Developing Web Applications

Laboratory Sessions for COMP212
Department of Computer Science
University of Otago

Semester 2, 2016

Contents

Part I

Introduction

Chapter A

COMP212 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. There are several catch up labs scheduled to help with this, and sample solutions will be released after each of those. As a result, we will not accept labs for assessment after the following catch up labs. For example, assessed labs up to Lab 7 will not be accepted for marking after catch up Lab 8.

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, and are hosted on Unitube (`http://unitube.otago.ac.nz/`). 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 select papers to construct a course of study at university. 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 11:59 p.m. August 26 and 11:59 p.m. October 7, 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 COMP212. 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.

Assessment Summary

Assessed Component	Due Date	Marks
Assessed Labs	Special	16%
Assignment 1	11:59 p.m. August 26	17%
Assignment 2	11:59 p.m. October 7	17%
Final Exam	TBA	50%
TOTAL		100%

Assessed Labs will be accepted for marking up until the following catch up labs. For example, Labs 3 to 7 will be marked up until 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 Mills. 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 plagiarism which are available from `http://www.otago.ac.nz/study/plagiarism/`.

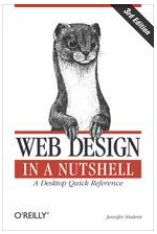
A.3 Workload

COMP212 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 COMP212 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

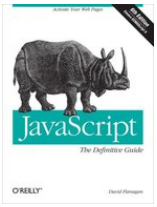
There is no single text for COMP212, but there is a range of useful resources. Several books are given below, which are available through Safari Books Online. Otago has a subscription to this service, so you should be able to reach these resources from any university computer. Individual resources will 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 COMP212. It provides information about HTML and CSS as well as an introduction to JavaScript.



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

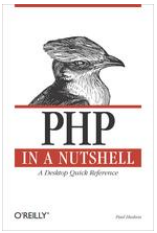
http://my.safaribooksonline.com/9781449393854

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	11 Jul	Introduction	HTML and CSS Revision
		Servers and Networks	Getting Started With Unix
2	18 Jul	JavaScript 1	Hello, JavaScript (1%)
		JavaScript 2	JavaScript Arrays and Objects (1%)
3	25 Jul	JavaScript 3	JavaScript Closures
		JavaScript 4	JavaScript Cookies (1%)
4	1 Aug	HTTP and Networks	JavaScript Form Processing (1%)
		jQuery 1	Catch Up
5	8 Aug	jQuery 2	Introduction to jQuery (1%)
		jQuery 3	jQuery, Ajax, and XML (1%)
6	15 Aug	Deployment scripting	jQuery Animation (1%)
		Clients and Servers	Managing Deployment (1%)
7	22 Aug	CGI Scripting	Version Control Systems
		Revision 1	Catch Up
	29 Aug	Break	
8	5 Sep	PHP 1	Hello, PHP (1%)
		PHP 2	PHP Form Processing (1%)
9	12 Sep	PHP 3	PHP Cookies and Sessions (1%)
		Intro to Databases	PHP and XML (1%)
10	19 Sep	PHP + Databases	Getting Started with MySQL (1%)
		Security - Authentication	MySQL and PHP (1%)
11	26 Sep	Security - Injection	Authentication and Sessions (1%)
		Security - Encryption	More Authentication (1%)
12	3 Oct	Alternative Technologies	Catch-Up/Assignment
		Case Study	Catch-Up/Assignment
13	10 Oct	Revision 2 + 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 notes – particularly the lecture scheduled just before the lab, but again material presented early in the course is still relevant later on.

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 is *incremental development*. You start with the most basic working example 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 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: 11:59 p.m. August 26. Worth: 17%

It is very important that you protect your assignment work from unauthorised copying. We have set up secured directories on `sapphire` for this purpose. You should put *all* files related to your assignment work under `/devel/username/projects/assignments`. This directory is password protected, in order to prevent other students from copying your work.

