have a storage module, a printing module, a user-menu module and so on. If you find a module getting large you might want to break it into sub-modules for different areas of functionality. While you can try and decide on each module and the things it should do at the start of a development, it is often hard to do this. Programmers talk about "refactoring", where they move behaviors in and out of different modules, or change what they do. This should be a continuous process during a development. You should not stick with a poor design just because it was the first thing you thought of.

If a function detects an error, should it return an error value or throw an exception?

A function can return a value that indicates whether it worked. However, a failure of the function would only be detected if the program that called it checks the returned value. However, a function that throws an exception when it detects an error will interrupt the flow of the code calling it, which is a more active form of error reporting. The worst thing that a program can do is fail silently. If it crashes, at least you know it is broken. But if fails to save data but doesn't display an error, that is much more of a problem because you might use it for a while before you discover the fault. Viewed in this light throwing an exception is a good idea because there is less chance the error will be missed.

Why do we need the `finally` part of a `try-catch` construction? Can't we just put code after the `try - catch` construction?

This is a good question. But we do need `finally`. Perhaps the code in the `catch` block throws another exception, in which case the code following the `try - catch` will never get to run. Perhaps the `try - catch` construction is running inside a function and when the `catch` block runs it returns from that function. A `finally` block is the only way we can make sure that code gets run.

When should you write your tests?

Testing is not something you should think about after you've written your code. It should be part of the development process. Over the years I've discovered lots of issues with my code design while writing the tests for it. Designing tests also forces you to think hard about the specification.

How many tests should you write?

One of the problems with test creation is knowing when to stop. There should be at least one test for every function and then one to test any failure conditions.

Where are cookies stored?

Cookies are stored in the local filestore of the browser.

Are cookies the only way an application can store data on a per-user basis?

You can also use browser local storage to store data, but this has to be performed by a program running in the browser itself. It is also possible for a browser and a server to maintain session information in the url used to address the site. We will investigate this in the next chapter.

Chapter 10:

Store Data

What you will learn

In this chapter we are going to build and deploy a database powered application. Along the way we'll discover how JavaScript applications can interact with file store and databases and how the asynchronous features of the language let us work with processes that may take some time to complete. We'll discover how to design and connect to database storage using the MongoDB database and Mongoose middleware. Then we'll prepare the Tiny Survey application for deployment, in the process creating a development environment and then finally we'll configure and deploy a working application.

File Data storage

The Tiny Survey application is almost ready for deployment. However, it does have one major flaw. The present version stores the survey data in an array variable in the program. When the program stops the variables are discarded. We need to create a version which persists the survey data. We could store the survey data in files on the server. The [node.js](#) framework provides a library of functions we can use to interact with files. So, let's set about adding file storage to Tiny Survey. We can start with a look at how we can open a file and write to it.

The first thing we need to do is import the file functions. We use [import](#) to bring existing JavaScript code into our programs. You can find out all about it in chapter 4 in the section [JavaScript Heroes: Modules](#). The statement below uses a form of [import](#) we've not seen before. It uses a wild card character (*) to indicate that the import is to fetch all the items in the source.

```
import * as fs from 'node:fs/promises';
```

This brings in the “promise” versions of the input/output functions. What do we mean by promise? Time for some more JavaScript heroes.

JavaScript Heroes: promise, then and await

JavaScript is a “single threaded” language, which means that a JavaScript program can only ever do one thing at a time. The engine running a JavaScript program keeps a list of active events and works through the list running functions connected to each event. When you click a button on a web page the event handler function is added to the event list and will run at some point in the future. All event handlers should complete as quickly as possible. If one gets “stuck” all of JavaScript gets stuck. Functions will be added to the event list, but not run. If JavaScript in a web page gets stuck the page becomes unresponsive to actions on the page. If JavaScript in a node.js server gets stuck it means that the server stops responding to incoming web requests. However, we know that some actions that the programs must perform take a long time (in computing terms) time to complete. We can do this using the “asynchronous” features of JavaScript.

Asynchronous code

A JavaScript function or method can be declared as *asynchronous* by adding the keyword [async](#) at

the start of the function or method declaration. This means that it doesn't return a result, it returns a `Promise` object. What does this mean? Let's go back to my childhood for an explanation. It turns out that there were two ways that mum could get me to tidy my bedroom. One was to stand over me and watch me do it. The other was for mum to accept a promise from me that I would tidy my bedroom at some point in the future. Mum preferred the second arrangement as it meant she was free to go off and ask someone else to tidy their room. When mum was standing watching me tidy my room she was synchronized to my actions. She couldn't leave until I'd finished. If she accepted my promise she could do other things while I was tidying. She wasn't synchronized to my actions. I was tidying my room *asynchronously* with her.

A method which is called asynchronously returns instantly with a reference to a `Promise` object describing the task that the method was asked to perform. The `Promise` object exposes methods that we can use to interact with the task. One of these methods is called `then`. The `then` method accepts a function to be run when the promise is fulfilled.

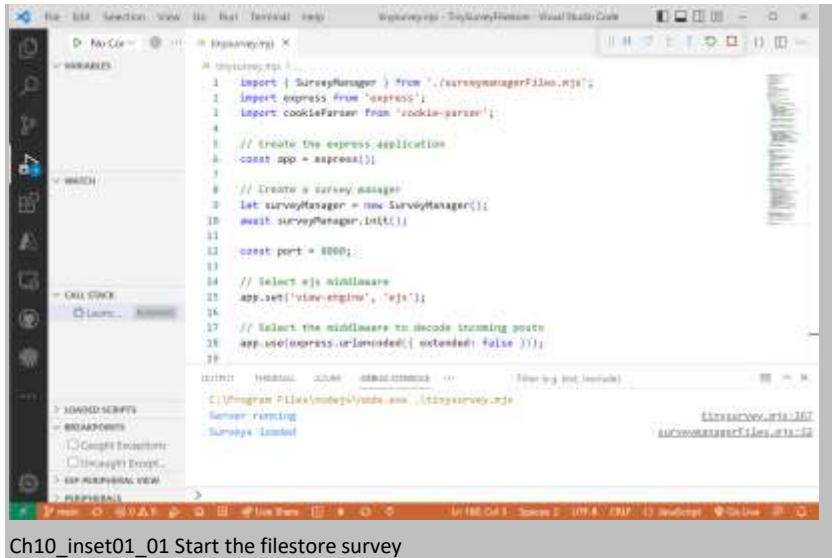
The `storeSurveys` function below shows how we use this. This function takes the survey values, makes them into a JSON string and then saves them in a file using the `writeFile` function from the `fs` library. The `writeFile` function has been declared as asynchronous, so it returns a promise. The code calls the `then` function on the promise to make the function display a message when the surveys have been stored.

```
storeSurveys() {  
  let surveysString = JSON.stringify(this.surveys);  
  fs.writeFile(this.fileName, surveysString).then(()=>console.log("File Written"));  
  console.log("Started storing");  
}
```

Make Something Happen

Saving surveys

The starting point for this Make Something Happen is a version of Tiny Survey you can download from GitHub here: <https://github.com/Building-Apps-and-Games-in-the-Cloud/TinySurveyFileStore> Clone this repository onto your machine and open it with Visual Studio Code. If you are unsure how to get the sample application, use the sequence described in the section **Get the example application** in Chapter 8. Just change the address of the repository that you clone. Open the application in Visual Studio Code and open the `tinysurvey.mjs` source file and start the program running. Make sure the **DEBUG CONSOLE** is selected in the terminal view.



```
tinySurvey.js
1 import { SurveyManager } from './surveymanagerfile.mjs';
2 import express from 'express';
3 import cookieParser from 'cookie-parser';
4
5 // Create the express application
6 const app = express();
7
8 // Create a survey manager
9 let surveyManager = new SurveyManager();
10 await surveyManager.init();
11
12 const port = 8080;
13
14 // Select ejs middleware
15 app.set('view-engine', 'ejs');
16
17 // Select the middleware to decode incoming posts
18 app.use(express.urlencoded({ extended: false }));
19
```

Ch10_inset01_01 Start the filestore survey

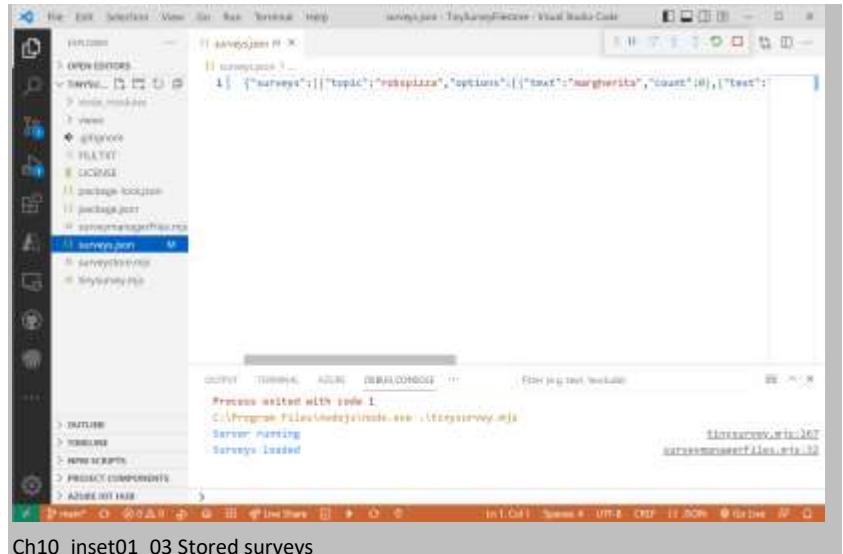
Visual Studio will show you the program running in the debugger. The debug console will show the messages “Server running” and “Surveys loaded”. Use the browser to navigate to **localhost:8080/index.html** and enter a survey and select an option. Now return to the Visual Studio Code window.



```
C:\Program Files\nodejs\node.exe ./tinySurvey.mjs
Server running
Surveys loaded
Started storing
File Written
```

Ch10_inset01_02 Saving surveys

When a survey or a vote is entered the surveys are saved by the method `storeSurveys` we saw earlier. It prints the messages above. Click the red square in the **debug controls** to stop the program and then restart it. If you return to the browser and re-open the survey you created you will find it has been retained. If you return to Visual Studio code you will find that there is a new file in the project called **surveys.json**. Open that file in the editor.



