Developing Web Applications

Laboratory Sessions for COSC212
Department of Computer Science
University of Otago

Semester 2, 2021

Contents

I Introduction **5**

A COSC212 Course Overview **7**
A.1 Lectures, Labs, and Assessment . . . 7
A.2 Appropriate Behaviour . . . 9
A.3 Workload . . . 9
A.4 Reading Material . . . 9
A.5 Course Overview . . . 10

B Getting Help in the Labs **13**
B.1 Helping Yourself . . . 13
B.2 Incremental Development . . . 14
B.3 Getting Help . . . 14

C Assignment 1 **15**
C.1 Requirements (Must-Haves) . . . 18
C.2 Deliverables . . . 19
C.3 Marking Scheme . . . 20

D Assignment 2 **21**
D.1 Requirements (Must-Haves) . . . 21
D.2 Deliverables . . . 22
D.3 Marking Scheme . . . 22

II Getting Started **25**

1 HTML and CSS Revision **27**
1.1 Before We Begin . . . 27
1.2 HTML and CSS . . . 28

2 Git **33**
2.1 Git concepts . . . 33
2.2 Using Git in PhpStorm . . . 35
2.3 Working with the Repository . . . 37
2.4 Pushing To GitLab . . . 38
2.5 Start Tracking Classic Cinema with Git . . . 41
2.6 Enable Automated Publish-on-Push (Pipeline) . . . 41
2.7 Working From Other Locations . . . 42

III Client-Side Scripting **43**

3 Hello, JavaScript **45**
3.1 Hello World in JavaScript . . . 45
3.2 Unobtrusive JavaScript . . . 46
3.3 More Complex Behaviour . . . 46
3.4 Debugging JavaScript . . . 48
3.5 Showing and Hiding Elements . . . 49
3.6 JSLint and JSHint . . . 52
3.7 Strict Mode . . . 53

4 JavaScript Arrays and Objects **55**
4.1 JavaScript Arrays . . . 55
4.2 JavaScript Objects . . . 57

5 JavaScript Closures **61**
5.1 Loading Multiple Scripts . . . 61
5.2 JavaScript Scope and Closure . . . 62

6 JavaScript Cookies **65**
6.1 JavaScript and Cookies . . . 65
6.2 A Classic Cinema Shopping Cart . . . 67

7 Web Storage and IndexedDB **69**
7.1 Web Storage . . . 69
7.2 IndexedDB . . . 70

8 JavaScript Form Processing **79**
8.1 Error Reporting . . . 80
8.2 Form Checking and Regular Expressions . . . 81
8.3 Successful Form Processing . . . 83

9 Maps **85**
9.1 Displaying a Map with Leaflet and OpenStreetMap . . . 85
9.2 Adding Markers to the Map . . . 86
9.3 Adding Detail to the Markers . . . 87
9.4 Working with Overlays . . . 88
9.5 Centering the Map . . . 89

10 Introduction to jQuery **91**
10.1 Getting Started with jQuery . . . 91
10.2 jQuery Shopping Cart . . . 93

11 jQuery, Ajax, and XML **95**
11.1 Ajax . . . 95
11.2 Classic Cinema Reviews . . . 98

12 jQuery Animation **101**
12.1 Back to the Beginning . . . 101
12.2 More Animations with jQuery . . . 101
12.3 Animating the Classic Cinema . . . 102

13 Sapphire, Permissions and (S)FTP **103**
13.1 Connecting to `sapphire` . . . 103
13.2 The Unix File System . . . 105
13.3 Navigating the Unix Filesystem . . . 106
13.4 Putting a Web Page on the Development Server . . . 107
13.5 Files and Permissions . . . 109
13.6 Unix Scripting . . . 111
13.7 Accessing `sapphire` from Off-Campus . . . 113
13.8 Further Information . . . 115

14 Catch Up **117**

IV Server-Side Scripting **119**

15 Hello, PHP **121**
15.1 Hello World in PHP . . . 121
15.2 PHP Includes . . . 123

16 PHP Form Processing **127**
16.1 PHP Form Processing . . . 127
16.2 Validating Form Input . . . 129
16.3 Classic Cinema Checkout Validation . . . 129
16.4 Scripts and Security . . . 131

17 PHP Cookies and Sessions **133**
17.1 PHP Cookies . . . 133
17.2 PHP Sessions . . . 136

18 PHP and XML **139**
18.1 Storing Classic Cinema Orders . . . 139
18.2 Displaying the Orders . . . 141
18.3 Deleting XML nodes . . . 142

19 PHP and JSON **143**
19.1 Storing JSON . . . 143
19.2 Using the JSON data to build an HTML page . . . 145

20 Getting Started with MySQL **147**
20.1 Connecting to MySQL . . . 147
20.2 Create a Database . . . 148
20.3 Create a Database User . . . 148
20.4 Creating a Users Table . . . 149
20.5 Validating Users and Passwords . . . 151
20.6 Adding Information to the Table . . . 151

21 MySQL and PHP **153**
21.1 Connecting to MySQL from PHP . . . 153
21.2 Adding a New Entry into your Users Table . . . 154
21.3 SQL Injection Attacks . . . 155

22 Authentication and Sessions **157**
22.1 Logging In . . . 157
22.2 Restricting Access . . . 159
22.3 Logging Out . . . 159

23 More Authentication **161**
23.1 Restricting Access to Orders . . . 161
23.2 Even Finer Order Control . . . 162
23.3 Adding Reviews . . . 163

24 Catch-Up/Assignment **165**

25 Catch-Up/Assignment **167**

Part I

Introduction

Chapter A

COSC212 Course Overview

This course is about building Web Applications. Sometimes these are referred to as “Software as a Service” or “Cloud Computing” applications, but it makes little difference from a developer’s point of view. The key point is that a computer program is provided on the Web; users need not install any software beyond a Web browser to access it, and vendors can roll-out new versions instantly.

Some examples of Web Applications are TradeMe, Facebook, Xero, GMail, Wikipedia, Rotten Tomatoes, Amazon, and the list goes on. All of these use Web pages to convey their content, but they cannot just be called Web "pages" – pages are there to be *read*, but applications are there to be *used*. Web applications can collect data, search through content, and allow access to services such as banking, ordering, booking, and updating content. This course is going to explore in some detail how this is done.

A.1 Lectures, Labs, and Assessment

Lectures

There are two lectures per week, at 3 p.m. on Mondays and Wednesdays. Do not miss them, as almost all of the examination material will be drawn from the lectures. Handout notes will be made available before the lectures, but these are no more than brief speaking notes. You are expected to add detail to them. The lectures will also cover the crucial information that will help you to complete the labs and assignments with a minimum of stress.

Laboratory Work

There are two labs per week, each in a two-hour time slot. Labs will be held in Owheo G06 (Lab A). 16 of the labs are assessed, and these are each worth 1% of your final grade. Non-assessed labs usually build up to the following assessed labs, so should not be skipped. Lab sessions are held on Tuesday/Wednesday and Thursday/Friday of each week. You will be allocated to a lab stream at the start of the course, but can attend the other lab times if there is room.

We expect labs to take around two hours to complete *on average* – some students may take shorter or longer, and some labs may require more or less effort. The lab times are opportunities for you to get help on the exercises. Working on the weekly lab work outside of these times is expected and recommended. Ideally you should make a start on labs *before* your scheduled lab session, so that you can get help with any problems you may encounter.

The laboratory sessions build up a running example, and so it is important that you keep up to date with them. This is why we require assessed labs to be marked off within 7 days of the assigned lab.

Throughout the lab book you will see highlighted sections marked with icons in the margins. These come in various forms:

Objectives, at the start of each lab, give a brief overview of what the lab covers, and what you should achieve by the end of it.

Task 1.1: Tasks identify key steps in completing the lab. There will typically be several tasks in a single lab, which build up to complete the overall objective.

Short videos accompany some of the labs. These might show what the finished result of the lab looks like, demonstrate some tool or technique, or give a bit more background material.

Warnings identify things that might go horribly wrong. There are not many of these, but it is a good idea to pay attention to them.

Assignment notes indicate how the lab work relates to the assignment. These can help you get started on the assignment early, avoiding a rush as the deadline approaches.

Assignments

There are two assignments for this course. Assignment 1 is based on client-side scripting with JavaScript, Ajax, and XML, and is worth 17% of your final grade. Assignment 2 adds server-side scripting with PHP, and is worth 17% of your final grade. The two assignments are linked, and are concerned with writing a web application where people can book dogs. The assignments build on information from the lectures, and skills gained in the laboratories.

More details of Assignment 1 are given in Chapter C and details of Assignment 2 can be found in Chapter D. The due dates are 06:00 p.m. Sunday August 29 and 06:00 p.m. Sunday October 3, and late assignments will be penalised at the rate of 10% per working day. Late penalties are applied as a multiplier, so if your raw mark was 70% and you submit your work 2 days late, your final grade will be $(1 - 2 \times 0.1) \times 70 = 56\%$.

The Final Examination

The final exam is worth 50% of your grade for COSC212. The exam will consist of mostly short answer questions, and previous years' examinations are available online from the University Library at `http://www.otago.ac.nz/library/exams/`. While your coding abilities will help you to do well, the exam is more about testing your level of knowledge and insight. Practical skills are tested and assessed through the lab exercises and assignments.

Assessment Summary

Assessed Component	Due Date	Marks
Assessed Labs	Continuous	16%
Assignment 1	06:00 p.m. Sunday August 29	17%
Assignment 2	06:00 p.m. Sunday October 3	17%
Final Exam	TBA	50%
TOTAL		100%

Assessed Labs will be accepted for marking for 7 days after the lab. Late *assignments* will be penalised at 10% per working day (or fraction). Further extensions for labs and assignments can only be granted by the course co-ordinator, Dr Steven Livingstone. Reasons for extensions shall usually be limited to serious illness or bereavement, and documentation will be required.

A.2 Appropriate Behaviour

- We expect you to treat the laboratory and your colleagues with respect.
- Any work you submit to us for assessment must be your own.
- Your attention is drawn to the departmental regulations on computer use which can be found at `http://www.cs.otago.ac.nz/regs.html` and the university's regulations on academic integrity, especially:
  - The University's Academic Integrity Policy
  - The Student Academic Misconduct Procedures
  - Academic Integrity: A Brief Guide for Students

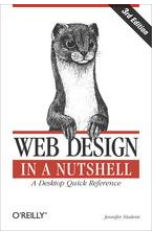
A.3 Workload

COSC212 is an 18 point paper, which equates to about 180 hours of work. Allowing for the mid-semester break and exam revision at the end of the semester, this is spread over roughly 15 weeks, giving an *average* workload of 12 hours per week. Half of this is taken up with lectures and scheduled laboratory sessions. The remaining 6 hours per week may be independent reading and study, additional time on the lab exercises, and time working on the main assignments.

Note that the 12 hours per week is an average estimate only. You may find that you need to spend more or less time than this. You may also find that the amount of time you need to spend on COSC212 varies through the semester, although with two labs scheduled in most weeks the course does reward a consistent commitment of time.

A.4 Reading Material

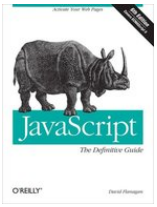
There is no single text for COSC212, but there is a range of useful resources. Several books are given below, which are available through Safari Books Online. Otago has a subscription to this service, so you should be able to reach these resources from any university computer. Individual resources will be recommended throughout the course, but a few key ones are:



Robbins, Jennifer Niederst, *Web Design in a Nutshell.* O'Reilly Media, 2006.

`http://my.safaribooksonline.com/0596009879`

The main text for COMP112, this is still useful for COSC212. It provides information about HTML and CSS as well as an introduction to JavaScript.



Flannagan, David, *JavaScript: The Definitive Guide.* O'Reilly Media, 2001.

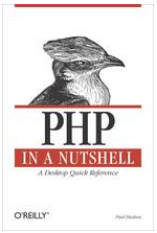
http://my.safaribooksonline.com/9781449393854

Flannagan provides a more detailed look at JavaScript, which is the language we'll be using for much of the first half of the course. It discusses the language in detail, and introduces the jQuery library.

Crockford, Douglas, *JavaScript: The Good Parts.* O'Reilly Media, 2008.

http://my.safaribooksonline.com/9780596517748

Crockford gives a critical view of the JavaScript language and identifies a range of issues. It also provides advice about the best way to use the language while avoiding these pitfalls.



Hudson, Paul, *PHP in a Nutshell.* O'Reilly Media, 2006.

http://my.safaribooksonline.com/0596100671

Gives an overview of PHP, which is the language that we will be using in the later parts of the course for server-side scripting.

A.5 Course Overview

After a brief introduction, the first main part of the course focusses on client-side scripting. This allows web developers to link programs to events in the user's browser, allowing complex behaviour in response to events such as mouse clicks or text entry. We will then look at server-side scripting, where data from the user is sent to the Web server, which then runs a program to generate a Web page in response. We will then consider two key topics in Web application development – maintaining *state* between Web pages, and security and authentication. Finally we will examine some common types of Web application to see how these components fit together.

A timetable for the planned lecture topics and laboratories is given in Table A.1. Note that this is subject to change, particularly the second half of the course.

Table A.1: Course Timetable

Week	Starting	Lecture	Lab
1	6 Jul	Introduction	HTML and CSS Revision
		Deployment	Git
2	13 Jul	JavaScript Intro	Hello, JavaScript (1%)
		JavaScript Arrays and Objects	JavaScript Arrays and Objects (1%)
3	20 Jul	JavaScript Scope and Closure	JavaScript Closures
		JavaScript Forms and Storage	JavaScript Cookies (1%)
4	27 Jul	Http and Networks	JavaScript Form Processing (1%)
		Web Mapping	JavaScript Form Processing(1%)
5	3 Aug	jQuery Intro	Maps (1%)
		jQuery, Axjax and XML	Introduction to jQuery (1%)
6	10 Aug	jQuery Animation	jQuery, Ajax, and XML (1%)
		Deployment scripting	jQuery Animation
7	17 Aug	Clients and Servers	Sapphire, Permissions and (S)FTP
		CGI Scripting	Catch Up
	24 Aug	Break	
8	31 Aug	PHP 1	Hello, PHP (1%)
		PHP 2	PHP Form Processing (1%)
9	7 Sep	PHP 3	PHP Cookies and Sessions (1%)
		Intro to Databases	PHP and XML (1%)
10	14 Sep	PHP + Databases	PHP and JSON
		Security - Authentication	Getting Started with MySQL (1%)
11	21 Sep	Security - Injection	MySQL and PHP (1%)
		Security - Encryption	Authentication and Sessions (1%)
12	28 Sep	Alternative Technologies	More Authentication (1%)
		Case Study	Catch-Up/Assignment
13	5 Oct	Revision + Exam	No Lab
		(Slack)	No Lab

Chapter B

Getting Help in the Labs

Demonstrators will be available to help in the scheduled laboratory times. As with previous papers you will have taken in the department, the DemoCall system will be used to manage requests for help. Remember that the labs can get busy, particularly towards the end, so it is important to do what you can before the lab so that you can get the most help when it is available.

B.1 Helping Yourself

You should not rely on the demonstrators as your first point of call for help. There are many resources that are available to you for help and advice, including:

This lab book – make sure that you have read the lab material. Sometimes the answer to your problem might just be a page away! Remember that the labs build on one another, so material from an earlier lab will still be relevant to later ones.

The lecture material – particularly the lecture scheduled just before the lab, but again material presented early in the course is still relevant later on. Note that this should include your own notes taken during the lectures, since the handouts are only brief bullet points.

Reference books – the ones suggested in Section A.4 are all available online.

Online documentation – Sites such as SitePoint (`http://www.sitepoint.com/`), WebPlatform (`http://www.webplatform.org/`), the Mozilla developer network (`https://developer.mozilla.org/`) and W3Schools (`http://www.w3schools.com/`) have a wide range of material. The online PHP documentation (`http://php.net/`) is one factor in the language's success.

The Internet at large – entering your error messages or 'How do I ...' into search engines often turns up quick and easy solutions to particular problems. Sites such as `http://stackoverflow.com/` often give explanations and pointers to resources as well as solutions to problems.

Your fellow students – you are encouraged to discuss the course with your peers, and to talk about different solutions to problems. You do need to be careful about the rules for plagiarism etc., but as a general guide talking about problems is OK, but sharing code is not. It is OK to tell someone that their HTML layout problem can be solved using the CSS float property, but not OK to write their CSS for them.

Remember, however, that no single source is 100% reliable, and so the more you make use of, the better the information you have will be.

B.2 Incremental Development

One very useful technique for identifying and solving problems when programming is using *incremental development*. You start with the most basic working example of your program you can, and gradually add functionality, testing as you go. This means that when something goes wrong you have a pretty good idea where the issue is – it is probably in something you've written in the last 10 minutes. In order to diagnose the problem you need to ask yourself two questions – 'When was it last working?' and 'What have I changed since then?' If the answer to the first question is 'last week sometime' and to the second 'about 200 lines of code in 5 files', you'll have a much harder time than if your answers are 'five minutes ago' and 'just the CSS for that element'.

B.3 Getting Help

When you do need help from us, don't be afraid to ask, but you should expect us to ask questions like:

- 'What exactly is the problem?'
- 'What have you tried to do to fix it?'
- 'Where have you looked for solutions?'
- 'When was your code last working?'
- 'What have you changed since then?'

We expect you to have good answers to questions like these. We also expect for your code to be professionally presented. Poorly laid out and un-commented code is very hard for us to understand or debug, and if the lab is busy we cannot spend time sorting out which brackets match or which tags are closed.

Chapter C

Assignment 1

Due: 06:00 p.m. Sunday August 29. Worth: 17% The assignments build a Web application to manage bookings for a particular *Dog Rental Service* (we leave the exact name up to you). This company has dogs available for hire (by the hour) to fill that dog-shaped void in some peoples' lives. To cater for these poor, unfortunate people the *Dog Rental Company* will rent you a dog (or dogs) for an afternoon walk along the beach or a companion to watch T.V. with.

- Assignment 1 provides an interface where *staff* can see which dogs are available and at what times/-dates, and for customers to see what dogs are available as well as read reviews from previous customers. You will also add an interactive map for people to view local walking tracks, beaches and camping spots.
- In Assignment 2 you will add the ability for people to make bookings and for administrators to cancel bookings.

It is very important that you protect your assignment work from unauthorised copying. Make sure that your repository is set to private and add teaching staff (Nick Meek, Karen Gray, Steven Livingstone and Veronica Liesaputra) to the repository, so only you and the teaching staff can access the website and the sources.

Assignment 1 uses client-side scripting in JavaScript. In order to complete this assignment you will need to know material presented in the first five weeks of the course. However, you can begin to work on the assignment earlier than that, and can start designing your site from week 1.

The data for the assignment is driven from JSON files, and sample files are available from Blackboard in the `Assignments` section. Note that the information provided in these JSON files is an example only, and your application should reflect any changes to these files. You *may not alter the structure* of the JSON files by adding properties. There are four JSON files:

- `animals.json` Provides information about each dog and the price per hour to rent the dog.
- `reviews.json` Lists the reviews of previous customers.
- `bookings.json` Provides information about which bookings have already been made*.
- `POI.geojson` Lists the locations (georeferences) of walking tracks etc..

* You can book up to three (3) dogs at the same time.

Example Data Files

animals.json

```
{
  "animals": {
    "dogs": [
      {
        "dogId": "DW-001",
        "dogName": "Fido",
        "dogType": "Poodle",
        "dogType": "Small",
        "description": "Excellant␣lap␣dog,␣doesn't␣shed.",
        "pricePerHour": "3.0"
      },
      {
        "dogId": "DW-002",
        "dogName": "Rover",
        "dogType": "Springer␣Spaniel",
        "dogType": "Medium",
        "description": "Very␣energetic,␣loves␣water.␣Good␣with␣kids",
        "pricePerHour": "5.0"
      },
            {
        "dogId": "DW-003",
        "dogName": "Rex",
        "dogType": "Retreiver",
        "dogType": "Large",
        "description": "Eats␣everything,␣very␣friendly.␣Slobbers␣a␣lot.",
        "pricePerHour": "5.0"
      },
            {
        "dogId": "DW-004",
        "dogName": "Digby",
        "dogType": "St.␣Bernard",
        "dogType": "Huge",
        "description": "Excellent␣horse␣replacement",
        "pricePerHour": "10.0"
      }

    ]
  }
}
```

bookings.json

```
{
  "bookings": {
    "booking": [
      {
```

```
        "dogId": ["DW-001", "DW-002"],
        "name": "Garner␣Family",
        "pickup": {
          "day": "15",
          "month": "8",
          "year": "2021",
          "time": "13:30"
        },
        "numHours": "4"
      },
      {
        "dogId": ["DW-003"],
        "name": "Mr␣and␣Mrs␣Jones",
        "pickup": {
          "day": "15",
          "month": "8",
          "year": "2021",
          "time": "10:00"
        },
        "numHours": "6"
      }
    ]
  }
}
```

reviews.json

```
[
  {
    "title": "Great␣Company",
    "author": "Test␣User",
    "reviewcontent": "Great␣company,␣great␣dogs,␣great␣prices!"
  },
  {
    "title": "Had␣a␣Ball!",
    "author": "Test␣User2",
    "reviewcontent": "Had␣a␣great␣time␣throwing␣the␣ball.␣Lovely␣to␣have␣a␣dog␣just␣when␣I␣want␣one."
  },
  {
    "title": "Friendly␣and␣Helpful",
    "author": "Test␣User3",
    "reviewcontent": "The␣staff␣couldn't␣be␣nicer␣and␣the␣dogs␣are␣fantastic."
  }
]
```

POI.geojson

```
{
```

```
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
        "marker-color": "#7e7e7e",
        "marker-size": "medium",
        "marker-symbol": "",
        "type": "campsite",
        "name": "Car␣Rental␣Office"
      },
      "geometry": {
        "type": "Point",
        "coordinates": [
          170.5153065919876,
          -45.90540437585189
        ]
      }
    },
    {
      "type": "Feature",
      "properties": {
        "marker-color": "#7e7e7e",
        "marker-size": "medium",
        "marker-symbol": "triangle",
        "name": "St␣Clair␣Beach",
        "type": "landmark"
      },
      "geometry": {
        "type": "Point",
        "coordinates": [
          170.49052566289902,
          -45.911734850052895
        ]
      }
    }
  ]
}
```


C.1 Requirements (Must-Haves)

Your Web application must allow the visitors to:

- See a list of dogs including breed and description.
- See preview images of the dogs including descriptions and so on that change as the user selects different dogs.
- Enter the date on which they wish to pickup the dog and find out which options are available for that time.

- Select a(n) available dog option(s), and save the information to localStorage. Look at `bookings.json`, that is the information that will be needed to complete the booking later.
  - The information should be saved in localStorage.
  - There is no need to proceed to booking yet, although this will be required in Assignment 2.
- See the location of the Company Office on the map, as well as the following points of interest:
  - At least three parks in Dunedin
  - At least three walking tracks near Dunedin
- The points of interest should be stored in the `POI.geojson` file in GeoJSON format.
- Visitors should be able to show or hide the parks and/or walking track markers on the map.
- Visitors should use a page called `index.html` as their main entry point to the site.
- You should also provide a separate view of the booking information for Company Office staff. This view should be separate from the customer's view, but need not be secured at this stage.

Company Office staff should be able to:

- See the details of all existing bookings.
  - This *does not* need to show incomplete bookings stored in the localstorage.
- Company Office staff should use a page called `admin.html` to view this information.

Additionally:

- You should use HTML5 and CSS3 for your site, and you should validate your code.
- You can use jQuery UI for building the user interface elements and providing a easy to navigate UI (e.g., `datepicker`, `controlgroup`).
- You should also use JSLint to identify potential issues with your JavaScript code

C.2 Deliverables

Your site should be stored in the department's GitLab server (`altitude.otago.ac.nz`) and served using GitLab Pages e.g., `https://<yourusername>.cspages.otago.ac.nz/assignment1`. Make sure that your repository is set to private and add teaching staff (Nick Meek, Karen Gray, Veronica Liesaputra, Steven Livingstone) to the repository, so that only you and the teaching staff can access the website and the sources. In addition you must submit a copy of your solution as described below.

Your submission should include all of the files needed for your site, but you should exclude any unnecessary files to reduce the amount of space used. You should also include a report, in PDF format, called `assignmentOne.pdf`. This report should include:

- A link to your site, e.g., *https://<yourusername>.cspages.otago.ac.nz/assignment1*.
- A list of any third-party code or libraries used in your submission. Note that all of the core functions should be implemented by you.
- A list of the JSLint options you chose when checking your code, and explanations as to why you chose these options.

- A description of how you tested your application, what test cases you used, and how you ensured that it worked correctly.
- Any outstanding issues or problems that you are aware of with your application as submitted.
- A brief description of the enhancements (if any) that you made beyond the basic requirements.

Your report should be brief, no more than two pages long in a 12pt font. You should be sure to present your report and your code professionally and thoughtfully.

Submitting Your Assignment

To submit your assignment email your report to Nick Meek (nick.meek@otago.ac.nz). Ensure the subject line of the email is: *212 Assignment 1: <your student ID>*.

Apart from files that we provide to you, the core jQuery library, jQuery UI and any libraries that we use in the Maps lab, all the code and markup that you submit must be your own work. The use of other third party libraries, whether CSS, HTML, JavaScript, or other languages, is *not* permitted. Images must either be your own work, in the public domain, or used under a license that allows re-use for educational purposes. Appropriate attribution should be given for all images.

C.3 Marking Scheme

The marking scheme for Assignment 1 is given in Table C.1. This marking scheme is provided as a guide only—we reserve the right to award or deduct marks for items not included in this scheme.

Table C.1: Assignment 1 marking scheme

Correctness		
Valid HTML/CSS/JavaScript	10	
General usability & appearance	5	15
Booking Functions		
Allow visitors to view dog options and details	10	
Allow visitors to view reviews	5	
Allow visitors to save and view potential bookings	10	
Allow staff to view bookings	10	35
Map Functions		
Display of office location	5	
Display of nearby locations	5	
Usage of GeoJSON for storing locations	10	
Show/hide functionality	5	25
Style and Testing		
Appropriate use of comments	5	
Appropriate separation of HTML/CSS/JS	5	
Appropriate JavaScript structures	5	
Report and testing strategy	10	25
Total		**100**

Chapter D

Assignment 2

Due: 06:00 p.m. Sunday October 3. Worth: 17%

It is very important that you protect your assignment work from unauthorised copying. When your files are stored in your network home directory, or on remote servers, you should ensure that these locations are not readable by other students.

Assignment 2 uses server-side scripting to add to the Web application developed in Assignment 1. In order to complete this assignment, you will need to know material presented up to week 9 of the course.

Assignment 2 builds on Assignment 1, and a sample solution to Assignment 1 will be made available early in the second half of the semester. This sample solution will provide a basis for students who did not complete Assignment 1. You can use either your own solution to Assignment 1, or the sample solution. If you choose to build on the sample solution, then you should be aware that it is a very minimal implementation of the core requirements of Assignment 1, and so will need some extension and customisation.

D.1 Requirements (Must-Haves)

The web application from Assignment 1 should be extended with server-side functionality. This will involve writing PHP scripts which update the JSON files in response to user input.

Your web application must allow a visitor to:

- Make a booking for a particular dog.
  - These bookings must be recorded in the JSON files on the server.
  - The use of localStorage in Assignment 1 was an interim step since there was no server-side scripting. For Assignment 2 bookings should be stored in JSON files, not in localStorage.

Your web application must allow staff to:

- Cancel any booking.
- Add or remove dogs from the list of those available in `animals.json`. You should consider what happens if there are existing bookings for a dog that is removed.
- Edit the details of existing dogs in `animals.json`.

D.2 Deliverables

As with Assignment 1, all the files that make up your site should be stored in the department's GitLab server (`altitude.otago.ac.nz`). Make sure that your repository is set to private and add teaching staff (Nick Meek, Karen Gray, Veronica Liesaputra, Steven Livingstone) to the repository, so that only you and the teaching staff can access the source code of your website.

In addition, you should write a report, in PDF format, named `assignmentTwo_`*`yourusername`*`.pdf`. This report should include:

- The URL of the repository that contains the source code for your assignment.
- A description of your application.
- A list of any third-party code or libraries used in your submission. Note that all of the core functions should be implemented by you.
- A description of how you tested your application, what test cases you used, and how you ensured that it worked correctly.
- Any outstanding issues or problems that you are aware of with your application as submitted.
- A brief description of the enhancements (if any) that you made beyond the basic requirements.

Your report should be brief, no more than two pages long in a 12pt font. You should be sure to present your report and your code professionally and thoughtfully.

Submitting Your Report

After you have ensured that your repository contains the files that you want to have marked, you should email your report to Nick Meek (nick.meek@otago.ac.nz). Ensure the subject line of the email is: *212 Assignment 2: <your student ID>*.

D.3 Marking Scheme

The marking scheme for Assignment 2 is given in Table D.1. This marking scheme is provided as a guide only—we reserve the right to give or take marks for items not included in this scheme.

Table D.1: Assignment 2 marking scheme

Correctness		
Valid HTML/CSS/JavaScript/PHP	10	
General usability & appearance	10	20
Required Functions		
Admin page needs logon to access	10	
Allow visitors to make a booking	10	
Allow staff to cancel any booking	10	
Allow staff to add, remove, and edit dogs	20	50
Style and Testing		
Appropriate use of comments and PHP structures	10	
Report and testing strategy	20	30
Total		**100**

Part II

Getting Started

Lab 1

HTML and CSS Revision

The objective of this lab is for you to review the HTML and CSS material you learned in COMP112. You will download some HTML files, correct some validation errors, and use CSS to style the pages.

1.1 Before We Begin

Check your lab stream on the notice board beside CS reception. There are two labs per week, and you should be streamed into one session on Tuesday or Wednesday and one on Thursday or Friday. You can come along to labs at other times as well, as long as there is space, and can use the lab whenever classes are not scheduled. COSC212 labs are held in Lab A, but you can use any of the labs when working in your own time.

The COSC212 Lab Environment

The lab machines are Apple iMacs running macOS, and so should be familiar to you from the other courses you have done in the department.

Your main tool will be an editor, and while you could use a plain text editor, an integrated development environment (IDE) offers more support for a productive workflow. A web-focused IDE available on the lab machines is PhpStorm, and we recommend that you use this for most of your development. Occasionally you may need to edit a file while working in a command prompt. There are several editors available from the command prompt, the main ones being Emacs, Vi, and Nano. If you are familiar with Emacs or Vi then you can use one of those, otherwise, we recommend Nano as a simple editor for small files.

Your username and password will be the same as it was for your last course in the department, and you should find that your home directory has followed you too. If you have forgotten your username or password, they don't seem to be working, or your home directory is missing, then there will be help in the labs during the first week to resolve any problems.

In addition to the lab machines, you will use a remote server, `sapphire.otago.ac.nz`, which will provide web server capabilities for this course. You'll learn more about `sapphire` in Lab 13.

Obtaining a PhpStorm License

You need a license to use PhpStorm. Fortunately JetBrains grant free Educational licenses to bona fide students. Open a browser and go to `https://www.jetbrains.com/shop/eform/students` and fill out and submit the form. Ensure you apply using your University of Otago email address. Otago is known to JetBrains and so using your Otago email address streamlines the whole process.

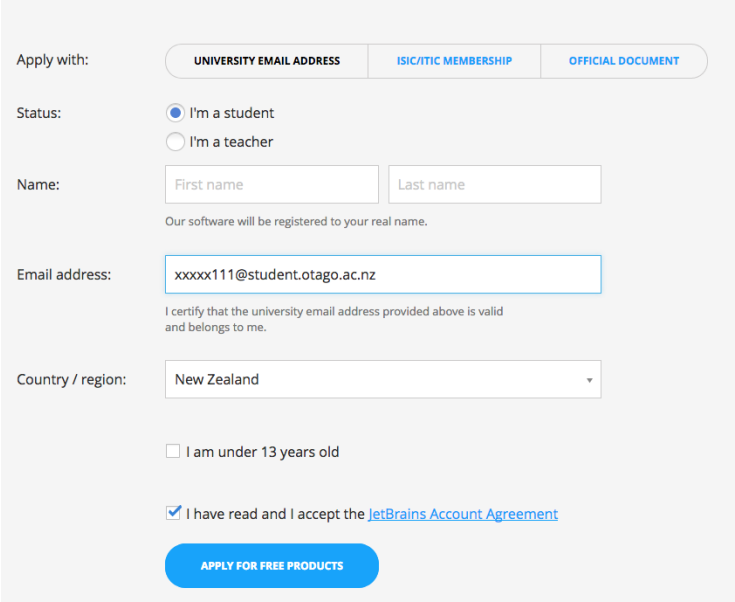


Figure 1.1: JetBrains License Application Form

After submitting the form check your student email, there will be an email from JetBrains leading you through the rest of the process. It only takes a minute or two and then you have a 1 year license to a whole range of JetBrains software in addition to PhpStorm.

1.2 HTML and CSS

HTML and CSS should be familiar technologies for you from your studies in COMP112. HTML is used to provide *logical structure* to a web page, while CSS is used to specify how that structure *should look* when displayed. As you should recall from COMP112, there are several versions of HTML and CSS. In this course we will be using HTML5 and CSS3

Task 1.1: The files that you need for this lab are available from the course Blackboard Pages in the Lab Materials section. You should download them to your local directory, and unzip them.

Watch the short video 'PhpStorm - Setup' available from Blackboard> Course Documents > Screencasts. This shows you how to activate PhpStorm and create a new project in PhpStorm using the lab files.

Before we start to write the CSS, open the file `index.html` in a web browser. You should see some basic HTML, without any fancy formatting. There is a list of navigation links at the top, and some content below.

Task 1.2: Begin by validating the HTML. The W3C validator can be found at `http://validator.w3.org/`.

You can validate the HTML either by File Upload, or by Direct Input. You should see several warnings

on each page, and a few errors as well. Note that if you use the PhpStorm IDE, many HTML errors will be identified and indicated as you type – just one of the advantages of an IDE over a plain text editor.

In general you should treat all Warnings as Errors – this is generally good practice in Computer Science. The validator should indicate that there is one issue that is common across all of the HTML pages, and one page that has some additional problems. Note that a single mistake can sometimes cause more than one error message to appear.

Task 1.3: Correct the HTML so that the pages successfully validate. You should be sure to understand why each of the warnings appears before trying to correct them.