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 assignment is to build a Web application for a round-robin tournament. Such tournaments are common in the initial stages sporting fixtures. In a single round-robin schedule, each team or player has one match against every other team or player.[1] For example, in Group F of the UEFA Euro 2016 competition, there are four teams: Austria, Hungary, Iceland, and Portugal. Each team plays the other three once, and there are six games in total.

Teams score points for each game, which are separate from their score in that game. For the purposes of this assignment we'll assume that a team earns 2 points for a win, 1 for a draw, and 0 for a loss.[2] For example, in Group F of UEFA Euro 2016, Hungary:

- Won 2–0 against Austria, earning 2 points
- Drew 1–1 against Iceland, earning 1 point
- Drew 3–3 against Portugal, earning 1 point

for a total of 4 points.

In the case of two teams having the same number of points, the total score against ('Score Against') them is subtracted from their total score ('Score For') in all games. For example, Hungary above scored a total of $2 + 1 + 3 = 6$ goals and their opponents scored a total of $0 + 1 + 3 = 4$ goals against them, for a difference of $+2$.

Iceland also won one game and drew two, for 4 points, but they scored a total of 4 goals, and their opponents a total of 3, for a difference of $+1$. Since Hungary had a higher score difference, they ranked first in the group.

[1] In many competitions each team plays the others twice – once at home and once away. In the ANZ Netball Championship each NZ team plays the other NZ teams twice and the Australian teams once, and vice-versa. We will ignore these complications to keep the assignment simple.

[2] UEFA Euro 2016, and other games where draws are common give 3 points for a win. Other competitions, such as the Super Rugby award points for close losses and other aspects of the game. Again, we will ignore these complications.

Here are the outcomes of all the matches in Group F:

- Hungary won 2–0 against Austria
- Portugal and Iceland drew 1–1
- Iceland and Hungary drew 1–1
- Portugal and Austria drew 0–0
- Iceland won 2–1 against Austria
- Hungary and Portugal drew 3–3

This gives a final ranking (again, assuming 2 points for a win) as shown in Table C.1.

Rank	Team	Played	Won	Drawn	Lost	For	Against	Diff.	Points
1	Hungary	3	1	2	0	6	4	+2	**4**
2	Iceland	3	1	2	0	4	3	+1	**4**
3	Portugal	3	0	3	0	4	4	0	**3**
4	Austria	3	0	1	2	1	4	−3	**1**

Table C.1: Group F rankings for UEFA Euro 2016.

The assignment is to build a Web application that displays the results of a round-robin tournament. Assignment 1 provides an interface where people can see the results of the games and the ranking table. You will also provide a form where administrators can update the information about the games, including the scores, dates, and venues for games. At this stage this form will not result in changes to the underlying data, as that requires server-side scripting and is the focus of Assignment 2.

The data for the assignment is driven from XML files, and sample files are available from Blackboard in `Course Documents` → `Assignment Files` → `Assignment XML files`. Note that the information provided in these XML files is an example only, and your application should reflect any changes to these files. There is one XML file for the tournament results and one listing the possible venues for each match:

- `venues.xml` Lists the possible venues for matches.
- `tournament.xml` Lists the matches in the tournament, including the team names, the venue for each game, and the scores of completed games. Note that some games may not have been played, and these will have no score provided.

C.1 Requirements

Your Web application must allow people to:

- See the results of the matches in the tournament, including which games have been played, the scores, and the dates and locations of upcoming matches.
- See a ranking table of the teams based on the games that have been played. This table should have the columns shown in Table C.1.
- Provide an administrative page or pages where the results of matches can be entered, and the dates and locations of matches can be set.

- The administrative pages need not update the XML (that comes in Assignment 2) but should apply at least the following validation rules:
  - Each match must be in one of the venues listed in `venues.xml`
  - If there are two games on the same day, then they must be in different venues.
  - A team cannot play two games on the same day.
- On successful validation, a page should be displayed summarising the changes that would be made to the XML file, and asking for confirmation.
- On unsuccessful validation, the issues should be clearly indicated to the user, with careful attention paid to making this as easy to use as possible.