This file contains the survey contents as a JSON string. The file is updated each time a new survey is entered and after an option has been selected.

CODE ANALYSIS

Asynchronous survey storage

You may have some questions about how the asynchronous file storage works.

Question: Why does the code in the `storeSurvey` method have the messages the wrong way round?

If you understand the answer to this question, you understand how asynchronous code works in JavaScript. The code we are talking about is the two statements in the `storeSurveys` method. You can see them below. The first statement writes the surveys into a file and the second statement prints that the storing process has started. It seems that these are the wrong way round. How can the code print "Started storing" first. However, if we look at the behavior in the "Make It Happen Saving Surveys" above we can see that it works. What is going on?

```
fs.writeFile(filename,surveys).then(()=>console.log("File Written"));
console.log("Started storing");
```

The key to understanding what is happening is remembering that the function in the `then`

part of the first statement runs after the `writeFile` method has completed. Just because the code appears in the program first does not mean it runs first. The `writeFile` function has been made asynchronous so that calling code doesn't wait for it to finish. The code above just logs a message after `writeFile` has been called, but it could go on and do other things. If you are still finding this confusing, consider another way of expressing what we are doing. The code below uses a variable called `writePromise` to hold the promise returned by the call to `writeFile`. The `then` function is called on `writePromise` to specify the function to be called when the promise is fulfilled.

```
let writePromise = fs.writeFile(filename,surveys);
writePromise.then(()=>console.log("File Written"));
console.log("Started storing");
```

Question: What happens if we change the contents of the `surveys` array before the output file has been written?

This is a good question. The answer is that your program will probably fail. When you make an asynchronous call, you create another sequence of program execution. This can be dangerous. We can use it to shoot ourselves in the foot. The code below is from a very badly written data storage app. Users enter a data record which is then saved. After the record has been saved it is set to `null` ready for next use. Can you spot the bug?

```
fs.writeFile(filename,record).then(()=>console.log("File Written"));
record = null;
```

Remember that the statement following the call of `writeFile` is executed before the `writeFile` function has completed because `writeFile` is an asynchronous function which returns a promise immediately after it was called. The code would delete the record storage before the file has been written. The correct way to write this code is as follows:

```
fs.writeFile(filename,record).then(()=> {
  console.log("File Written");
  record = null;
});
```

Now the record is cleared in the `then` part of the code. This runs after `writeFile` has completed.

Question: What happens if `writeFile` never completes?

There is always the possibility that a file operation can fail. Perhaps the drive is full or disconnected. In that case the surveys would not be saved and the message "File Written" will never be displayed. Unfortunately, the survey program will keep on running, giving the user the impression that all is well. Later in this chapter we will discover how we can manage conditions like these.

Question: How does a promise actually work?

At the start of this section we observed that JavaScript is single threaded. But then we've introduced asynchronous operation, implying that two things can happen at the same time when a JavaScript program is running. How can this be so? We get asynchronous operation by very clever use of events. Imagine that writing a file involves waiting for the disk to be ready, sending a block of data to the disk, and waiting for the disk to finish the write operation. The program asks for the disk to get ready and creates an event that will fire when the disk is ready. When the disk is ready the event fires and runs some code which sends the data to the disk and then creates an event which will fire when the write is complete. When an event "fires" the handler for the event is added to the JavaScript event list to be run when JavaScript gets around to it. You can think of a promise as a binding of an event to code which will run when the promise is fulfilled and move an operation onto the next step.

We've seen that we can use `promises` and `then` to create a program which can apparently do two things at the same time. The `storeSurveys` function above appears to write a message while `writeFile` function is writing to a file. We've seen how this ability can be dangerous, in that it makes it possible for one sequence of execution to interfere with another, as in the case above where a record was cleared before it had been stored. It turns out that what we want most of the time is something that will wait for an action to complete without blocking a JavaScript program from execution. We can use another JavaScript hero, `await` to do this for us.

The await keyword

We've seen how we can use the `then` part of a promise to perform a "background" action while our program runs. But most of the time we don't want to do anything while we wait for an action to complete. We can do this by using the JavaScript `await` keyword. The `loadSurveys` method below reads the surveys back into the Tiny Survey application. It uses the `fs.readFile` function to read the contents of the file into a string and then uses `JSON.parse` to convert the string into an array of JavaScript objects which can be used to construct new `Survey` instances.

```
async loadSurveys() {
  try {
    let surveysString = await fs.readFile(this.fileName);275
    let surveyValues = JSON.parse(surveysString);276
    let result = new Surveys();277
    surveyValues.surveys.forEach(surveyValue => {278
```

²⁷⁵ Await the read file

²⁷⁶ Convert to a list

²⁷⁷ Make an empty survey store

²⁷⁸ Work through the incoming surveys

```

        let survey = new Survey(surveyValue);279
        result.saveSurvey(survey);280
    });
    console.log("Surveys loaded");
    this.surveys = result;281
}
catch {282
    console.log("Survey file not found - empty survey created");
    this.surveys = new Surveys();283
    this.storeSurveys();284
}
}

```

The call of `fs.readFile` is preceded by the `await` keyword. The `await` keyword takes the `Promise` that is returned by `fs.readFile` (because `fs.readFile` is an asynchronous function) and returns it to the caller of `loadSurveys`. It will then run the rest of the `loadSurveys` method when the `fs.readFile` completes. The `loadSurveys` method is made `async` because it returns a `Promise` object. The great thing about `await` is that it makes asynchronous code look a lot like synchronous code. We want the program to wait until `fs.readFile` has finished before it processes the contents of the stored file.

When JavaScript sees an `await` it does some fancy footwork behind the scenes and creates a `Promise` and a `then` construction which contains the code which follows the asynchronous call. Programmers call this kind of thing *syntactic sugar*. A piece of syntactic sugar is a language feature which doesn't make anything possible you couldn't do before. It just makes something easier. Just like how adding sugar to your food doesn't always improve your diet, it just makes the food taste nicer. You could completely ignore the `async` keyword and write all your calls of asynchronous methods using `then` but using `async` does result in neater looking code. There is a lot more to the JavaScript `Promise`. You can fire off multiple tasks and use `Promise.all` to wait until all of the tasks have completed. You can use `Promise.race` to run multiple tasks and complete when one of them finishes (this is how you can implement timeouts on operations).

²⁷⁹ Create a new survey instance from the incoming data

²⁸⁰ Store the survey

²⁸¹ Set the surveys to the loaded result

²⁸² Exception thrown if `readFile` fails

²⁸³ Make an empty survey

²⁸⁴ Store it

Using async in Tiny Survey

The Tiny Survey application uses a [SurveyManager](#) class to manage interactions with the survey storage. The [SurveyManager](#) loads and saves surveys and fetches survey data and options for display in the web pages. Below you can see the first part of the [SurveyManagerFiles](#) class which uses asynchronous storage. The class has acquired an [init](#) method which loads the surveys.

```
class SurveyManagerFiles {  
  
    constructor() {  
        this.fileName = "surveys.json";  
    }  
  
    async init() {  
        await this.loadSurveys();  
    }  
    // rest of SurveyManager  
}
```

CODE ANALYSIS

SurveyManagerFiles class

You may have some questions about this class.

Question: Why don't we load the surveys in the constructor for the [SurveyManagerFiles](#) class?

The previous version of [SurveyManager](#) stored the surveys in a list variable which was created in the constructor for the class. This meant that the storage was created when manager was created. This version of the class has a separate [init](#) method which loads the surveys. You might be wondering why this is. It is because a constructor for a class cannot be an asynchronous function. A constructor is a method that must return an object, not a promise to make an object.

Question: What happens if the [init](#) function never returns?

We've already noticed that operations involving files may fail or not complete. If the promise returned by [loadSurveys](#) is never resolved the [init](#) function will never return because it is using an [await](#) on that promise. The sequence of statements in the [tinysurvey.mjs](#) program shown below means that if the surveys are not loaded the web server never starts listening.