The changes you make to the HTML probably won't have any effect on how the pages appear in the browser. Modern browsers are often very good at guessing what invalid HTML is 'supposed' to mean. Relying on this, however, is a very bad idea – you cannot tell what browser the user will use, or what that browser will do to your (invalid) HTML. Adhering closely to the HTML standards is the best way to make sure that your pages can be viewed as intended by as many people as possible.

Task 1.4: Create a CSS style sheet and link it to the HTML in order to make the pages look like Figure 1.2, Figure 1.3, and Figure 1.4. Full-resolution versions of these images are included in the Zip file. You should not need to change the structure of the HTML, except as it relates to the validation errors you identified earlier, but you may wish to add `id` or `class` attributes to some tags. Your CSS should validate using the W3C validation service at `http://jigsaw.w3.org/css-validator/` with no errors or warnings.

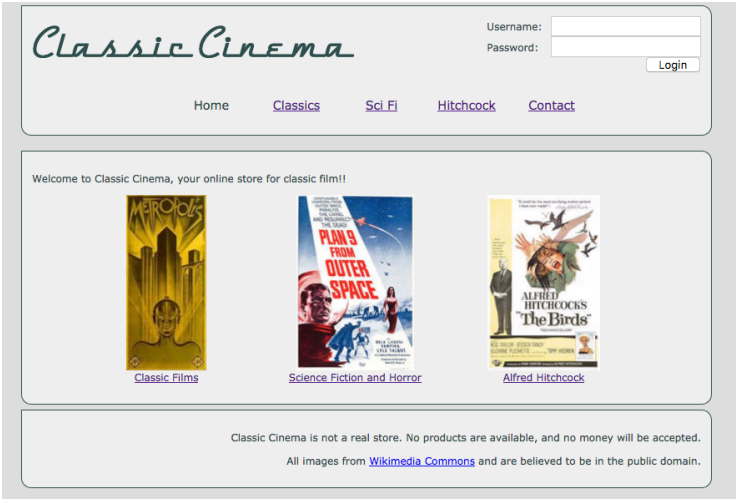


Figure 1.2: CSS style applied to `index.html`

A few details that may not be clear from the images:

- The main body font should be Verdana, if available, or the browser's sans-serif font if not.
- The main headings (`<h1>` and `<h2>`) use the DymaxionScript font, which can be downloaded from `http://www.fontsquirrel.com/fonts/DymaxionScript`. If you click on 'Webfont Kit' you can download a ZIP file with the required files, and with a CSS stylesheet which includes a `@font-face` declaration that you may find useful.

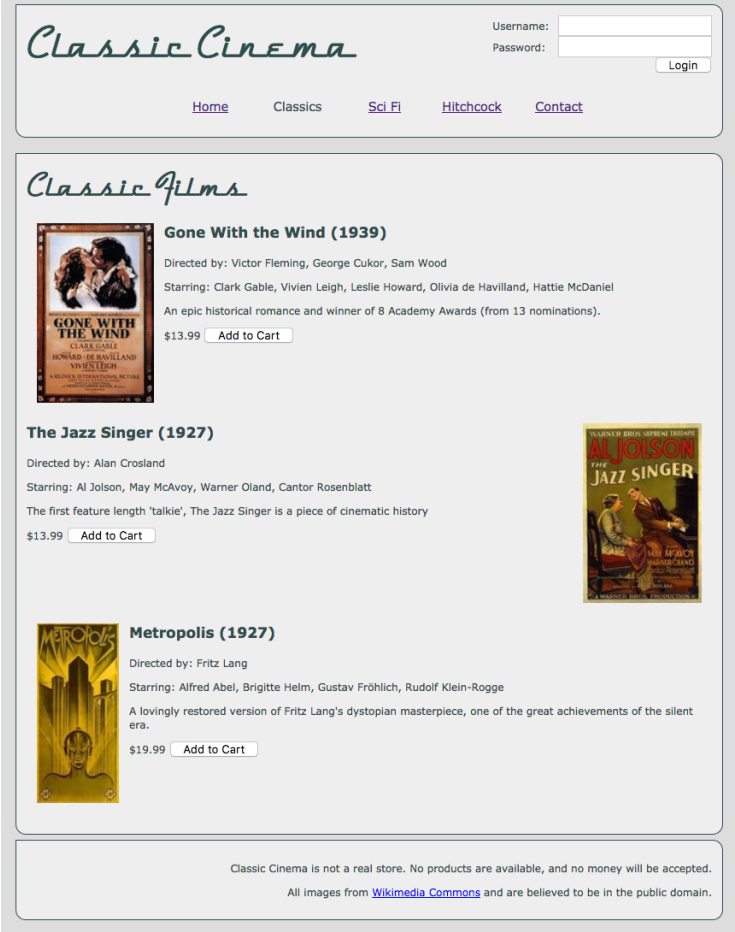


Figure 1.3: CSS style applied to `classic.html`

- Body text is 10pt size, navigation links are 12pt, and headings (h1 to h3) are 48pt, 36pt, and 14pt.
- The darker grey background has hex value `DDDDDD`, and the lighter grey boxes hex value `EEEEEE`.
- The header's position is 'fixed'.
- The body is 700px wide with automatic margins left and right.
- The main block elements have 5 pixel margins, 10 pixel padding, and a 1 pixel wide black border with rounded corners as shown with a radius of 20 pixels.

For the assignment: No, it's not too early to start thinking about the assignment! You can start to plan how you'd like to structure your pages with HTML and how you'd like to use CSS to give them a pleasing appearance.

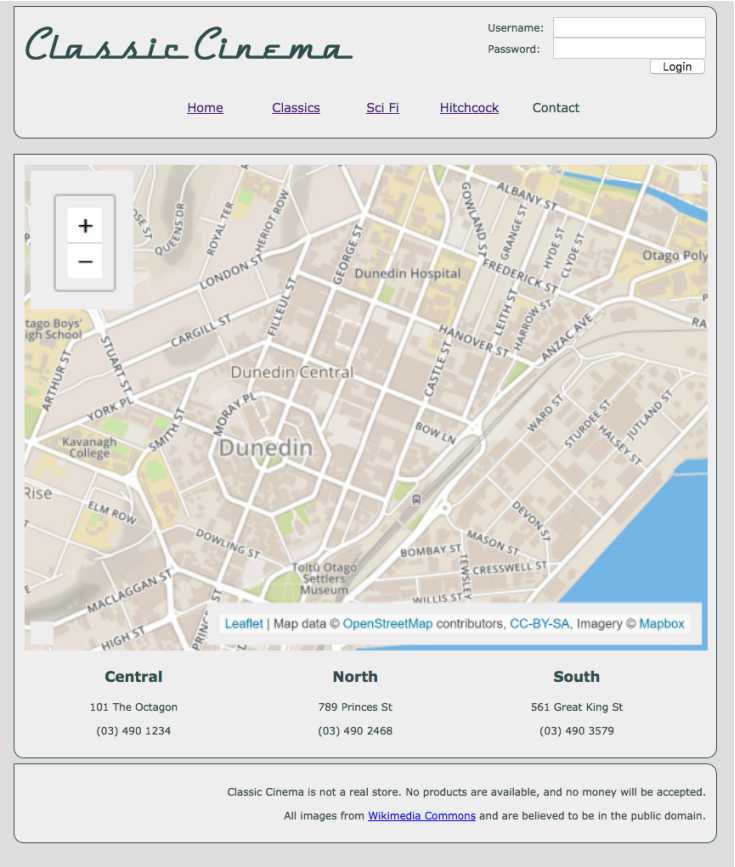


Figure 1.4: CSS style applied to `contact.html`

Lab 2

Git

In this lab you will learn how to use a version management system to allow managing multiple versions of website content. In particular, you will explore how to use a tool called Git, and see that such systems can provide an alternative to deployment using (S)FTP.

In COMP112 you worked on a single version of a file (HTML or CSS) and then uploaded it to a server via FTP. It was your responsibility to manage backups of any past versions of the content that you had created so, for example, you could revert to an earlier version of the content.

Often developers of web content, and of software code, will want to be able to maintain a structured history of past versions of the content that they create. This facilitates reverting undesirable changes back to "known good" content, as well as being able to examine the differences between specific versions, and being able to record metadata such the rationale behind particular committed versions—e.g., do they fix a bug, or add new content? Version management systems are one way to meet this need. Note that version management systems are also referred to using many other names, including source code management (SCM), revision control systems, and more generally as version control (VC).

In this lab, we will explore the use of the Git version management system. Git is an extremely popular version management system. Although Git was built originally for the purpose of handling the development of the Linux kernel, more recently it has facilitated some extremely effective approaches to open "community coding" that allow for very large numbers of contributors, such as the support for the "pull request" workflow provided by online services such as GitHub, Bitbucket and GitLab. The Department has its own GitLab server at https://altitude.otago.ac.nz There are plenty of other excellent version management systems that have different strengths and weaknesses, such as Subversion and Mercurial.

We will focus on how you can use Git to maintain past versions of website content, and how to use it to deploy content onto a live server. Note that version management systems are also frequently used to allow teams to work collaboratively on content, where the version management system will coordinate how to merge the work of different team members in a convenient way.

2.1 Git concepts

Before we start using Git, there are a number of concepts that need to be introduced.

Repository (repo). In version management systems, the term *repository* is used to describe the place where all of the past versions of content will be stored. In Git, this is (usually) on the same machine (in the same directory) as the project it is tracking, in a hidden .git directory.

Commits and revisions. Version management systems will not usually record every change that you make to a file. You need to instruct a tool such as Git when you have reached a state of the files that

Figure 2.1: Git: The popular view - from XKCD

you want it to remember for you. Your act of requesting Git to remember a particular version of files is known as *committing* a *revision* to the repository.

2.2 Using Git in PhpStorm

You can of course access all of Git's functionality from the command line, however we will use the assistance provided by PhpStorm.

In this first set of tasks you will see how to use it within PhpStorm.

Task 2.1: Create Some Test Content. ✓

Create a directory called 'gitTest' in your home directory.
Open PhpStorm and 'Create New Project'.
At the next step navigate to and open the directory you just created; click Create.
Create an `index.html` file by right-clicking on *gitTest* in the Project view.
Add the following content:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Testing use of git</title>
  </head>
  <body>
    <h1>Testing use of git</h1>
    <p>Some initial text.</p>
  </body>
</html>
```

Save.

Task 2.2: Create an Empty Git Repository ✓

From the menu choose VCS > Import into Version Control > Create Git repository.
Click Open on the next dialogue, the repository will be created in the Project root directory.

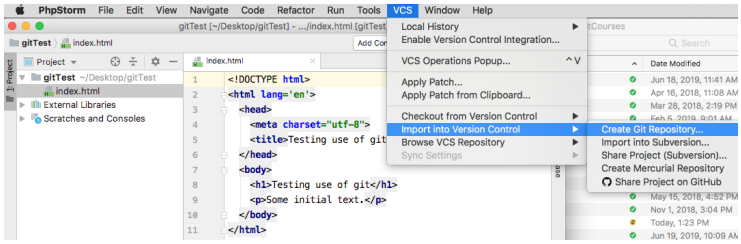


Figure 2.2: Enabling Git in PhpStorm

You have now created an empty git repository in your gitTest directory. You probably won't see it in the PhpStorm view, it's in a hidden directory.

Navigate to your gitTest directory in Finder and reveal the hidden files. There is an macOS shortcut to toggle showing hidden files: CMD + SHIFT + .

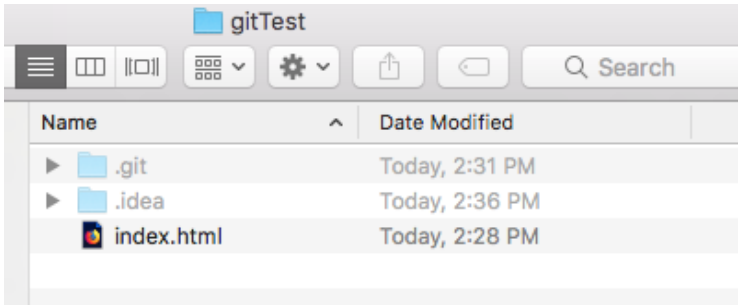


Figure 2.3: The hidden files created by Git and PhpStorm

Task 2.3: Adding Files to the Repository

A file must be 'added' to the repository in order for it to be tracked.

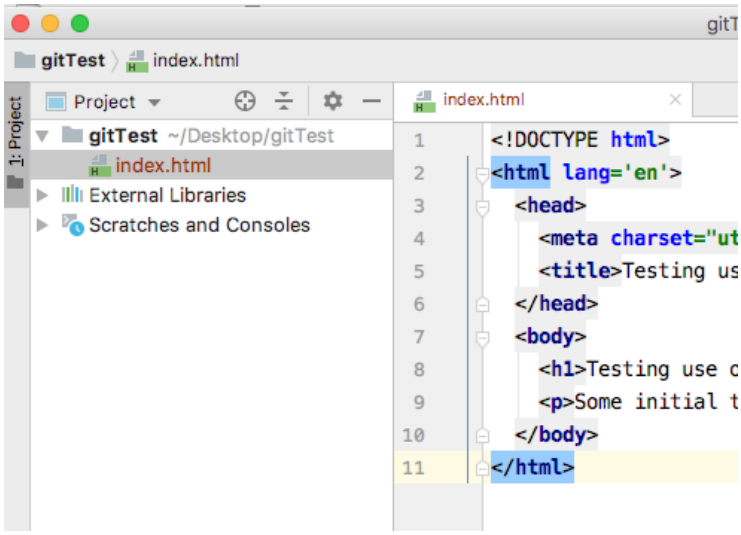


Figure 2.4: Colour indicates un-versioned files

Notice that index.html is coloured brown in the project view in PhpStorm. This indicates that the file is 'un-versioned'. For a full list of all the colour code see Colour Codes

Right-click on index.html and add it to the repository.

Notice it turns green in PhpStorm's project view.

Task 2.4: Committing to the Repository

Now that Git knows it should be tracking the changes to index.html we need to tell it to record its current state or *commit* it to the *repository*.

Right-click on gitTest in the Project view and choose Git > Commit directory

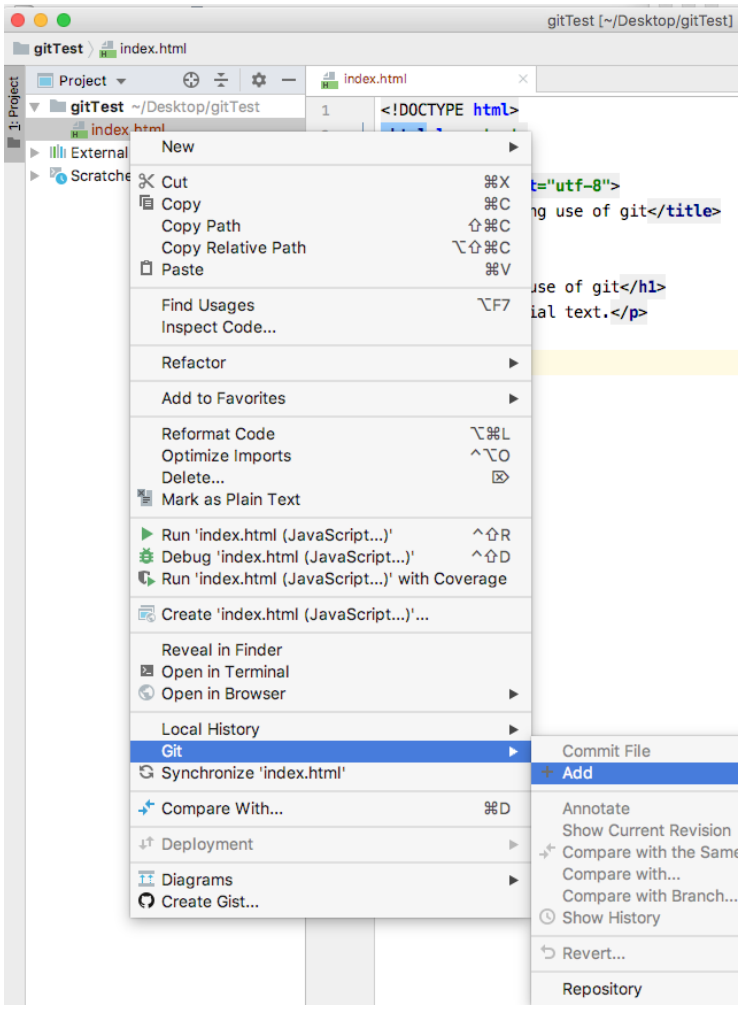


Figure 2.5: Adding index.html to the git repository

Note that `index.html` is checked as the file to commit. There are a number of files in the Unversioned Files folder that you don't really want to be versioned, mainly a lot of PhpStorm configuration files.

Add a meaningful commit message, 'Initial commit' or similar would be appropriate, and click the blue commit button.

If your code contains errors you will be given the option of reviewing them. By all means do so but you will still need to commit them afterwards.

2.3 Working with the Repository

Add some more content to `index.html`. just a few paragraphs of anything, Lorem Ipsum would be ideal.

Save and then commit the directory again, don't forget a suitable comment; "added some content".

Add a link to a stylesheet in the head section of `index.html`:

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

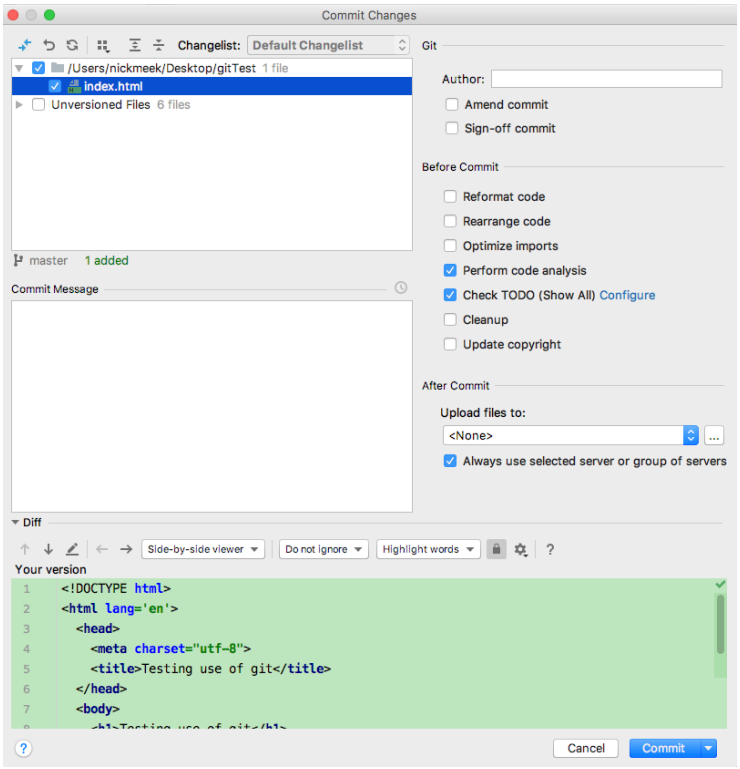


Figure 2.6: Committing index.html to the repository

Make a new stylesheet by right-clicking on 'gitTest' and choosing the appropriate option.

Add some basic style just so you know it all linked up properly:

```
html{
  background-color: rgb(230, 240, 255);
  color: rgb(0, 0, 4);
}
```

You may be asked if you want to add the file to the project; you do. If you aren't prompted then add style.css in the same way as you added `index.html`.

Save and commit the project again.

At the bottom of the PhpStorm window you will find a 'Version Control' button. Clicking it will toggle a panel that gives information about the current project. Click on the Log tab to see a representation of the history of your project. This is also one of the places where you can access more advanced features of Git such as branching, checking out previous versions etc. Feel free to experiment with these features but be careful, it is all too easy to get in to a horrible mess!

2.4 Pushing To GitLab

Currently any changes that are made (and committed) in gitTest are being tracked by Git and that, as a system, is useful all by itself. But it has a limitation in that the git repository is on your local machine (OK your network home in the CS context but that is a special case). What would be more useful is if the

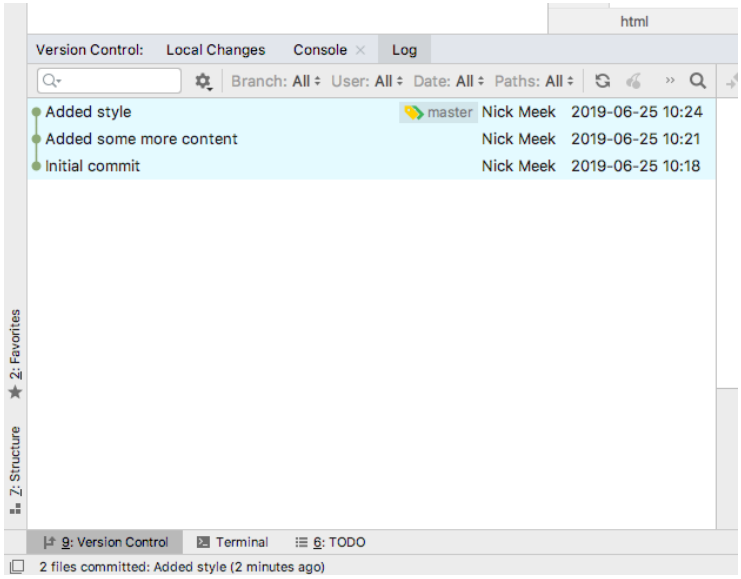


Figure 2.7: The commit log after 3 commits

repository was stored elsewhere. Then not only would you have an off-site backup (always a good idea) but also it is easy to work from different locations.

Task 2.5: Create an Empty Repository on Altitude and Pushing to it. ✓

In a browser open `https://altitude.otago.ac.nz`, your usual CS credentials should work. Click on the orange fox-head icon at top-left to be taken to your Dashboard. The student in Figure 2.8 already has one repository.

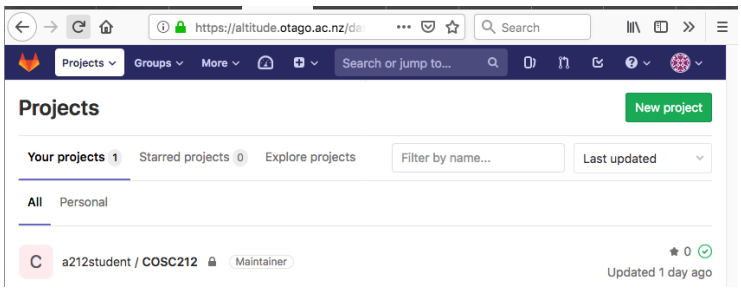


Figure 2.8: The GitLab Dashboard

Click the New Project button to create a new empty repository.
Give your project a sensible name.
Set the Visibility Level to *Public*.
Ensure the 'Initialise with Readme' box is NOT checked and click Create Project.

Click on the Clone button at top-left and copy the "Clone with HTTPS" url. You are not actually going to clone the directory you just need the address where the GitLab repository is.

Switch back to PhpStorm and from the VCS > Git menu choose 'push'. On the resulting dialogue click "Define remote" and then paste in the url you copied earlier. Leave the name as 'origin', see Figure 2.9

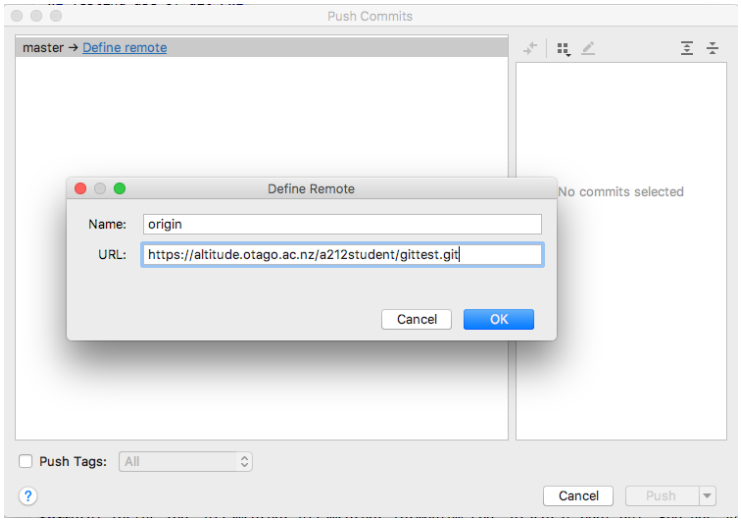


Figure 2.9: Defining the remote repo

Click OK, you will probably be warned about the authenticity of the remote host, it's us so don't worry. Now click *Push*, see Figure 2.10.

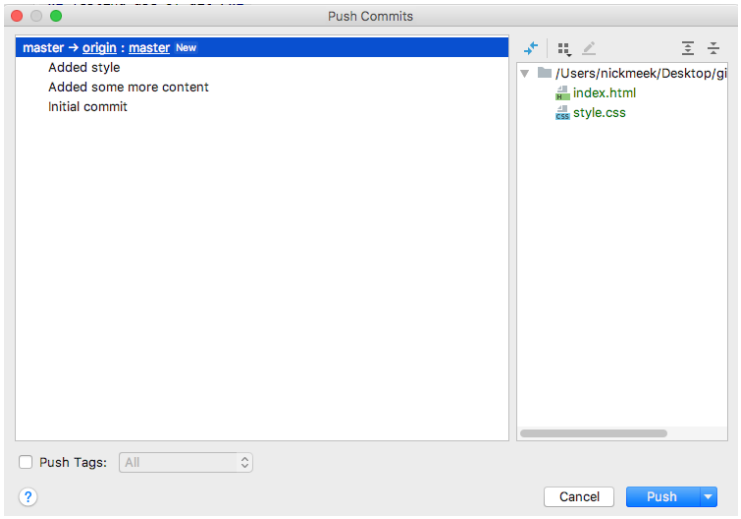


Figure 2.10: Pushing local commits to the remote repo.

You will be asked to login, again your usual CS credentials should work and your local repository will be pushed to the remote. Now there are two copies of the repository which each contain all the files

and the full history of the project.

Switch back to the Dashboard view in GitLab (on altitude) and select your gitTest project.

Find the "Add README button and add a short message.

Commit the changes.

Now the online repository is one step ahead of your local repo.

Switch back to PhpStorm. Obviously the README file is not there.

From the VCS > Git menu choose 'pull'.

Accept the defaults. After a few seconds the repos will be sychronised and the README will appear in your local project files.

Feel free to experiment with the other features but maybe use this gitTest account because things can easily get quite tangled.

2.5 Start Tracking Classic Cinema with Git

There is a finished version of Lab 1 (Classic Cinema) on Blackboard; download it to your local machine. This is the version we will develop throughout the semester. You might want to delete the version you worked on in Lab 1 to avoid confusion.

Task 2.6: Following the same steps as you did for the exercise above firstly make a local Git repository for Classic Cinema, add all the files* to it and commit them to the local repo. Then make a Classic Cinema repo on altitude and push your local repo to that. From now on we will mark work from the Git repository. You should get in the habit of committing frequently.

*Include subdirectories like `/images` but don't include files starting with an "`._`". In fact you can safely delete any file in your project that starts with a "`._`".

2.6 Enable Automated Publish-on-Push (Pipeline)

One common workflow pattern is for web pages to be published directly from the GitLab repository. That means that the web-server always has the most recently pushed files to serve. It is also a faster way of working than some as many of the tedious tasks are completely automated.

For the first half of this course we will use this push-to-publish workflow.

To enable it you need to add a magic file to the root directory of your project; it *must* be called *.gitlab-ci.yml*.

```
pages:
  stage: deploy
  script:
    - mkdir .public
    - cp -r * .public
    - mv .public public
  artifacts:
    paths:
      - public
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
```

You need to *add* it to the repository and then *commit* it. Now, next time you push, that file will be pushed too and then the processes in GitLab will process the file and publish your pages which can be viewed at https://*your username*.cspages.otago.ac.nz/*yourProjectName*/ You can also find the URL in Settings > Pages on GitLab

You will be asked to allow an API, allow it. If your project is set to Private (the default) then you will need to authenticate to see the pages. If you want everyone to be able able to see your pages then you will need to set it to Public in GitLab in Settings > General > Visibility, project features, permissions. Scroll down to Save.

2.7 Working From Other Locations

Now that you have Classic Cinema on GitLab you can easily work at home on the same files.

Task 2.7: Simulate working from another location using GitLab.

Make a folder on your Desktop called *MyFlat*.

In GitLab go to your ClassicCinema project and click on the Clone button and then copy the HTTPS address.

Switch back to PhpStorm and close any open projects and then from the dialogue choose Checkout from Version Control > Git.

Paste in the url you copied earlier and select the MyFlat directory as the location.

The files will be copied from the repository to MyFlat. Open the project.

Make some trivial change to `index.html`.

Save and commit.

Push. The change is pushed back to GitLab.

Close the project and open Classic Cinema from your Home directory. This project is now one step behind.

In PhpStorm choose VCS > Git > pull. Your repository is synchronised with the one on GitLab and the changes are reflected in the local project files.

If you are working in more than one location you will need to follow this workflow:

1. Pull
2. Edit > Save > Commit (any number of times)
3. Push

Delete the MyFlat directory when you have finished experimenting.

Part III

Client-Side Scripting

Lab 3

Hello, JavaScript

In this lab you'll start to work with JavaScript for client-side scripting. You will write your first JavaScript programs, use JavaScript to alter the HTML in a web page, and connect JavaScript code to basic events.

Note: This lab is assessed, and is worth 1% of your final grade. Please make sure that you get a demonstrator or teaching fellow to mark it off as completed.

3.1 Hello World in JavaScript

To get started, let's look at a basic script to say "Hello World". The JavaScript for this is very simple:

```
alert("Hello, World!");
```

and to add this to an HTML page we can use the `<script>` tags, along with HTML comments like this:

```
<script>
  alert("Hello, World!");
</script>
```

In the following tasks and labs you will be developing the Classic Cinema site from the initial static web site into a web application that provides a far richer user experience. In many of the following tasks you will create small test files to examine and test JavaScript concepts. You don't want to pollute your Classic Cinema site with these test files so you have a number of work-flow options:

- Make a new project for each lab and make all the test files in there. The advantages of this system are that you have a clear record of each lab and the Classic Cinema site is uncontaminated by unnecessary files.
- Make a folder in Classic Cinema specifically for test/lab files. The advantages are that is is quick and easy and you have a full record of all labs however the Classic Cinema site has unrelated files in it.
- Use Scratch files. PhpStorm allows you to make 'scratch' files. Advantages are it is fast and easy and doesn't really contaminate Classic Cinema. Downsides are that you have to do a little extra work to keep things organised.

Task 3.1: Make a new HTML file and put the "Hello World" script anywhere in the file. Open the file in your browser and see what happens – you should get a message box.

3.2 Unobtrusive JavaScript

Just as it is good practice to separate content (HTML) from presentation (CSS), you should keep behaviour (JavaScript) isolated as well. You can include JavaScript from an internal file in the `<script>` tags:

```
<script src="somefile.js"></script>
```

You can then put your JavaScript into the file `somefile.js`, away from the HTML.

This approach has several advantages:

- You get a clean separation between content, presentation, and behaviour.
- Browsers (or other agents) that don't understand JavaScript will not read the file.
- There is no confusion as to what is JavaScript and what is HTML – for example, what if you wanted to make an alert box with the text "`Missing </script> tag`"?
- You can easily re-use JavaScript code across multiple pages, and any changes only need to be made in one place.

Task 3.2: Re-write your hello world page so that the JavaScript is unobtrusive. Your JavaScript file will just need the single line

```
alert("Hello, World!");
```

The script tags can go almost anywhere in the page, but the usual places are:

- In the `<head>` section – this lines up with CSS includes and means that your JavaScript is loaded by the time the page is rendered. It also maximises compatibility with some (*very* old) browsers.
- At the end of the page, just before the `</body>` tag. This means that the HTML loads more quickly, as it can be rendered before the (possibly complex) scripts are loaded and parsed.

Whichever option you choose, be consistent so that it is easy to locate the `<script>` tags.

3.3 More Complex Behaviour

Task 3.3: The "Hello World" program isn't very interesting, so let's make a slightly more complicated version. Follow the instructions below to do this. If you run into problems with your JavaScript, read the next section, Debugging JavaScript, and see if that helps.

Begin by making an HTML page with the following elements in the body:

- A form with the following elements in it:
  - A text entry box with id `name`
  - A button element with id `hello`
- An empty paragraph with id `result`

Now link the HTML to a file containing the following JavaScript:

```
function sayHello() {
  var name;
   var target;
   name = document.getElementById("name").value;
   if (name.length === 0) {
     name = "World";
   }
   target = document.getElementById("result");
   target.innerHTML = "Hello, " + name + "!";
}

function setup() {
   var button;
   button = document.getElementById("hello");
   button.onclick = sayHello;
}

if (document.getElementById) {
   window.onload = setup;
}
```

The general pattern of this JavaScript is quite a common one, and is easiest to read from bottom to top. There are three main parts to the code:

1. At the bottom we have some script commands that are outside of any functions. These are run when the script loads. In this case, we check to see that the page has something that looks like a Document Object Model (DOM), and attaches some function (`setup`) to an event (`window.onload`). This means that once the page has finished loading, the function `setup` will be called. Note that there are no brackets on `setup` here. If we had

   ```
   window.onload = setup();
   ```

   the *result of calling* `setup` would be attached to the event.

2. Next we have the `setup` function. This uses the DOM to get a reference to an element on the page (the element with id `hello`). It then attaches a function (`sayHello`) to an event (the `onclick` event of the button). This means that the function will be called whenever the event occurs.

3. Finally we have the `sayHello` function. It uses the DOM to get the value of the textbox with id `name`, and changes the text of the paragraph with id `result`. This function is fairly simple but illustrates some key points

   - We can access HTML elements through the DOM, and can read and modify their properties.
   - We have the usual sort of programming constructs – variables, conditional statements etc.

Now when you press the button, the paragraph text should change to greet the person named in the text entry box. If there is no name in the box, then the output is "Hello, World!". There is one issue, however. Pressing 'Enter' in the entry box will cause the form to submit, which reloads the page. To overcome this we need to do two things:

- We need to add an event handler, so that the `sayHello` function is called when the form is submitted. If we get a reference to the form element from JavaScript, we can set its `onsubmit` event, just like we set the button's `onclick` event.
- We need to add the line `return false;` to the end of the `sayHello` function. Returning false to a form submission causes the form submission to fail. This is often used when validating form input – you return true if the form is OK and gets submitted, or false if there is a problem and the form cannot be submitted.

Task 3.4: Alter your code so that submitting the form has the same effect as pressing the button.

3.4 Debugging JavaScript

Debugging web applications is often difficult. You can only access them through the browser, but many browsers offer quite a lot of support for finding errors. How you get to these tools depends on the browser you use:

Safari You need to enable the developer tools. In Safari preferences, click 'Advanced' and then check the box beside 'Show Develop menu in menu bar'. Opening one of these will give you a separate pane at the bottom of the browser window, and you can switch between different tools there. These are shown in Figure 3.1.

Google Chrome A similar set of tools can be brought up by pressing Alt-Cmd-I (Mac) or Ctrl-Shift-I (Windows).

Internet Explorer Has similar tools available by pressing F12 (Windows).

Firefox has various Web Developer tools in the Firefox menu .

In most browsers you can also right click (or Ctrl-click on Macs with a one button mouse) on an element and choose "Inspect element" from the pop-up menu. This will open the developer view of the HTML source code, with the element you clicked on highlighted.

Assuming you're using Safari or Chrome, there are a number of useful tools here. 'Elements' lets you inspect HTML elements and their CSS properties. You can visually highlight their padding and margins, and see what CSS styles are being applied to them. 'Scripts' (in Safari) or 'Sources' (in Chrome) lets you see the source code for any JavaScript that has been loaded. Finally the 'Console' tab will let you see error messages from JavaScript, and you can also type JavaScript commands directly into the Console and they will take effect immediately.

Task 3.5: Using the console, make the text of the paragraph in your HTML page from Section 3.3 say "The Console has control".

When developing JavaScript it is often useful to output some debugging information. You can do this in a number of different ways:

- You can add an element to your web page and inject HTML into the DOM:

```
document.getElementById("debug").innerHTML = "My debug message";
```

- You can raise alert messages:

```
alert("My debug message");
```

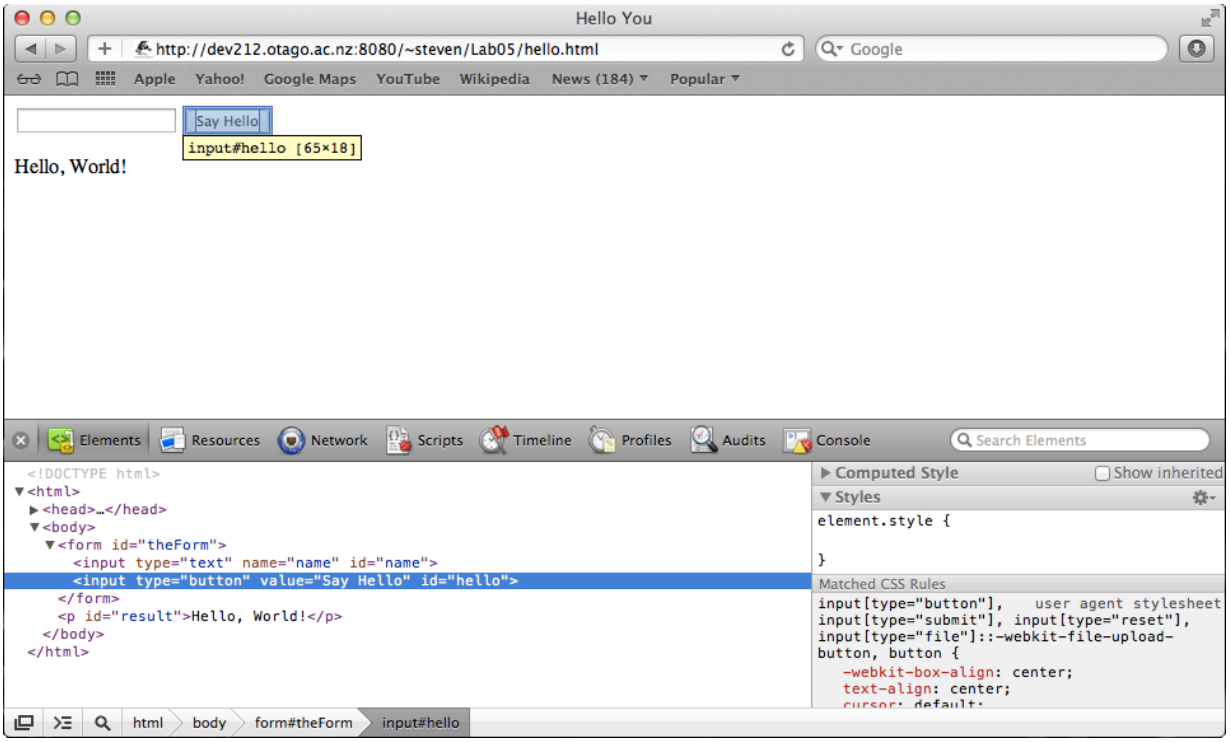


Figure 3.1: Developer Tools in Safari under OS X

- You can write information to the console:

```
console.log("My debug message");
```

Task 3.6: Experiment with these three methods of reporting debug information. What are the advantages and disadvantages of each?

3.5 Showing and Hiding Elements

Task 3.7: Let's return to the Classic Cinema example from earlier. Each film in the web site has an image, a title, and a number of other details. Using the following instructions, make it so that clicking on the film titles (the `<h3>` elements) will show or hide the image and other details.

To get started, we can try out some of the JavaScript commands in the console. This is a good way to experiment to figure out what you need to put in a script. Start by navigating to one of the category pages – in this example we'll look at the "Classics" page.

Let's start by getting a list of all of the films in the page. The `<div>` tags surrounding each film have `class="film"`, so we can use the following JavaScript statement to get the films:

```
document.getElementsByClassName("film");
```

Type this in the console, and you should get a list of films in square brackets. These indicate that this is an array of elements, and when we write the final script we're going to need to iterate over the array. For now, let's just get the first element of the array and assign it to a variable so that we can refer to it easily:

```
var theFilm = document.getElementsByClassName("film")[0];
```

The console shows the result of each statement – in this case you should see `undefined`. This does not mean that `theFilm` is undefined, but rather that the *result of the assignment* of a value to a variable has no value. To see what value a variable has, just enter it in the console:

```
theFilm;
```

Next we want to get the title of this particular film in order to attach an event to it. Again, we'll leave the actual event to the final script, but at the console we can get the title as the only `<h3>` element that is a child of `theFilm`:

```
var theTitle = theFilm.getElementsByTagName("h3")[0];
```

Note that when we get elements by tag name we get an array, even though we know there should only be one element returned. This is because you can't generally tell how many elements with a given tag (or class) there might be – in fact, it is good practice to check how many there before assuming that there are a particular number (or any at all). For now, however, we just grab the first element using the index `[0]`, and trust that it is the only one.

This element, which we have stored in `theTitle`, is what we'll attach an event to, and so it is the starting point for the event handler. So, given a reference to the title, we want to get access to the associated details. To do this, we need to think about the DOM a bit more. The DOM has a tree structure, and a part of the structure of this page is shown in Figure 3.2.

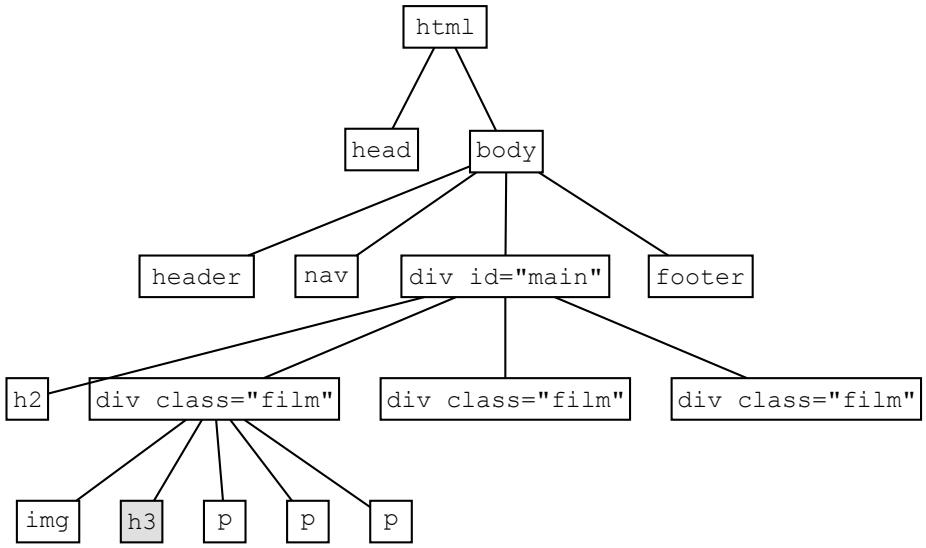


Figure 3.2: Part of the DOM tree structure for the Classic Films page. Given a reference to a level 3 heading element (shaded), we want to reach the related paragraph elements.

So, to get the `<p>` tags associated with a given `<h3>` tag we need to go up one level in the tree and then search for the relevant child nodes:

```
var paragraphs = theTitle.parentNode.getElementsByTagName("p");
```

Again, this returns a list which we'll need to iterate over to affect each element, but to hide the first paragraph you set the CSS display property to 'none':

```
paragraphs[0].style.display = "none";
```

Try that out in the browser, and then bring the paragraph back using

```
paragraphs[0].style.display = "block";
```

Of course, we want the display to toggle between the two states, so we use a conditional statement to switch between them:

```
if (paragraphs[0].style.display === "none") {
  paragraphs[0].style.display = "block";
} else {
  paragraphs[0].style.display = "none";
}
```

To enter multiple line commands in the console you need to press Option-Enter (Mac) or Shift-Enter (Windows), or you can just enter the whole statement on one (long) line. Now we can show or hide a single paragraph, and we can extend this to all of the paragraphs by iterating over all of the elements in the array.

Writing the Script

Now that you know the basic commands that will do what you want, it is time to put them together in a script. A skeleton for this follows the same pattern as the 'Hello' script:

```
function showHideDetails() {

}

function setup() {

}

if (document.getElementById) {
   window.onload = setup;
}
```

The setup function needs to loop over all the films and add an `onclick` event handler to their titles. For now we won't worry about the details of `showHideDetails`, but will just put in an alert so that we can see that it is being called.

```
function showHideDetails() {
  alert("Not implemented yet");
}

function setup() {
  var films, f, title;
  films = document.getElementsByClassName("film");
  for (f = 0; f < films.length; f+=1) {
    title = films[f].getElementsByTagName("h3")[0];
    title.onclick = showHideDetails;
  }
```

```
}
```

Task 3.8: Make a script called `showHide.js` with the code above in it. Link this to the category pages of the Classic Cinema site. Check that clicking on the film titles brings up the message box.

Before we get on to the details of `showHideDetails`, there is one minor thing to fix up. The cursor doesn't change as you move over the titles, which means that users won't know that they can click on them. You can fix this by adding the line

```
title.style.cursor = "pointer";
```

inside the for loop in `setup`.

All that remains is to full in the function `showHideDetails`. Pseudocode for this is as follows: This has a similar pattern to setup, except that the list of tags we loop over is in relation to the item clicked. A reference to the `<h3>` item that was clicked can be accessed via the variable `this`. For example, you can change the title colour of the clicked heading like this:

```
this.style.color = "red";
```

Likewise, we can use `this` like the variable `theTitle` we had in the console earlier to get to the paragraph and image elements. For example to find and iterate over the paragraphs we could do something like

```
function showHideDetails() {
  var paragraphs, p;
  paragraphs = this.parentNode.getElementsByTagName("p");
  for (p = 0; p < paragraphs.length; p+=1) {
    // Show or hide paragraphs[p]
  }
}
```

Task 3.9: Complete the function `showHideDetails` so that it switches between hidden and block display for the relevant image and paragraph elements.

3.6 JSLint and JSHint

JavaScript as a language was released on the world early in its development. The widespread use of JavaScript means that it is very hard to change the language, as removing any functionality would likely break a lot of websites. One of the results of this is that there are many parts of the language that can cause problems if not used carefully. Douglas Crockford, author of *JavaScript: The Good Parts* has provided a tool that checks against the rules and guidelines he proposes in his book. This is called JSLint, and is available online at http://www.jslint.com/.

By default JSLint is very strict about a number of things, but you can change some of the settings to suit your own preferences. For example, I usually set "Assume a browser" to true when writing JavaScript for web pages, and tolerate `for` statements. The `for` statement checking is there because in many situations it is better to use Array methods, but some things in JavaScript (like the result of `getElementsByTagName()`) look like arrays but aren't. The choice of settings can come down to personal preference, but the important thing is that you *understand the choices* you make. JSLint can't make you write good code, but it can help identify issues that you need to think about.

An alternative to JSLint is JSHint, http://www.jshint.com/. JSHint started as a fork of JSLint but with a community deciding on the options rather than just Douglas Crockford. It is generally less strict, but is also very configurable.

You can use JSLint or JSHint directly in PHPStorm, but going into Preferences → Languages & Frameworks → JavaScript → Code Quality Tools → JSLint *or* JSHint, and selecting 'Enable'. The default settings for JSHint are shown in Figure 3.3.

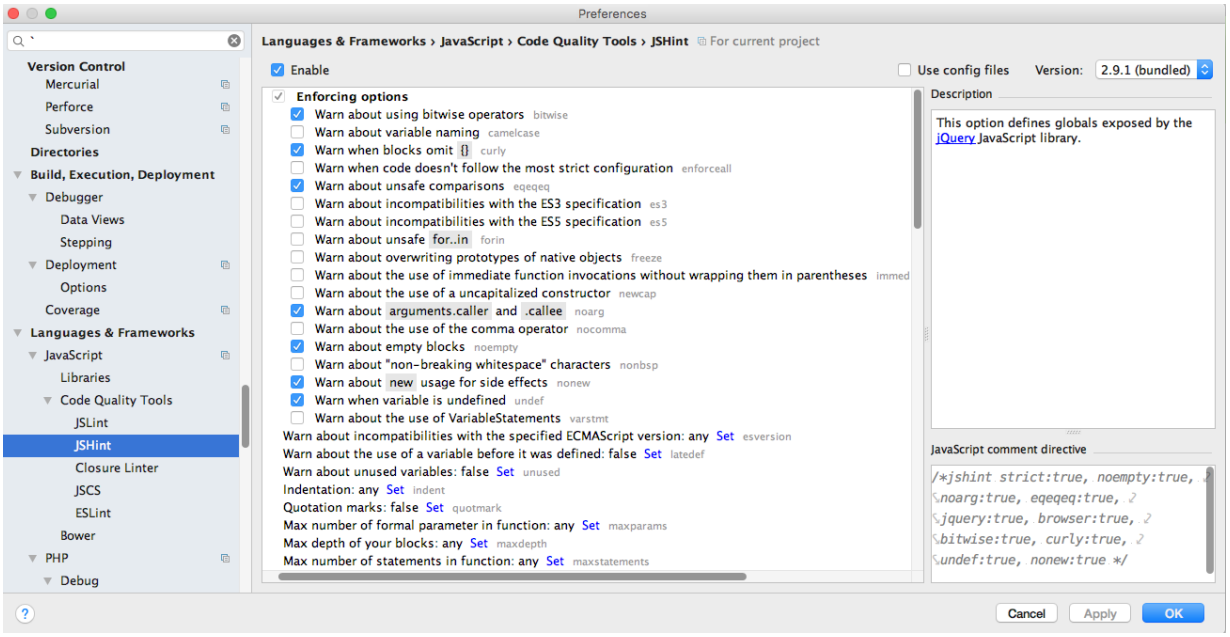


Figure 3.3: Setting up JSHint in PHPStorm.

Task 3.10: Use JSLint or JSHint to check the contents of your `showHide.js` script. You should correct any errors you get, or adjust the settings to tailor the checker to your personal style. *Don't, however, just adjust settings to avoid error messages*.

If you see warnings about 'Strict mode' or 'Missing "use strict" statements', then read on...

3.7 Strict Mode

Tools like JSLint and JSHint can help you check your code, but they don't prevent browsers from running code that does potentially harmful things. To overcome this, modern JavaScript interpreters support a *strict mode*, where various additional rules are enforced, such as raising an error when undeclared variables are used.

So how do you put your code in strict mode? It is quite simple, you can just put the line

```
"use strict";
```

at the top of your file, and all the code in that file will be in strict mode. That might look a little odd, since it is just a string which isn't assigned to a variable or doing anything, however modern JavaScript interpreters will recognise it as a directive to enter strict mode. This is also quite elegant since older interpreters will just ignore the string, and so won't break when they encounter strict code.

While the approach above is an easy way to put all of the code in a file in strict mode, it is often considered better style to declare each function to be strict (or not) separately. This gives more control when converting older code, and since functions are the basic building block of most JavaScript code,

they are a logical place to put the strict directives. Making a function strict is easy as well – you just add the directive at the start of the function:

```
function square(x) {
  "use strict";
   return x*x;
}
```

For the assignment: We've not yet covered all the JavaScript you will need for the assignment, but you can start to think about what sorts of behaviour you'd like to implement. What actions will trigger this behaviour?

Lab 4

JavaScript Arrays and Objects

In this lab we will look at how to represent complex data in JavaScript through arrays and objects. We'll use these to make a more dynamic front page for the Classic Cinema site.

Note: This lab is assessed, and is worth 1% of your final grade. Please make sure that you get a demonstrator or teaching fellow to mark it off as completed.

4.1 JavaScript Arrays

Like many other programming languages, JavaScript provides arrays to store lists of things. Arrays are indexed using square brackets, and created with the **new** keyword, which we'll see more of later in this lab. Here is a simple example of using an array in JavaScript:

```
// Create an empty array
var numbers, ix;

numbers = [];

// Some information to the array
numbers[0] = "Zero";
numbers[1] = "One";
numbers[2] = "Two";
numbers[3] = "Three";

// Display the contents of the array
for (ix = 0; ix < numbers.length; ix+=1) {
  alert(numbers[ix]);
}
```

As well as acting like a normal array, JavaScript arrays have functions associated with them that allow them to act like a stack, to search and sort them, and various other operations:

```
var item;

item = "Four";
numbers.push(item);
```

```
while (numbers.length > 0) {
  item = numbers.pop();
   alert(item);
}
```

Task 4.1: Enter the commands in the previous two snippets at the console. Remember that you will need to use Option-Enter or Shift-Enter to write multi-line commands. What is the value of `numbers` once all the commands have been run?

An Image Carousel

Now let's apply JavaScript arrays to revamp the front page of the Classic Cinema site. Instead of having all of the categories listed, we'll have a changing page which iterates over the different categories. For now we'll just have a changing image, but later in the lab you'll see how to add more complex information.

Task 4.2: Edit the HTML for `index.html` and replace the list on the front page which displays an image and link for each category with an empty `<div>`.

- Keep a copy of the list somewhere though, as you may want to refer to it.
- Give the `<div>` a unique `id` so that you can refer to it later.
- You should also create a new script and link it to the front page.

Next we'll start writing the script by making an array and fill it with the images we want to cycle through. We'll also need a counter to keep track of which image we're up to. We do this in the `setup` function that gets called on page load, and store them in variables with global scope. This means that they can be accessed by another function, `nextImage`, which will handle changing from one image to the next.

```
var imageList, imageIndex;

function nextImage() {
  // Need some code in here using imageList and imageIndex
}

function setup() {
  imageList = [];
   imageList.push("images/Metropolis.jpg");
   imageList.push("images/Plan_9_from_Outer_Space.jpg");
  imageList.push("images/Vertigo.jpg");
  imageIndex = 0;
}

if (document.getElementById) {
  window.onload = setup;
}
```

As it stands, this won't do anything, we need to fill in a couple of details. The first is the function `nextImage`, which needs to do the following:

- Use the current value of `imageIndex` to get the address to the next image to display

- Update the contents of the `div` you created earlier so that its `innerHTML` is a `<img>` element that displays this image.
- Add one to `imageIndex` so that the next call to `nextImage` uses the next image in the list.
- Make sure you've not gone off the end of the list – if you have, go back to the start

You may find it useful to be able to concatenate strings in JavaScript to achieve this. The `+` operator is used for string concatenation, so

```
var str = "A string" + " another string";
```

assigns the value "A string another string" to the variable `str`.

Once you've written this function, link the script to the `index.html` page of the Classic Cinema site. When you load page you should look in the developer tools 'Console' tab to make sure that no errors have been reported. If there are errors, the console will give a message and line number to help you identify and fix them. Once there are no errors in the console, you should be able to type

```
> nextImage();
```

in order to call this function and update the image on the front page. If you call it four times, you should loop back to the first image.

Finally we need to add a couple of lines to the `setup()` function. Firstly we call `nextImage` so that the contents of the front page are set up properly when the page is loaded. Secondly, we need to call `nextImage` every few seconds so that the content changes regularly. We can do this with a special event using the `setInterval` function. This function takes two parameters – a function to call, and a time in milliseconds. For example,

```
setInterval(myFunction, 3000}
```

would cause `myFunction` to be called every 3 seconds.

Task 4.3: Modify your `setup` function so that `nextImage` is called when the page is loaded, and every 2 seconds thereafter.

4.2 JavaScript Objects

Along with arrays, JavaScript has objects, like a lot of other programming languages. An object in JavaScript is essentially a collection of data and functions or methods. As with other programming languages, objects allow us to *encapsulate* state (data) and behaviour (methods) Also in common with languages like Java and Python is the idea of *inheritance*. JavaScript, however, uses prototype-based rather than class-based inheritance. For now, the distinction won't concern us too much, but in order to make the best use of objects and inheritance in JavaScript you should to understand the difference between the two.

Returning to the Classic Cinema site, we have replaced the list of categories on the front page with a rotating 'carousel' of images. This is a more dynamic page, but we've lost some functionality – in the static page, the images had associated text, and links through to the relevant category pages. We can wrap this information up in an object. The data elements of our object will be:

- The name of the category
- An image to represent the category
- A page to link to.

There are three ways we can create such an object in JavaScript. The first one is to create an empty object, and then to add the various data elements to it:

```
var movieCategory;
movieCategory = new Object();
movieCategory.title = "Classic Movies";
movieCategory.image = "images/Metropolis.jpg";
movieCategory.page = "classic.html";
```

The second way is a short-hand using curly braces, called *object literal* notation:

```
var movieCategory = {
  title: "Classic Movies",
   image: "images/Metropolis.jpg",
   page: "classic.html"
};
```

Finally, we can create a *object constructor* function which makes a new object:

```
var movieCategory;

function MovieCategory(title, image, page) {
  this.title = title;
   this.image = image;
   this.page = page;
}

movieCategory = new MovieCategory("Classic Movies",
                                  "images/Metropolis.jpg",
                                                  "classic.html");
```

The **this** keyword refers to the 'owner' of the function. By default, most global functions are owned by the global object `window`. When a function is called as a member of an object, however, **this** refers to the object that it is called on.

All three of these are useful in different circumstances. The ability to add elements to objects, as in the first example, lets you modify an existing object by adding or changing data elements or methods. The object literal notation is an easy way to create an object that you'll only use once, and JSLint favours using an empty object literal like this:

```
movieCategory = {};
```

to the **new** `Object()` syntax. Finally, object constructor functions are a useful way to make a number of different objects with the same structure.

As well as data elements, objects can contain functions or methods. For example, we might give our category objects a method to create the HTML we need to display them on the front page. These can be existing functions:

```
function someFunction() {
  // Code goes in here
}

movieCategory.makeHTML = someFunction;
```

Or can be defined directly using anonymous functions.

```
movieCategory.makeHTML = function () {
  // Code goes in here
}
```

Anonymous functions are commonly used in JavaScript. They are just like other functions, except that they don't have a name. In many languages this would not be particularly useful, but since JavaScript treats functions as 'first-class' entities, you can assign functions to variables or pass them to other functions. Anonymous functions are used when you want to make a function that is simply going to be assigned to a variable or passed to another function, and then never used again.

Thinking up names for these functions and making sure they don't conflict with each other or with other function or variable names gets harder as your code gets larger. These names also clutter your program with information that is never going to be used. Anonymous functions let us create single-use functions and use them without all these problems. They are a common feature of functional languages (like JavaScript) and are becoming more common in other languages, such as C++ and C#.

You can also use anonymous functions to define member functions directly inside a constructor like this:

```
function MovieCategory(title, image, page) {
  this.title = title;
   this.image = image;
   this.page = page;
  this.makeHTML = function() {
    // Details go in here
  }
}
```

The body of `makeHTML` will use the object's data elements to create an HTML string like this:

```
<img src='images/Metropolis.jpg'><figcaption>Classic Movies</figcaption>
```

For our purposes, we need to be able to create a set of objects which represent different categories, so a constructor function would be a good idea. Since the function to make HTML for a category is unlikely to be used without reference to a specific object, declaring it using an anonymous function is probably most appropriate.

Task 4.4: Write a function, `MovieCategory`, which creates an object to represent a category with the three data elements described above. Add a function `makeHTML` to this object which creates a string containing the HTML required to display the category on the front page.

If you put this function in a script which you link to the `index.html` page of the Classic Cinema site, you can test it in the console like this:

```
> var testCategory = new MovieCategory("Test thing", "path/image.jpg", "page.html");
undefined
> testCategory.makeHTML();
"<a href='page.html'><figure><img src='path/image.jpg'><figcaption>Test Thing
    </figcaption></figure></a>"
```

Task 4.5: Update your script to use a list of objects describing the three categories instead of a list of images. The content of the `<div>` you made on the front page should be the result of `makeHTML` instead of a simple image, but should still change every few seconds.

For the assignment: Arrays and objects are fundamental to structuring data in JavaScript programs. Think about how you could use them to represent the data stored in the XML files. What things are best stored as objects, and what as arrays?

Lab 5

JavaScript Closures

In this lab we will examine JavaScript scope and look at the concept of *closure*, where data and functions are bound together into a common context. Closure is an important concept in functional languages like JavaScript, and we'll look at one common use of it in JavaScript – creating modules that can be used to safely load multiple scripts in a single page.

5.1 Loading Multiple Scripts

In the next lab we'll be developing a script, `login.js` which will be included in all of the pages of the Classic Cinema site. However, there are already several scripts included in the site. The front page has a script that cycles through the categories to display, and each category page has a script which shows and hides the details of items when you click on a movie's title. Including more than one script isn't a problem in principle – you just use multiple script tags like this:

```
...
<script src="script1.js"></script>
<script src="script2.js"></script>
...
```

However, you do run into problems when functions or other elements are defined in more than one script. Most of the scripts you've developed so far have something like the following:

```
function setup() {
  ...
}

if (document.getElementById) {
   window.onload = setup;
}
```

This is OK for a single script, but suppose both `script1.js` and `script2.js` have this form, and we include them both as described above. The browser first loads up `script1.js`, gets a definition for the function `setup`, and assigns this to the `window.onload` event. So far, so good.

It then loads `script2.js`, and finds a new definition for `setup`. This is perfectly valid – functions are 'first-class citizens' in JavaScript so you can assign new values to them freely. However, it overwrites the definition of `setup` from `script1.js`, and then assigns `setup` to `window.onload` again. This has the effect of calling the new version of `setup` when the document finishes loading, but loses the one from `script1.js`.

5.2 JavaScript Scope and Closure

What we need to do is to control the *scope* of the variables that we introduce. If several different scripts define a `setup` function, we need to stop them from interferring and know which one we're talking about. The only construct that defines a new scope in JavaScript is the function. One example of this is the object constructor function syntax, which makes use of a function to create a new object, and can be used to make information (data or functions) public or private like this:

```
function Constructor() {

   // Private members
   var privateData = "secret";

   function privateFunction() {
      //...
   }

   // Public members
   this.publicData = "available";

   this.publicFunction = function() {
      //...
   }
}

var myObj = new Constructor();
myObj.publicData = "changed"; // OK - public member
myObj.privateFunction();      // Not OK - private member
```

This combination of a function and an environment of variables is called a *closure*, and is an important aspect of many functional languages, including JavaScript. Closures have many uses, but essentially they allow us to bind together related data and functions into a coherrent, and contained, unit.

JavaScript Modules

One common use of closures in JavaScript is the definition of a *module*. The first thing that this does is to minimise the number of functions and variables we put into the global namespace. Having lots of different variables in global scope is an issue because it leads to a high risk of name collisions – if two scripts define functions or variables with the same name, one will over-write the other. The JavaScript module pattern uses a closure to achieve this:

```
var MyModule = (function() {
   var pub = {};

   // Private data and functions
   var hiddenValue = "secret";
   function hiddenFunction() {
      //...
   }

   // Public data and functions
   pub.value = "available";
```

```
   pub.func = function() {
      // can use hiddenValue and hiddenFunction here if you want
   }

   return pub;
}());
```

The anonymous function here is used to create a private scope for a module. Any variables or methods that we want to expose publicly are put into an object, which I've called `pub` to reflect this. This object is returned from the function, and the `()` around the whole function are used to assign the result (the object `pub`) to the variable `MyModule`.

The end result of all this is that the only thing put into the global scope is the variable `MyModule`. Through this we can access the public data as `MyModule.value` and `MyModule.func`. The private data and functions aren't visible externally via `MyModule`, but can be used within the public functions.

As an example, here is how we could re-write the movie category carousel using the module pattern. In this case I have chosen to keep as much information as possible private – in fact, only the `setup` function needs to be exposed publicly.

```
var Carousel = (function(){
   var pub = {};

   var categoryList = [];
   var categoryIndex = 0;

   function nextCategory() {
      var element = document.getElementById("carousel");
      element.innerHTML = categoryList[categoryIndex].makeHTML();
      categoryIndex += 1;
      if (categoryIndex >= categoryList.length) {
         categoryIndex = 0;
      }
   }

   function MovieCategory(title, image, page) {
      this.title = title;
      this.image = image;
      this.page = page;

      this.makeHTML = function() {
         return "<a href='" + this.page + "'><figure>" +
             "<img src='" + this.image + "'>" +
             "<figcaption>" + this.title + "</figcaption>" +
             "</figure></a>";
      };
   }


   pub.setup = function() {
      categoryList.push(new MovieCategory("Classic",
                "images/Metropolis.jpg", "classic.html"));
      categoryList.push(new MovieCategory("Science Fiction",
```

```
            "images/Plan_9_from_Outer_Space.jpg", "scifi.html"));
      categoryList.push(new MovieCategory("Alfred Hitchcock",
            "images/Vertigo.jpg", "hitchcock.html"));
      nextCategory();
      setInterval(nextCategory, 2000);
   };

   return pub;
}());


if (document.getElementById) {
   window.onload = Carousel.setup;
}
```

Adding Event Handlers

The second thing that we need to be able to do when loading multiple scripts is to add functions to events without overwriting any existing event handlers. Suppose we re-write `showHide.js` as a module, so the setup function becomes `ShowHide.setup()`. We want to call this function when the page is loaded, but there might be some other functions to be called then as well. In most browsers this is done using the `addEventListener` function:

```
window.addEventListener('load', ShowHide.setup);
```

However, in older versions of Internet Explorer, you would use:

```
window.attachEvent('onload', ShowHide.setup);
```

In order to work across as many browsers as possible, you can check which behaviour is supported and use that:

```
if (window.addEventListener) {
   window.addEventListener('load', MovieCategories.setup);
} else if (window.attachEvent) {
   window.attachEvent('onload', MovieCategories.setup);
} else {
   alert("Could not attach 'MovieCategories.setup' to the 'window.onload' event");
}
```

Task 5.1: Modify your existing scripts for the Classic Cinema site to use the module pattern, and to allow multiple scripts to safely add multiple functions to the `window.onload` event.

For the assignment: Closures and modules are a useful way to organise your code. Think how the functionality you may need for the assignment can be broken down into parts. These parts might become separate modules in your JavaScript code.

Lab 6

JavaScript Cookies

In this lab we'll use Cookies to let the user add items to a shopping cart. Cookies allow us to store information as the user navigates around a website, or between visits to the site.

Note: This lab is assessed, and is worth 1% of your final grade. Please make sure that you get a demonstrator or teaching fellow to mark it off as completed.

6.1 JavaScript and Cookies

Cookies allow us to store information on the client side. They are essentially a text file in which the browser can store name-value pairs. This information is associated with particular web sites, and the browser is responsible for protecting cookies from unauthorised access.

In JavaScript you can access cookies via the `document.cookie` global variable. This stores cookies as a string, in name-value pairs separated by semi-colons. So if we had a cookie called 'username' with a value 'someuser' and one called 'email' with a value 'someuser@somedomain.com', the value of `document.cookie` would be

```
username=someuser; email=someuser@somedomain.com
```

Assigning values to `document.cookie` is a little different to assignment to other variables. You can assign multiple name-value pairs, and they get added to the cookie value. It is only when you re-assign the same name that the value is overwritten.

When you assign a value to a cookie, you can also assign other information, like an expiry time and a path on the server that restricts what files can access the cookie. If no expiry time is given, the cookie lasts until the browser is closed and then is deleted. If no path is given then all pages in the same domain as the current page can see the cookie.

Putting all this together, we can create some helper functions for cookies. These are adapted from http://www.quirksmode.org/js/cookies.html, but have been put into a module so that they are easier to reuse. See the QuirksMode.org page for more details about how these functions work.