You should use a page called `index.html` as the main entry point to the site, and give careful consideration to the appearance and usability of your site. Marks will be awarded for thoughtful presentation of the information in a way that is easy to navigate and understand.

You should use HTML5 and CSS3 for your site, and should validate your code. You should also use JSLint or JSHint to identify potential issues with your JavaScript code

C.2 Deliverables

Your site should be available at `http://dev212.otago.ac.nz:8080/~username/assignments/index.html`. Note again that this directory is password protected, so only you and the teaching staff can access it. 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 list of the JSLint or JSHint 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.

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

Apart from the core jQuery library, all code and markup 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.

You should put all of the files for your assignment (HTML, JavaScript, CSS, XML, images, your report, etc.) in the folder on sapphire called `assignments/` in your web development directory on `sapphire` (`/devel/username/projects/assignments/`).

Once all of the files are in place you should run (from the console on `sapphire`) the command

```
~steven/public/bin/submitAssignment1.sh
```

This script will create an archive of your assignment and submit it. You will be left with a copy of the archive in your home directory on `sapphire`. This will be named with your username plus a timestamp. You should keep this file as a record of your submission.

The script does some basic checking and will report if it cannot find the sort of files it expects. If you run the script multiple times, you will make multiple submissions. Each will generate an archive with a different timestamp. Your last submission will be marked, and late penalties applied accordingly.

C.3 Marking Scheme

The marking scheme for Assignment 1 is given in Table C.2. 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.2: Assignment 1 marking scheme

Correctness and Usability		
Valid HTML/CSS/JavaScript	10	
General usability & appearance	10	20
Required Functions		
Display the current tournament status, including scores of completed games	10	
Display the ranking table for the current tournament status	10	
Allow the entry of scores, dates, and venues for games	10	
Appropriate behaviour on successful validation of data	5	
Appropriate behaviour on unsuccessful validation of data	5	40
Style and Testing		
Appropriate use of comments	5	
Appropriate separation of HTML/CSS/JS	5	
Appropriate JavaScript structures	5	
Report and testing strategy	15	30
Total		**100**

Chapter D

Assignment 2

Due: 11:59 p.m. October 7. Worth: 17%

It is very important that you protect your assignment work from unauthorised copying. We have set up secured directories on `sapphire` for this purpose. You should put *all* files related to your assignment work under `/devel/username/projects/assignments`. This directory is password protected, in order to prevent other students from copying your work.

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 XML files in response to user input.

Your web application must allow the administration pages to be used to:

- Update the score, date, and/or venue for a particular game
  - These updates should be reflected in the contents of `tournament.xml/`
  - The client-side validation required in Assignment 1 must be repeated on the server-side.
  - *It is not sufficient to rely on client-side validation. Ever.*
- Create a whole new tournament, specifying a list of venues and a list of teams.
  - Creating a new tournament will replace the contents of `tournament.xml` and `venue.xml` completely.
  - There should be at least one venue provided.

D.2 Deliverables

As with Assignment 1, your site should be available at `http://dev212.otago.ac.nz:8080/~username/assignments/`. Note again that this directory is password protected, so only you and the teaching staff can access it. 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 `assignmentTwo.pdf`. This report should include:

- 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.

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

Apart from the core jQuery library, all code and markup you submit must be your own work. The use of other third party libraries, whether CSS, HTML, JavaScript, PHP, 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.

You should put all of the files for your assignment (PHP, HTML, JavaScript, CSS, XML, images, your report, etc.) in the folder on sapphire called `assignments/` in your web development directory on `sapphire` (`/devel/username/projects/assignments/`).

Once all of the files are in place you should run (from the console on `sapphire`) the command

```
~steven/public/bin/submitAssignment2.sh
```

This script will create an archive of your assignment and submit it. You will be left with a copy of the archive in your home directory on `sapphire`. This will be named with your username plus a timestamp. You should keep this file as a record of your submission.