Anyone trying to access the survey website will get a “site not found” message. In the next section we will look at better ways of managing errors like these.

```
let surveyManager = new SurveyManager();285
await surveyManager.init();286
const port = 8080;287
app.listen(port, () => {288
  console.log("Server running");
})
```

Question: How do we call asynchronous functions in Express routes?

The Express framework runs code to deal with incoming web requests. We know that any function which contains an `await` must be marked as asynchronous. How do we do this? The code below shows how this is done. The method handling the `request` and `response` items is marked as asynchronous by using `async`.

```
app.post('/recordselection/:topic', async (request, response) => {
  ...
})
```

Handle file errors

The code we have written so far would handle a failed file open by never starting the Express server. This means that browsers trying to connect to the Tiny Survey application would display “Can’t reach this page” messages when the user tried to start a survey. The application should do better than this. At least it should display a message saying that the site is broken, otherwise people might think they have typed in the wrong address. To do this we need to design some form of error handling for the Tiny Survey application.

Programmer’s Point

You should design your error handling at the start

We’ve already seen that you start the design of an application by considering the

²⁸⁵ Make the survey manager

²⁸⁶ Wait for it to load the data

²⁸⁷ Set the server port

²⁸⁸ Start the server listening

workflow of the application. You should do something similar with errors. Before you build the program you need to consider what could go wrong and how the code will deal with it. The previous version of Tiny Survey was completely self-contained. It didn't rely on any other service which might fail. However, a version of Tiny Survey which uses a file to store surveys is reliant on that file, so we need to consider what the application should do if the file cannot be opened when the program starts.

We need to manage the situation where the application is active and responding to requests, but the database is not available. We can do this by creating a flag variable in the application which tracks whether the database is connected as shown below:

```
let surveysLoaded = false;
```

The initial value of the `surveysLoaded` flag is `false`. Now we can add some code that will set the flag to `true` when the surveys have been loaded. The code below initializes the `surveyManager` by calling the `async` method `init`. When the `init` method completes the arrow function in the `then` part is called. That sets the `surveysLoaded` flag to `true`, telling the application that the storage is now loaded. If the application gets a request when the survey is not initialised it can display an error page.

```
surveyManager.init().then(() => {289  
    surveysLoaded = true;290  
});
```

The code below handles the `index.html` route. It runs when a browser requests the `index.html` page. It checks to see if the surveys are loaded. If they are the `index.ejs` page is rendered. If not, the `statusCode` of the response is set to 500 (which means server fault) and the `error.ejs` page is rendered. We've seen the status code before. It is how a server lets a browser know whether their request was successfully dealt with. A status code of 200 means success, a code of 404 means the requested resource was not found and 500 means there was something wrong at the server end.

```
// Home page  
app.get('/index.html', (request, response) => {
```

²⁸⁹ Arrow function runs when init has completed

²⁹⁰ Sets the database flag to true

```
if (surveysLoaded) {291  
    response.render('index.ejs');292  
}  
else {  
    response.statusCode = 500;293  
    response.render('error.ejs');294  
}  
};
```

The `error.ejs` file contains the page shown in Figure 10.1 below.

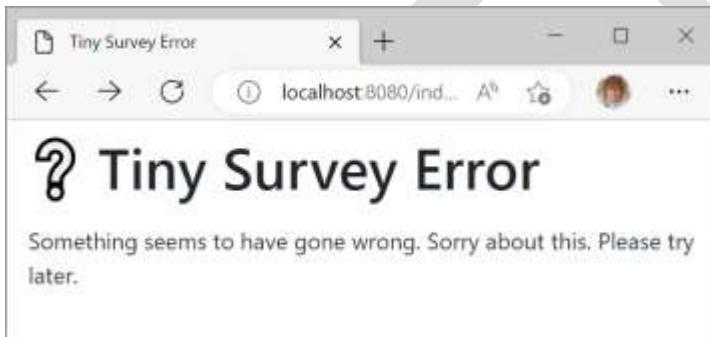


Figure 10.42 Ch-10_Fig_01 Tiny_Survey_errors

Create error handling middleware

We've seen how we can add code to a route handler which displays an error page if the survey data has not been loaded. We could work through the `tinysurvey.mjs` file and add the test to every single route. However, there is a much easier way of adding this error handling to a route, and that is by creating our own piece of *middleware*. We've talked about middleware before as code that is inserted into an application to add functionality. We've added middleware to the Tiny Survey application to render pages and process cookies. Now we are going to add middleware to a route which will deal with this error condition in a route.

The code below looks very like the test that we have just added to the `index.html` route.

²⁹¹ Are the surveys loaded?

²⁹² Render the index page if it is

²⁹³ Set the response status to 500

²⁹⁴ Render the error page

However, it also looks a bit like a route handler. The `checkSurveys` function can be inserted as a piece of middleware into a route. It will be called when the route is processed. It is provided with three parameters. The first two are the `response` and the `request` objects that are passed into a route. The third is a reference called `next` which refers to the next function in the chain of middleware. If the surveys have been loaded the middleware will call the `next` to continue the processing of this path. If the surveys have not been loaded the function sets the status code to 500 and then renders the error page. Note that in this case the `next` function is not called because the route cannot continue to the next stage. The `surveysLoaded` flag is imported from the `tinysurveys.mjs` file and the `checkSurveys` function is exported from the file.

```
import { surveysLoaded } from '../tinysurvey.mjs';295

function checkSurveys(request, response, next) {296
    if (surveysLoaded) {297
        next();298
    }
    else {299
        response.statusCode = 500;300
        response.render('error.ejs');301
    }
}

export { checkSurveys };
```

The code above is placed in a file called `checkstorage.mjs` which is in a folder called `helpers` which is part of the solution. It is imported into `tinysurvey.mjs` as shown in the code below:

-
- ²⁹⁵ Import the loaded flag
 - ²⁹⁶ Create our
 - ²⁹⁷ Do we have loaded surveys?
 - ²⁹⁸ Call the next piece of middleware
 - ²⁹⁹ Surveys not loaded
 - ³⁰⁰ Set the status code to server error
 - ³⁰¹ Render the error page

```
import {checkSurveys} from './helpers/checkstorage.mjs';
```

Once the `checkSurveys` function has been imported it can be added to a route. The code below shows how this is done. The name of the function is added to the path. When the path is followed the `checkSurveys` middleware will run before the body of the handler function. If the `checkSurveys` middleware function doesn't make a call to `next` the body of the path will not run at all.

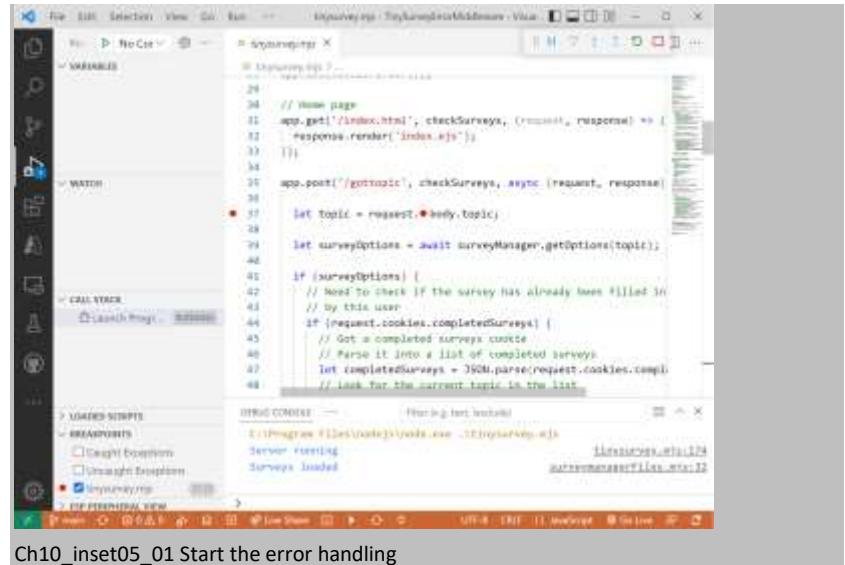
```
app.get('/index.html', checkSurveys, (request, response) => {
  response.render('index.ejs');
});
```

We can add a call of `checkSurveys` to all the routes in the `TinySurvey` application so that none of them will work if the storage is not loaded. We can also add a chain of multiple middleware functions if we like. In chapter 11 when we created a password protected site we will add the password testing element as a piece of middleware.

Make Something Happen

Using middleware

The starting point for this Make Something Happen is a version of Tiny Survey you can download from GitHub here: <https://github.com/Building-Apps-and-Games-in-the-Cloud/TinySurveyFileStoreErrorMiddleware> Clone this repository onto your machine and open it with Visual Studio Code. If you are unsure how to get the sample application, use the sequence described in the section **Get the example application** in Chapter 8. Just change the address of the repository that you clone. Open the application in Visual Studio Code and open the `tinysurvey.mjs` source file and start the program running. Make sure the **DEBUG CONSOLE** is selected in the terminal view.