```
var Cookie = (function () {

    var pub = {};

    pub.set = function (name, value, hours) {
        var date, expires;
```

```
        if (hours) {
            date = new Date();
            date.setHours(date.getHours() + hours);
            expires = "; expires=" + date.toGMTString();
        } else {
            expires = "";
        }
        document.cookie = name + "=" + value + expires + "; path=/";
    };

    pub.get = function (name) {
        var nameEq, cookies, cookie, i;
        nameEq = name + "=";
        cookies = document.cookie.split(";");
        for (i = 0; i < cookies.length; i += 1) {
            cookie = cookies[i].trim();
            if (cookie.indexOf(nameEq) === 0) {
                return cookie.substring(nameEq.length, cookie.length);
            }
        }
        return null;
    };

    pub.clear = function (name) {
        pub.set(name, "", -1);
    };

    return pub;
}());
```

Task 6.1: Make a file called `cookies.js` and include it from an HTML page. Experiment with the `Cookie.get`, `Cookie.set`, and `Cookie.clear` functions. If you want to experiment with expiry times, it might be easier to set them in minutes or seconds rather than hours.

Special Characters, Objects, and Cookies

The cookie functions as given may not work in all cases. What if the name or value of a cookie contains characters like `=` or `;`? What if we want to store complex data like an object or an array in a cookie?

To avoid problems with unusual characters in cookies, we can URI encode names and values when we store them, and then decode them when we retrieve them. URI encoding will replace most characters other than letters and numbers with escape codes starting with `%`. For example, `=` is encoded as `%3D`, and `;` as `%3B`. The numbers after the percent sign are the ASCII code for the character as a hexadecimal value. The JavaScript functions `encodeURIComponent` and `decodeURIComponent` that can be used for this.

Task 6.2: Modify the `Cookie` functions so that they use URI encoding to store and retrieve cookies.

To store objects or arrays in a cookie, we need to convert them to strings. One way to do this is with the JavaScript Simple Object Notation, or JSON. We'll see more about JSON later in the course, but it provides a way to convert complex data structures to and from strings. Note that JSON doesn't preserve

all of the details about an object. It can only record data, so any functions or methods will be lost. To convert some object, `obj`, to a string you can use `JSON.stringify(obj)`. To convert a string, `str`, to an object you can use `JSON.parse(str)`.

Task 6.3: In the console, create an object with several fields, for example:

```
var person = {};
person.firstname = "Tom";
person.lastname = "Sawyer";
person.age = 16;
```

Then, still in the console, `JSON.stringify` the object to a string. Save this string in a new variable and convert it back into an object.

6.2 A Classic Cinema Shopping Cart

At the end of the HTML for each film on the Classic Cinema site looks like this:

```
<section class="film">
  ...
  <p>
    $<span class="price">13.99</span>
    <input type="button" value="Add to Cart" class="buy">
  </p>
</section>
```

Note that the price and button are identified by `class`, not by `id`. This means that we will be able to write one piece of code to handle buying all of the elements.

Next we need to call a JavaScript function when the "Add to Cart" buttons are pressed. You can get a list of all the buttons with `document.getElementsByClassName(...)`. You can then iterate over this list to add a `onclick` event to each one.

Task 6.4: Make a new JavaScript file called `cart.js`, link it to the three category pages, and add an `onclick` event to each of the "Add to Cart" buttons. For now the event can just be a stub that raises an alert, but you should put your code inside a module so that it works well with other functions on the pages.

Next, let's start to make these buttons functional. What we want to do is to store a list of movies in the cart, and each entry in the list will need to record the title of the movie and the price. This suggests an array of objects, but for now we'll worry about making one object in the callback for the "Add to Cart" buttons.

Task 6.5: Change the function called when the "Add to Cart" buttons are pushed so that it creates an object which has two data fields – a `title` and a `price`. You should use DOM scripting to extract the correct values for these fields from the web page. Remember that the variable `this` inside the callback will refer to the `<input>` element that was clicked.

Making an object inside a callback can be a bit tricky to debug, since you can't see its value easily. What you can do is use JSON to convert the object to a meaningful string, and raise an alert like this:

```
alert(JSON.stringify(myObject));
```

Which should display something like

```
{"title":"King Kong (1933)", "price":"13.99}
```

Alerts are a useful way to monitor the progress of your code, but it can get annoying clicking "OK" to make them go away all the time. An alternative is to send messages to the console in the browser, which you can do with the `console.log` function.

Now we need to keep an array of items in a cookie. The basic approach is as follows:

- Ask for the list of items stored in the cookie
- If it is found
  - Add the new item to the list
- Otherwise
  - Make a new list with just the new item in it
- Store the updated (or new) list in the cookie

Remember that to store complex objects in cookies you can use the `JSON.stringify` and `JSON.parse` functions.

Task 6.6: Update your code so that items are added to an array which is stored in a cookie. You should be able to add items from different pages to the cart, and check that it is working by inspecting the value of `document.cookie` in the console.

Now that we have a shopping cart, we need to be able to view it.

Task 6.7: Make a new page which displays the shopping cart contents (or a relevant message if it is empty). You should list the items along with their prices, and give the total cost of the items in the cart. To add up the total price you'll need to convert strings to numbers. The JavaScript function `parseFloat` can do this.

Lab 7

Web Storage and IndexedDB

In this lab you will look at other technologies that you could have used in the development of the Classic Cinema site.

7.1 Web Storage

Currently Classic Cinema uses cookies for client-side storage. As you know the interface is a little awkward and in fact you used the Cookies.js closure to simplify using them. Apart from being awkward cookies have other drawbacks: they are limited to 4kb and they are included with every server request. To address these issues two very similar technologies were included in the HTML5 specification:

- window.localStorage – stores data with no expiration date
- window.sessionStorage – stores data for one session (data is lost when the browser tab is closed)

Task 7.1: Update Classic Cinema to use Web Storage rather than cookies. Think carefully about whether you should use localStorage or sessionStorage. See `https://www.w3.org/TR/webstorage/#storage` for the api

The full API has more information but here is the short version:

- *length* attribute returns the number of key/value pairs.
- *getItem(key)* returns the current value associated with the given key or null.
- *setItem(key, value)* sets item key to value value.
- *removeItem(key)* the key/value pair with the given key is removed from the list associated with the object.
- *clear()* causes the list associated with the object to be emptied of all key/value pairs.

In use you would expect to see lines of JavaScript similar to:

```
window.localStorage.setItem('myKey', 'myValue');
```

You can of course enter lines like the one above into the console to become familiar with the API. The developer tools supply an easy way to view the contents of all sorts of things.

When you are familiar with how Web Storage works update your Classic Cinema JavaScript files so that they use Web Storage rather than cookies.

When you are finished you will be able to remove Cookies.js from Classic Cinema and everything will work just the same.

7.2 IndexedDB

IndexedDB is the replacement for Web SQL Database which is now deprecated by the W3C. IndexedDB is a bit more complicated than Web Storage and so we won't convert Classic Cinema to using it however you will develop a good chunk of the closure that implements a indexedDB solution for Classic Cinema.

Task 7.2: Start a new JavaScript file called ClassicCinemaIDB.js and add the following code:

I used Chrome to develop this and I found that not only deleting the database but also quitting Chrome and restarting it between code updates was the most reliable way to develop.

```
var ClassicCinemaIDB = (function () {
    /* A closure to demonstrate basic use of indexeddb
     * Adapted from
         https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Using_IndexedDB
     * Nick Meek 2019
     */
    'use strict';

    //if indexeddb is not supported then no use continuing
    if (!window.indexedDB) {
        window.alert("Your browser doesn't support a stable version of IndexedDB.");
        return;
    }
    var pub = {}; //usual public interface
    var db;
    const dbName = "ClassicCinema";

      //Some movie data to get us started
    const movieData = [
        {
            title: "Gone With the Wind (1939)",
            director: ["Victor Fleming", "George Cukor", "Sam Wood"],
            starring: ["Clark Gable", "Vivien Leigh", "Leslie Howard", "Olivia de
                Havilland", "Hattie McDaniel"],
            comments: "An epic historical romance and winner of 8 Academy Awards
                (from 13 nominations).",
            price: 13.99
        },
        {
            title: "The Jazz Singer (1927)",
            director: ["Alan Crosland"],
```

```
        starring: ["Al Jolson", "May McAvoy", "Warner Oland", "Cantor
            Rosenblatt"],
        comments: "The first feature length 'talkie', The Jazz Singer is a piece
            of cinematic history",
        price: 13.99
    },
    {
        title: "Metropolis (1927)",
        director: ["Fritz Lang"],
        starring: ["Alfred Abel", "Brigitte Helm", "Gustav Frohlich", "Rudolf
            Klein-Rogge"],
        comments: "A lovingly restored version of Fritz Lang's dystopian
            masterpiece, one of the great achievements of the silent era.",
        price: 19.99
    }
];

    var request = indexedDB.open(dbName, 1); // Request to open a db

//an error handler for the request
request.onerror = function (event) {
    alert("Opening db error " + request.error);
};

//a success handler
request.onsuccess = function (event) {
    alert("Database successfully created.")
}

//This function runs before onsuccess, if needed i.e.
// no db exists or
// version number is higher than existing db
request.onupgradeneeded = function (event) {
    db = event.target.result;

    // Create an objectStore to hold information about our movies. We're
    // going to use "title" as our key path because it's guaranteed to be
    // unique.
    var movieObjectStore = db.createObjectStore("movies", {keyPath: "title"});

    // Use transaction oncomplete to make sure the objectStore creation is
    // finished before adding data into it.
    movieObjectStore.transaction.oncomplete = function (event) {
        // Store values in the newly created objectStore.
        var customerObjectStoreTransaction = db.transaction("movies",
            "readwrite").objectStore("movies");
        movieData.forEach(function (m) {
            customerObjectStoreTransaction.add(m);
        });
    };
```

```
    };

        return pub;

}());
```

Create an HTML page (`IDBTest.html`) and place a link to `ClassicCinemaIDB.js` in the `<head>` section.

Open `IDBTest.html` in a browser (screenshots are from Chrome Figure 7.1, Figure 7.2). If there are any errors they will be shown in the console. If all is well then the Application tab has useful information.

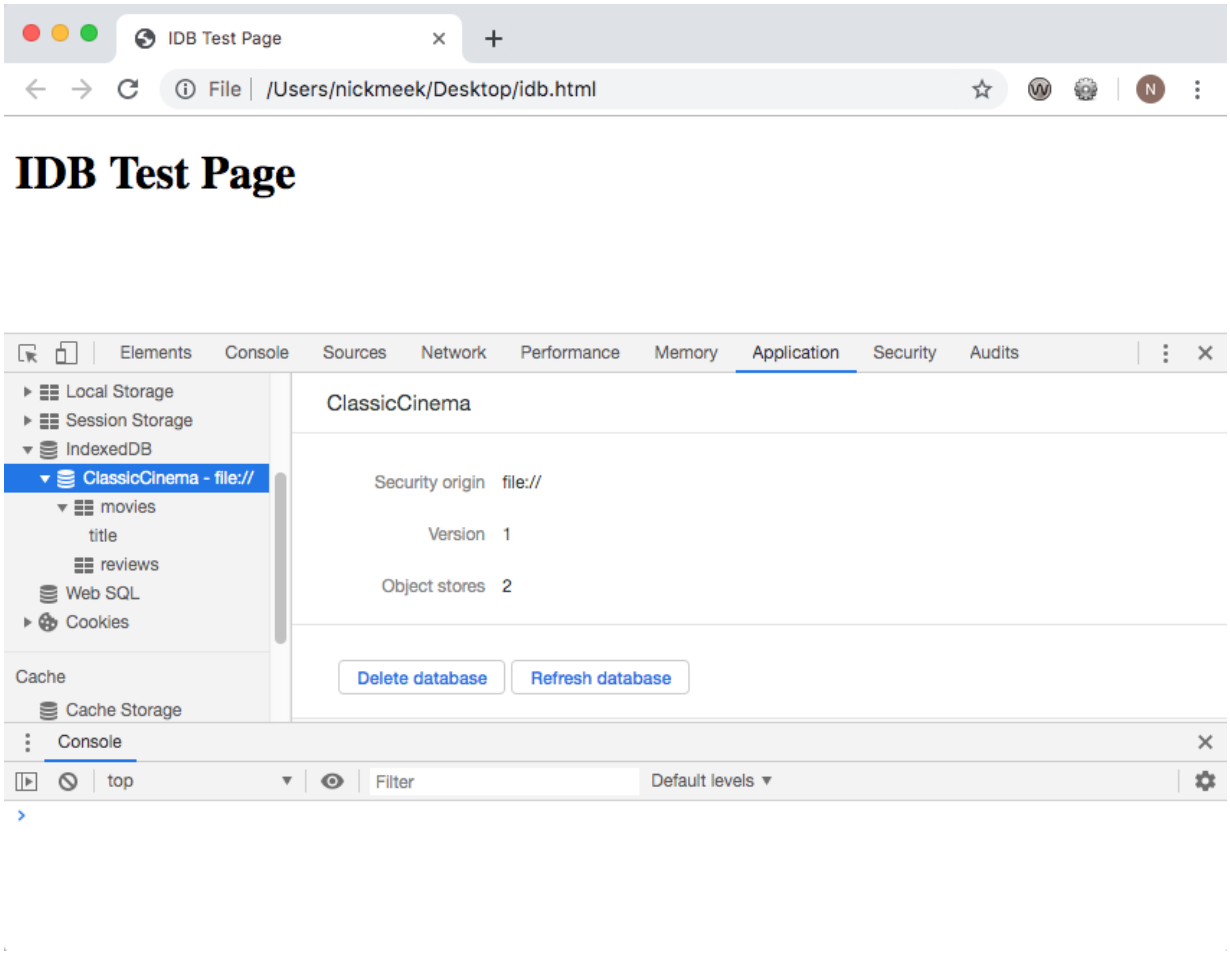


Figure 7.1: Chrome showing ClassicCinema db.

Note the line:

```
var customerObjectStoreTransaction = db.transaction("movies",
    "readwrite").objectStore("movies");
```

This starts a transaction and then returns a reference to the appropriate object store. You will need to do this quite often so encapsulating this in a function seems like a good idea.

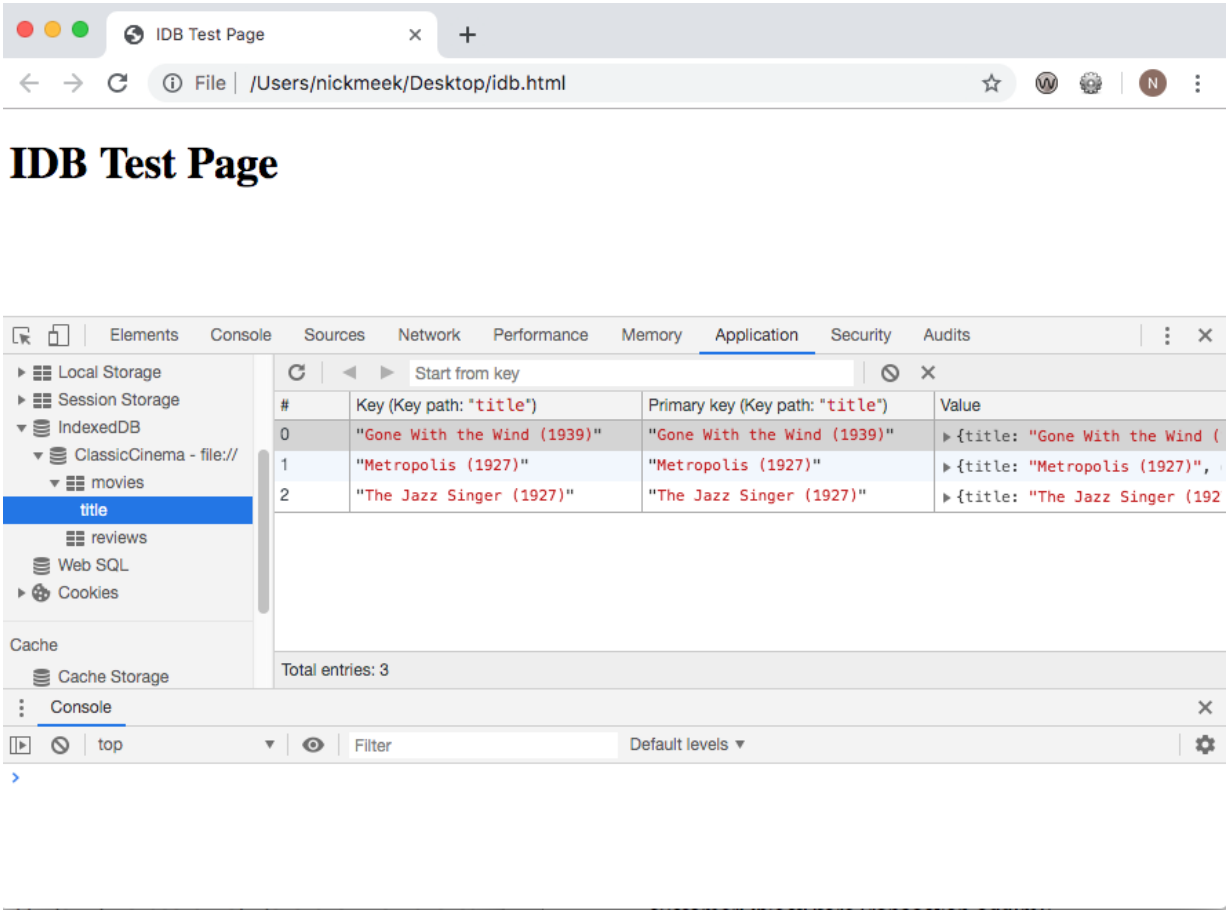


Figure 7.2: Chrome showing ClassicCinema db.

Task 7.3: Add the following to your closure and then update the code to use the function.

```
/**
 * Utility function to start a transaction on the specified objectStore(s)
 * and return the object store.
 * @param {string} store_name
 * @param {string} mode either "readonly" or "readwrite"
 * @return objectStore
 */
function getObjectStore(store_name, mode) {
    var tx = db.transaction(store_name, mode);
    return tx.objectStore(store_name);
}
```

Task 7.4: Write a function that displays the details of a specified movie.

To achieve this you need to do two things:

- Create an index on the movies object store so that you can directly access any movie given its title.
- Write a function that utilises that index to retrieve and display the information.

Add the following to the `request.onupgradeneeded` function:

```
// Create an index to search movies by title. We want to ensure that
// no two movies have the same title, so use a unique index.
movieObjectStore.createIndex("title", "title", {unique: true});
```

It creates an index called 'title' of the title properties of the objects in the store.
Add the following function to your JavaScript:

```
/**
 * Displays the title, price and rating of a specified movie
 * @param {string} title The title of the movie you want to find.
 */

pub.getMovieInfo = function (title) {
    var store = getObjectStore("movies");
    var request = store.get(title);
    //error handler
    request.onerror = function (e) {
        alert("getMovieInfo error " + request.error);
    };
    //success handler
    request.onsuccess = function (e) {
        alert(request.result.title + "\n" + request.result.price + "\n" +
            request.result.director);
    };
};
```

Test your code by calling the new function from the console (Figure 7.3).

Task 7.5: Add a function to add a movie to the movies object store.

Add the following function to your code.

```
/**
 * Adds a movie to the objectStore
 * @param {object} movieData The object you want added to the store.
 * The movieData object must take the following form:
 * title: {string}
 * director: [<string>]
 * starring: [<string>]
 * comments: {string}
 * price: {float}
 */
pub.addMovie = function (movieData) {
//start the transaction and get the store
    var store = getObjectStore("movies", "readwrite");
    var request = store.add(movieData);
```

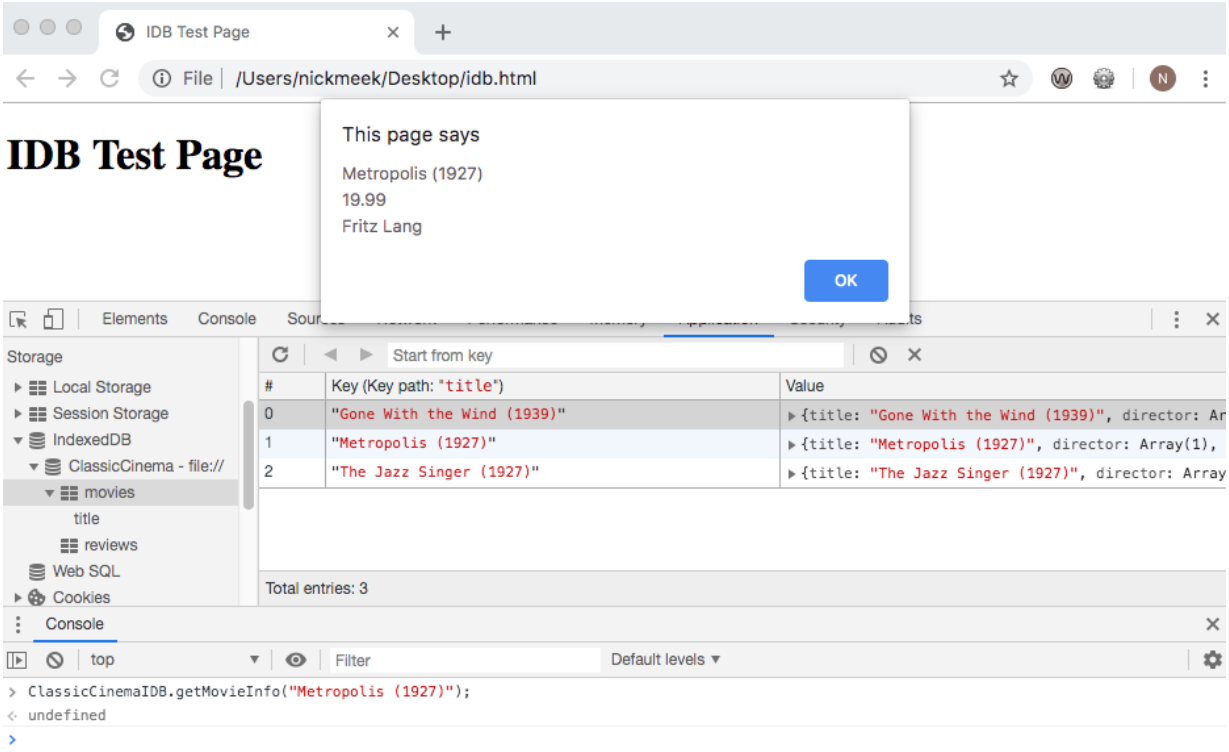


Figure 7.3: The getMovieInfo function working correctly

```
        //success handler
        request.onsuccess = function (event) {
            alert("movie Added");
        };

        //error handler
        request.onerror = function (event) {
            alert("addMovie error", this.error);
        }
    };
```

Test that the code works in the console (Figure 7.4, Figure 7.5).

Task 7.6: Create and object store to hold reviews and add a function to add reviews to it.

First create the object store, remember you can only create object stores in the request.onupgradeneeded function:

```
        // Create an objectStore to hold reviews We're
        // going to use a key generator to make our keys; it's guaranteed to be
        // unique.
```

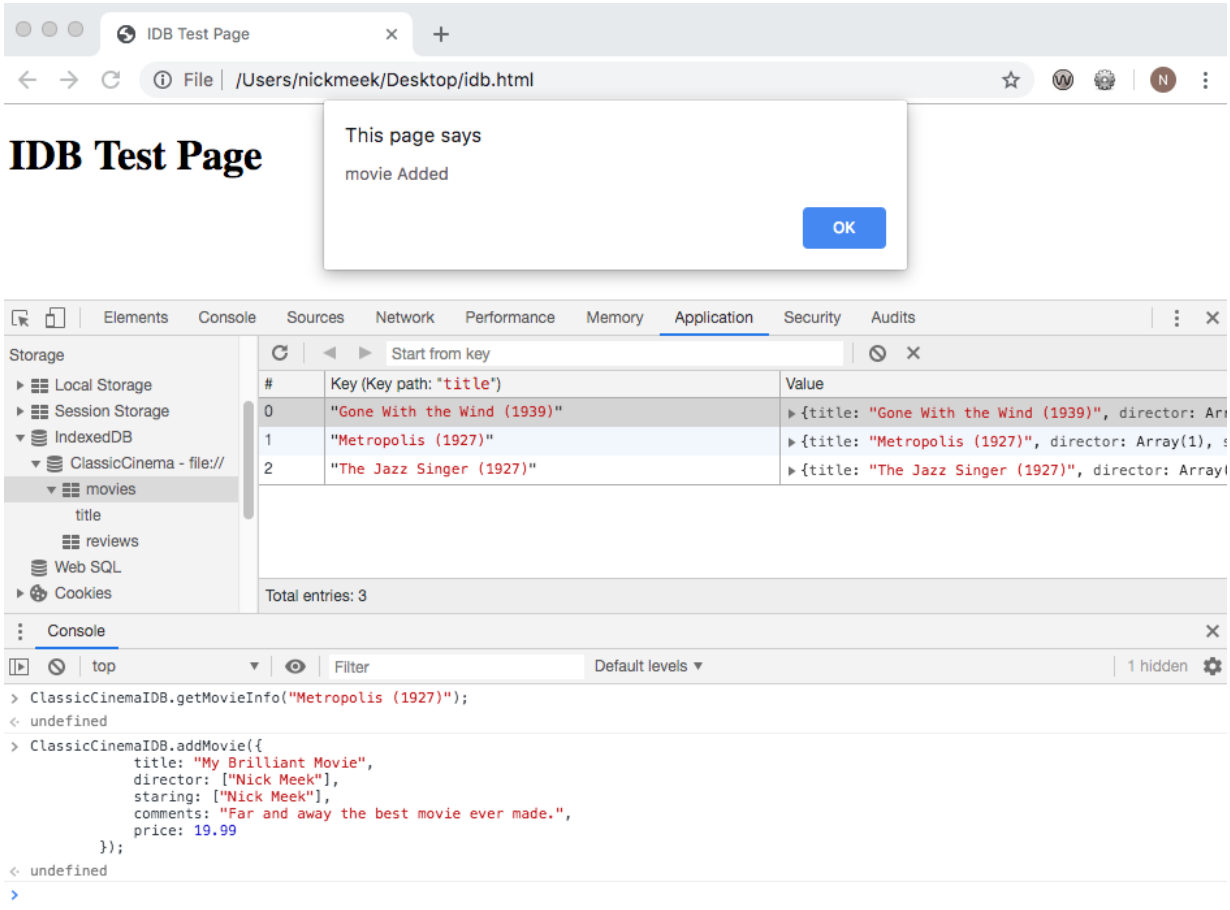


Figure 7.4: Adding a movie to the object store

```
    var reviewObjectStore = db.createObjectStore("reviews", {autoIncrement:
        true});
```

Then add the function to add items to it:

```
/**
 * Adds a review to the objectStore
 * @param {object} review The object you want added to the store.
 * The review object must take the following form:
 * user: {string}
 * title: {string}
 * review: {integer}
 */

pub.addReview = function (review) {
    var store = getObjectStore("reviews", "readwrite")
    var request = store.add(review);
```

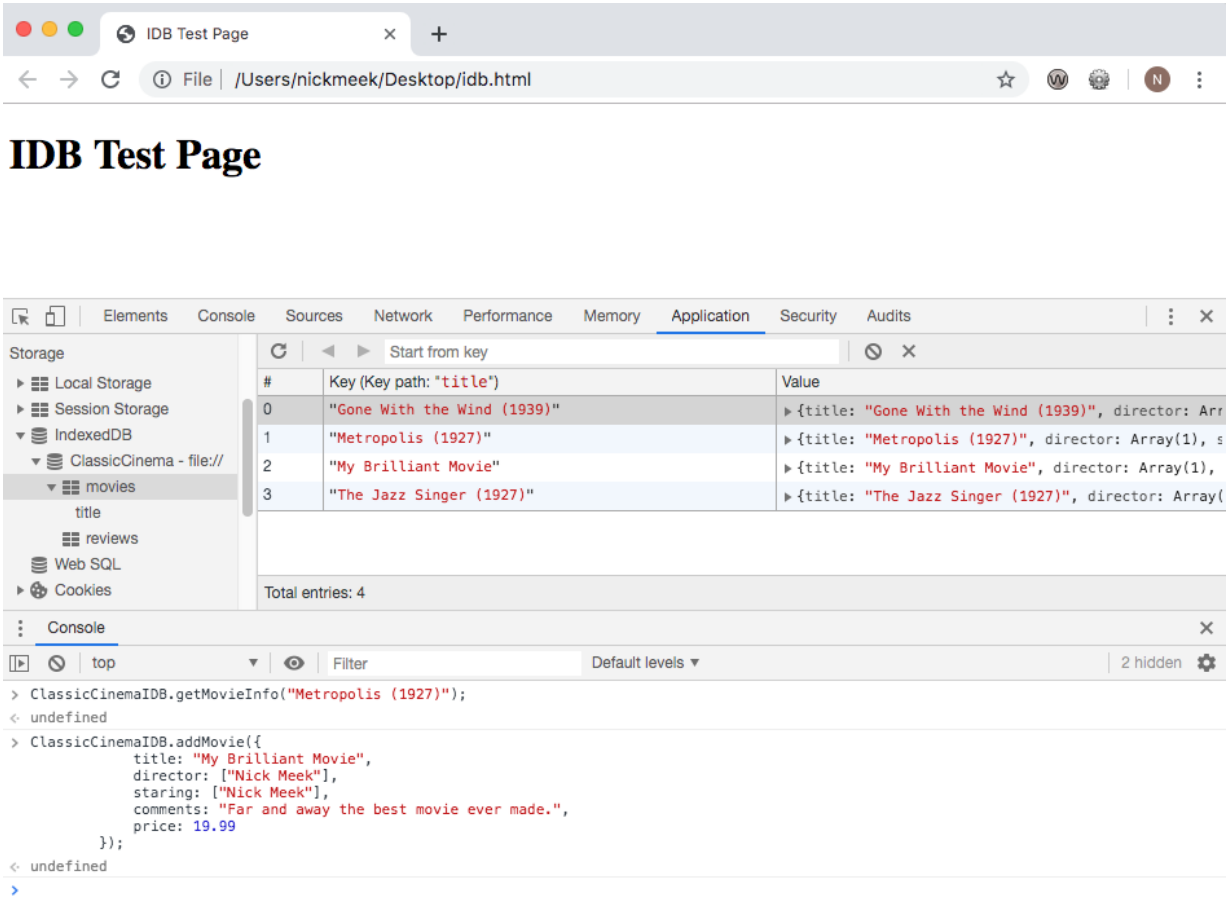


Figure 7.5: Adding a movie to the object store

```
        //success handler
        request.onsuccess = function (event) {
            alert("review Added");
        };

        //error handler
        request.onerror = function (event) {
            alert("addReview error", this.error);
        }
    };
```

Task 7.7: Write a function to display all the reviews for a particular movie.

Hint: Add a few reviews from an array as you did with movies to make testing easier.
To begin with you will need to create an index:

```
        // Create an index to search reviews by user. We may have duplicates
```

```
        // so we can't use a unique index.
        //reviewObjectStore.createIndex("user", "user", { unique: false });
```

Now you will need to write a function that does the finding and displaying. Read about the 'cursor' in the documentation.

Lab 8

JavaScript Form Processing

In this lab we look at how JavaScript can be used to process information entered from a form. We'll add a dummy checkout facility to the shopping cart developed in the last lab, including some basic form validation.

Note: This lab is assessed, and is worth 1% of your final grade. Please make sure that you get a demonstrator or teaching fellow to mark it off as completed.

Most web applications are driven by information entered by the user through HTML forms. This information often needs to be validated, and there are several ways in which this can be done. The three main forms of form validation are:

- HTML validation, which has been much improved in HTML5.
- Validation using JavaScript, which can occur when the user submits the form, or as they enter text.
- Server-side validation, which we will look at later in the course.

The first two forms of validation are fairly easily bypassed, so cannot be relied upon to prevent harmful data from being submitted. They can, however, greatly enhance the user's experience by providing prompt and helpful guidance.

Task 8.1: Add a form to the shopping cart page. This form should not be shown when the cart is empty, and should have fields for delivery information and credit card details. Sample HTML for this form is available on Blackboard – note that the form itself, as well as each element inside the form, has an `id` for easy reference from JavaScript.

This form contains some basic HTML validation. Several of the fields are marked as `required`, and most browsers will use this to check that the appropriate parts of the form have been filled in. Some fields also have a `maxlength`, which stops users from entering too much data. Finally, the `deliveryEmail` input element has `type="email"`, which is one of several new input elements which are available in HTML5. There is also a `type="number"` option, which could have been used for fields like the postcode and credit card number. However, this has an issue with leading zeros – 0123 is a valid New Zealand postcode, but as a 'number' can be converted to 123, which is not.

Task 8.2: Experiment with the HTML5 validation, and what it accepts for the various fields.

HTML5 validation is easy, but it can be a bit limited. Rules for validation can become quite complex, so it is often desirable to add some custom validation routines. Also, browser support for HTML5 validation is not quite standardised, so we can use JavaScript as an additional check.

Task 8.3: Before we get started with JavaScript validation add the attribute `novalidate` to the checkout form tag. This will prevent HTML5 validation, which will make it easier to test your JavaScript validation.

To see how we can validate form information in JavaScript, look at the file `sampleValidator.js`, which can also be found on Blackboard. There is a fair bit of code here, but the general module structure should be familiar. It is easiest to understand this by starting from the bottom.

First we have the usual code to setup some functionality. In this case the `setup` function attaches a function (`validateCheckout`) to our form's `onsubmit` event. This function will be called when the form is submitted, and if it returns `false` (or another 'falsey' value) then the form will not be processed further. If the function returns `true` (or another 'truthy' value), the form will be passed on to whatever `action` is specified in the form's HTML.

The `validateCheckout` function works by keeping an array of error messages, which is initially empty. For each field of the form, a separate function is called to check its value. The use of individual functions makes the code easier to understand, since each part (function) is kept quite small, and also facilitates code reuse. This pattern is repeated, with basic checks (like checking that a field is not empty) getting their own functions (`checkNotEmpty` in this case).

The checks that are made by the sample code are:

- The card validation date must be in the future.
- A credit card number and verification code must be provided, and consist only of digits.
- The credit card number and verification code must meet certain rules, depending on the card type:
  - For American Express cards: The credit card number must be 15 digits long and start with a '3'; the validation code must be 4 digits long.
  - For MasterCard cards: The credit card number must be 16 digits long and start with a '5'; the validation code must be 3 digits long.
  - For Visa cards: The credit card number must be 16 digits long and start with a '4'; the validation code must be 3 digits long.

Each of the smaller validation functions takes the `messages` array as a parameter. Since arrays in JavaScript are objects, they are passed by *reference*. This allows functions like `checkCreditCardNumber` to add messages to the array, and these changes will be seen in `validateCheckout`.

Finally, `validateCheckout` prints the error messages to the console and returns true or false depending on whether there are any messages or not. If there are no messages, then the form is OK and the function returns `true`. If there are messages, then there is some error in the input, and so `false` is returned which prevents the form from being submitted.

Task 8.4: Have a look through the sample validation functions, and ask about anything that you don't understand. Add the validation routines from `sampleValidator.js` to your checkout page. Check that the credit card validation routines are working OK.

8.1 Error Reporting

At the moment, the error messages are logged to the console as an array. This is not particularly useful, so we will display a list of messages on the checkout page. To do this you will need to do three things:

- Create an HTML element to display a list of messages, and give it an ID so that you can refer to it easily from JavaScript.
- Write a function in the checkout validation script that will take a list of messages and display them in the element you just created.
- Call this function if there are any messages to display at the end of the `validateCheckout` function.

Task 8.5: Follow these steps to add error reporting to the Checkout page. You should think carefully about what sort of HTML elements to use, when and where to display errors, and how to make them clear to the user.

8.2 Form Checking and Regular Expressions

Now let's look at the parts of the form concerning delivery details. This currently has no JavaScript validation, and we will add some based on the following rules:

- All fields must be filled out, except the second line of the address which is optional.
- The postcode must consist of exactly 4 digits. Note that '0410' is a valid New Zealand postcode (for Kaitaia), but '410' is not.
- Proper email validation is very complex, but we will use the following simplified rules:
  - Emails have the form `name@domain`
  - The `name` and `domain` components consist of blocks containing the letters `a-z` and `A-Z`, digits `0-9`, and special characters _ and `-`.
  - The `name` and `domain` have one or more such blocks, separated by `.`s.

Regular expressions are often used to check the contents of a string. There is already an example of a regular expression in the sample validation script:

```
function checkDigits(textValue) {
  var pattern = /^[0-9]+$/;
  return pattern.test(textValue);
}
```

Here `pattern` is the regular expression, and strings can be tested as to whether they match the pattern or not. Let's look at the pattern in more detail:

- The forward slashes (`/`) indicate the start and end of the pattern, like `"` or `'` are used for the start and end of strings.
- The `^` (called 'caret' or less formally, 'hat') means the start of the string.
- `[0-9]` means any of the characters from '0' to '9', so any digit.
- `+` means one or more of the previous item.
- Finally, `$` indicates the end of the string.

So all together the pattern means, "The start of the string, followed immediately by one or more digits, followed immediately by the end of the string".

We can make other patterns to describe a range of types of string. For example, the `name` and `domain` parts of an email address can be matched (following the rules described above) by

```
/^[a-zA-Z0-9_\-]+(\.[a-zA-Z0-9_\-]+)*$/
```

This looks complicated, but can be broken down as follows:

- The expression `[a-zA-z0-9\_-]+` matches one of the 'blocks' of allowed characters.
- If we call this `X`, then the pattern looks like `^X(\.X)*$`, which is much simpler.
- This means the start of the string (`^`); then a block of characters (`X`); then zero or more (`*`) instances of a `.` followed by a block of letters (`X`); then the end of the string (`$`).
- Note that there is a `\` before the `.`, this is because `.` has a special meaning in JavaScript regular expressions and so needs to be 'escaped'. The backslash (\) indicates that the following character doesn't have its normal special meaning, but should be treated as a literal character.

Task 8.6: Add functions to check the delivery address input, and to report appropriate messages to the user. You should use regular expressions to check the postcode and email address.

Interactive Validation

The validation rules so far have been applied when the form is submitted, but this doesn't always give the best user experience. Often it is better to filter the input as it is entered. For example, the postcode, credit card number, and card verification code elements require that the input consists solely of the digits 0-9. Rather than checking for this after the form is submitted, it is better just to prevent invalid input in the first place.

This can be done in JavaScript by attaching an `onkeypress` event to the form element. This is linked to a function that is sent an `event` object when a key is pressed in the form element, and this object can then be queried to see what key was pressed. If the function returns true, then the key is accepted and the form element is updated; if the function returns false, then the key is rejected and the form element's value remains unchanged. For example, to accept only lower case letters we could use something like:

```
function checkLowerCase(keyPressEvent) {
  var keyCode = keyPressEvent.keyCode;
  if (keyCode < "a".charCodeAt(0)) {
    return false;
  } else if (keyCode > "z".charCodeAt(0)) {
    return false
  } else {
    return true;
  }
}

document.getElementById("myElement").onkeypress = checkLowerCase;
```

Note that the `keycode` property of the event object gives you the character code (or Unicode value) of the key that was pressed. You could compare this against known values – the code for 'a' for example, is 97. However, remembering these codes is not particularly useful, the use of such 'magic numbers' in code makes it harder to understand. For these reasons, it is better to compare with explicitly computed character codes, and JavaScript strings have a `charCodeAt` method that returns the character code at a given index.

Task 8.7: Add interactive validation to your checkout page, so that the user cannot enter anything except digits into the postcode, card number, and card verification code input boxes.

8.3 Successful Form Processing

So far we have looked at what happens when there is an error in the form validation. But what happens if the form is successfully submitted? The actual answer is that the data should be sent to the server for processing, but that will have to wait for later in the course when we look at server-side scripting. For now, we can make it look as if the order has been processed by emptying the shopping cart and making the action of the form a thank-you page

Task 8.8: Update your script so that when the form is successfully submitted the shopping cart is emptied and a message is displayed, thanking the customer for their order.

For the assignment: The administrative functions for your assignment will require form processing. Think about how you can apply what you've learned in this lab to that task. Some of this gets easier to implement with jQuery, which we'll get on to soon, but you can think about the process now, and maybe write some validation functions.

Lab 9

Maps

In this lab we will add an interactive map to the Classic Cinema site using the Leaflet JavaScript library[a], and annotate the map with points of interest.

[a] http://leafletjs.com

Note: This lab is assessed, and is worth 1% of your final grade. Please make sure that you get a demonstrator or teaching fellow to mark it off as completed.

The Contact page for the Classic Cinema site has a placeholder map, but you are probably used to seeing interactive maps on websites. Embedding a map from a provider such as Google or Bing in an `iframe` is one common way to do this. This approach, however, limits the interaction between the map and the rest of your site. To go further you need to register for a key to access the maps through an API, and often this involves giving credit card information. Fortunately, there are free services and APIs that you can use. In this lab we'll be using the Leaflet JavaScript API (`http://leafletjs.com/`) to display OpenStreetMap (`https://www.openstreetmap.org/`) maps.

9.1 Displaying a Map with Leaflet and OpenStreetMap

To get started, download the Leaflet library from (`https://leafletjs.com/`). At the time of writing, the latest version was 1.3.1. You'll need to add a link to the Leaflet JavaScript and CSS files to your page, as well as a new JavaScript file for your code to the `<head>` of `contact.html`:

```
<link rel="stylesheet" href="leaflet.css"/>
<script src="leaflet.js"></script>
<script src="map.js"></script>
```

Your new file, which we've assumed is called `map.js`, should follow the usual module pattern to prevent conflicts with other files. As before, you should have a `setup` function which is called once the document is loaded.

The contact page has an image of a map inside the `<div>` with id `map`. We'll replace this with an interactive map, and with Leaflet this is very easy:

```
map = L.map('map').setView([-45.875, 170.500], 15);

L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
            { maxZoom: 18,
```

```
        attribution: 'Map data &copy; ' +
          '<a href="http://www.openstreetmap.org/copyright">' +
          'OpenStreetMap contributors</a> CC-BY-SA'
    }).addTo(map);
```

This code should go in the `setup` function in `map.js` and the `map` variable doesn't need to be made public, since we will interact with it via specific methods. Note that the variable `L` here is provided by `leaflet.js` and allows you to access its functions using a variant of the module pattern we've already seen.

The first line declares a map, and sets its view to be centred on a set of co-ordinates ($45.875\,\text{deg}$ South, $179.5\,\text{deg}$ East, which is near The Octagon). It also sets a zoom level (in this case 15). The next line loads a `tileLayer`. Online maps are generally divided into square tiles, which are provided as needed by a *tile server*. These tiles can be street maps, satellite imagery, or renderings of geospatial data. The values in curly brackets are filled in with data relevant to the tiles being fetched – see the online documentation for details. A set of options is then provided, including the maximum zoom level and the attribution to display. The OpenStreetMap data is provided under a Creative Commons licence which requires attribution of the source and sharing-alike of any derivative works (CC-BY-SA). The code above provides the attribution required by OpenStreetMap.

Task 9.1: Add the Leaflet libraries and the code above to the contact page to replace the static map image with an interactive map. Be sure to follow the module structure from Lab 5 when writing `map.js`. You will need to add a width and a height to the #map figure element in the CSS otherwise the element will collapse to zero height and you will not be able to see the map.

9.2 Adding Markers to the Map

We can add additional content, such as markers and overlays to the map. Lets start with a simple marker:

```
L.marker([-45.875, 170.500]).addTo(map);
```

This should add a new marker in the middle of the map when it loads.

Task 9.2: Add a marker to the map at -45.875, 170.500. You should be able to add this to the `setup` function in `map.js`.

We can add markers wherever we want, but finding the right co-ordinates to use can be tricky. Leaflet uses the WGS844[1], which is very commonly used. Google Maps, for example uses it, so we could use that to find the co-ordinates of places we're interested in. We can do better than that though – we can add an event handler to the map, so that when we click on the map it will report the location:

```
function onMapClick(e) {
   alert('You clicked the map at ' + e.latlng);
}

map.on('click', onMapClick);
```

Task 9.3: Add the function `onMapClick` to `map.js` and bind it to the map so that you can click and

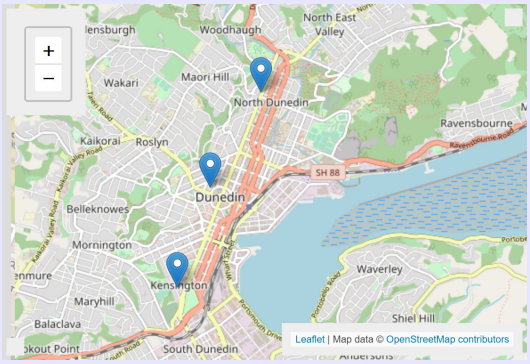
[1] https://en.wikipedia.org/wiki/World_Geodetic_System

find out the location of points on the map. You should think carefully about where these pieces of code go, and how they relate to the module structure in `map.js`.

Now we can start adding markers for the three store locations listed on the contact page. These locations are fictitious addresses, so don't line up with real buildings. The best thing to do is to find some empty area of the map, and place the markers there. Here are some suggestions:

- The Central location is in the Octagon, so you might place it in front of St Paul's, or in the open space in the middle of the Octagon.
- The North location suggests somewhere around North Ground, between St David and Dundas Sts.
- The South location would lie in the Oval, south of the Exchange.

Task 9.4: Add makers for each of the store locations. Here are my marker locations if you're having trouble finding the addresses:

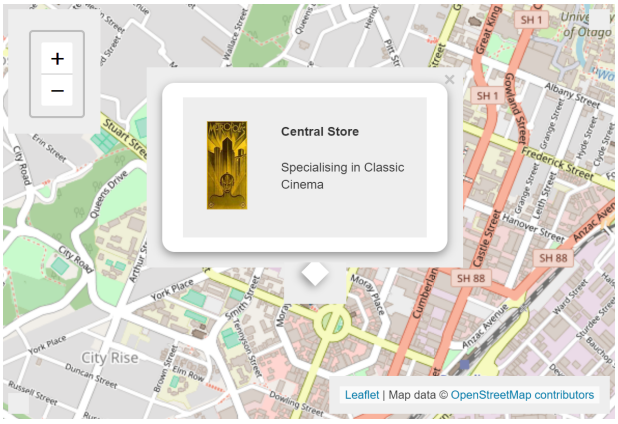


9.3 Adding Detail to the Markers

The markers you have are fairly generic and don't give much information about what location they are indicating. Fortunately, it is easy to add more detail to the markers. You can add some HTML which will be displayed when the marker is clicked. To do this, we need to store the marker in a variable, and then we can bind the HTML to popup:

```
centralMarker = L.marker([-45.873937, 170.50311]).addTo(map);
centralMarker.bindPopup("<b>Central Store</b>" +
                        "<p>Specialising in Classic Cinema</p>");
```

Since this is HTML, you can add images and any other elements you like this:



Task 9.5: Add popups to each of the three store markers, making each one a specialist in one of the three categories. Add a picture to each, and use CSS to style them appropriately. The variables that store the markers (such as `centralMarker` in the example above) should be member variables of the Map closure, since we'll be using them later.

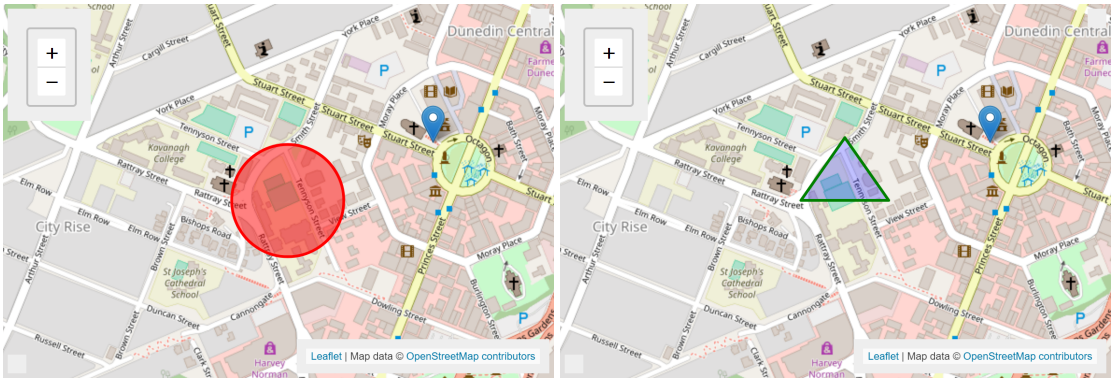
9.4 Working with Overlays

The locations we've placed on the map don't ideally show any buildings, but this is a bit odd. We can add overlays to 'create' fake buildings on the map. For example, to draw a red circle with radius 100m at the starting point on the map, we could use the following code:

```
L.circle( [-45.875, 170.500],
          { radius: 100,
            color: 'red',
            fillColor: 'red',
            fillOpacity: 0.5 } ).addTo(map);
```

We can also draw polygons by specifying a list of corner locations:

```
L.polygon( [ [-45.874, 170.500],
             [-45.875, 170.499],
             [-45.875, 170.501] ],
           { color: 'green',
             fillColor: 'blue',
             fillOpacity: 0.25 } ).addTo(map);
```



9.5 Centering the Map

The map allows the user to see the locations of the hotel, but how can they find the locations in the first place? What we want is for the user to be able to centre the map on each of the three locations. To do this, we'll add `onclick` events to the three headings. This is much the same as showing and hiding the details of the films (see Lab 3).

We start by make a function to centre the map – this will need some Leaflet functions, but for now it can just report the element that triggered the event:

```
function centreMap(e) {
    console.log(this);
}
```

Next, we bind this to the click event on the headings of the three stores. This is done in the same way as the show/hide script in Lab 3 – we get a list of all the headings

```
headings = document.getElementsByTagName("h3");
```

then loop over this list and set each element's `onclick` property to be `centreMap`.

> **Task 9.6:** Set up the `onclick` events on the level 3 headings on `contact.html`, and make sure that the appropriate information is being logged to the console. ✓

Now we need to actually centre the map. First we need to figure out which marker to centre the map on, which we can do by inspecting `this.textContent` in `centreMap` to see what heading we've clicked. Once we know what marker has been clicked, we can centre the map. Recall that you were instructed to store the markers themselves as variables within the `Map` closure. This means they can be accessed by `centreMap`, and we can use the marker location to change the map view. For example, to centre the map on the 'Central' location, we could use:

```
if (this.textContent === 'Central') {
  markerLocation = [centralMarker.getLatLng()];
  markerBounds = L.latLngBounds(markerLocation);
  map.fitBounds(markerBounds);
}
```

The first line inside the conditional statement gets the latitude and longitude of the marker, and puts it in an array. The second line takes an array of locations (in this case, just one location) and finds a bounding box around them. The final line zooms the map view to fit this bounding box.

Task 9.7: Update `centreMap` so that clicking on each heading centres the map on that location.

For the assignment: The assignment requires several map functions to be implemented. Many of those can be done using the material in this lab, but others may require a bit of research.

Lab 10

Introduction to jQuery

In this lab we'll look at jQuery, which is a popular and useful library of JavaScript functions. We'll use jQuery's facilities to rewrite the shopping cart in a simpler form.

Note: This lab is assessed, and is worth 1% of your final grade. Please make sure that you get a demonstrator or teaching fellow to mark it off as completed.

10.1 Getting Started with jQuery

jQuery is a commonly used library of JavaScript functions. Since jQuery is written in JavaScript it doesn't add anything new to the language, but it does make a lot of tasks a whole lot easier. In fact, it makes many things so much easier that at times it can feel like a completely different language. Much more information about jQuery, including the library itself, documentation, and tutorials, can be found at `http://jquery.com/`. Their website also contains some useful tutorials about general JavaScript programming – follow the link from the front page to the jQuery Learning Center.

jQuery Versions

Previously there were two main development lines for jQuery – the 1.x and 2.x releases. The 2.x versions include some newer features but do not support Internet Explorer 8 or earlier. Removing support for older versions of Internet Explorer also decreases the code size noticeably, since a whole lot of workarounds can be removed. The latest version, (3.4.1 at the time of writing), is a successor to the 2.x line, and since we are not too concerned with old versions of Internet Explorer is the one we'll use.

There are also two versions of the code released – *minified* and *uncompressed*. Minified JavaScript is processed to make it as small as possible. Basic minification removes all comments and unnecessary whitespace, and replaces function and variable names with single characters. The goal is to reduce the file size without changing the behaviour.

Task 10.1: Download the latest compressed (minified) and uncompressed versions of jQuery's 3.x branch from `http://jquery.com/download/`.

The minified version of 3.4.1 is about one third the size of the uncompressed one. If you open them up in a text editor, you will see the effects of minification. There is a short comment at the top (containing a copyright notice) but then it gets pretty hard to read:

```
!function(a,b){"use strict";"object"==typeof module&&"object"==typeof module.exports?
module.exports=a.document?b(a,!0):function(a){if(!a.document)throw new Error("jQuery
requires a window with a document");return b(a)}:b(a)}("undefined"!=typeof window?
window:this,function(a,b){"use strict";var c=[],d=a.document,e=Object.getPrototypeOf,
f=c.slice,g=c.concat,h=c.push,i=c.indexOf,j={},k=j.toString,l=j.hasOwnProperty,...
```

and so on.

If you want to see how jQuery does things, then the uncompressed file is the place to look, but there is a lot of code there. The high-level structure, however, is a more advanced form of the module pattern that we've seen already. The module is exposed both as `jQuery` and as `$`. The `$` syntax may look a bit odd, but since `$` is a valid character in JavaScript variable names, it is just another variable name. In particular, it is one which is short to type and unlikely to be used in other code.

Hello, jQuery

As usual, we'll start with a 'Hello World' application. We start with a simple HTML file which includes jQuery, our own JavaScript file, and has a two `<div>`s, one with a unique ID so we can refer to it easily.

```
<!DOCTYPE html>

<html>
  <head>
    <title>Hello, jQuery</title>
    <meta charset="UTF-8">
    <script src="jQuery-1.11.3.min.js"></script>
    <script src="helloJQuery.js"></script>
  </head>

  <body>
    <div id="unique"></div>
    <div></div>
  </body>
</html>
```

Now we can fill in `helloJQuery.js`. In this script we just want to make a setup function that sets the contents of the `<div>` to "Hello from jQuery". In plain JavaScript this would be something like:

```
function setup() {
  var myDiv = document.getElementById("unique");
  myDiv.innerHTML = "Hello from jQuery";
}

window.onload = setup;
```

Using jQuery this becomes

```
function setup() {
  $("#unique").html("Hello from jQuery");
}

$(document).ready(setup);
```

The line `$(document).ready(setup);` is equivalent to `window.onload = setup;` with two main differences. Firstly, it is possible to set multiple functions in different scripts without them overwriting

each other. Secondly, it is called as soon as the DOM is ready to be manipulated rather than when the page is fully loaded. This may be before all of the content (such as images) have loaded.

The `$(...)` syntax looks a little odd at first, but `$` is just a variable declared in the jQuery library, and it is (like all JavaScript objects) able to be treated as a function. You can also use `jQuery(...)` to get the same result (since both `jQuery` and `$` refer to the same object), but the `$` syntax is shorter.

This function is the main way in to most of the jQuery facilities, and so gets a lot of use. There are actually several different versions of this function, depending on what types of parameters you pass to it. The line

```
$("#unique").html("Hello from jQuery");
```

passes a CSS-style selector as a string, and returns a list of all elements that match that selector. Another useful way to call the `$` function is with an existing element. Suppose, for example, that the variable `element` is a reference to some HTML element. We can then find its parent node with

```
$(element).parent();
```

The code `$(element)` makes a jQuery representation of the element, just like `$("#unique")` makes a jQuery representation of the element with the id `unique`.

This list returned by the call to `$` isn't just a simple array, however, as you can call functions on it. In this example we call the `html` method on the array, which returns the HTML contents of each element in the list. The `html` method has several forms, two of which concern us for now. Called with no arguments (`html()`) it returns the HTML contents of the first element in the list. Called with a string (as in this example), it updates the contents of all elements in the list to the string.

Task 10.2: Try out the `helloJQuery` example. What do you think would happen if you change the selector from `#unique` to `div`? Try it and find out.

The fact that jQuery syntax is so compact can also help make your code shorter and more readable. For example, compare this code which changes all list items to be red:

```
var listItems, i;
listItems = document.getElementsByClassName('li');
for (i = 0; i < listItems.length; ++i) {
   listITems[i].style.color = 'red';
}
```

with this:

```
$('li').css({color: 'red'});
```

Not only is the jQuery code much shorter, but it avoids the need to declare variables, which makes your code less prone to errors to do with scope and initialisation.

10.2 jQuery Shopping Cart

As well as providing a much simplified syntax for selectors, jQuery provides improved event handling and DOM manipulation functions. It also provides a consistent interface across different browsers, although you need to use the 1.x libraries to support older versions of Internet Explorer.

As an example of an event, consider clicking on a button. In normal JavaScript we'd use something like this:

```
document.getElementById("buttonID").onclick = eventHandler;
```

With jQuery it is written like this:

```
$("#buttonID").click(eventHandler);
```

Apart from being more compact, this syntax allows multiple event handlers to be easily attached to a single event:

```
$("#buttonID").click(eventHandler1);
$("#buttonID").click(eventHandler2);
$("#buttonID").click(eventHandler3);
```

The event handlers themselves are also passed an event object which provides information about what happened. For a mouse click event this includes what DOM element was clicked, the mouse position (x,y), which button was clicked, and so forth. For example, the following code will report the type of tag and the screen location for all mouse clicks in a document:

```
$(document).click( function (e) {
  // e is the event
  console.log("Mouse clicked on a " + $(e.target).prop("tagName") +
              " element at " + e.pageX + "," + e.pageY);
});
```

The parameter `e` passed to the function is the event object; `e.target` returns the item that triggered the event (was clicked on in this case); and `e.pageX` and `e.pageY` give the mouse position relative to the top left corner of the document. See the Events section of the jQuery API documentation for more information.

DOM manipulation also gets a make-over. With jQuery it is easier to add children or siblings to an element in the DOM tree, to remove elements from the DOM, and to inspect computed CSS styles such as size and position. Some useful functions, which are methods that can be called on jQuery document element objects, are:

- `.append(content)`, which adds content inside an element, after any existing content within that element.
- `.before(content)` and `.after(content)` which add content immediately before or after an element, as children of that element's parent.
- `.empty()`, which deletes any content inside an element, including any child elements.
- `.remove()`, which deletes an element, all of its content, and its children.

The Manipulation section of the API documentation has details on these, and other DOM manipulation functions.

Task 10.3: Rewrite the code to add items to the shopping cart, display the cart, and validate the checkout so that they use jQuery for selectors, events, and DOM manipulation.

For the assignment: Use of the jQuery library in your assignments is encouraged. It can make your code much clearer and easier to understand, as well as easier for you to write.

Lab 11

jQuery, Ajax, and XML

In this lab we will look at Ajax, which allows pages to retrieve information from the server. The X in Ajax comes from the original use of XML to format this information, and we will look at how XML can be processed in JavaScript with jQuery.

Note: This lab is assessed, and is worth 1% of your final grade. Please make sure that you get a demonstrator or teaching fellow to mark it off as completed.

11.1 Ajax

Ajax comes from the acronym AJAX for Asynchronous JavaScript and XML, and is a way for a web page to dynamically update using information from the server without having to reload the whole page. The basic idea is as follows:

- In response to an event some client side code (probably JavaScript) is triggered.
- This code makes a request for some information from the server. This request can happen *asynchronously*, which means that the web page remains responsive rather than waiting for the response from the server.
- The server returns some information, possibly in XML but other formats (plain text, HTML, JSON, etc.) are common as well.
- The receipt of this information is a second event, which triggers some more client-side code to make use of it.

There are various jQuery functions to simplify this process, and we'll look at some of them in this lab.

Hello, Ajax

Let's look at a fairly minimal Ajax example using jQuery. We'll make a button which, when clicked, causes some HTML to be loaded from the server.

- First make a basic HTML page called `helloAjax.html` – it needs to include jQuery and another script, `helloAjax.js`.
- Put a button on the HTML page, along with some element (a `<div>` is a reasonable choice) to load the HTML into.

- Give these elements the IDs `helloButton` and `helloResult`.
- Then make a second HTML document, `ajaxResponse.html` with the single line

  ```
  <b>Hello from Ajax!</b>
  ```

  Note that this file has no `<head>`, `<body>`, or anything. It isn't a complete HTML document, just a snippet that we'll insert into the other page.
- Finally we have the script itself:

  ```
  function doAjax() {
    $("#helloResult").load("ajaxResponse.html");
  }

  function setup() {
    $("#helloButton").click(doAjax);
  }

  $(document).ready(setup);
  ```

Task 11.1: Follow the instructions above and make sure that clicking on the button displays "Hello from Ajax!" in your web page.

Ajax without jQuery

Ajax with jQuery is often very simple, but this hides a lot of complexity. To get some idea of what is going on, let's look at the code that would be required to do the same thing without jQuery:

```
var xmlHttp;

function createXmlHttpRequest() {
  if (window.ActiveXObject) {
    xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
  } else if (window.XMLHttpRequest) {
    xmlHttp = new XMLHttpRequest();
  } else {
    alert("Could not create the XML-HTTP Request object");
  }
}

function handleStateChange() {
  if (xmlHttp.readyState === 4) {
    if (xmlHttp.status === 200) {
       document.getElementById("helloResult").innerHTML = xmlHttp.responseText;
    }
  }
}

function doAjax() {
  createXmlHttpRequest();
```

```
  xmlHttp.onreadystatechange = handleStateChange;
  xmlHttp.open("GET", "ajaxResponse.html", true);
  xmlHttp.send(null);
}

function setup() {
   document.getElementById("helloButton").onclick = doAjax;
}

if (window.addEventListener) {
   window.addEventListener('load', setup);
} else if (window.addEventListener) {
   window.attachEvent('onload', setup);
} else {
   alert("Could not attach 'setup' to the 'window.onload' event");
}
```

There's a fair amount of code here, so let's break it down. The `setup` and the event handling at the end should be familiar to you by now, so there's just the top three functions to look at. These all use the global variable `xmlHttp`, which stores information about the Ajax request.

Let's start with the `doAjax` function, where most of the work is done. Firstly we initialise the Ajax request object, `xmlHttp` by calling `createXmlHttpRequest`. We then assign a function to call in order to process the result, in this case `handleStateChange`. These two functions are discussed below. Next we open a connection by specifying three things:

- The HTTP protocol we are going to use (`GET` in this case, since we're just retrieving information).
- The resource on the server that we wish to access (`ajaxResponse.html`).
- Whether we want to operate asynchronously or not (`true` means we do).

We then `send` the request to the server.

The `createXmlHttpRequest` function initialises `xmlHttp`. Since the way to do this varies between browsers, we have the usual code pattern of checking for behaviour before using it.

The `handleStateChange` function is used to deal with the response from the server. First we check that the Ajax request has completed properly. The `xmlHttp.readySatus` value starts at 0 and increases as the Ajax request is made. It is set to 4 (its final value) when the data has been returned from the server and is ready to use. The `xmlHttp.status` is the HTTP status code for the request. A value of 200 indicates 'OK', while other codes can represent errors (such as 404 for 'resource not found'). If these values are both good, we can access the data returned from the server as `xmlHttp.responseText`.

General jQuery Ajax Requests

The jQuery `.load` function we used in our first Ajax script does a very specific thing. It uses Ajax to retrieve some HTML and inserts it into the page. The `.load` function in our earlier is shorthand for a more general form of Ajax request in jQuery:

```
$.ajax ({
  type: "GET",
  url: "ajaxResponse.html",
  cache: false,
  success: function(data) {
```

```
    $("#helloResult").html(data);
  }
});
```

The `.ajax` function takes a variety of parameters, which are usually specified using object literal notation. In this case we specify what protocol to use (`type`), the resource to fetch (`url`), and what to do when the result comes back (`success`). Setting `cache` to `false` makes sure that we get the latest version of the file. The function given as a value to `success` receives the resulting data as a parameter. There are many other options that can be provided, and these are discussed in the jQuery documentation.

11.2 Classic Cinema Reviews

Let's return to the Classic Cinema site. There is a folder on Blackboard which contains several XML files with reviews for many of the movies on the site. The names of these files are the same as the names of the images for each movie, but with a `.xml` extension instead of `.jpg`. Copy this folder to the same place as the Classic Cinema site (or a copy of it). We will be using this to provide review information on request using jQuery and Ajax.

We first need to add a button to each movie, and a `div` to hold the reviews that looks like this:

```
<input type="button" class="showReviews" value="Show Reviews">
<div class="review"></div>
```

This could be done by manually editing the HTML, but this is tedious when there are many items. Instead we can write some JavaScript to add this information to each item. With jQuery this can be quite short – you just need to get a list of all the films, and then add some HTML elements to them. The function to add a child element in jQuery is `append`, and you can just pass through the HTML you want to add as a string. For example, to append an item to a list with id `theList` you could use

```
$("#theList").append("<li>The new item");
```

The buttons can then be linked them up to some basic JavaScript using the module pattern and jQuery:

```
var Reviews = (function() {
  var pub = {};

  function showReviews() {
    console.log("Show Reviews called");
  }

  pub.setup = function() {
    $(".showReviews").click(showReviews);
  }

  return pub;
}());

$(document).ready(Reviews.setup);
```

Task 11.2: Follow the instructions above to make 'Show Reviews' buttons on the Classic Cinema page, and make sure that they are triggering the `showReviews` function when clicked. Think carefully about where you should create the `input` and `div` elements

Fetching the XML

Next we'll start to fill in the `showReviews` function. The first thing that we need to do is to fetch the XML. This is a call to jQuery's `.ajax` function:

```
$.ajax({
  type: "GET",
  url: xmlSource,
  cache: false,
  success: function(data) {
    parseReviews(data, target);
  }
});
```

We need a few things here. The first is the filename to fetch, `xmlSource` in the code above. The second is a reference to the HTML element where we're going to put the review information, `target` in the code. Finally, we need to write a function `parseReviews` to parse the XML document and inject some HTML into the page.

Finding `target` is fairly easy since we have a reference (through `this`) to the button. We just need to navigate the DOM to get a reference to the related `<div>` with class `review`:

```
var target = $(this).parent().find(".review")[0];
```

We go up one level (`parent()`), then search for the class name. Since this returns a list, we take the first element (there should be only one).

Finding the filename starts with the related image. You can get a reference to that using a similar approach to finding a reference to the `div`. Once you have the image's file name, you can use JavaScript's `replace` function to make the corresponding XML file's name. If the image is "…/images/-Gone_With_the_Wind.jpg', then the XML file is "…/reviews/Gone_With_the_Wind.xml', so we need to replace '.jpg' with '.xml', and '/images/' with '/reviews/'.

Task 11.3: Extend your code so that `parseReviews` is called when the button is clicked. For now it should just log the values of `data` and `target` to the console.

Working with XML

Finally we have to write the `parseReviews` function. This will parse the data in the XML `data` and generate HTML to go inside `target`. Let's start by looking at the structure of the XML files:

```
<reviews>
  <review>
    <user>Alice</user>
    <rating>5</rating>
  </review>
  <review>
    <user>Bob</user>
    <rating>4</rating>
  </review>
</reviews>
```

We want to present this as a list of users and their ratings. The HTML should look like this:

```
<dl>
  <dt>Alice:</dt> <dd>5</dd>
```

```
  <dt>Bob:</dt> <dd>4</dd>
</dl>
```

XML objects have a document model, just like an HTML page. For the XML stored in the variable `data`, we can find all of the `review` elements with

```
data.getElementsByTagName("review");
```

Alternatively, we could make a jQuery object out of the XML and use

```
$(data).find("review");
```

We can then process these further by iterating over each review. jQuery provides a convenient way to do this with the `.each` function:

```
$(data).find("review").each(function () {
  // This code is executed for each review
  // You can refer to the review with the `this' variable, eg:
});
```

This will apply the function provided to every element in the list of `review` tags. For each review we can extract the user and rating with jQuery DOM calls, and get their contents via the `textContent property`.

Our function `parseReviews` now looks like this:

```
function parseReviews(data, target) {
  $(data).find("review").each(function () {
    var rating = $(this).find("rating")[0].textContent;
    var user = $(this).find("user")[0].textContent;
  });
}
```

All that remains is to insert the values we've extracted into the HTML inside `target`, which can be done with the `append` method.

Task 11.4: Add code so that when the XML is loaded, a list of users and their ratings is displayed on the page. What happens if the button is pressed more than once? What happens if the XML on the server is changed?

Two of the films – *The Jazz Singer* and *The Man Who Knew Too Much* – do not show any reviews. However, there is a slight difference between the two. *The Jazz Singer* has an XML file with no `review` entries, while *The Man Who Knew Too Much* does not have an XML file at all.

Task 11.5: Update your code so that in both of these situations, a short message is displayed on the page saying that there are no reviews for the item.

For the assignment: This lab covers the last material that is needed for the first assignment.

Lab 12

jQuery Animation

We finish our jQuery labs with a look at jQuery's animation tools. These work by varying CSS properties over time, to make HTML elements move, fade in or out

Note: This lab is assessed, and is worth 1% of your final grade. Please make sure that you get a demonstrator or teaching fellow to mark it off as completed.

12.1 Back to the Beginning

Let's start by returning to Lab 3, where you wrote code to show and hide the details of a movie. With jQuery, we can make the `showHideDetails` function a lot shorter:

```
function showHideDetails() {
  $(this).siblings().toggle();
}
```

First we make a jQuery object out of `this`, and get all its siblings (that is all of the children of its parent, but not the node itself. Recall that `this` in this case refers to the heading element for a particular film. By accessing its siblings, we get all of the elements associated with the film, except the heading itself. Finally, we call jQuery's `toggle` function. jQuery has a wide range of animation and related functions, but two of the most basic are `show` and `hide`. The `toggle` function switches between showing and hiding an element based on its current state. For more information about these and similar functions, look in the jQuery API under 'Effects'.

Task 12.1: Update your script to use jQuery to show and hide elements. Look into the options you can pass to `toggle` in order to change the animation or to add additional behaviour.

12.2 More Animations with jQuery

Next we'll look at a more complex jQuery animation – a bouncing ball. This uses the most general jQuery effect function, `animate`. The code for this example is in the folder `~steven/public/ball/`, and you'll also need a copy of the jQuery library. Make a copy of the code and look at the HTML, CSS, and JavaScript code. The HTML and CSS are straightforward, but there is a reasonable amount of code in the `moveBall` JavaScript function. Most of this is computing the path for the ball to follow, but the animation itself is done in a single line:

```
$("#ball").animate({paddingTop: ty+"px", paddingLeft: tx+"px"},
                   t, "linear", function() {moveBall(vx, vy);});
```

The `animate` property takes up to four parameters, which are all used in this case:

- First we provide some target CSS values. The animation will transform the object's current CSS values to these over time. Here we give the target `paddingTop` and `paddingLeft` values. Increasing these makes the ball move to the right and down inside its containing `div`, while decreasing them moves it up and to the left.
- Next we give the duration of the animation – by default this is 1 second (1000 ms), but it might be shorter if the ball will hit an edge before then.
- Next we give an 'easing' function. This describes how the speed of the animation changes over time. The default easing value (`swing`) is an elongated 'S' shaped curve that starts off slow, speeds up, then slows down. This makes for a smooth transition in most cases, but here we want a constant speed so use `linear`.
- Finally we give a function to call when the animation is complete. Calling `moveBall` from within itself creates an endless cycle of motion.

Task 12.2: Load `ball.html` in your browser, and try calling `moveBall` with different parameters from the console. See what happens when you change the easing value to `swing`.

The main part of designing an animation is determining what CSS values to change. You can use any CSS property with a numeric argument, and jQuery will smoothly blend from its current value to the new value that you specify.

Task 12.3: Alter the code so that the ball's `opacity` CSS value changes from 0.0 when at the left of the `div` to 1.0 at the right – you can compute the target opacity by dividing the target x-value by the width of the `div`.

12.3 Animating the Classic Cinema

In Lab 4 we replaced the static front page of the Classic Cinema site with a more dynamic one. With jQuery animations we can do even more exciting things. In general you should be cautious with animation effects – it is often a case of 'less is more', and too much can overwhelm or confuse the user. For now, however, the main point is to learn how to use `animate`, so be bold!

Task 12.4: Use jQuery animations to make a more interesting transition between categories on the front page of the Classic Cinema site. New categories might move in from one side of the page to cover the previous one; each category might fade in then out; or you could combine multiple effects.

For the assignment: Animation is not required for the assignment, but might be useful as part of an enhancement. However, you should be sure that any animations (or other enhancements) are actually adding to the usefulness of the site, and not distracting from the content.

Lab 13

Sapphire, Permissions and (S)FTP

In this lab you will change your workflow so that rather than your work on Classic Cinema being automatically published every time you push to GitLab you will start to use `sapphire` as your web server and you will transfer files there using (S)FTP. You will continue using Git and GitLab for version control.

You will learn how to put web pages on the development server, and some of the basics of the Unix command line including permissions and simple scripting.

The web server is `sapphire.otago.ac.nz`. It is a Linux machine running the Apache Web server and a MySQL database server. It is actually running two Web servers – one to act as our *development* server, and one for our *production* server. For now we will just be using the development server. Your username on `sapphire` is the same as your departmental username, but your initial password is different.

13.1 Connecting to `sapphire`

To connect to sapphire, open a terminal on one of the lab machines, and type

```
ssh sapphire
```

You may get a warning that the authenticity of the host cannot be established. Just enter 'yes' in response. You will then be prompted for a password. This is *not* the same as your CS department password – you will learn your password for sapphire in the labs.

Task 13.1: The first thing you should do is to change your password to something more secure. The command for this is `passwd`, so type this at the prompt and press enter. You will be asked to enter your current password and then your new password twice.

Things to Keep in Mind

There are a few things that you should keep in mind when using Unix commands on `sapphire`:

Tab completion If you start to type a command or filename, and have entered enough to uniquely identify the rest, you can press tab to complete it. If what you have typed is not unique, it will complete as much as it can before it becomes ambiguous. Pressing tab twice will list all possible completions from what you have already typed.

Up arrow Unix keeps a history of commands you have typed recently, pressing the up and down arrows will take you through the list of recent commands. You can use this to repeat commands, or you can go back to an old command and edit it slightly before running it.

Manual pages Most of the commands in the Unix environment have manual pages. You can access these with the command `man` followed by the command you're asking about. To get to the manual page for `man`, just type

```
man man
```

The *pager* program that the manual pages use lets you scroll through the text with the arrow keys, and the 'q' key will quit out of it.

Case sensitivity Unix commands, directories, and filenames are case-sensitive

This means that `ls` is not the same as `LS`, and `~astudent` is different from `~AStudent`. For URIs, the specification states that scheme names like `http` or `ftp` and host names like `dev212.otago.ac.nz` are case-*insensitive*. The files remain case-sensitive, however, so the following are the same:

```
https://dev212.otago.ac.nz:8443/~astudent/file.html
Https://Dev212.Otago.ac.nz:8443/~astudent/file.html
HTTPS://DEV212.OTAGO.AC.NZ:8443/~astudent/file.html
```

but they are not the same as

```
https://dev212.otago.ac.nz:8443/~astudent/File.html
```

Secure shell You should connect to `sapphire` using `ssh` (secure shell).

On Linux and OS X `ssh` is a standard command line tool, along with `scp` (secure copy) and `sftp` (secure FTP). On windows you'll need to install some software – one application that provides a terminal with `ssh` is PuTTY, and WinSCP provides secure file transfer. When using `ssh` all traffic between you and the server is encrypted, including your initial login credentials, so is protected from packet sniffing and other similar attacks.

Home directories You have several directories that can be referred to by `~username` in different contexts. It is important to keep these clear.

- There is your computer science home directory

  `/home/cshome/[a..z]/username`

  which is used by lab machines running Linux or OS X, and is stored on the department's central servers.

- There is your home directory on `sapphire`, which is also

  `/home/cshome/[a..z]/username`

  but this is local to `sapphire`, and distinct from the general CS home directories. This is the place to put personal files, configuration files, and the like.

- The `~username` syntax is also used to refer to your web space on `sapphire` in URLs. The configuration on `sapphire` means that

  `https://dev212.otago.ac.nz:8443/~username`

refers to the directory

`/devel/username/projects.`

This is where you will put web pages and related files that you want the web server to be able to access.

13.2 The Unix File System

The Unix file system, like most file systems is a tree of directories, each of which can contain other directories and/or ordinary files. The base of this tree is referred to as the 'root directory' and at the command line is denoted by `/`. Note that 'root' is also used to refer to the primary administrative account on a Unix machine. Paths to files that start with `/` are *absolute paths* – they specify how to get to a file from the root directory. Other paths are *relative paths* – they specify how to get from the current directory to the file.

The important parts of `sapphire`'s file system for this course are shown in Figure 13.1. You have a standard Unix home directory on `sapphire`, which is `/home/cshome/<u>/<username>`, where `<username>` is your normal CS username, and `<u>` is the first letter of your username. For example, if your username is `astudent`, then your home directory would be located at `/home/cshome/a/astudent`. Your home directory is where you can store any general files, but is not normally accessible by the web server.

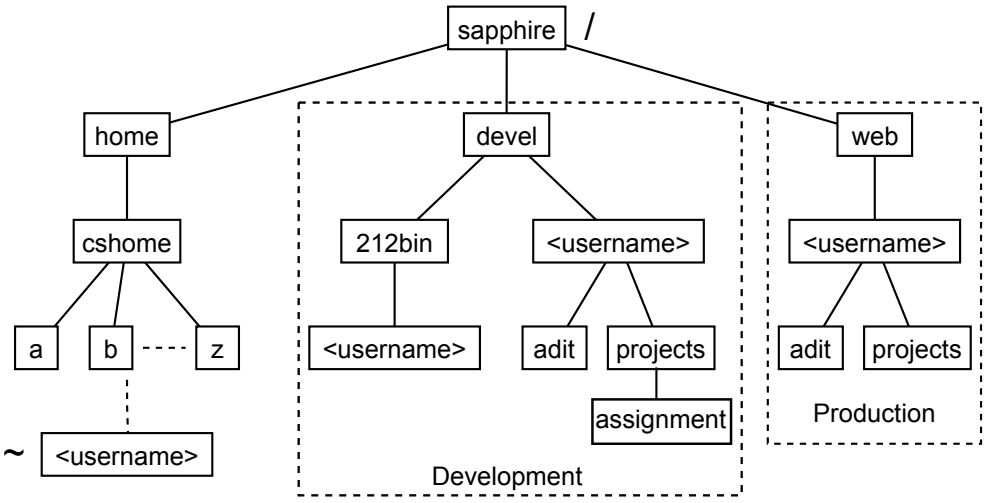


Figure 13.1: The filesystem on `sapphire`.

The other two important places are your development and production web site directories. For now we'll just worry about the development space, but the production space is generally similar. The main place where you put your html, css, and other web page files is `/devel/<username>/projects`. If you put some file, `file.html` say, in this directory it can be seen in a browser by navigating to `https://dev212.otago.ac.nz:8443/~<username>/file.html`.

There are a few important details here. Firstly, the `:8443` part of the address tells the browser to connect to port 8443 rather than the standard HTTPS port (which is port 443). Port 443 is used for the production server. (Note that unsecured HTTP connections use port 80, but we will use HTTPS in COSC212) Secondly, your web pages won't be directly available from computers outside of the campus network for security reasons. If you want to work from home, you can get around this restriction using a technique known as *SSH tunnelling*, which is explained in Section 13.7.

Under your development space there is a directory called `assignment`, where you should put all of your assignment work. Files in this directory will require authentication to be viewed, and so cannot be viewed (and so copied) by other students.

13.3 Navigating the Unix Filesystem

There are several commands used to navigate around the Unix filesystem. Some of the more commonly used ones are:

- `cd <dirname>` (change directory) changes to a new directory, specified as an argument. `cd` with no arguments changes to your home directory, while `cd ..` moves up one directory.
- `pwd` (print working directory) prints the path of the current directory.
- `ls` lists the files in a directory.
- `mkdir <dirname>` makes a new directory in the current directory.
- `cp <filename> <newname>` copies files.
- `mv <filename> <newname>` moves files.
- `rm <filename>` removes files.
- `rmdir <dirname>` removes an *empty* directory. You can remove directories *and everything in them* with `rm -r <dirname>`.

Be very careful with destructive commands like `rm`. Unix systems often do exactly what you ask without checking for confirmation, and there is no undo comamnd.

Most of these commands take options which usually start with a `-`. For example, the bare command `ls` lists the files in the current directory. The command `ls -l` gives more detail about each file (`-l` is for long).

You can use 'wildcard' characters in filenames, which allow for basic pattern matching. The wildcards are:

- `?` matches any single character, except a leading dot.
- `*` matches any zero or more characters, except a leading dot.
- `[]` defines a class of characters, you can use `-` to define a range, and `!` for negation: `[1-5]` is the class of digits from '1' to '5', while `[!a-z]` matches any character *except* lower case letters.

```
rm *.html
```

will remove all files that end in `.html`. What does the following command do?

```
ls -l ../[a-z]*.jpg
```

Task 13.2: Change to the web development directory following the instructions below:

First check where you are when you log in. The command

pwd

should return something like

```
/home/cshome/a/astudent
```

except with your username at the end.

Next change to your web development directory using the `cd` command. You can do this with an absolute path like this:

```
cd /devel/astudent/projects
```

How would you get from your home directory to your web development directory using a relative path?

Use `pwd` to confirm that you're in the right location, and then `ls` to list the files that are there - there should only be one (template.html) which we shall use shortly.

Task 13.3: Create a default web page on the development server. ✓

Now that we're in the development directory, let's make a default home page. There are a number of text editors available from the command line, with Vi and Emacs being the most popular. If you've had experience with one of those, then feel free to use it. Otherwise, you can use Nano, which is a bit easier to get started with (but less powerful once you've mastered the others). To start editing a new file you could just type

```
nano index.html
```

Often when developing web pages, however, you don't want to start with an empty file. Rather, you start from an existing file and modify it. To do this you need to make a copy of an existing file, and then open it in the editor. There is a basic HTML template called `template.html` in your projects directory. You can copy this file to a file called `index.html` with the command

```
cp template.html index.html
```

If you then start editing this file with nano, it should appear as shown in Figure 13.2.

The text at the bottom gives you a reminder of some commonly used commands. The `^` character represents the control key, so `^X` is shorthand for Ctrl-X, and means exit. Nano (for historic reasons) uses WriteOut and Read File to Save and Open files.

You should be able to view your page in a browser, by navigating to

```
https://dev212.otago.ac.nz:8443/~username/index.html
```

Since the web server on `sapphire` is configured to look for default pages, you can also view an 'index' page by specifying just the directory:

```
https://dev212.otago.ac.nz:8443/~username/
```

13.4 Putting a Web Page on the Development Server

Editing files directly on `sapphire` is sometimes useful, but often it is more convenient to develop files elsewhere and transfer them across.

Using PhpStorm's FTP capabilities

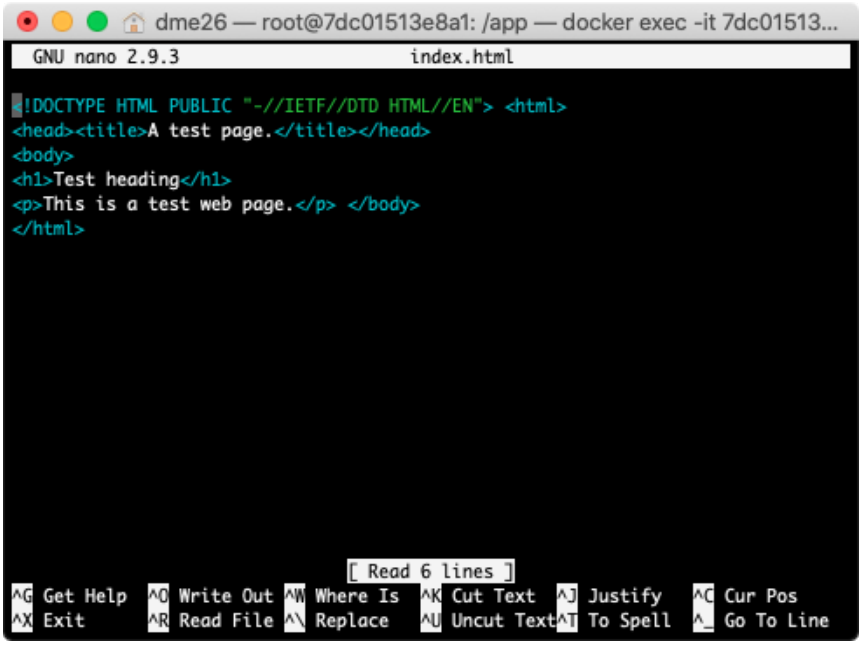


Figure 13.2: Editing `index.html` with Nano.

Watch the short video 'PhpStorm - FTP Setup' available from Blackboard in the Lab Materials/Videos. This shows you how to setup PhpStorm's FTP client.

Using FileZilla

There is a short video demonstrating the use of FileZilla in Lab Materials/Videos if you need a stand-alone FTP client.

FileZilla is in the Applications folder, and when you launch it you will get a window with a lot of sections. Across the top are entry boxes for Host, Username, Password, and Port. Fill these in as follows:

Host: `sftp://sapphire` – you need to use secure FTP to connect to `sapphire`.

Username and Password: your computer science username and your password for `sapphire`.

Port: 22 – the standard port for secure FTP connections.

You will get a warning about 'The server's host key is unknown', this is expected. Check the 'Always trust this host' box, and then click OK – this will prevent the message from reappearing when you connect in the future.

Once you have connected you will see navigation tools for the local site (the lab machine) and the remote site (`sapphire`). You can navigate through directories with the tree view in the top pane under each site, and drag files between the bottom panes. These are illustrated in Figure 13.3.

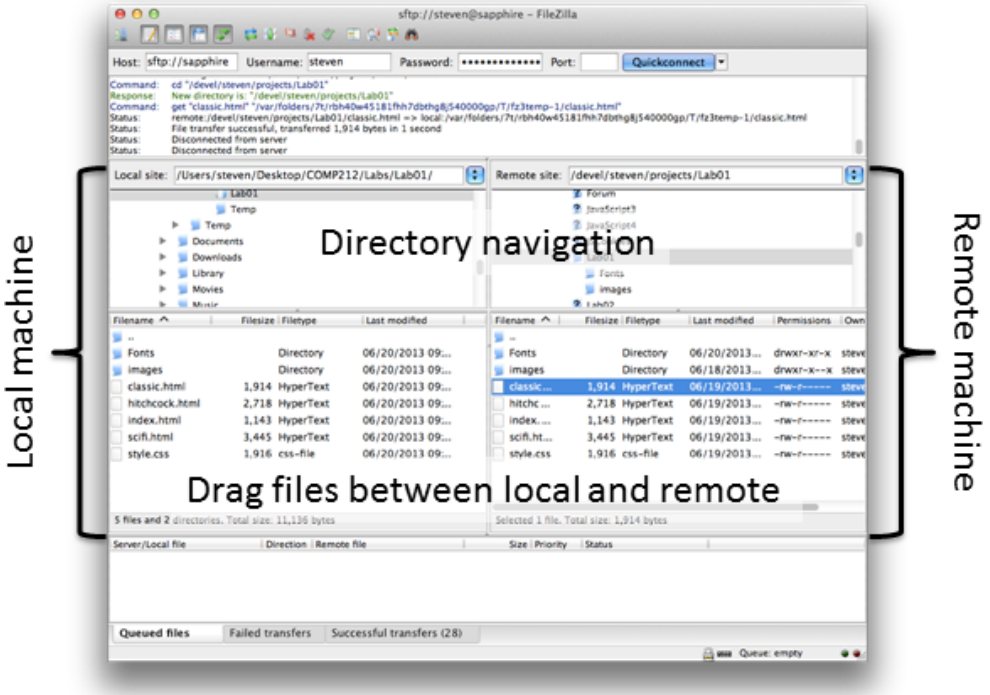


Figure 13.3: The FileZilla FTP client.

13.5 Files and Permissions

All files on a Unix system have a set of permissions which describe which users can access them. The `ls` command with the `-l` option will give you this information (and some more). If you have completed the lab work so far, then `ls -l` in `/devel/projects/username` should give you a result like this:

```
-rw-r--r-- 1 astudent svr212  172 Apr 18 13:41 index.html
drwxr-sr-x 2 astudent svr212 4096 Apr 19 10:29 Lab01
```

Task 13.4: Find out what the rest of the information provided by `ls -l` means.

For now, we're interested in the first block of characters which describe the permissions, and the owner (`astudent` in this case) and group (`svr212`). There are 10 flags which tell you who can do what to the file. You should read these in chunks like this:

```
    -       rw-     r--      r--
    d       rwx     r-s      r-x
directory  user    group    other
```

- The first flag tells you whether the item is a directory (`d`) or an ordinary file (`-`).
- The next three blocks tell you whether the file is readable (`r`), writable (`w`), or executable (`x`).

- The three blocks relate to the user who owns the file (`u`), a group associated with the file (`g`), and other users of the system (`o`).
- For files executable means that the file is a script or program that can be run by the appropriate user or group.
- For directories you can *list* files if it is readable, *add, delete, edit, and rename* files if it is writable, and *search* the directory if it is executable. To be able to search a directory that you can't list means that you can access a file if you know the name, but you can't find out about what files are there.

Each file also has an owner (`astudent` in the example) and a group (in this case `svr212`). For files on `sapphire` to be accessible on the development server, they must be:

- Somewhere in the file system under `/devel/<username>`.
- Readable by the `svr212` group.
- In a directory that is executable by the `svr212` group.

The special flag `s` in the group executable space of the directory means that new files created in that directory inherit the group value. This flag is set on your `devel/username/projects` directory, so any files which you make in the development space are by default in the `svr212` group.

Task 13.5: Create a new file in your *home directory* on `sapphire`. What permissions does this file have? What group is it in?

Changing Permissions

The `chmod` (change mode) command is used to change permissions on files. The basic syntax is

```
chmod <permissions> <filename>
```

but there are several ways that you can specify the permissions.

Firstly, you can set absolute values for the user (`u`), group (`g`) or other (`o`) permissions. For example,

```
chmod u=rwx,g=rx,o=- index.html
```

will give the user all permissions, the group read and write, and remove all permissions from others on `index.html`.

Secondly you can add or subtract permissions. For example

```
chmod g+x index.html
```

will give the group execute permissions from `index.html`, while

```
chmod go-r index.html
```

will remove read permissions from the file for the group and others.

Finally you can use a numeric shorthand for the permissions. You can view the read, write, and execute flags as binary values, and interpret them as a number from 0 to 7, as shown in Table 13.1. Three numbers can then be used to specify permissions for user, group, and other. For example,

```
chmod 644 index.html
```

will set the permissions of `index.html` to `-rwr--r--`.

Table 13.1: Numeric values for Unix file permissions

Permissions:	`---`	`--x`	`-w-`	`-wx`	`r--`	`r-x`	`rw-`	`rwx`
Binary:	000	001	010	011	100	101	110	111
Value:	0	1	2	3	4	5	6	7

Task 13.6: Check that you can access `index.html` from a browser. If you have been trying out `chmod` commands on `index.html`, you will need to reset the permissions to (at least) `-rw-r-----` before you begin.

Task 13.7: Remove the read permissions on `index.html` from the `svr212` group. Try to reload the page in your browser – it shouldn't work. Add the read permission back to the group and make sure you can access the page again.

Task 13.8: Create two directories, `public` and `outbox`, in your home directory on `sapphire`. Set the permissions on `public` so that everyone can enter and search it, and read (but not write) any files inside of it. Set the permissions for `outbox` so that other people are not able to enter the directory, or list its contents, but if they know the name of a file in it then they should be able to read that file.

Some of the files you create on `sapphire` may contain information that you don't want to share with your classmates. This may include authentication details such as password information, as well as work for assignments. It is your responsibility to ensure that permissions are set so that other users cannot access this information.

Permissions in FileZilla

The FileZilla FTP client (like most others) also allows you to set permissions. If you select a file on the Remote site, and right- or Ctrl-click on a file on the remote server you get a pop-up menu. The 'File permissions...' option will bring up a dialogue box where you can set the file permissions.

13.6 Unix Scripting

Unix scripts allow you to automate common tasks. Any command that you use in your shell can be put into a file, and the file can then be run as a 'shell script'. If you recall the file from Lab 2, .gitlab-ci.yml. The commands in that are a bash script.

A Basic Shell Script

To be a shell script, a file needs two things. Firstly, it needs to be executable, and secondly it needs the first line to be

```
#!/bin/sh
```

This 'magic line' starts with the characters `#!` (often read 'hash-bang', or 'shebang') which indicate that this is a script, and then the path to the program that will interpret the script – in this case the shell, which can be found at `/bin/sh`.

Task 13.9: Log on to `sapphire` and navigate to your home directory. Make a new file called `hello.sh`, this will be our 'hello world' shell script. Set the permissions on this file to be executable by you, and edit it so that it contains the following text:

```
#!/bin/sh
# `Hello world' script
echo Hello, world!
```

The first line is the required magic to make this a shell script, the second line is a comment – in shell scripts all characters after a # are ignored by the interpreter. Note that the magic line is also a comment in this sense – it is not executed by the interpreter. Finally we have a command that prints "Hello, world" to the console.

Task 13.10: Save your file, quit the editor, then enter `./hello.sh` as a command at the console.

Scripts and Permissions

Since permissions can be set from the command line, it is possible to write scripts to change them. The permissions on files in `/devel/username/projects` determine whether pages are available online or not. In order to take our files off line, we need to do the following:

- Change into the directory `/devel/username/projects`
- Remove the read permissions from the group for all of the files in the directory.

To remove permissions on all of the files you will need to use a wildcard. Since the files from HTML and CSS Revision are in a sub-folder, they will not normally be affected by the `chmod` command. To overcome this you can use the `-R` (for recursive) option to `chmod`, like this:

```
chmod -R <some permissions> <some file(s) or folder(s)>
```

This will work its way down any folders (and their sub-folders, and so on) applying the permission changes to everything in that sub-tree.

Many Unix commands have recursive options (usually `-R` or `-r`). These should be used with care because you can affect a lot of files with a single command.

Task 13.11: Write a script to ensure that all of the files in your web directories are online, but are invisible to other users on sapphire.

Task 13.12: Write another script that takes all of your files offline, but does not move or remove them.

Permissions and Groups

Recall that the directories in your development space on `sapphire` have a special flag that means that files within them inherit their group id. This is important as it means that new files are accessible by the web server. Changing the group permissions on a directory can remove this flag, but there is a script that runs every few minutes to reset it. If you have changed permissions and find that new files are not accessible online when you expect them to be, check that they are in the `svr212` group, and check that the

execute position for the group on the containing directory is `s`. If not, wait until the background script has reset the permissions on the directory and re-create the new file.

13.7 Accessing `sapphire` from Off-Campus

The development server on `sapphire` is configured only to accept connections from on-campus. This is also true of the other server names that `sapphire` responds to, such as `dev212.otago.ac.nz`. This allows us to give you the ability to write potentially unsafe Web applications without too many security concerns. However, you can get around this by connecting to `sapphire` indirectly.

This is done by forwarding your connection to `sapphire` or `dev212` through another computer via *SSH Tunnelling*. This computer needs to be on-campus, able to accept external connections, and accessible by you. Fortunately there is a machine with just these attributes – `hex.otago.ac.nz`. What you need to do is to associate a network port on your home machine with the Web server on `sapphire`, and direct traffic through `hex`. Exactly how you do this depends on what operating system you are running on your home machine.

Connecting from Mac or Linux

There is a short video demonstrating the process of connecting to `sapphire` from OS X and windows via an SSH Tunnel. This is available from Blackboard in Lab Materials/Videos. However, note that some of the server name details may have changed, so you should read the text below carefully as well as watching the video.

If you are running OS X or some variety of Linux this process is quite simple. Simply enter the command

```
ssh -L 9999:dev212.otago.ac.nz:8443 <your cs username>@hex.otago.ac.nz
```

This associates port 9999 on your home computer with port 8443 (the development Web server) on `dev212`, forwarding traffic through `hex.otago.ac.nz`. You need to provide your usual departmental username, and will be prompted for your password.

The first time you do this you will get a message about encryption keys. Answer 'yes' to add `hex.otago.ac.nz`'s public encryption key to your home computer's list. As long as you remain logged in to `hex` you will be able to direct your browser to `http://localhost:9999/~<username>/file.html` and get whatever page you'd normally see on campus at `https://dev212.otago.ac.nz:8443/~<username>/file.html`.

You can use multiple instances of `-L` to link other ports as well. For example, to access web pages as above, and also to use SSH and SFTP (which operates on port 22) to transfer files to `sapphire`, you could use:

```
ssh -L 9999:dev212.otago.ac.nz:8443 -L 2222:sapphire.otago.ac.nz:22 user@hex...
```

You could then view web pages as above, and also connect to `sapphire` for SFTP through `localhost` port 2222.

Connecting from Windows

Connecting from Windows is a little more complicated, since you'll need to install some software. You will need a SSH client, and two popular options are PuTTY (`http://www.chiark.greenend.org.uk/~sgtatham/putty`) and Tunnelier (`http://www.bitvise.com/tunnelier`). Both are free for personal

use, and Tunnelier also provides SFTP for secure file transfer. If you use PuTTY for SSH connections, then WinSCP (`http://winscp.net/eng/index.php`) is an alternative file transfer option.

Whether you use PuTTY or Tunnelier, you need to set up some basic details for the SSH connection. The host name is `hex.otago.ac.nz` and you are connecting through port 22 (the default for SSH). This is shown in Figure 13.4.

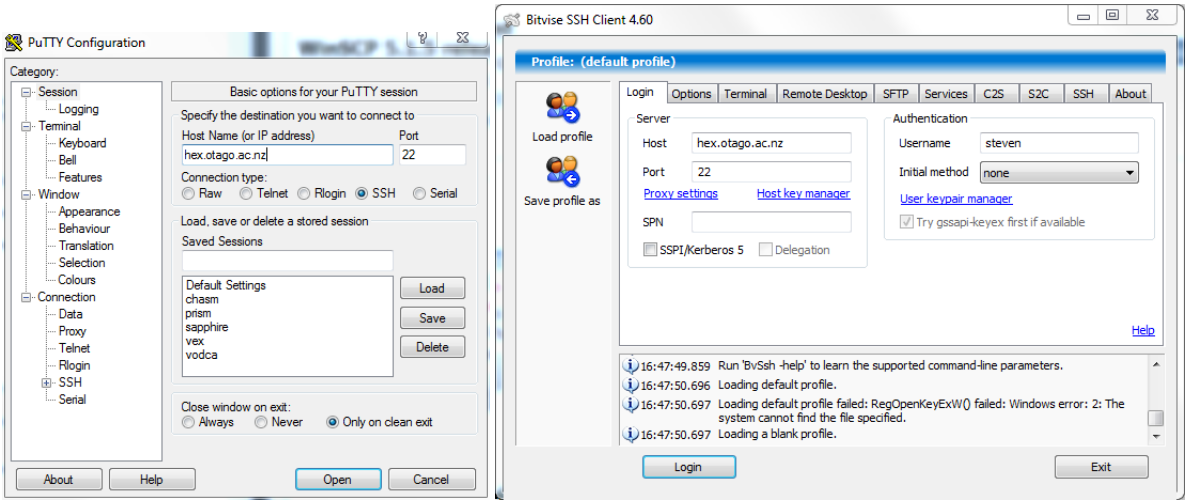


Figure 13.4: Connecting with PuTTY (left) and Tunnelier (right).

Next you need to set up the port forwarding. In PuTTY you can do this in Connection->SSH->Tunnels. Set the Source port to 9999; Destination to `dev212.otago.ac.nz:8443`, and then click add. In Tunnelier this is done in the C2S (client-to-server) tab. Click Add, then change the List. Port to 9999; Destination Host to `dev212.otago.ac.nz:8443`; and Dest. Port to 8443. You can add other ports (such as a link to port 22 for SSH and SFTP) as required. These details are shown in Figure 13.5

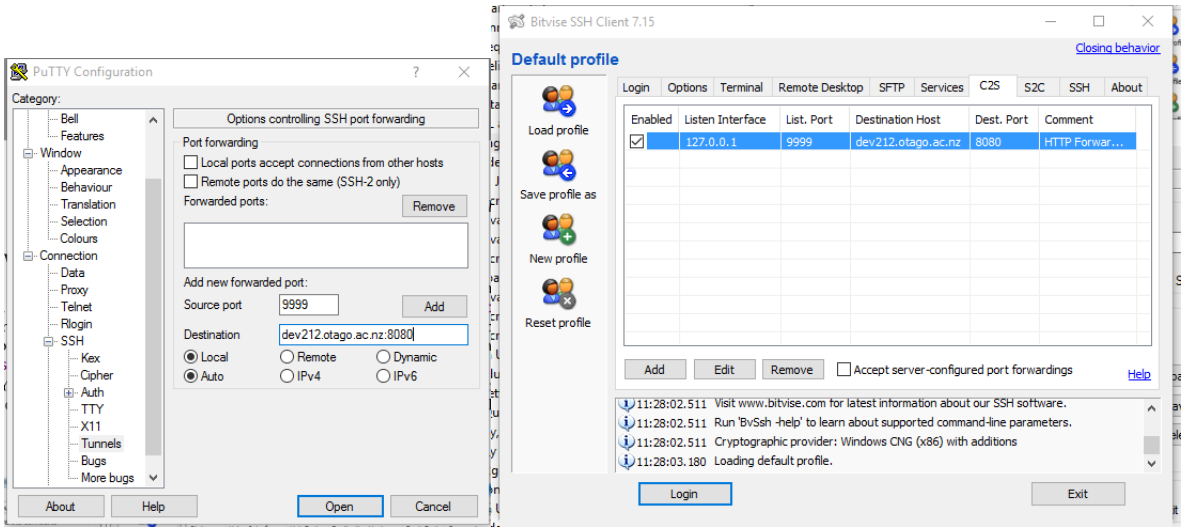


Figure 13.5: Setting up port forwarding with PuTTY (left) and Tunnelier (right).

Now click the Open (PuTTY) or Login (Tunnellier) button. You'll be prompted about adding RSA encryption keys, which is OK, and for your username/password. As long as you keep the connection to `hex` open, you will be able to direct your browser to `http://localhost:9999/~<username>/file.html` and get whatever page you'd normally see on campus at `https://dev212.otago.ac.nz:8443/~<username>/file.html`.

13.8 Further Information

As mentioned earlier, there is help available for most Unix commands via the `man` utility. The *Advanced Bash Scripting Guide* is a useful reference, and is available from `http://tldp.org/LDP/abs/html/`. As its title suggests, this covers some advanced material but the introductory stuff is very good as well.

Unix has a long history, in part due to its underlying philosophy. Eric Raymond's *The Art of Unix Programming* is both an overview of Unix programming and an attempt to capture the philosophy and culture behind it. The book is freely available at `http://www.faqs.org/docs/artu/`.

For the assignment: We have provided a directory for your assignment files, which is protected from other people. Think about how you could protect your files from unauthorised access if you didn't have that.

Lab 14

Catch Up

There are no new exercises for this lab. The first assignment is due at the end of this week, however.

If you have already completed all of the labs, then there are a number of ways you can make good use of the time:

- You can continue to work on the assignment.
- You can investigate other JavaScript tutorials online to reinforce what you have learned.
- You can try out other things in JavaScript to gain more practice and experience.
- You can look ahead to future labs to get a head start on those.

Part IV

Server-Side Scripting

Lab 15

Hello, PHP

In this lab you will start to work with PHP, which will be our primary language for server-side scripting. You will write your first PHP pages, and use PHP to factor out some of the common content of the Classic Cinema site.

Note: This lab is assessed, and is worth 1% of your final grade. Please make sure that you get a demonstrator or teaching fellow to mark it off as completed.

PHP is a commonly used language for server-side scripting. Earlier methods of server-side scripting, such as CGI scripts, produce web pages by printing out large amounts of HTML. So, if you were doing server-side scripting with Java, you'd probably have a lot of statements like