Visual Studio will show you the program running in the debugger. Set a breakpoint at line 37 as shown above. Now use the browser to navigate to localhost:8080/index.html and enter a survey topic and click Start. Now return to the Visual Studio Code window. You will find that the server has hit the breakpoint that you set. Now you are going to do something completely awesome. You are going to change the contents of the `surveysLoaded` variable using the debug console. The console is at the bottom of the Visual Studio windows. Enter the following into the console and press enter.

```
surveysLoaded = false
```

The debug console can be used when a program is paused at a breakpoint. It is just like the debugger console we've seen in the browser Developer Tools. We can type in JavaScript statements and they will be executed. We can change or view variable values.

```

 37 let topics = request.query.topics;
 38
 39 let surveyOptions = await surveyManager.getOptions(topics);
 40
 41 if (surveyOptions) {
 42   // Need to check if the survey has already been filled in
 43   // by this user
 44   if (surveyOptions.completedSurvey) {
 45     // Get a completed survey's details
 46     // Parse it into a list of completed surveys
 47     let completedSurveys = JSON.parse(request.responseText);
 48     // Look for the current user in the list

```

Ch10_inset05_02 Change surveyLoaded in the console

If we set `surveysLoaded` to `false` it makes the application think that the surveys have not been loaded. Press the right pointing arrow in the debug controls to resume the server running. You will not see the error page just yet because the middleware will run at the start of the next route handler. Navigate to `localhost:8080/index.html` again in the browser and you will see the error page displayed.

Database storage

We can use files on the server to hold the data for the Tiny Survey application, but this is not very efficient. The entire survey file is overwritten every time something changes. It would be useful if there was something we could use that would let us manipulate individual records in the survey store rather than having to write the whole file each time. It turns out that there is, and it is called a database.

JavaScript can be used with many different database engines. We are going to use one called MongoDB (<https://www.mongodb.com/home>). This is a document-oriented database. Records are stored as documents with properties that hold the data in the document. The contents of a document are defined by a schema. The great thing about MongoDB is that the design of a document can be changed very easily. If you discover that you need to add another item to a document (for example, we might suddenly decide we want to store the creation date of a survey) you can do this by changing the schema for the document and everything will keep working.

Start with MongoDB

If you install the MongoDB database server on your machine it will run as a service when your machine starts running. If you want to put your database in the cloud there are paid hosting plans

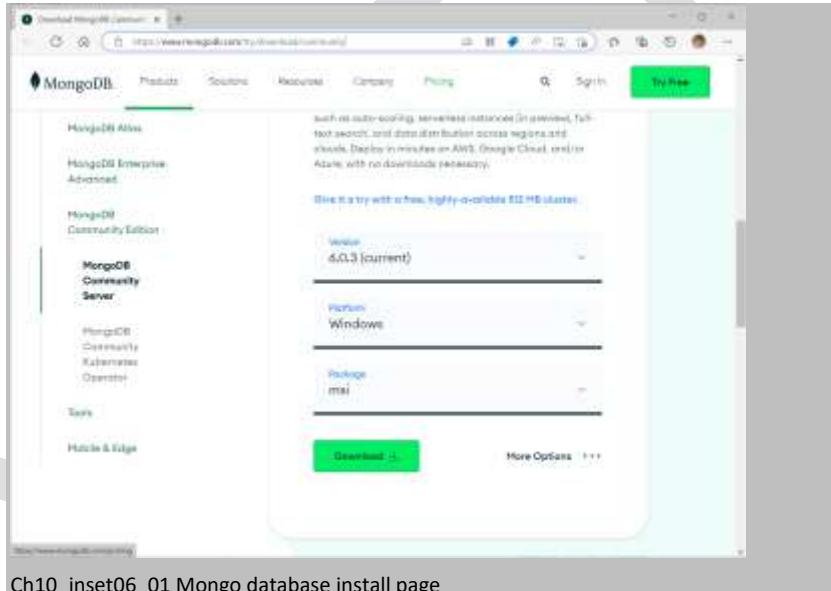
available as well as a free service you can use for small projects.

Make Something Happen

Install MongoDB and MongoDB Compass

Installing MongoDB will allow you to develop programs on your computer which use database storage. When the database is not being used it only consumes a very small amount of memory and processor power. First you need to open your browser and visit the web page:

<https://www.mongodb.com/try/download/community>



Ch10_inset06_01 Mongo database install page

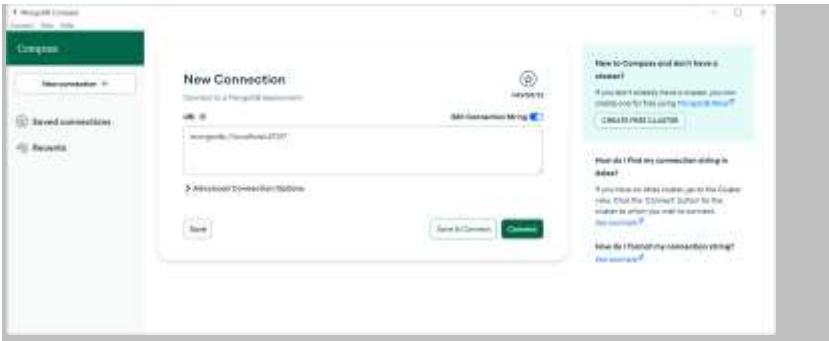
Select the latest version, choose your platform, click the Download button to get the installer and then run it on your machine. Follow the instructions to install MongoDB. Make sure you select the option to install the MongoDB Compass application, we will be using this to manage our databases.

Once it has been installed the MongoDB server will run in the background and wait for applications to connect to it. The server will store the database information in files on the local machine. We can interact with the server using the MongoDB Compass application which is installed alongside MongoDB. The application uses a network connection to interact with the database server. You can use it to view and edit the contents of database records and to create new databases. We are going to use it to create a database for the Tiny Survey application.

Make Something Happen

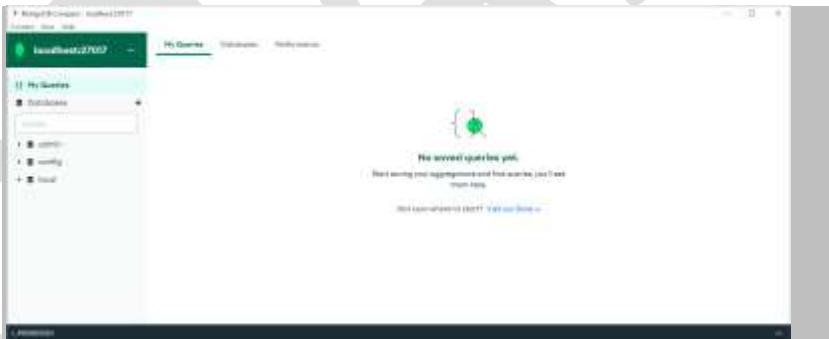
Create a database

The first thing you need to do is start the MongoDB Compass application. This should be in your program files on your machine. It has the name **MongoDBCompass**.



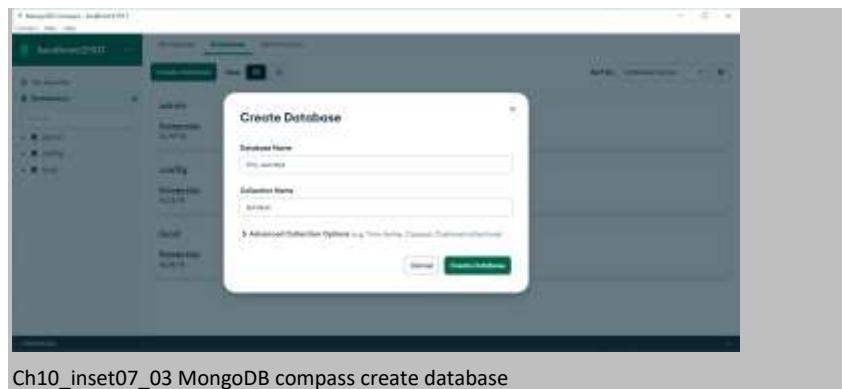
Ch10_inset07_01 Starting MongoDB compass

When Compass starts it displays a dialog that lets you select the database connection you want to use. You can use Compass to manage database servers on distant machines, but we are going to use it to manage the databases on our machine. It should select a connection on localhost as shown above, so just click the green **Connect** button to connect to the server on your machine.



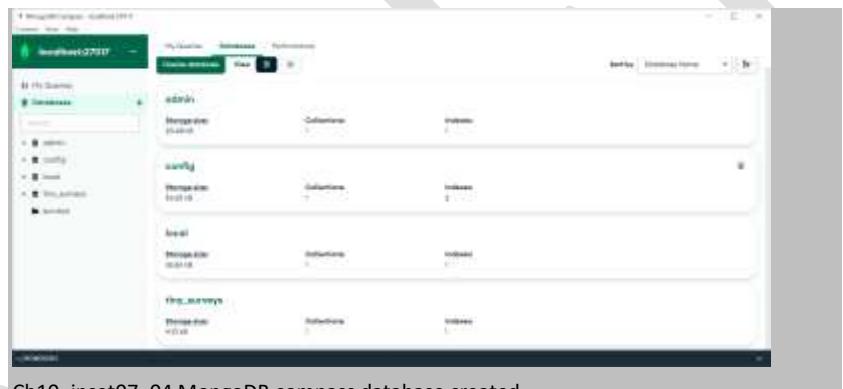
Ch10_inset07_02 MongoDB compass home page

Now we need to create a database for the Tiny Surveys application. Select the Database tab in the right-hand window and click the green “Create database” button. Create a database called **tiny_surveys** containing a collection called **surveys** as shown below.



Ch10_inset07_03 MongoDB compass create database

The [tiny_surveys](#) database and the [surveys](#) collection will appear in the list of databases on the left-hand side of the window.



Ch10_inset07_04 MongoDB compass database created

The [tiny_surveys](#) database provides survey storage for the Tiny Survey program. We are going to use an object modelling framework called Mongoose to access the database from our JavaScript code.

Mongoose and schemas

We add Mongoose to an application using [npm](#) as shown below.

```
npm install mongoose
```

We can then import mongoose into our code and use it to interact with a MogoDB database. The first thing we need to do is tell Mongoose what our data looks like. Mongo is a *document driven*

database. A particular record in the database is a document. Multiple documents are stored in *collections*. The `tiny_surveys` database contains a single collection which contains `survey` documents. A document is described to Mongoose using a JavaScript object called a *schema*. We've already seen how a class definition tells JavaScript how to make an instance of a class. A schema object tells Mongoose how to make a MongoDB document.

Below you can see the schema file for the Tiny Surveys application. It defines two schemas. The first is for `option` and the second is for `survey`. A `mongoose.Schema` object is initialized with an object literal that contains the property definitions. If you look at the code below you will see that the `optionSchema` object contains a `text` and a `count` property and that the `text` property is a string and the `count` property is a number. The `surveySchema` defines an object which contains a `topic` property and an array of `options`. The code below creates a Mongoose model from this schema and exports it.

```
import mongoose from "mongoose",302

var optionSchema = mongoose.Schema({303

  text: {304
    type: String,305
    required: true306
  },
  count: {307
    type: Number,308
    required: true309
  }
});
```

³⁰² Import the Mongoose library

³⁰³ Create an Option schema

³⁰⁴ Text property

³⁰⁵ Type of the property

³⁰⁶ Text is required

³⁰⁷ Count property

³⁰⁸ Type of count property

³⁰⁹ Count is required

```

var surveySchema = mongoose.Schema({310
  topic: {
    type: String,311
    required: true312
  },
  options: {313
    type: [optionSchema],314
    required: true315
  }
});317

let Surveys = mongoose.model('surveys', surveySchema);316
export { Surveys as Surveys };317

```

If we had additional documents that we wanted to store in the application we would create a further schema classes and use them to create models to export.

The SurveyManagerDB class

The file store version of Tiny Survey uses a [SurveyManagerFiles](#) class which stores surveys in a file. We are going to create a new version of this class called [SurveyManagerDB](#) which will use a database to store surveys. We will then change the import statement in the [tinsurveys.mjs](#) application to use this storage manager rather than the one created to use files. So, let's look at how we modify the storage manager to use a database rather than a file.

The code below shows the first part of the [SurveyManagerDB](#) class. It imports the mongoose library and the survey database model that was exported by the above code. It is traditional to put schema files in their own folder (called models) in an application project. They are then imported into the application to manage the data storage for the application. The constructor for

³¹⁰ Create a Surveys schema

³¹¹ Survey topic

³¹² Topic is required

³¹³ Survey options

³¹⁴ A list of options

³¹⁵ Options are required

³¹⁶ Make the model

³¹⁷ Export it

`SurveyManagerDB` does nothing. The `init` method in the class connects to the database. We could use the address of a distant server, but in this case we are using the MongoDB server running on the local machine. The `connect` method is asynchronous, so the `init` method uses `await` when calling it.

```
import mongoose from 'mongoose';318
import { Surveys } from './models/survey.mjs'319

class SurveyManagerDB {

    constructor() {
    }

    async init() {
        await mongoose.connect('mongodb://localhost/tiny_surveys');320
    }

    // rest of SurveyManager
}
```

We will have to create database versions of each of the survey management methods. Each of the methods will now interact with the database rather than with the survey storage classes. Below is `storeSurvey` function. This is provided with an object which contains the topic and options for a survey which is to be stored. The first time a survey is stored the `storeSurvey` method must make a new document and save it. If a survey is already in the database, it must be updated with the contents of new value.

```
async storeSurvey(newValue) {321
    let survey = await Surveys.findOne({ topic: newValue.topic });322
    if (survey != null) {323
```

³¹⁸ Import the mongoose library

³¹⁹ Import the Surveys schema

³²⁰ Connect to the database

³²¹ Survey storage method

³²² Search for the survey by topic

³²³ Does the survey exist?

```

        await survey.updateOne(newValue);324
    }
    else {
        let newSurvey = new Surveys(newValue);325
        await newSurvey.save();326
    }
}

```

The `findOne` method is used to find a single document in the collection which matches the given criteria; in this case the document must have a topic which matches the one in the new value. If `findOne` returns `null` it means that the document was not found. A survey loaded from a collection has all the properties defined in the schema and these can be used directly. Below you can see the database version of the `getCounts` method which finds a survey and then creates the object to be displayed in the from the topic and the options in the `displayResults` page.

```

async getCounts(topic) {
    let result;327
    let survey = await Surveys.findOne({ topic: topic });328
    if (survey != null) {329
        let options = [];330
        survey.options.forEach(option => {
            let countInfo = { text: option.text, count: option.count };331
            options.push(countInfo);332
        });
    }
}

```

³²⁴ If it exists, update it using the new values

³²⁵ If it doesn't exist, make a new Survey document

³²⁶ Save it in the database

³²⁷ Create an empty result object

³²⁸ Find the survey to be displayed

³²⁹ Check if the survey was found

³³⁰ Make an array of option results

³³¹ Loop through the options

³³² Make an option result

³³³ Add it to the list of options

```
        result = { topic: survey.topic, options: options };334  
    }  
    else {335  
        result = null;336  
    }  
    return result;337  
}
```

Make Something Happen

Use a database

The starting point for this Make Something Happen is a version of Tiny Survey you can download from GitHub here: <https://github.com/Building-Apps-and-Games-in-the-Cloud/TinySurveyDatabase>. Clone this repository onto your machine and open it with Visual Studio Code. For this application to work you must have installed MongoDB and created the database as shown above.

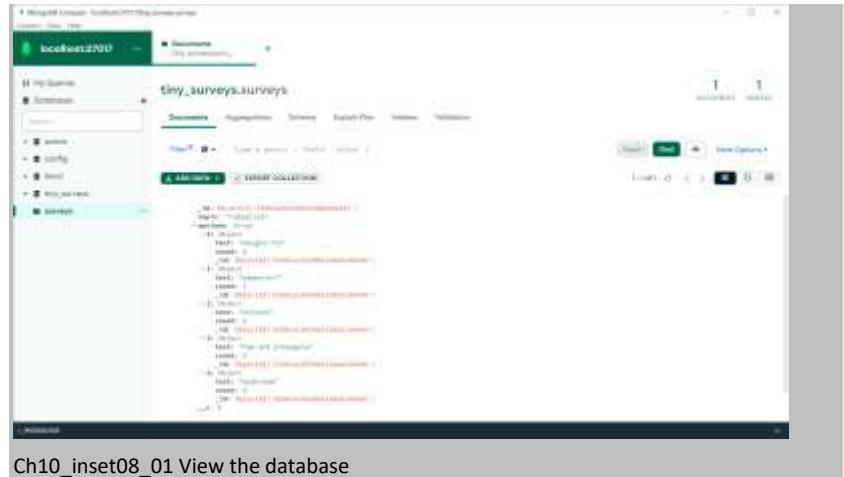
Open the application in Visual Studio Code and open the `tinysurvey.mjs` source file and start the program running. Now use the browser to navigate to `localhost:8080/index.html` and enter a complete survey. You should find that the application works in the same way as before. Now start **MongoDBCompass**, connect to the server and view the **surveys** collection. You should discover that there is a document there containing a topic and options that you entered. If you are already running **MongoDBCompass** you can use **View>Reload Data** to refresh the display.

³³⁴ Build the result object

³³⁵ If the survey was not found

³³⁶ Set result to null

³³⁷ Return the result



Ch10_inset08_01 View the database

You can use the **MongoDBCompass** application to change the contents of documents and delete documents and even entire databases.

Testing asynchronous code

When we create asynchronous code we have to slightly change the way that our tests work. The tests themselves will call asynchronous functions and the testing framework will call the tests asynchronously. Below is the code that calls one of the tests for the database version of Tiny Surveys.

```
test('Store and retrieve options', async () => {
  const result = await optionsTest();338
  expect(result).toBe("");339
});
```

Make Something Happen

³³⁸ Await the result of the test

³³⁹ Check the test result

Test the database

If you have left the application open from the previous exercise you can just continue with this one. Otherwise, follow the instructions in Make Something Happen Use a database to set things up. There are some tests in the application already, all we have to do is run them. Open up the Terminal in Visual Studio Code and give the command and press enter:

```
npm test
```

The tests will run and pass as shown below.