```
System.out.println("<p>Some text that never changes</p>");
```

PHP avoids these difficult to handle print statements by using special tags to indicate which parts of a page are PHP code. All other text is passed through unchanged. The effect is that you can create "islands" of PHP code surrounded by a sea of HTML. This makes it easy to start from static HTML and work towards dynamic server-side scripts. This easy transition from plain HTML, along with the extensive documentation extensive documentation and tutorials available from `http://www.php.net/`, are large factors in PHP's popularity.

15.1 Hello World in PHP

As usual, we begin with a basic script using the new language to say "Hello, World!". The special tag that indicates a block of PHP is `<?php ... ?>`, so our hello world program in PHP looks like this:

```
<!DOCTYPE html>

<html>
  <head>
    <title>Hello from PHP</title>
    <meta charset="utf-8">
  </head>
  <body>
      <p><?php echo "Hello, World!"; ?></p>
  </body>
</html>
```

Task 15.1: Make a file on `sapphire` called `hello.php`, and put the script above in it. You may need to refer back to Lab 13 in order to start Docker Desktop, reach the Docker Desktop dashboard, and to start up your LAMP container.

The following points are useful to keep in mind about PHP scripts:

- PHP is embedded inside HTML, rather than producing entire HTML pages (although some HTML is typically produced by the PHP code).
- PHP scripts go in the `app` directory that you used for HTML content in Lab 13.
- In your containers the default Unix file permissions should work when you create new files. The PHP scripts do not need to be executable by the web server, but they do need to be readable. The web server does not run the scripts themselves, rather it runs a PHP interpreter, which in turn reads the contents of the script and returns the resulting HTML.
- There is nothing in the file to tell the web server that this is a PHP script—the extension `.php` does that.

PHP Variables and String Interpolation

Now you have your first PHP script going, but it isn't very interesting—to make things slightly more complex let's introduce a variable:

```
<?php
  $name = "World";
  echo "Hello, $name!";
?>
```

Note that PHP names all start with a `$`, which make things a little confusing when you are moving between PHP and JavaScript with jQuery. Also note that within double quotes, variables are replaced by their values. This is called *string interpolation*, and is a bit more convenient than using the string concatenation like this:

```
<?php
  $name = "World";
  echo "Hello " . $name. "!";
?>
```

Here, `.` is PHP's string concatenation operator, like `+` in JavaScript.

PHP, like JavaScript, lets you use either single or double quotes for strings, but there is a small difference—string interpolation only works inside double quotes.

Task 15.2: Update your script to use a variable to store the name to greet. Try changing the value of the variable `$name`, and replacing the double quotes in the `echo` statement by single quotes.

Debugging PHP Errors

Small programs like this 'Hello, World" example let us check that the basics are working before we move on to bigger things. Another useful test is to introduce a *deliberate* error to see what happens. We can then use this to learn how to diagnose problems.

Task 15.3: Remove the semicolon from the first line of your PHP script and reload your page. You should see an error message.

We have enabled display of errors within the browser, since it is more convenient for supporting software development. In production systems, display of error messages may be a security risk, and are often disabled, so that PHP pages with an error simply display a blank page, or an error page that does not include details about the error.

On such systems, there is still a way to get some more information about what went wrong. Errors that arise during PHP processing are stored on `sapphire` in a file called `/var/log/apache2/error.log`. If you need to filter this file for a particular pattern, you can use the Unix command `grep`:

```
grep <search-term> /var/log/apache2/error.log
```

If you find (later on) that there are still too many errors reported, you can just look at the latest ones by passing the result of the `grep` on to the `tail` command—

```
grep <search-term> /var/log/apache2/error.log | tail -n10
```

—which will list just the last 10 (or however many you specify with `-n`) errors.

Task 15.4: Look in the error log to see what error is reported by the missing semicolon. Compare the details in the server's log file with those shown in the web browser.

Unfortunately you don't get quite the result you might want—PHP doesn't recognise the missing semicolon as the source of the error, but rather reports the message `unexpected T_ECHO`. This means that when parsing the PHP file an `echo` token (`T_ECHO`) was encountered where one wasn't expected. This is because the missing semicolon means that PHP tries to continue the command on to the next line, where it encounters the error.

You can often find more information about specific PHP errors online, and at least the line number in the error will tell you where in your code to start looking. Remember, however, that it is worth looking at the lines before (and sometimes after) the reported error as well.

It is also a good idea to check the PHP error log when developing your script, even if things seem to be working well. Some issues cause warnings rather than errors, and these will also be recorded in the log.

15.2 PHP Includes

One simple, but useful, application of PHP is including one `.php` file within another. This is done with one of two PHP statements, either:

```
<?php include ("somefile.php"); ?>
```

or

```
<?php require ("somefile.php"); ?>
```

The difference between the two is that `require` will raise an error if it cannot find the specified file, while `include` just issues a warning.

Include files are useful because you often find that you have common content on many pages of a site. For example, all of the Classic Cinema pages have much the same content in the header, navigation, and footer. If we want to change this information (such as adding a link in the navigation) then we have to update all of the pages. This is extra work and leads to errors, and so having this common content in one place is a good thing. Note that when you enter an `included` or `required` page you leave PHP mode. If the contents of the file are intended to be interpreted as PHP code they must be contained in `<?php ... ?>` tags.

PHP and the Classic Cinema Site

To get started with a PHP-powered version of the Classic Cinema site, we can make a copy of it and rename the files from `.html` to `.php`. If this was all we were going to do, you would also need to update the links in the navigation and the front-page carousel to refer to the PHP rather than HTML versions. We'll soon be updating the navigation anyway, but it is worth changing the carousel links at least.

Task 15.5: Make a copy of the Classic Cinema site, renaming the `.html` files to `.php`. Update the carousel links to refer to the PHP pages—if you are using the sample solution you will need to change the parameters passed to `MovieCategory` in `Carousel.setup`.

The new site should look the same as the old one, except that the navigation links won't work. Clicking on the carousel links should work, however. Don't worry about the navigation links for now. Updating them in all of the pages would be tedious, and include files allow us to update a single file which affects all the pages in the site.

Let's start by replacing the footer information with a simple include file. All of the files have the main content in a `<div>`, followed by the an HTML `<footer>` element and close tags for the `<body>` and `<html>` elements. We will replace this with an include file.

Task 15.6: Make a new PHP file called `footer.php` which contains the `<footer>` tag and its contents. Replace this HTML in the other classic cinema pages with a PHP include.

The information before the main `<div>` is *almost* the same on every page, but there are some differences:

- Different JavaScripts are included on each page—for example, only `index.php` needs `carousel.js`.
- The navigation item to the current page is not an active link.

There are several ways that we could deal with this. We could just include all the scripts on every page, and have all of the navigation links active on every page. This is simple but has two main issues. Firstly we will be loading unnecessary JavaScript files, which will increase page load times. Secondly it won't be as clear to the user where they are on the site.

Task 15.7: This solution isn't ideal, but it is the easiest way to get started and will provide a basis for a better solution later. Write a `header.php` script which includes all of the JavaScript files required by any page, and which has all links active in the navigation pane. Don't forget to change the navigation links to point to the `.php` files.

An alternative would be to just use include files for the constant content, and to keep the page-specific stuff separate. Using the sample solution, the pages would then look something like this:

```
<?php include("header1.php"); ?>
<!-- <script> tags go here -->
<?php include("header2.php"); ?>
<!-- Navigation <li> links go here -->
<?php include("header3.php"); ?>

<div id="main">
<!-- Main content goes here -->
</div>
```

```
<?php include("footer.php"); ?>
</body></html>
```

This is better, but leads to quite a lot of fragmentation which can get confusing. For example, for the navigation we'd have a lot of `<li>` tags in the main file for each page, but the containing `<ul>` tag would be in `header2.php` and the corresponding `</ul>` would be in `header3.php`. Sometimes this can't be avoided, but it can get a bit messy.

The final solution is to declare some variables at the top of each page, then use them to alter the way the first PHP include operates. These variables need to deal with the scripts to include, and the selection of different navigation links. We will declare these variables at the top of each PHP page, before `header.php` is included. They will still be visible inside `header.php`, and so can be used to control what is displayed.

Including Different Scripts

To include different scripts, we can just give a list of scripts that are needed by each page. For example, the sample solution's index page just requires the jQuery library and `carousel.js`. We can use this to define an array of scripts at the top of `index.php` before we include `header.php`:

```
<?php
  $scriptList = array('jquery-1.11.1.min.js', 'carousel.js');
  include('header.php');
?>
```

Inside `header.php` we can iterate over this array to make a list of script tags:

```
foreach ($scriptList as $script) {
  echo "<script src='$script'></script>";
}
```

PHP's `foreach` provides a convenient way to loop over the elements of an array. In this case each element of `$scriptList` is assigned to `$script` in turn. We then use string interpolation to output the relevant HTML for each script.

This code assumes that `$scriptList` has been set, and is an array. We can check this using an `if` statement:

```
if (isset($scriptList) && is_array($scriptList)) {
  foreach ($scriptList as $script) {
    echo "<script src='$script'></script>";
  }
}
```

Task 15.8: Update the `header.php` script to use a `$scriptList` variable to include just the required scripts for each page. Remember to set `$scriptList` in each page before including `header.php`.

Fixing the Navigation Links

The second issue with the header content is that the navigation links to the current page need to be disabled. The basic approach is to check which page we are on, and to output either an active link or plain text as appropriate. For example, the navigation entry for the Home page might be formatted like this:

```
if ($currentPage === 'index.php') {
  echo "<li> Home";
} else {
  echo "<li> <a href='index.php'>Home</a>";
}
```

If we are already on `index.php` then we just output plain text. Otherwise we output a link to `index.php`. However, we still need some way to determine what page we are currently viewing.

There are a number of ways in which we could do this. We could just set the value of `$currentPage` at the top of each page, before including `header.php`. For example, at the top of `index.php` we could have

```
<?php
  $scriptList = array(...);
  $currentPage = 'index.php';
  include('header.php');
?>
```

This is OK, but has two main problems. Firstly, the value is hard-coded, so we need to remember to change it if we move or rename a page. Secondly, it depends on the developer remembering to set `$currentPage` before including `header.php`. In some cases (such as with `$scriptList`) this may be the way to go, but in this case there is a better solution.

PHP makes a number of variables available which describe the context in which the script is being executed. These are known as *superglobals*, because they are always in scope and so can be accessed within any function or script. One of these is called `$_SERVER` and provides information about the web server and execution environment. It is an associative array, and so maps names to values, and the particular name-value pair we're interested in here is `_SERVER['PHP_SELF']`, which is the filename of the script the web server is running. This gives the full pathname of the file, but we can extract just the bit we need with

```
$currentPage = basename($_SERVER['PHP_SELF']);
```

This is an interesting option because it gives the name of the PHP script that is being *executed by the server*, which is the page that the user originally requested and not the included file. This means that we could use `$_SERVER['PHP_SELF']` within `header.php` to find out what page of the site we are on. Using this approach would mean that we would not have to remember to set `$currentPage` in each script, which is much safer and more convenient.

Task 15.9: Update your `header.php` script so that it doesn't display links to the current page in the navigation elements.

Lab 16

PHP Form Processing

In this lab we will look at server-side validation of form data using PHP. We'll look at how to access information submitted via a form, and at regular expressions in PHP.

Note: This lab is assessed, and is worth 1% of your final grade. Please make sure that you get a demonstrator or teaching fellow to mark it off as completed.

16.1 PHP Form Processing

Let's begin with a simple extension of the Hello World program, by allowing users to enter their name via a form. We start with a simple HTML page, `helloForm.html`, which includes a form with the following properties:

- The action of the form should be `processHello.php`, and it should pass information using the `GET` method.
- The form should have a submit button and a single text input with name `user`, and a suitable label.

We now need to write the PHP file `processHello.php`. In order to access this information we use the superglobal variable `$_GET`. There is a corresponding variable, `$_POST`, for forms submitted with `method='POST'`. These variables are arrays, which are indexed by the name of the form elements, so to access the value input in the form above we can use `$_GET['user']`.

Task 16.1: Write the HTML page `helloForm.html` and a PHP page, `processHello.php`, that takes the input from `helloForm.html` and makes an HTML page which greets the user by name.

Securing the Form

Even this simple form is open to an injection attack—what happens if the user enters some HTML in the form, say `<b>Injected!</b>`? While entering some bold text isn't too harmful, it would be possible to include much more interesting and dangerous content. Fortunately this is easy to prevent. PHP provides a function which will HTML-encode special characters, so that '<' becomes '`<`', etc. This function, `htmlentities`, takes a string as a parameter, and returns the encoded string as the result.

A second issue is that people can navigate directly to `processHello.php` without submitting the form. We can prevent this by checking to see if a value for `user` has been submitted, and redirecting the

browser to `helloForm.html` if it has not. Redirection is done by sending back some HTML headers, and the PHP function `header` does that:

```
if (!isset($_GET['user'])) {
  header("Location: helloForm.html");
  exit;
}
```

The `header` function is used to return HTTP header information to the client. In this case we are sending a header which triggers redirection to `helloForm.html`. Since headers must be returned before any actual output is sent, this must be one of the first things on the page. It must go before any HTML is generated or any other content returned to the client. Note that even a blank line outside of `<?php ... ?>` tags can be considered content.

Task 16.2: Update your `processHello.php` script to prevent HTML injection, and to redirect the user to `helloForm.html` if no user information is provided.

All-in-One Form Processing

There are several common patterns for PHP form processing that you should become familiar with. The first pattern, which we have just seen, is using one page to gather the form information and another page to process it. An alternative is to do the form collection and processing in a single file. The basic pattern is to check if the form has been submitted. If not, then the form is displayed. Otherwise, the form processing can proceed.

```
if (isset($_GET['some_expected_name'])) {
  // Generate the response to the form submission
} else {
  // Generate the form for submission
}
```

A slight refinement on this pattern is useful when there might be errors that arise during form processing. Such errors often mean that the form needs to be resubmitted, leading to the following pattern:

```
$formOK = false;
if (isset($_GET['some_expected_name'])) {
  $formOK = true;
  // Form processing goes in here
  // If any errors arise then two things happen:
  //  1) $formOK is set to false
  //  2) A message is reported via HTML
}
if (!$formOK) {
  // Display the form
}
```

Mixed-Mode Processing

Often PHP files have several different `<?php ... ?>` blocks. While these may look independent, it is important to remember that the whole file (including any `included` or `required` pages) is considered a single program by the web server. This means that it is possible to drop out of PHP mode in order to send some HTML through to the client.

For example, in the all-on-one form processing pattern, the form generation is usually just HTML, so we can simple leave PHP mode to do that:

```
<?php
if (isset($_GET['submit'])) {
  // Generate the response to the form submission
} else {
?>
<form name="myForm" action="<?php echo $_SERVER['PHP_SELF'];?>" method="GET">
  <!-- Rest of form goes in here as plain HTML -->
  <input type="submit" name="submit" value="Submit">
</form>
<?php } ?>
```

Some of the PHP blocks, like `<?php } ?>` may look odd in isolation, but they make sense when considered together. Note also that the action of the form is set via PHP to `$_SERVER['PHP_SELF']` rather than being hard coded—this means that the page will always submit to itself, even if it is renamed.

Task 16.3: Rewrite the hello world example as an all-in-one script. When processing the form you should check that the value entered is not empty—the test `(strlen(trim($_GET['user']))> 0)` will do this. The `trim` function removes whitespace from either end of a string, and `strlen` returns the length of a string.

16.2 Validating Form Input

While we have looked at client-side validation, it is only useful to improve the user's experience. Client-side validation cannot be relied upon to provide secure checking of form input. For `GET` requests, the values are passed through as part of the URL, and so can easily be edited. Faking `POST` requests is slightly more complicated, but still easy to do. For this reason, your PHP scripts should have a healthy suspicion (or, better yet, paranoid mistrust) of any information received from the user.

We've already seen one way to help make sure that input from the user is safely processed—the `htmlentities` function. As well as such basic data sanitisation, you should repeat any validation that you wish to carry out on the data on the server side. As with JavaScript, and many other languages, PHP offers regular expressions to help with this. As an example, consider checking to see if a username consists of letters, numbers, and underscores only, and is not empty. In JavaScript we used—

```
function isValidUSername(str) {
  var pattern=/^[A-Za-z0-9_]+$/;
  return pattern.test(str);
}
```

The syntax for PHP is very similar (they both inherit their regular expression syntax from Perl):

```
function isValidUsername($str) {
   $pattern='/^[A-Za-z0-9_]+$/';
   return preg_match($pattern, $str);
}
```

16.3 Classic Cinema Checkout Validation

Note: In this exercise, and in the next few labs, you will need to be submitting the checkout form for

validation and further processing. Filling out the form again and again can get tiresome, but there are three things to remember:

1. You can use the browser's back button to return from the validation page to the form, and the browser should remember what you had entered.
2. You can refresh the validation page to resubmit the form data. You'll get a warning from your browser because it is a POST request, but since we're not dealing with actual orders or money there's a limit to how much harm can be done.
3. You can use the form-filling script supplied on Blackboard.

As a more significant example, let's return to the Classic Cinema checkout validation. To properly test the server-side validation, we'll need to disable the client-side validation and make a few other changes. The following notes assume that you are working from the sample solution to the last set of labs. If you are working on your own solution, then some of the details will be different. We need to do the following things:

- Set the checkout form up to submit to a PHP page—in `checkout.php` (yours might be called `cart.php`) update the form's details so that its action is `validateCheckout.php` and its method is `POST`.
- Disable client-side validation. If you are using the sample solution you can do this by changing the `setup` function in `checkoutValidation.js` to—

  ```
  pub.setup = function () {
  //*
    $("#checkoutForm").submit(validateCheckout);
      $("#cardNumber").keypress(checkKeyIsDigit);
     *//
  };
  ```

  You should also ensure that HTML5 validation is disabled in the form, `novalidate`.

  We disable the client-side validation for testing, because otherwise it is difficult to pass through invalid data. Without testing on both invalid and valid data, we can't be sure our server-side validation is correct.

- Make a new PHP page to validate the form. Call this `validateCheckout.php`, and with the header and footer files from the last lab it is easy to make a new page that looks like a proper part of the site:

  ```
  <?php
    $scriptList = array('jquery-x.x.min.js');
    include("header.php");
  ?>
    <main>
      <p> Placeholder for checkout validation </p>
    </main>
  <?php include("footer.php"); ?>
  </body>
  </html>
  ```

Task 16.4: Follow the instructions above to set up a placeholder script for server-side validation of the form. Once you have done this, you should be able to click on the submit button of the checkout form and get to the placeholder page.

Now that we have the form validation page being called, the next step is to do the actual validation. Some functions to help with this are available in the file `validationFunctions.php` available on Blackboard. These functions all return true or false, depending on whether the checks that they make pass or fail.

Task 16.5: Add code to `validateCheckout.php` to validate the checkout form. You may find an incremental approach useful—add validation for one element at a time, checking for PHP errors and warnings as you go. For now the validation page should either report success or give a list of errors. We'll look at dealing with the shopping cart in the next lab.

16.4 Scripts and Security

It is often convenient to separate your PHP files into different parts, such as `header.php` and `footer.php`, or to have PHP files which contain specific functions, such as `validationFunctions.php`. This separation of content is a good thing, but these individual components don't make any sense on their own. Browsers should not be able to load these components, except as parts of 'proper' pages like `checkout.php`. This will become especially important later on, when PHP files will include sensitive information such as database passwords.

The solution to this problem is to put the include files somewhere that the web server processes can read the files, but which are not accessible to a browser. The usual way to do this is to place them in a directory outside of the directories that the Web server will allow access to (such as `/app` on `sapphire`). The Web server would still have access to this directory, but there would be no URL associated with them.

However, doing this on `sapphire` would require that the Web server would have access to your home directory. We'd rather not do this, because it would make it almost impossible for you to have a private directory to keep your work in. An alternative solution is to use `.htaccess` files. These are files which allow you to override system-level configuration of the Web server. In particular, you can use an `.htaccess` file to prevent pages from a particular directory from being served to browsers. This is done by putting a file in the directory called `.htaccess` which contains the single line:

```
deny from all
```

There are a lot of other things you can do with `.htaccess` files—see `http://httpd.apache.org/docs/2.4/howto/htaccess.html` for more information.

Task 16.6: Create a directory for the PHP files that should not be accessed directly in the Classic Cinema site, and set up an `.htaccess` file to protect them. Remember to update the paths used when you include these files in other PHP scripts.

Lab 17

PHP Cookies and Sessions

In this lab we'll look again at cookies, and see how you can retrieve and set cookie values from PHP. Since cookies are directly accessible from both JavaScript and from PHP, they are a convenient way to share information between client- and server-side scripts. We'll also look at sessions—the server-side equivalent of cookies.

Note: This lab is assessed, and is worth 1% of your final grade. Please make sure that you get a demonstrator or teaching fellow to mark it off as completed.

In the JavaScript labs we used two different ways to maintain state between visits to different pages. The first of these was cookies—a text file stored on the client which can store information relevant to a particular user's interaction with the site. The second was LocalStorage, in many ways similar to cookies.

In this lab and the next, we'll revisit methods of preserving state from the perspective of PHP. We'll access the shopping cart stored in LocalStorage, and transfer that information to an XML file storing all of the orders made on the site. We'll also introduce a new way to maintain state—the session. Later in the labs we'll look at a final method for storing information in a web application—the database. Databases provide a more structured approach to data management than file-based approaches, but we will only be able to include a brief introduction in this course. More information about database systems is provided by COSC344 and COSC430.

17.1 PHP Cookies

Cookies in PHP are accessed through the superglobal variable `$_COOKIE`, and the **`setcookie`** function is used to change their values. Here is a short PHP program that updates a counter every time a user visits the page:

```
$counter = 1;
if (isset($_COOKIE['counter'])) {
   $counter = (int) $_COOKIE['counter'];
}

echo "<p> You have been here $counter time(s) recently</p>";

setcookie('counter', $counter+1, time()+3600, '/');
```

The `$_COOKIE` variable is an associative array (or dictionary) which maps names to values. The **`setcookie`** function takes four parameters—the name of the cookie, its new value, its expiry time, and path on the server which can access the cookie. The expiry time is given in seconds, and you can get the current Unix timestamp (seconds since the start of 1970) with the **`time`**`()` command. Here the cookie is set to expire 3600 seconds (i.e., 1 hour) after being set. Setting the expiry time to 0 will make a cookie that expires when the browser closes; setting it to any other time in the past (such as `time()-3600`) will make the cookie expire immediately.

The path is optional, but for consistency with the JavaScript `Cookie` module you should use the path `/`, otherwise you may find that **`setcookie`** does not work properly alongside your client-side scripts.

Task 17.1: Make a simple PHP page with the counter code in it. Check that the counter increments with each visit to the page, and that it persists when the browser is closed and re-opened. Experiment with shorter expiry times to check that the cookie does expire properly.

The Cart, Cookies and LocalStorage

Returning to the Classic Cinema site, the shopping cart is stored in LocalStorage (originally in a cookie). When the PHP page which processes the cart succeeds, we should access the cart and do something with it. Eventually we'll copy the cart into an XML file and then clear LocalStorage, but for now we'll just report on the cart contents.

Task 17.2: Update `validateCheckout.php` from the previous lab so that on successful validation of the form the contents of the cart are retrieved and displayed.

This is where the cookie and LocalStorage solutions diverge. If the cart was still stored in a cookie then you could access it server-side as it is sent along with every page request and available in PHP via the `$_COOKIE` super-global.

With LocalStorage this is not the case: you need to incorporate into `validateCheckout.php` some AJAX that sends the data to another PHP page that will process the data and return the results.

In the following few labs the instructions for completing the task using cookies are in the boxes with the 'fork' symbol.

As the data is not automatically sent to the server (with LocalStorage) you will need to do this manually as it were by invoking a jQuery AJAX call.

Here is a function that invokes an AJAX call.

```
$.ajax({
    type: "POST",
    url: 'processCartContents.php',
    cache: false,
    data: cartData,
    datatype: 'JSON',
    contentType: "application/json; charset=utf-8",
    success: function(data) {
        alert(data);
    },
    error: function(data){
        alert("Ajax Failed");
    }
```

```
    });
```

Note that the `dataType` and `contentType` are both specified.

Task 17.3: Create a JavaScript closure called `GetCartContents.js` with the AJAX code above in the setup function and then place a link to the script from `validateCheckout.php`.

Later we will want to do something with the cart (like store it server-side) but for now we will just show that we can transfer the data to the server and display the contents.

The `data` should be the stringified array of objects just as stored in LocalStorage. The `url` needs to point to a `.php` file that exists, so make a file with the appropriate name that contains just the following—

```
<?php
   echo "Success";
?>
```

Test that the AJAX call is working by submitting the form at `checkout.php`, you should receive the "Success" alert if it is.

Task 17.4: Modify `processCartContents.php` so that it prints the cart contents.

The following line will retrieve the POSTed information and decode it to an array of objects:

```
$arr = json_decode(file_get_contents("php://input"));
```

You can then step through the array and get the values from the elements:

```
foreach ($arr as $value){
    echo $value->title;
}
```

Once that is done we are left with three problems:

- There is no HTML markup to display the items.
- The items appear in an alert rather than in the page itself.
- The items are displayed whether the validation succeeded or not.

The first is easily fixed.

Task 17.5: Modify the code in `processCartContents.php` so that the contents are printed in an HTML table.

The second is also easy.

Task 17.6: Make an empty `<div>` in the `<main>` section of `validateCheckout.php` and place the table there rather than in the alert.

Now just to stop the table displaying whenever the page is loaded and only display it when validation succeeds.

The reason that the table is always displayed is that `GetCartContents.js` is loaded every time `validateCheckout.php` is loaded. Actually you only want to do that if the validation succeeds.

Task 17.7: Change where `GetCartContents.js` is loaded so that it is only loaded if the validation is successful.

The value of the cart displayed contains the information you need, but recall that it is encoded in a JSON string. Also, the JavaScript cookie functions URI encode the cookie values, although this will not be apparent in the value you retrieve. PHP automatically URI encodes cookie values when using **`setcookie`** and then decodes them when retrieving their values.

The installation of PHP on `sapphire` also has an extension for dealing with JSON strings. The function `json_decode($string)` will take a string in JSON format and return a PHP representation of the contents. In the case of the shopping cart this will be an array of PHP objects, and each object will have two properties—a title and a price. You can access elements of the array with numeric indices starting from 0, and access properties of an object as `$object->property`. To get the price of the first item in the list we could use:

```
$cart = json_decode($_COOKIE['shoppingCart']);
$firstPrice = $cart[0]->price;
```

Since the contents of the cart is stored in an array, we can loop over all of the elements with a **`foreach`** statement. The element we get on each iteration is an object with a title and price.

Task 17.8: Update `validateCheckout.php` so that the cart contents are displayed in a table with columns for title and price, rather than as a JSON string.

17.2 PHP Sessions

Sessions are in many ways the server-side equivalent of cookies. Sessions store information about a user's interaction with a web application as they move from page to page. Unlike cookies, sessions are stored on the server, and so are more secure. However, sessions have two main limitations. Firstly, they are linked to a single interaction with the web site (usually through a cookie which expires when the browser closes), and so are not able to permanently store data. Secondly, because they are strictly a server-side mechanism they cannot be accessed from client-side scripts—this is why they are more secure.

Using Sessions in PHP

Session variables in PHP are accessed via the `$_SESSION` superglobal variable. This is very similar to `$_COOKIE` in that it is an associative array (i.e., dictionary) that maps names to values. However, in order to use this you need to first issue the command **`session_start()`**. Since this command may need to send some header information, it needs to be done before any body content is generated. The easiest way to do this is to make it the very first thing on a page.

Counting with Sessions

As a simple example, let's look at another counter. To make things a little more complex, we'll have separate counters for each of three pages. The first page for this exercise `sessionCounter1.php` is available on Blackboard.

Task 17.9: Copy `sessionCounter1.php` to somewhere in your web development directory. Load the page up and check that refreshing the page causes the counter to increase. The links to pages 2

and 3 won't work yet.

Let's take a look inside `sessionCounter1.php`. Right at the top we start a session, and then increment a session variable—the value associated with the name `counter1` is either increased by one, or set to 1 if this is our first visit. We can tell if it is our first visit, because in that case the session variable won't have been set.

The other PHP code, in each of the list items follows a similar pattern. For each of the counters (`counter1`, `counter2`, and `counter3`) we report their values. If they haven't been set, then the corresponding page hasn't been visited, so the counter is assumed to be 0.

Task 17.10: Create two new pages, `sessionCounter2.php` and `sessionCounter3.php` in the same directory as the first. These will be use `$_SESSION['counter2']` and `$_SESSION['counter3']` respectively. ✓

The new pages will be almost the same as `sessionCounter1.php`, so copies of that are a good place to start. Update the code at the top to increment the corresponding counter, and don't forget to make sure that each page has links to the other two, and to update the title and heading so that you can tell what page you are on easily.

Now, if you navigate around the pages, you should see the counters track how many times you have visited each page. If you quit the browser completely and re-open the pages, the counters should be reset, since the cookie that tracks the session will be cleared.

Sessions and the Cart

When you go to checkout from the Classic Cinema and submit some incorrect information, the form is cleared. This is annoying. Client-side scripting can help, by stopping the form from being submitted with incorrect data, but we can only be sure that things have gone correctly after server-side validation. Sessions offer a solution—we can store the form values submitted to `validateCheckout.php` in a session, and then use these to populate the form fields when we generate it in `checkout.php`.

For example, in `validateCheckout.php` we might copy the cookie value for the `name` field to the session like this—

```
$_SESSION['name'] = $_POST['name'];
```

Then, in `checkout.php` we can use this (if it is present) to fill in the value for the Name box like this:

```
<input id="name" type="text" name="name" <?php
  if (isset($_SESSION['name'])) {
    $name = $_SESSION['name'];
    echo "value='$name'";
  }
?> >
```

If the form does validate successfully, then we need to clear all of the session values, so that the form is cleared. We can clear either one session variable like this—

```
unset($_SESSION['name']);
```

—or all of them like this—

```
$_SESSION = array();
session_destroy();
```

Task 17.11: Update the Classic Cinema site so that session variables are used to remember values when the form is incorrectly submitted. Since you will be doing a similar task in many different places, you should consider using PHP functions to reduce the amount of repeated code. If the form passes server-side validation, then clear the session completely.

Lab 18

PHP and XML

In this lab we'll look at how to read, manipulate, and write XML files using PHP. We will use the SimpleXML extension which provides a basic interface for dealing with XML files. More complete XML parsers are available, see `http://www.php.net/` for more details.

Note: This lab is assessed, and is worth 1% of your final grade. Please make sure that you get a demonstrator or teaching fellow to mark it off as completed.

18.1 Storing Classic Cinema Orders

Currently when an order is made on the Classic Cinema site, nothing much happens. The order is stored in LocalStorage on the client's machine, but there is no permanent record made of it. In a real web application, a record of the order would be made in a database, the payment would be processed, and then back-end systems would pick the order up and organise for it to be fulfilled. For the purposes of these labs, we'll do something a bit simpler—we'll record the orders in an XML file and make a page where all of the orders can be viewed.

Let's get started by creating an XML file to store the orders. This will start off being empty, so is a fairly simple text file:

```
<?xml version="1.0"?>
<orders>
</orders>
```

Task 18.1: Make a new file, `orders.xml`, as described above in the Classic Cinema site directory. Make sure that the web server has permissions to read and write to this file, but take steps to ensure that it cannot be directly accessed in the browser.

Reading, Modifying, and Writing XML files

The SimpleXML extension provides routines for reading and writing XML files. Here is a basic script that copies one XML file to another:

```
// $sourceFile and $targetFile are assumed to be set
$xml = simplexml_load_file($sourceFile);
$xml->saveXML($targetFile);
```

For the Classic Cinema orders, we want to read `orders.xml`, add a new item to the list of orders, and then write it back to `orders.xml`. During the course of this lab it is likely that you will make some mistakes, and end up with a bit of a mess in `orders.xml`. Don't worry, you can always go back to an empty orders file and start over.

Let's start with what we'd like the orders file to look like once an order has been added. Suppose I ordered a copy of *Metropolis* for $19.99, and *Tarantula* for $7.99. After processing, `orders.xml`, might look something like this:

```
<?xml version="1.0"?>
<orders>

  <order>

    <delivery>
      <name>David Eyers</name>
      <email>dme@cs.otago.ac.nz</email>
      <address>University of Otago, PO Box 56</address>
      <city>Dunedin</city>
      <postcode>9054</postcode>
    </delivery>

    <items>
      <item>
        <title>Metropolis (1927)</title>
        <price>19.99</price>
      </item>
      <item>
        <title>Tarantula (1955)</title>
        <price>7.99</price>
      </item>
    </items>

  <order>

</orders>
```

Additional orders would be added as new `<order>` tags within the main `<orders>` tag. Note that the credit card information is not stored here—storing credit card information in plain text files is a bad idea. Usually what happens with online ordering is that the credit card details are used to validate a payment and then either stored in an encrypted form (for convenience) or forgotten entirely.

So how can we do this? Here's an example to get started with:

```
$orders = simplexml_load_file('orders.xml');
$newOrder = $orders->addChild('order');
$delivery = $newOrder->addChild('delivery');
$delivery->addChild('name', $_POST['name']);
// and so on...
$orders->saveXML('orders.xml');
```

The key function here is `addChild`, which makes a new XML tag as the child of another one. The initial result of loading the XML file is a reference to the root tag, i.e., `<orders>` in this case. We can then directly make a new `<order>` tag under that using `addChild('order')`. The result returned by this is a

reference to the newly created tag, which allows us to continue adding further layers as needed. We can also specify the text content of the tag as a second parameter to `addChild`.

Task 18.2: Update `processCartContents.php` (`validateCheckout.php` if you are using cookies) so that it stores the order details in `orders.xml`. While debugging you may find it easier to write the XML to a different file, so that you don't have to keep replacing `orders.xml` with a corrected version. If you do this, make sure that the file already exists and is writable by the web server. You will ned to think carefully where you are able to get the information from. The cart data (an array or Title/Price objects) arrives as part of the AJAX request but where do you get the customer details from? You can't get them from the `$_POST` array because that has made it to this script. You did however store all this information in the `$_SESSION` as part of the form validation process...

Once the order has been processed and stored in `orders.xml`, you can safely clear LocalStorage. You can't do that from PHP so you will need to do it as part of the success function in `GetCartContents.js`.

Task 18.3: Once you are confident that orders are being stored correctly, update your code so that LocalStorage is cleared. Check that after successful validation, the checkout page shows that the cart is empty, and that you can make a new order.

Once the order has been processed and stored in `orders.xml`, you can safely clear the cookie storing the order on the client side. As well as setting the cookie expiry time to be in the past, it is best to clear the value cached in `$_COOKIE`:

```
setcookie('cookieName', '', time()-3600, '/');
unset($_COOKIE['cookieName']);
```

Task 18.4: Once you are confident that orders are being stored correctly, update your code so that the cookie is cleared. Check that after successful validation, returning to the checkout page shows that the cart is empty, and that you can make a new order.

18.2 Displaying the Orders

Finally, we'll make a new page which displays the orders made on the Classic Cinema site.

Task 18.5: Make a new page, `orders.php`, for this purpose. Again, you should be able to user `header.php` and `footer.php` to easily make a new page that fits in with the rest of the site. Update `header.php` to include a link to the orders page.

To make the content of the orders page we need to iterate over all the orders, and print out the details of each one. Iterating over all the orders is quite straightforward with SimpleXML, and the SimpleXML object representation makes it easy to get specific child nodes:

```
$orders = simplexml_load_file('orders.xml');
foreach ($orders->order as $order) {
  $name = $order->delivery->name;
  echo "</p>Name: $name<p>";
}
```

We can iterate over all the orders, and easily access their sub-elements by name, and use a similar **foreach** loop to iterate over all of the items in the order. There are several other methods of SimpleXML objects that you might like to use as well. In particular the `xpath` method lets you search an XML document for nodes of a particular type. For example, to get a list of all of the `items`, regardless of which order they are in we could use `xpath` like this:

```
$xml = simplexml_load_file("orders.xml");
$items = $xml->xpath('//item');
foreach ($items as $item) {
  // do something with $item
}
```

The documentation at http://www.php.net/manual/en/book.simplexml.php has more details of the SimpleXML functions in PHP, and W3Schools has an XPath tutorial at http://www.w3schools.com/XPath/.

Task 18.6: Update `orders.php` so that it displays a list of orders, giving the delivery details and items ordered for each.

18.3 Deleting XML nodes

One thing which we have not covered, but which may be useful in the assignment is deleting nodes from and XML document in PHP. Basically this is done by unsetting the SimpleXML node, but it is not quite as simple as using—

```
unset($node);
```

SimpleXML nodes are a collection of data, and the actual node is the first element in this collection. Viewing this collection as an array, we can delete the node with—

```
unset($node[0]);
```

Using this technique and `xpath`, we can delete all of the orders in `orders.xml` as follows:

```
$xml = simplexml_load_file('orders.xml');
$orders = $xml->xpath('order');
foreach ($orders as $order) {
  unset($order[0]);
}
$xml->saveXML('orders.xml');
```

Lab 19

PHP and JSON

In this lab we'll look at how to read, manipulate, and write JSON files using PHP.

In Lab 17, you met the `json_decode` and `json_encode` functions briefly, and demonstrated that you could use PHP **`foreach`** loops to iterate over the objects returned from `json_decode`.

In this lab, we will explore PHP's JSON functionality in more detail, using the same sorts of functionality that you implemented in Lab 18, but with JSON instead of XML.

19.1 Storing JSON

We will use the JSON data from Lab 9 to demonstrate reading and writing JSON content on the server, so that changes to the data are persistent. (Note that all valid GeoJSON content is valid JSON content.)

Task 19.1: Choose or create a directory on `sapphire` in which you can place PHP scripts and access those scripts using a web browser. In that directory, create a JSON file, e.g., named `map-data.json`. Copy the GeoJSON reproduced below, into that JSON file.