A screenshot of the Visual Studio Code interface. The left sidebar shows project files including 'surveymanager', 'surveymanager.settings', 'THREE.js', 'node_modules', 'public', 'views', 'deployment.json', 'environments.js', 'errors', 'indexes', 'internationals', 'gitignore', 'jest.config.js', 'LICENSE', 'package-lock.json', 'participate.js', and 'surveymanager.test.js'. The right pane shows the 'surveymanager.test.js' file with code for testing a survey manager. Below the editor is the terminal window, which displays the command 'npm test' and its output: '2 passed, 2 total'. At the bottom, the status bar shows the path 'C:\Users\seanlsource/repos\GitHub\TinySurveyDatabase>'.

```
Ch10_inset08_01 Test the database
```

Each test runs in isolation from the next and connects and disconnects from the database as it does this. The `surveymanager` class has an additional method (shown below) that is called to disconnect the test from the database when the test has finished.

```
async disconnect(){
    await mongoose.disconnect();340
}
```

³⁴⁰ Disconnect the database connection

Tests and data storage

The previous versions of Tiny Survey stored their data in memory, so nothing was persisted when they stopped running. However, the tests above interact with the live database, which is not really a good idea. Running the tests above would overwrite the contents of a “Robspizza” survey as one is created during the tests. In the next section we will look at how we can manage the use of different resources for test and deployment versions of applications.

Refactoring Tiny Survey

In Chapter 9 we identified some things that mark out an application as “professional”. Now we are going to discover another. Professional applications are well structured. The Tiny Survey application has evolved from the original workflow, and it could really do to be better structured. We are just about to make some significant additions to the code to allow users to log in to the service, and this would be a good point to tidy up the structure. This won’t affect the way the code works, or at least it shouldn’t.

Create route files

The first thing we are going to do is separate all the routes out into files. This will make the main program smaller and make it possible for work to be shared around a group. Below is the route file for the index page of the application. It creates an `Expres.Router` instance which implements the `get` handler for the index page. This instance is then exported from the files as `index`.

```
import express from 'express';
import { checkSurveys } from '../helpers/checkstorage.mjs';341
const router = express.Router();342

// Home page
router.get('/', checkSurveys, (request, response) => {343
  response.render('index.ejs');344
});
```

³⁴¹ Import middleware

³⁴² Create an empty router

³⁴³ Set a get behavior for the router

³⁴⁴ Render the index.ejs file for this route

```
});  
export { router as index };345
```

Previously the `tinysurvey.mjs` file contained code that assigned the handler for the route to the Express application. In the refactored version the route handler above is imported and then added the route. The code below shows how this is done.

```
import {index} from './routes/index.mjs';346  
app.use('/index.html', index);347
```

You can regard each route as piece of middleware that is added to a framework built by Express to underpin the application. Note that all the route files are held in a `route` folder.

Make Something Happen

Look at the refactored version

The starting point for this Make Something Happen is a version of Tiny Survey you can download from GitHub here: <https://github.com/Building-Apps-and-Games-in-the-Cloud/TinySurveyRefactored>. Clone this repository onto your machine and open it with Visual Studio Code. Select the Explorer view and expand all the folders in the solution. Then open the `TinySurvey.mjs` file and look at the way that the routes are now imported and used. If you run the application, you will discover that it works in exactly the same way. However, it is much easier to find your way to different parts of the application.

³⁴⁵ Export the router with the name `index`

³⁴⁶ Import the route code

³⁴⁷ Connect the route to the index path

```

    // App.js
    1  import SurveyManager from './SurveyManager';
    2  import SurveyResults from './SurveyResults';
    3  import configReader from './config-reader';
    4
    5  import {Indexer} from './indexer/indexer';
    6  import {GetSurvey} from './get-survey/get-survey';
    7  import {InsertSurvey} from './insert-survey/insert-survey';
    8  import {UpdateSurvey} from './update-survey/update-survey';
    9  import {DeleteSurvey} from './delete-survey/delete-survey';
    10
    11  // Create the express application
    12  const app = express();
    13
    14  // Select the middleware to receive incoming posts
    15  app.use(express.urlencoded({ extended: false }));
    16
    17  // Set the cors/cross origin middleware
    18  app.use(cors());
    19
    20  // Add the middleware
    21  app.use(indexer);
    22
    23  // Compute the route handled by the reader
    24  app.use('/tiny-survey', index);
    25  app.use('/tiny-survey/:id', reader);
    26  app.use('/tiny-survey/:id/:action', getSurvey);
    27  app.use('/tiny-survey/:id/:action', updateSurvey);
    28  app.use('/tiny-survey/:id/:action', deleteSurvey);

```

Ch10_inset10_01 View the refactored code

Tiny Survey deployment

In chapter 6, in the section “Deploy an application” we put the Cheese Finder game into the cloud using as an Azure hosted App Service. Now we are going to deploy Tiny Survey. However, before we do that, we might want to step back a bit and consider how the development process works. You might think that we just create a program and then put it in the cloud. After all, that’s what we did with the Cheese Finder application. However, the Tiny Survey application is different. It uses a database to store survey information. It also has a much higher number of user interactions than Cheese Finder and we have even built some tests for it (although presently these tests use the application database, which is not a good idea). It would be useful to have a “developer mode” for when the application is being built and a “production mode” which is used when the application is being deployed to customers. Developer mode could use a test database hosted on our local computer and production mode could use a database hosted in the cloud. It turns out that the designers of node and npm have thought of this and the [package.json](#) file can be used to configure these two different modes. To understand how this can be made to work we first have to discover how environment variables are used to configure applications.

Manage environment variables

An environment variable is a value that sends information from the environment into the application. When we configure Azure to host our application, we will create environment variables to pass information into the application. In Chapter 6, in the section “Set the port for the server” we used an environment variable to set the HTPP port for our Cheese Finder server. The code below is in the CheeseFinder application. It tests to see if the environment variable `PORT` exists. If it does the value of `port` is set to the value in the environment variable, otherwise `port` is set to

8080. This allowed the service hosting CheeseFinder to send a port value into the application by creating an environment variable called `PORT` which is set to the required number.

```
const port = process.env.PORT || 8080;348  
server.listen(port);349
```

There is another piece of information we would like to feed into the Tiny Survey application. This is connection string used for the database connection. At the moment the database connection string is written directly into the code in the `surveymanagerdb.mjs` file, as shown below:

```
await mongoose.connect('mongodb://localhost/tiny_surveys');
```

This is the statement that connects the database to the application. The argument to the call of `connect` is a *connection string*. It is a string that defines a connection to a MongoDB database server. The present version contains a localhost address because the database is hosted on the local machine. When we deploy this application into the cloud, we will use the address of a database hosted on a distant server. The connection string will contain a password that will authenticate the database connection. This kind of connection string really shouldn't be in the source code of our application because we might decide to make our code open source and publish it for other people to use. If we publish our code, we don't want to also publish the address and password for our database server.

We can solve this problem by creating an environment file which contains values which can then be picked up by the application. The environment file has the name `.env`. It contains a set of name-value pairs. Below is an environment file for the Tiny Survey application. It creates an environment variable called `DATABASE_URL` which is set to the string `mongodb://localhost/tiny_surveys`.

```
DATABASE_URL=mongodb://localhost/tiny_surveys
```

³⁴⁸ Set to 8080 or the PORT value

³⁴⁹ Start listening to the port

We can install a library called `dotenv` to use this file during application development. We do this using the `npm install` command as shown below.

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

Note that the install command has the option `--save-dev` on the end. This means that the package will be specified in the “`devDependencies`” part of the `package.json` file. We don’t need `dotenv` in the production code because we will configure environment variable values when we install the application on the host.

Once we have the `dotenv` library installed we can use it in `surveymanagerdb.mjs` by importing it and then initializing it as shown in the code below, which is at the top of the `surveymanagerdb.mjs` file. This code imports the `dotenv` element from the library and then calls `config` on it to set it up.

```
import * as dotenv from 'dotenv';350
dotenv.config();351
```

Once this has been set up the program can now use values declared in the environment file. This allows us to change the database connection statement to the one below:

```
await mongoose.connect(process.env.DATABASE_URL);
```

When the program runs the value of `DATABASE_URL` is loaded from the environment file and used at that point in the code.

CODE ANALYSIS

Environment variables

³⁵⁰ Get `dotenv` from the library

³⁵¹ Start `dotenv` running

You may have some questions about environment variables.

Question: Why are we using environment variables again?

Environment variables let us send values into our application. We do this rather than modify the code when we want these values to change. When our application is running in the cloud host, for example Azure, it will use environment variables to get setting values from the host. The .env file is a way of simulating these variables when we are developing the application.

Question: What stops the environment files being saved in the repository whenever we check the files in with Git?

We have seen that a repository can contain a [.gitignore](#) file which contains a set of patterns matching files that are not to be stored in the repository. We first use this in Chapter 8 in the section “Use gitignore” to stop Git from saving library files in the repository. The file also contains a pattern that matches the [.env](#) file, preventing it from being stored in the repository.

Question: What happens when we load our application onto a server in the cloud?

Good question. Later in the chapter we will see how to manage environmental variables for cloud-based applications.

Question: How do we tell people what our environmental variables do?

It might be a good idea to add some documentation to the project in the form of a README.md file. See below for how to do this.

Use the nodemon package

One of my first principles when I start a new project is to build a nice place to work. If you are going to do something many times, it makes sense to make this as easy as possible. The [nodemon](#) package can be downloaded using [npm](#) and it makes it much easier to run and debug your application. The command below installs [nodemon](#) into an application as a developer dependency.

```
npm install nodemon --save-dev
```

Once [nodemon](#) has been installed we can modify the [package.json](#) for the application to provide a way of starting it running. The [package.json](#) file contains a [scripts](#) section where we can put commands that can be run from the command line. The JSON below shows how we can add script entries that we can use from the command line. There are three script entries. The first, “test”, uses the [jest](#) package to run the tests on the application. The second, “devstart” starts [nodemon](#)

running and the third starts tinysurvey running normally.

```
"scripts": {  
  "test": "node --experimental-vm-modules node_modules/jest/bin/jest.js",  
  "devstart": "nodemon tinysurvey.mjs",  
  "start": "node tinysurvey.mjs"  
},
```

Once these entries have been added to [package.json](#) we can use commands from the terminal to trigger any of them. The command below would run the `devstart` script which would start `nodemon` running.

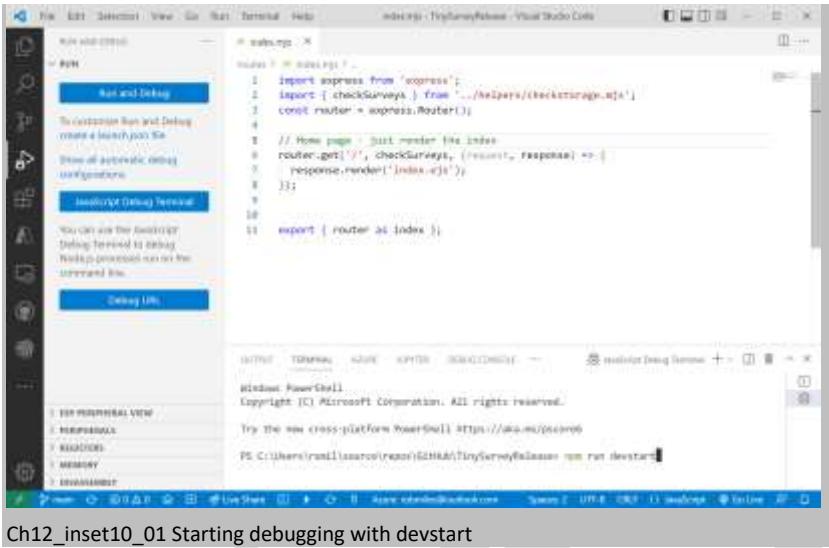
```
npm run devstart
```

The `nodemon` package starts the application running and then monitors all the source files in the application. If any of the files change `nodemon` will restart the application. This can be very useful when you are writing code because you can be sure that the program is always running with the latest version of the source code.

Make Something Happen

Use nodemon

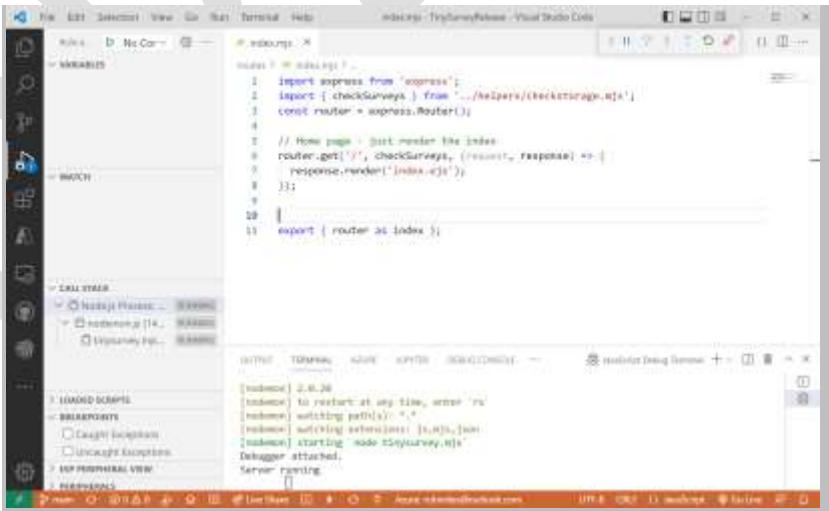
The starting point for this Make Something Happen is a version of Tiny Survey you can download from GitHub here: <https://github.com/Building-Apps-and-Games-in-the-Cloud/TinySurveyRelease>. Clone this repository onto your machine and open it with Visual Studio Code. Open the file `index.mjs` in the `routes` folder. Now select the **Run and Debug** option in the left-hand menu and click the **JavaScript Debug Terminal** button.



Ch12_inset10_01 Starting debugging with devstart

This will open a debugging terminal which attaches a debugger to any programs that are executed in it. It means that you will be able to put breakpoints in programs that you run in this terminal. Start the program running using the npm run command. Use it to run the devstart script as shown below. Press enter to run the command.

```
npm run devstart
```



Ch12_inset10_02 Running the application

The `nodemon` application is now watching all the files in the application. Make a change to the `index.mjs` file (add a comment) and then save it. You will see that the application is

automatically restarted.

Create a README.md file

If you want to tell people all about your project how your project works you can create a README file. README files are an important part of GitHub. A repository (and even each folder in the repository) can contain a README file which can look like a mini-website describing the project. README files have the language extension `.md` to indicate that they are formatted using **Markdown** syntax. This is a great way to quickly format text. Markdown files can also contain images and links to other pages. Below is a tiny sample of Markup that will tell you just about all you need to know about creating documents. For this to work the folder containing the README file must contain a folder called images which holds the file small-rob.jpg.

```
# This is a big heading
## This is slightly smaller

This is body text.

To make another paragraph we need a line break.

Make a numbered list by starting each line with a number followed by a period (full stop):

1. It's that
1. easy.

Make a bulleted list using stars:

* this is
* easy too

Put code into blocks by using ``
``Javascript
function doAddition(p1, p2) {
    let result = p1 + p2;
    alert("Result:" + result);
}
``

Links are easy too https://www.robmiles.com or with names [my blog](http://www.robmiles.com)
```

Finally, you can add pictures:
![picture of rob](images/small-rob.jpg)

Below you can see the Markdown as it appears in the browser. You can find a guide to the version of Markdown supported by GitHub here: <https://enterprise.github.com/downloads/en/markdown-cheatsheet.pdf> It is worth spending a bit of time learning some Markdown. You can add a Markdown preview plugin to Visual Studio Code which makes it easy to preview your pages as you create them.



Figure 10.43 Ch-10_Fig_02 Markdown output

Make Something Happen

Deploy Tiny Survey on Azure

The Tiny Survey application needs a MongoDB database to host the survey information. You can get hosting for MongoDB databases at mongodb.com. You will have to create an account, but you do not have to give any payment information. Follow the process at <https://www.mongodb.com/basics/create-database>. Create the database in the shared tier, this provides a small database you can use for testing. Then follow the process to create a username and password and use this to create the connection string for your database.

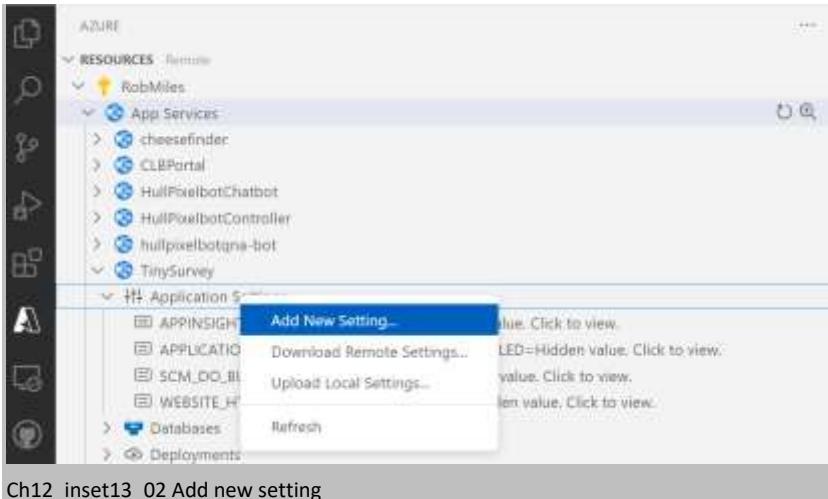


Ch12_inset13_01 Getting the connection string

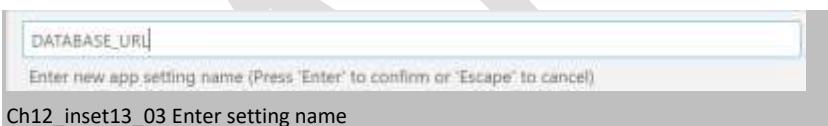
The screenshot above shows the connect instructions generated by the MongoDB Atlas web page. The variable `uri` holds the connection string. Remember to change the `<password>` placeholder to the password you have created.