```
{
  "type": "FeatureCollection",
  "features":[
     {
        "type": "Feature",
        "properties": {
           "description": "College of Education"
        },
        "geometry": {
           "type": "Point",
           "coordinates": [ 170.5199, -45.86759]
        },
        "id": 1
     },
     {
        "type": "Feature",
        "properties": {
           "description": "HCI lab"
```

```
        },
        "geometry": {
           "type": "Point",
           "coordinates": [170.51528, -45.86788]
        },
        "id": 2
     }
  ]
}
```

Reading, Modifying, and Writing JSON files

Reading a named JSON file into a PHP variable, and then writing this variable to another JSON file, can be achieved using the following PHP code:

```
$input_filename = "map-data.json";
$output_filename = "map-data-updated.json";

$json_input = file_get_contents($input_filename);
$json = json_decode($json_input,true);
$json_output = json_encode($json,JSON_PRETTY_PRINT)."\n";
file_put_contents($output_filename,$json_output);
```

In this case the code writes to a different JSON file from the one that it reads from. This is done here so that any bugs that might be in your code initially will not accidentally destroy the input JSON file. Whenever you are confident that your code is working as expected, you can of course set `$input_filename` and `$output_filename` to be the same name.

Task 19.2: Embed the above code within a PHP page so that you can run the code on `sapphire`, but also extend it to output the structure of the PHP `$json` variable into the page produced by the script. Consider using the PHP **`var_dump`** function and/or the PHP **`print_r`** function, combined with a `<pre>` tag. Also, check that the file named `$output_filename` has been created on `sapphire`, and that it contains JSON data.

The directory containing your PHP script must be writable by the webserver in order for it to be able to write the output file. By default this should work, but if it doesn't, recall that Unix permissions are covered in Lab 13.

Recall that the PHP objects created by the SimpleXML library provided facilities for you to query and modify an XML DOM tree. JSON data, however, is closer to how PHP would normally store information in variables. It is often easier to work with, for this reason.

Note that the second parameter to the `json_decode` call above, with value **`true`**, instructs PHP to parse the JSON data into nested associative arrays. When that parameter is **`false`** (or the parameter is omitted), then the JSON is parsed into PHP objects rather than associative arrays..

If the `json_decode` function call is successful, you can navigate your way through the nested associative arrays that match the structure of the data in the JSON file. For example `$json["features"]` will be a list of GeoJSON features, and `$json["features"][0]` will be the first feature in the list of features. In the data file used in this exercise, to give a complete example `$json["features"][1]["type"]` will be the string "Feature", which is visible on the fifth line of the JSON data, above.

Task 19.3: Copy your PHP script from the previous task into a new PHP file. Using what you have learnt from your scripts outputting the value of the `$json` variable in the previous task, make a modification to the value of the `$json` variable so that the name of the second feature in the GeoJSON has both words capitalised (i.e., "HCI Lab"). Check that your output JSON file on `sapphire` contains the updated version of the GeoJSON.

As usual, when dealing with PHP associative arrays, you can add new data by assigning values to keys that have not been used yet. To remove items you can use the **`unset`** function. For example. `unset($json["features"][0]);` would remove the GeoJSON entry for the College of Education.

Task 19.4: Duplicate your PHP script for the previous task, and extend your script to add a new GeoJSON entry for a building on campus. For example, the Owheo building is somewhere near longitude, latitude coordinates `[170.518204,-45.866727]`. Check that your resulting GeoJSON file contains the three expected features.

19.2 Using the JSON data to build an HTML page

Since `json_decode` creates standard PHP data structures—associative arrays in the examples in this lab—displaying structured HTML pages from JSON content is no different from displaying any other PHP data. Nonetheless, deriving a structured HTML page from server-side JSON data is an appropriate way to close off this lab exercise.

Task 19.5: Write a PHP script that reads in the above GeoJSON, and presents the GeoJSON features contained within it, using an HTML table that has columns for description, longitude and latitude.

For the assignment: This lab covers the last of the material you will need to complete the basic requirements of Assignment 2. The later labs may be useful for extension work, but are not required for the must-have functions.

Lab 20

Getting Started with MySQL

Starting in 2020 we will transition to developing PHP and using MySQL on your own 'Docker' containers as opposed to you working on the dedicated, shared web server (`sapphire.otago.ac.nz`) that we used in past years. This material may need to be updated during the year when we determine what works best within the Department of Computer Science labs.

In this lab we will begin a brief look at relational databases, and MySQL in particular. We won't be going into much detail, just enough to set up a table of usernames and passwords that we can use for authentication. Much more detail about database systems is taught in COSC344 and COSC430.

Note: This lab is assessed, and is worth 1% of your final grade. Please make sure that you get a demonstrator or teaching fellow to mark it off as completed.

20.1 Connecting to MySQL

To get started with MySQL, log in to `sapphire` using the shell. You can then connect to the MySQL database server that is running on `sapphire`. The command to do this, like most Unix commands, has a range of options which are introduced with `-` (i.e., a hyphen).

A common form of command invocation is explained below as you may see such invocations used elsewhere on the Internet, particularly when the MySQL database is already set up for you. Actually the invocation we will use will be much simpler, since you have complete control over your own private instance of MySQL within your container.

```
mysql -h some-server -D database-name -u username -p
```

The options above would have the following meaning:

- `-h` specifies which host you wish to connect to. In this case we would be specifying connecting to the database server running on a computer named `some-server`. It is common to run the database server on a different physical computer from the web server as they have different resource requirements and thus can be best optimised separately.

- `-D` specifies which database you wish to use on that server. For example, an organisation might provide separate databases for development and for production.

- `-u` specifies the username you wish to use to connect to the database. Often your database username will be different from your Unix account username.
- `-p` specifies that you will be expecting to provide a password.

In our case, the invocation from a Unix shell on your container is far simpler. You just need to run `mysql`. Because you are accessing the default Unix shell on your container as the `root` superuser, and connecting to the instance of MySQL running on that virtual server, `mysql` will not ask you for a password.

20.2 Create a Database

MySQL is a database server on which you can manage multiple independent databases. Before we begin creating relational database tables and populate those tables with data, we need to create the containing database.

Open a Unix shell on your container, and run the `mysql` command. Below a welcome message that indicates the version of the MySQL server that is running, you should see the MySQL command prompt:

```
mysql>
```

At the MySQL command prompt, issue the command to create a database named `webdb`. (The name does not really matter provided that you remember what it is, and use it consistently.) Also run the command that follows, which instructs MySQL to connect to the database that you just created. SQL statements in MySQL end with a semicolon (strictly speaking, the `USE` command doesn't need one but it doesn't hurt to add one anyway).

```
CREATE DATABASE webdb;
USE webdb;
```

Now that you have created a database, you can either run `mysql` and then the appropriate `USE` command. You can also add the database onto the command line invocation of `mysql`, which means you won't need to run a `USE` command, as you will have specified which database to connect to. For example:

```
mysql -D webdb
```

20.3 Create a Database User

It is common for database systems—particularly MySQL—to manage their own database-internal notion of a user with privileges, that is independent of the set of Linux users on a given computer. This can be useful in cases such as when one single Linux user might want to run multiple software systems that should all access isolated databases. For example, web application developers often want to run a development and a production system. Ideally beyond having two separate databases, it is useful to have two separate (database) users, so that there is less chance of interference of their database administration.

We will now create a web user for the database with a known password—the text content within and not including the single quotes that follows the `IDENTIFIED BY` keywords. On shared Linux servers, the password needs to be kept private. However only you have access to the Linux server running in your container, so we need not worry about others on the Internet discovering the username and password details from this lab book, and then attacking your database.

```
CREATE USER 'myphpscripts'@'%' IDENTIFIED BY 'Correct Horse Battery Staple';
GRANT ALL PRIVILEGES ON webdb.* TO 'myphpscripts'@'%';
```

This particular database user only has permission to work on the `webdb` database. So far you have been connecting to MySQL using the database root user, that is all-powerful. We created the less-powerful `myphpscripts` user since we will later use these credentials to access your MySQL database from within PHP scripts that you write. Ideally these PHP scripts will not have all-powerful access to your MySQL database server, since they do not need all-powerful access to operate. You can connect to your database on the command-line using the less-powerful `myphpscripts` user as follows, noting that MySQL will prompt you to enter the password shown within the preceding SQL listing:

```
mysql -u localhost -D webdb -u myphpscripts -p
```

20.4 Creating a Users Table

MySQL, like many popular database systems is a *relational database*, and uses SQL as the main language to manipulate data. We won't be going into the theory here, but will work through some examples to get a basic list of users and passwords up and running. Data in a relational database is stored in mathematical objects called relations (hence the name), but conceptually these are tables with rows and named columns. A simple users table might have columns for the username and password:

username	password
Adam	SuperSecret
Nick	NeverGuessThis

We can make a table like this with the following SQL command:

```
CREATE TABLE Users (
   username VARCHAR(255) NOT NULL,
   password CHAR(40) NOT NULL,
   PRIMARY KEY (username)
);
```

The **`CREATE TABLE`** command takes a list of columns and constraints. Here the two columns are `username` and `password`. Both columns are given a type, and are declared **`NOT NULL`**, which means that a value must be supplied for them. The type of `username` is **`VARCHAR`**`(255)`, which means a string of up to 255 characters (which should be long enough). The `password` column's type is a string where room for exactly 40 characters is reserved. The reason for the 40 character limit will become clear soon.

We also provide a constraint on the `username` column, by declaring it to be the **`PRIMARY KEY`** for the table. A primary key is used to uniquely identify rows in the table, and so must be unique, and cannot be NULL (i.e., an actual value must be given).

Task 20.1: Create the `Users` table in MySQL using the SQL command above.

Adding Users to the Table

We can now add a user to the table like this—

```
INSERT INTO Users (username, password) VALUES ('me', 'secret');
```

—and see what is in the table like this—

```
SELECT * FROM Users;
```

The * here means 'all columns'. Alternatively you can provide a comma-separated list of column names:

```
SELECT username FROM Users;
SELECT password FROM Users;
SELECT username, password FROM Users;
```

You can also provide restrictions on which row(s) are returned:

```
SELECT password FROM Users WHERE username='bob';
```

The **WHERE** clause specifies a condition, and only those rows that meet the condition are returned.

Task 20.2: Add a few hypothetical users into your `Users` table. What happens if you try to add two users with the same `username`? Try out some **SELECT** statements with different conditions in the **WHERE** clause.

More Secure Password Storage

Storing passwords as plain text in a database (or anywhere else) is a bad idea. A better idea is to store a *cryptographic hash* of the password, rather than the password itself. A cryptographic hash is a value computed from some text (the message) that has the following properties:

- It is easy to compute the hash from the message
- It is *very* hard to do any of the following:
  - Compute a message that gives a given hash
  - Modify a message without changing the hash
  - Generate two messages that have the same hash

Cryptography is a whole subject area in itself, but for now we will use the SHA-1 (Secure Hash Algorithm), which generates 40-digit hexadecimal hashes of strings. This is why our password column was set to 40 character strings.

SHA-1 is not particularly secure, but will serve for our purposes, and is supported natively by MySQL and PHP. You can compute the SHA-1 hash of a string in MySQL using the `SHA` function:

```
SELECT SHA('Some string');
```

Even very similar strings have very different SHA values:

```
SELECT SHA('String number 1');
SELECT SHA('String number 2');
```

We can now clear out our Users table, and add some more secure information:

```
DELETE FROM Users;
INSERT INTO Users (username, password) VALUES ('me', SHA('secret'));
SELECT * FROM Users;
```

Be very careful with SQL commands such as **DELETE FROM**, as they can cause large-scale data loss, and there is no straightforward undo facility.

Task 20.3: Clear out the `Users` table and add a few users with different passwords using SHA encryption.

20.5 Validating Users and Passwords

The main task which we will want to do is to validate a username and password against the database. If we are supplied a username and password (such as from a form on a website), we can check if it is in the database as follows:

```
SELECT * FROM Users
 WHERE username = 'the_username'
   AND password = SHA('the_password');
```

Here `the_username` and `the_password` are the values supplied. This will return exactly one row if the username-password pair is in the database (remember, usernames are unique), and zero rows otherwise.

Task 20.4: Try out some SQL commands to validate different usernames and passwords against your Users table. You should check when the username and password are both correct, both incorrect, or when either one alone is correct.

20.6 Adding Information to the Table

While username and password columns are sufficient to do basic authentication, often we want to store more information in the `Users` table. We could destroy the whole table and make a new one with extra columns, but it is easier to just add new columns. You can do this with the **`ALTER TABLE`** command. For example—

```
ALTER TABLE Users ADD COLUMN email VARCHAR(255);
```

—will add a column called `email`, that will accept strings of up to 255 characters to the `Users` table. The new columns will have missing (**`NULL`**) values for any existing rows. You can add values using the **`UPDATE`** command:

```
UPDATE Users
   SET email='dme@cs.otago.ac.nz'
 WHERE username='dme';
```

You can also use **`UPDATE`** to alter existing values, such as changing a password:

```
UPDATE Users
   SET password = SHA('new_password')
 WHERE username = 'dme';
```

You should be very careful to have the right **`WHERE`** condition in **`UPDATE`** statements. If there is no condition given then *all* rows in the table will be updated. It is often worth running a **`SELECT`** statement with the same condition first, to make sure it returns the rows you expect.

Task 20.5: Update the `Users` table to have an email column, and set email values for the existing users.

Lab 21

MySQL and PHP

In this lab we will see how to connect to a MySQL database from a PHP page. We'll see how to add new users based on a registration form, and how to prevent SQL injection attacks.

Note: This lab is assessed, and is worth 1% of your final grade. Please make sure that you get a demonstrator or teaching fellow to mark it off as completed.

To get started, we need to create a page with a form for users to register with the Classic Cinema site. This page should ask for the username, password, and email address (as well as any other columns you've added to the `Users` table).

Task 21.1: Create a new page, `register.php`, in the Classic Cinema site. Put a form on this page asking for a new user's details. You should provide a link to the registration form on each page—beside the Login button is the typical place for this link.

21.1 Connecting to MySQL from PHP

When we receive the contents of this form, we first need to check to see if the username is already in use. We can do this with a `SELECT` statement:

```
SELECT * FROM Users WHERE username = 'the_username';
```

This should return one row if the username is already used, and none if it is not. To make this check from PHP we need to do several things:

- connect to the database by creating a `mysqli` object;
- make a PHP string that has the SQL query we want to make in it;
- send this through to the database and get the result back;
- check to see how many rows are in the result; and
- free up the resources (the result object and the database connection) that we've used.

The PHP code to do this looks as follows:

```
$conn = new mysqli('localhost', 'myphpscripts', 'Correct Horse Battery Staple',
    'webdb');
if ($conn->connect_errno) {
  // Something went wrong connecting
}

$query = "SELECT * FROM Users WHERE username = 'new_username'";
$result = $conn->query($query);

if ($result->num_rows === 0) {
  // OK, there is no user with that username
} else {
  // Problem -- username is already taken
}
$result->free();

$conn->close();
```

Note that there are two usernames here—the one you use to connect to the database (`myphpscripts`) and the one you are trying to add to the database (from the form). Also remember that SQL uses single quotes (') around strings, so using double quotes (") around PHP strings that contain SQL lets you mix and match them more easily.

Task 21.2: Update `register.php` so that when a user submits the form the page checks to see if the username is available. For now, just report a message saying which case has occurred. This form would be a good candidate for all-in-one form processing as discussed in Section 16.1. Also, the code that connects to a database should be secured as much as possible from unauthorised access. Placing this in a separate script in an `.htaccess` protected directory is a good way to do this.

As well as checking to see if the username is available, you may wish to add some other verification to the username and/or password. You should be careful with this, however, as many rules annoy users and may add little to security. A simple but useful validation for passwords is that they should be quite long (8 characters minimum is common) and contain at least two or three categories of characters (uppercase and lowercase letters; numbers; punctuation characters). At a minimum you should check that the username and password aren't blank.

Another common validation is to ask the user to confirm their password. Since password fields do not display the text being typed, it is easy for mistakes to be made when entering a new password. Asking the user to type their new password twice helps to reduce these errors.

21.2 Adding a New Entry into your Users Table

Once we have an acceptable username and password, we need to add a new entry to the database. We already have a connection open to the database through the `mysqli` object, and we can use that to insert a new row:

```
$query = "INSERT INTO Users (username, password, email) " .
         "VALUES ('new_username', 'new_password', 'new_email')";
$conn->query($query);
if ($conn->error) {
  // Something went wrong
}
```

Encrypting the Password

The password needs to be encrypted before storage in the database, and we could either do this using SQL:

```
$query = "INSERT INTO Users (username, password, email) " .
         "VALUES ('new_username', SHA('new_password'), 'new_email')";
```

or in PHP:

```
$password = sha1('new_password');
$query = "INSERT INTO Users (username, password) " .
         "VALUES ('new_username', '$password', 'new_email')";
```

Task 21.3: Add code to `register.php` to store new users in the database. To check that it has worked, you can connect directly to the MySQL database from the command line, and `SELECT` all the rows from the Users table.

21.3 SQL Injection Attacks

Passing information from a web form into SQL raises the issue of an injection attack. SQL injection attacks can be particularly damaging, since data is often a web-based companies most valued asset, and privacy breaches can have far reaching effects.

To get started, let's look at how easy injection attacks can be against unprotected systems. We'll look at a simple page to update passwords in the Users table. There is a script to do this at https://altitude.otago.ac.nz/cosc212-share/php-sql-injection/-/blob/master/injection.php. Check the details at the top of the file, but they should match the details used in previous lab tasks. When the form is submitted the form reports the query that is run and the number of rows that are updated so that you can see what is going on.

Task 21.4: Take a copy of the above PHP script and name it `injection.php`. Check that you can update passwords with this script—if your database structure differs from that in the instructions, you may need to change the SQL query created by this page. Make sure that you have more than one user in the database.

Now let's conduct an injection attack. Enter `user'OR 'x'= 'x` as the username, and any value in the password field. How many rows are updated?

If you go in to the MySQL database and look at the contents of the Users table, everyone should have the same password. This has happened because the `WHERE` clause of the SQL statement which updates the password has been set to—

```
WHERE username='user' OR 'x' = 'x'
```

—which is always true, so all rows have been updated.

This means that the attacker now has knowledge of everyone's passwords—they have all been set to the value entered in the form. This is a bad thing.

Injection Prevention

There are several ways to avoid this problem. The best way is often to use a technique called *prepared statements*, but that is a bit more than we need for this course. A simpler (but not as robust) method is to use the `mysqli->real_escape_string()` method. This is similar to the use of URI encoding in cookie data or the PHP **`htmlentities`** function to prevent HTML injection attacks:

```
$safe_string = $conn->real_escape_string($unsafe_string);
```

If you need to reverse the process, you can use the **stripslashes** function:

```
$original_string = stripslashes($safe_string);
```

The `real_escape_string` function encodes special values with an escape character (the slash, \). This makes strings safe to use in SQL statements, and we can easily update `injection.php` so that the lines

```
$password = $_POST['newPassword'];
$username = $_POST['username'];
```

become

```
$password = $conn->real_escape_string($_POST['newPassword']);
$username = $conn->real_escape_string($_POST['username']);
```

which should prevent this sort of attack.

Task 21.5: Make suitable changes to `injection.php` and your registration code to protect against SQL injection.

Lab 22

Authentication and Sessions

In this lab we'll implement the login form on the Classic Cinema site. We'll see how to use server-side scripting to authenticate against the database, and how sessions can be used along with this authentication to restrict access to parts of the site.

Note: This lab is assessed, and is worth 1% of your final grade. Please make sure that you get a demonstrator or teaching fellow to mark it off as completed.

In this lab, we'll be adding login functions to the Classic Cinema site, using the Users table developed in the previous labs. To get started, we'll remove the CSS styling that hides the logout form. This will make both the login and logout forms visible. Don't worry, we'll fix this later in the lab.

Task 22.1: Update the login and logout forms so that they redirect to relevant PHP pages (`login.php` and `logout.php` would be good names). The login form (at least) should use the POST method—we don't want people bookmarking their login credentials.

Before we get started, let's take a broad look at how authentication is going to work:

- When a user logs in, their credentials will be compared against the database. If they match, a session variable will be set with the name of the user.
- We can then check the session variable to control access to content on the site:
  - Sometimes whole pages might be blocked. For example, we might not allow access to the registration page when logged in.
  - Sometimes specific functions might be enabled or disabled. For example, we might only allow users to rate films if they are logged in.
- The logout script will clear the session variable.

22.1 Logging In

First, let's make a page that logs the user in and sets a related session variable. This page has to do the following things:

- Connect to the MySQL database

- See if the username and password entered by the user matches an entry in the `Users` table.
- If so, log the user in with the username specified. If not, then report an error message.

We've already seen how to get information from a form in PHP (Lab 16), how to validate a username and password (Lab 20), and how to connect to the database from PHP (Lab 21). Putting these together should allow you to make a page which checks a username/password pair against the database and reports success or failure.

Task 22.2: Write `login.php` so that it responds to the login form by validating a username and password against the database, and reports success or failure. Don't worry too much about formatting the HTML content of the response—we'll see soon how to have the login redirect back to the originating page.

Logging in for a Session

The script you've just written authenticates the user. We now need to record the fact that the user is logged in across the site. The easiest way to do this is with a session variable:

```
$_SESSION['authenticatedUser'] = $username;
```

where `$username` is the username that has been successfully authenticated with a password.

Task 22.3: Update your script, `login.php`, so that it sets a session variable on successful login. Don't forget to call **`session_start()`** at the beginning of the script.

To use this session variable across the site we need to include **`session_start()`** at the top of every page that will make use of it. Since `header.php` is included in every page on the site, before any content is produced, we could put **`session_start()`** at the start of that, or we could call it individually for each page.

This leads to a minor issue that we may need to be sure in an included script that **`session_start()`** has been called. We can do this by checking whether there is already a session, and if not then starting a new one:

```
if (session_id() === "") {
  session_start();
}
```

Redirecting to the Last Page

Currently the login page displays success or failure. Ideally we'd like the login functions to work without users having to go to a login page and then back to the main site. To achieve this, the login script (regardless of success or failure) should redirect the browser back to the originating page rather than returning a page of its own. We've already seen how to redirect pages, with the following code:

```
header('Location: somepage.php');
exit;
```

The problem is determining what page to redirect the user to. One source of the previous page is the server variable `$_SERVER['HTTP_REFERER']`, which is supposed to be the page we want. However, this cannot be relied upon. Not all browsers set this correctly, and some can be configured to disable this function.

An alternative solution is to store the last page visited in a session variable, and the use that. Since this is under our control, and on the server, it is more reliable. At the start of each page (for example, in `header.php`) we can set:

```
$_SESSION['lastPage'] = $_SERVER['PHP_SELF'];
```

We can then use `$_SESSION['lastPage']` to determine where to return the user after logging in.

Task 22.4: Update `login.php` to redirect the user back to the page where they filled in the form once the form has been processed. If the session variable is not set, redirect them to `index.php`.

22.2 Restricting Access

Once we have the authenticated username stored in a session, we can use that to restrict which parts of the site we show. For example, if the username is set we want to show the logout form and not the login form. If the username is not set, then we want just the login form. We can easily do this as follows:

```
<?php if (isset($_SESSION['authenticatedUser'])) { ?>
  <!-- HTML to display the Welcome message and logout form -->
<?php } else { ?>
  <!-- HTML to display the login form -->
<?php } ?>
```

In addition to the login/logout forms, the registration link should not be available to users who have logged in. As well as removing the link if the user has logged in, you should prevent direct navigation to `register.php`. This can be done by checking if `$_SESSION['username']` is set when the page loads. If it is, then the user is already logged in and you can redirect them to another part of the site (such as the index, or the previous page).

22.3 Logging Out

The script to log users out of the page is quite simple—we clear the session variable and then redirect the user back to the originating page. Recall that we clear a session variable by **`unset`**ting it:

```
unset($_SESSION['username']);
```

Task 22.5: Update your Classic Cinema site so that only the login or logout form is displayed (as appropriate), and that the registration page is not accessible when a user is logged in. Make sure to fill in the Welcome message with the current users' username. Also update the site so that the checkout, validateCheckout, and orders pages are not available unless you have logged in. You should also remove the navigation links to checkout and orders when users are not logged in, replacing them with a suggestion to log in.

Since this is another example of a facility that may be required on multiple pages, you might want to factor the code for checking authentication out into a separate script.

Lab 23

More Authentication

In this lab we will extend the authorisation developed in the previous one to allow for special administrative access to some parts of the Classic Cinema site.

Note: This lab is assessed, and is worth 1% of your final grade. Please make sure that you get a demonstrator or teaching fellow to mark it off as completed.

23.1 Restricting Access to Orders

In the previous lab we restricted some parts of the site to users who were (or were not) logged in. This is useful, but sometimes we need finer-grained access control. For example, the orders list is currently visible by anyone logged in to the site. Most people do not need to see the list of *all* orders made on the site. We will first restrict access to `orders.php` to user(s) who have special privileges, and then relax this so that users can see their own orders when logged in.

Updating the Users Table

Firstly we need to identify which users have special access rights. We can do this by adding a new column to the `Users` table to record the type of access each user has. For the purposes of this lab we'll just need two types of access—normal users and administrators who can do a bit extra.

Task 23.1: Update the `Users` table (using an **`ALTER TABLE`** statement at the MySQL console) so that it has a new column called `role` of type **`CHAR`**`(5)`. Set the role of most users to `user`, but one or two users should have the role of `admin`. You may wish to add a new user to the database specifically for this purpose.

It's OK to manually update the existing users, but what about new users created through the registration page?

Task 23.2: Update `register.php` so that new users are given the role of `user`. This should be a simple change to the **`INSERT`** statement used to put the new user in the database.

Retrieving the User's Role

Recall that when logging in we query the database to see if the given username and password exists. Your code to do this should look something like this:

```
$query = "SELECT * from USERS" .
         " WHERE username = '$username'" .
         "   AND password = SHA('$password')";
$result = $conn->query($query);
if ($result->num_rows === 0) {
  // Bad username or password
} else {
  // Good - log in
}
```

In order to determine the role of the user we need to look inside the result of the query, rather than just checking if there is one. The result of the query is a table—a set of rows. In this case the query will either have no rows (if the username and password don't match any entries), or exactly one row. If there is a row, we can fetch it as an associative array as follows:

```
$row = $result->fetch_assoc();
```

The resulting row variable can be indexed by column name to give the value of that column. The role of the user can therefore be retrieved as

```
$role = $row['role'];
```

Task 23.3: Update the login script so that the current user's role is stored in `$_SESSION['role']`. You should also make sure that logging out clears the role from the session variable.

Checking to see if this stage is working is a bit tricky—it happens behind the scenes. For debugging purposes you may wish to display the role alongside the username in the Welcome message.

Restricting Access to the Orders Page

Now we can restrict access to the orders page—if the user heading to the page doesn't have the role of `admin`, they can be redirected to `index.php`

Task 23.4: Update `orders.php` so that only admin users can view the page. Normal users should be redirected to `index.php` for now. Don't remove the link to orders in the navigation for ordinary users though, we'll be needing that soon.

23.2 Even Finer Order Control

As it stands, the orders page is restricted only to administrators. This is probably too harsh—only administrators should be able to see *all* orders, but users should probably be able to review their own orders. To enable this we need to do two things:

- We need to record which user made each order.
- We need to use this information to filter the orders for display.

Task 23.5: Update `validateCheckout.php` to add the username to the XML that stores the orders. You do not need to collect the username in the checkout form—you can use the value from the session variable.

Note that older orders will not be associated with any username. To fix this we need to clear the history of orders and add in a few new ones that do have this information.

Task 23.6: Edit `orders.xml` so that it just contains the outer `<orders>` tags, and no individual order information. Make a few orders through the site, with different users. Make sure that the correct username is associated with each order, and that there you cannot make an order without being logged in.

Filtering the Orders

Finally we need to update `orders.php` so that it behaves as follows:

- If there is no user logged in `orders.php` should not be accessible—it should be removed from the navigation links, and direct access should be prevented.
- If the current user is of type `admin` then it should show a list of all orders on the site.
- If the current user is of type `user` then it should show just the orders associated with the current user, or a message along the lines of "You have not made any orders".

Displaying the orders for a given user is fairly straightforward. We can retrieve the username for each order, just like the other information stored in the XML file. This can then be compared to the username value stored in the session variable, and the order detail displayed only if the two are the same.

Task 23.7: Update `orders.php` so that it has the behaviour described above.

23.3 Adding Reviews

As a last piece of functionality we'll add the ability for logged-in users to add reviews to the website. This will be a simple form added after each movie with just two visible elements: a drop-down box where the user can select a rating from 1–5, and a button to submit the form.

When submitted, this form will add an entry to the appropriate XML file. To do this it needs to know what the name of the XML file is, and the name of the current user. You can use the `$_SESSION` variable to get the name of the user, but the name of the XML file will need to be provided as a hidden form element. Since the form for each film will be the same, apart from the XML file name, it makes sense to write a PHP function to create the relevant HTML:

```
function addReviewForm($xmlFileName) {
  if (isset($_SESSION['authenticatedUser'])) {
    echo "<form action='addReview.php' method='POST'>";
    echo "  <input type='hidden' name='xmlFileName' value='$xmlFileName'>";
      // Rest of the form goes in here
    echo "</form>";
  }
}
```

The script `action.php` will then use the form information, and the username stored in `$_SESSION` to update the relevant XML file. Remember from Lab 11 that one *The Man Who Knew Too Much* does not have an XML file, so you may want to make one for that film.

Task 23.8: Add the ability for users to add review to films on the Classic Cinema website. Make sure that the ability to add reviews is limited to users who are logged in.

Lab 24

Catch-Up/Assignment

There are no new exercises for this lab. The second assignment is due at the end of this week, however.

If you have finished all of the lab work and the assignment, you may want to think about how the Classic Cinema site could be improved. Some possibilities include:

- Using the Post-Redirect-Get pattern for pages that alter information.
- Replacing the XML-based order system with a database table.
- More generally, driving the content (categories, reviews, films) from database tables rather than XML and HTML files.
- Adding more back-end functions so that administrators can manage orders and track their progress from submission to completion.

Alternatively you might want to think about how the techniques you've learned about in this course could be applied to other web applications, or to a problem that is of personal interest to you.

Lab 25

Catch-Up/Assignment

There are no new exercises for this lab. The second assignment is due at the end of this week, however.

If you have finished all of the lab work and the assignment, you may want to think about how the Classic Cinema site could be improved. Some possibilities include:

- Using the Post-Redirect-Get pattern for pages that alter information.
- Replacing the XML-based order system with a database table.
- More generally, driving the content (categories, reviews, films) from database tables rather than XML and HTML files.
- Adding more back-end functions so that administrators can manage orders and track their progress from submission to completion.

Alternatively you might want to think about how the techniques you've learned about in this course could be applied to other web applications, or to a problem that is of personal interest to you.