Now that you have a working database you can install Tiny Survey in the cloud and connect it to this database. Start by opening the **TinySurveyRelease** application in Visual Studio and following the process described in the “Make Something Happen Create an Azure App service” exercise in chapter 6. Note that you won’t be able to call the service **TinySurvey** because I’ve already got that name.

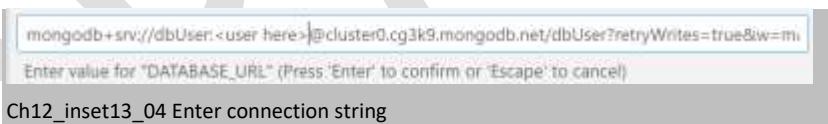
Don’t deploy the application at the end of the process because we need to set up the connection string environment variable before we deploy it. Find the App Services collection in Azure and select the **Tiny Survey** item in the list (you might not have as many services as I do). Right click **Application Settings** and select **Add New Setting** from the list which appears.



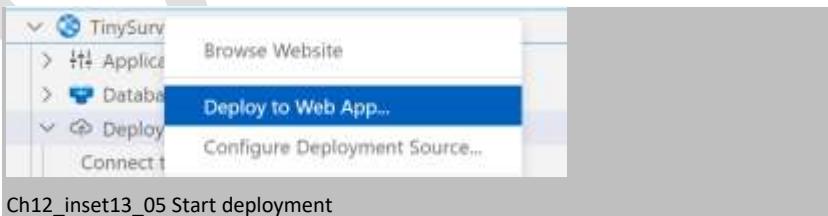
This will open a dialog to enter the name of the setting value, in this case DATABASE_URL



Type in the name and press Enter and you will be asked to enter the value for the setting.
Enter the connection string you got from the MongoDB setup.



Press Enter to save the string in the application settings. Now we can deploy the application.
Right click on TinySurvey and select "Deploy to Web App.."



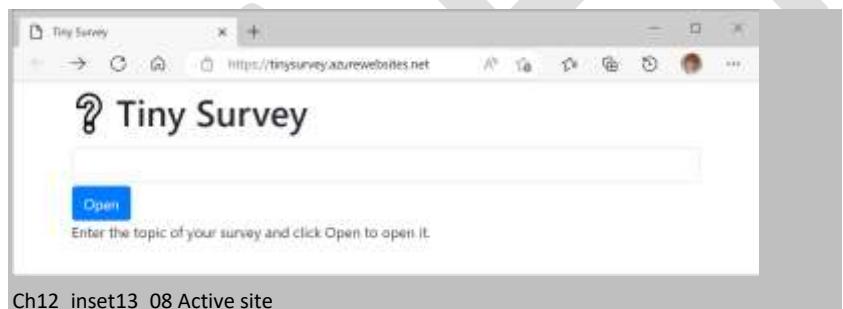
Now you will be asked to select the folder containing the application to be deployed. Select the repository folder.



A dialog box will pop up asking you to confirm the deployment. Click Deploy. The deployment will take a little while. At the end of it a dialog will appear at the bottom right of the Visual Studio Code window confirming the completion of the deployment.



Click **Browse Website** to view the application.



The browser will open your web application and start it running. Note that the url in the above page is the tinsurvey.azurewebsites.net which is the internet address for the application I uploaded. Yours will be different.

What you have learned

This has been another fun chapter. We started by considering how to read and write files. Then we found out how to manage asynchronous processes in JavaScript and made some error handling middleware. We've built our first MongoDB database and connected a JavaScript application to it using Mongoose. Then we've tidied up our code, built a proper development environment and finally configured and deployed the program. Here's a recap plus some points to ponder.

- The `fs` library in node.js provides file input/output to applications. The library functions all work asynchronously, returning a promise object describing the requested action. The code that receives the promise can then

continue without being blocked by the call to a function which may take some time to complete.

- A promise object exposes a `then` behavior which can be given a function to be called when the promise is resolved.
- It is frequently the case that an application wants to wait until a “promise returning” activity completes. In this situation the `await` keyword can be used to await the completion of a function returning a promise. A function or method that contains an `await` action must be marked as `async` by preceding the declaration with the keyword `async`. This is because the `await` action generates a promise which is returned to the caller of the code containing the `await`.
- It might appear that asynchronous operation allows a JavaScript application to run multiple threads of execution at the same time. However, this is not the case. JavaScript maintains call stack and an event list and uses these to switch rapidly between operations. Events are used to trigger new actions. For example, rather than waiting for a network transaction to complete the underlying JavaScript code nominates an event to fire when the network transaction has completed.
- Applications can use global flag variables to indicate the status of application and change how the application responds to inputs.
- We can write our own middleware functions and add them to the chain of functions that process an incoming web request and generate the response. The middleware function can cut short the handling of a web request handler if required (for example if a service is not available).
- MongoDB is a document-oriented database which holds data as set of collections of documents. Not all the documents in a particular collection must have the same properties. This makes MongoDB more flexible than standard table-based databases. We can install a MongoDB server on our computer and use it to manage data storage. We can also connect an application to a database on a distant server. We can get limited MongoDB hosting from Mongo at <https://www.mongodb.com/home>
- The Mongoose library can be added to an application using npm. It provides a library of functions that implement an object-oriented interface to MongoDB. An application can create *schema* objects which contain properties that describe the document contents. An application can create new documents and search for and update existing ones.
- Refactoring is the process of ensuring that the structure of the code in an application is a good match to the problem being solved. It is a constant

process because the understanding of a problem improves during the development and initial design decisions may need to be revisited. Refactoring also helps to make code easier to work on and impose a navigable structure to the elements of a solution.

- The structure of an Express application can be improved by separating the routes for various endpoints into different code files. The routes become middleware items which are added to the Express framework underpinning the application.
- Environment variables are a means by which setting information is communicated from the operating environment into a running application. They make it easier to manage the application (there is no need to change the content of the code to use different resources) and they also make the application more secure (the source code doesn't contain any resource information).
- The `devDependencies` part of the `package.json` file for an application contains packages to be used when developing the program but not to be deployed with it.
- The `package.json` file can contain a `scripts` part which contains scripts to be run when developing and testing the application.
- When developing an application, you can use the `dotenv` package and a `.env` file to create environment variables to configure an application for testing. When an application is deployed into the cloud the process of deploying the application into the host must include setting values for the environment variables the application uses.
- When developing an application, you can use the `nodemon` package to automatically run the application each time a file in it is changed.
- An application can contain a `README.md` file which contains instructions that describe how to run and deploy the application. This file is formatted using Markdown commands which provide simple but powerful formatting commands.
- When deploying a database application as an Azure App Service we need to create an Application Setting value containing the database connection string to be used when the application runs.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions".

What happens if a function contains an `await` but is not marked as `async`?

The JavaScript compiler will refuse to run such a function.

What happens if my code awaits a function that is not marked as async?

The JavaScript engine will just call the function in the usual way.

What happens if a function marked as async doesn't contain any awaits?

The function will run without problem, but the function will not be awaited.

What happens if code inside an asynchronous function throws an exception?

The exception must be caught by an exception handler that is wrapped around the asynchronous call. Then the exception handler for the call will then run at the time the exception is thrown. An exception construction which encloses a number of asynchronous calls will not catch any exceptions thrown by them.

What is special about a function that is to be used as middleware when processing a web transaction in Express?

The middleware function must have the correct parameters. These are request (a reference to an object describing the incoming request), response (a reference to an object which will describe the response) and next (a reference to the next middleware function in the chain of functions handling this request).

How does a middleware function cause the processing of a request to be abandoned?

If a middleware function detects that there is no point in continuing processing the request it doesn't call the next middleware function in the chain.

Why is a document-oriented database a good idea?

A document-oriented database doesn't insist on all the objects in a collection containing exactly the same properties. It is also possible to add new properties to documents by adding them to the schema object that describes the document contents.

What is the difference between MongoDB and Mongoose?

MongoDB is a document-database technology which can be used directly. Mongoose is a framework which allows database elements to be represented by JavaScript objects.

What would happen if two applications attempt to connect to the same database?

One of the great things about databases is that they were built to handle this. The database would keep working. However, if both applications start to "fight" over a particular

document in the database this might cause problems. The database will never fail in this situation, but the applications may become confused unless they have been designed to handle this.

What happens if I forget to set an environment variable when deploying an application?

The value will be undefined and the application will fail when it runs.

How many free applications and databases can I have?

Both Azure and MongoDB restrict you to a single free application or database. You can use a single database with multiple applications by having each application use a different document collection.