The script does some basic checking and will report if it cannot find the sort of files it expects. If you run the script multiple times, you will make multiple submissions. Each will generate an archive with a different timestamp. Your last submission will be marked, and late penalties applied accordingly.

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		
Allow administrators to update match details (score, venue, and date)	15	
Allow administrators to create a new tournament	20	
Server-side validation of data	15	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.

A short video is available on Unitube showing the final form of the Classic Cinema site. You can see this video at `https://unitube.otago.ac.nz/view?m=DMnX5I7Z0LL`. In this lab, you will start to develop this site.

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. COMP212 labs are held in Lab A, but you can use any of the labs when working in your own time.

The COMP212 Lab Environment

The lab machines are Apple iMacs running Mac OS X, and so should be familiar to you from the other courses you have done in the department. Your main tool will be a text editor, and you have several to choose from: Taco and TextWrangler are in their usual places in the application menu, and Emacs, Vi, and Nano are available from the command line.

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 `sapphire.otago.ac.nz`, which is the web server machine for this course. You'll learn more about `sapphire` in Lab 2.

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 Course Documents → Lab Files. You should download them to your local directory, and unzip them.

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.

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, apart from the warnings listed above. 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 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 1.1 and Figure 1.2. 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.

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 UpperEastSide font, which can be downloaded from `http://www.fontsquirrel.com/fonts/UpperEastSide`. 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.
- Body text is 10pt size, navigation links are 12pt, and headings (h1 to h3) are 36pt, 24pt, and 12pt.
- The darker grey background has hex value `DDDDDD`, and the lighter grey boxes hex value `EEEEEE`.
- The navigation box is 135 pixels wide, and the main content box is 600 pixels wide.
- The main block elements have 5 pixel margins, 10 pixel padding, and a 1 pixel wide black border with rounded corners in the top left and bottom right with a radius of 20 pixels.

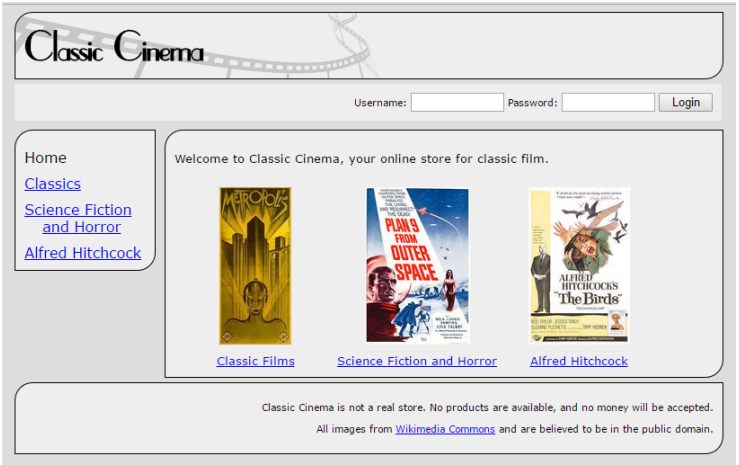


Figure 1.1: CSS style applied to `index.html`

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

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.

Lab 2

Getting Started With Unix

In this lab you will start to use `sapphire`, our Web server for this course. You will learn how to put web pages on the development server, and some of the basics of the Unix command line. Once you have put your web pages from the last lab on the server, you will see how to test it on different computers and browsers, and to overcome some of the issues that this raises.

The web server for this course 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.

2.1 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 2.1: 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:

```
http://dev212.otago.ac.nz:8080/~astudent/file.html
Http://Dev212.Otago.ac.nz:8080/~astudent/file.html
HTTP://DEV212.OTAGO.AC.NZ:8080/~astudent/file.html
```

but they are not the same as

```
http://dev212.otago.ac.nz:8080/~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

  `http://dev212.otago.ac.nz:8080/~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.

2.2 The Unix File System

The Unix file system, like most file system 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 2.1. 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.

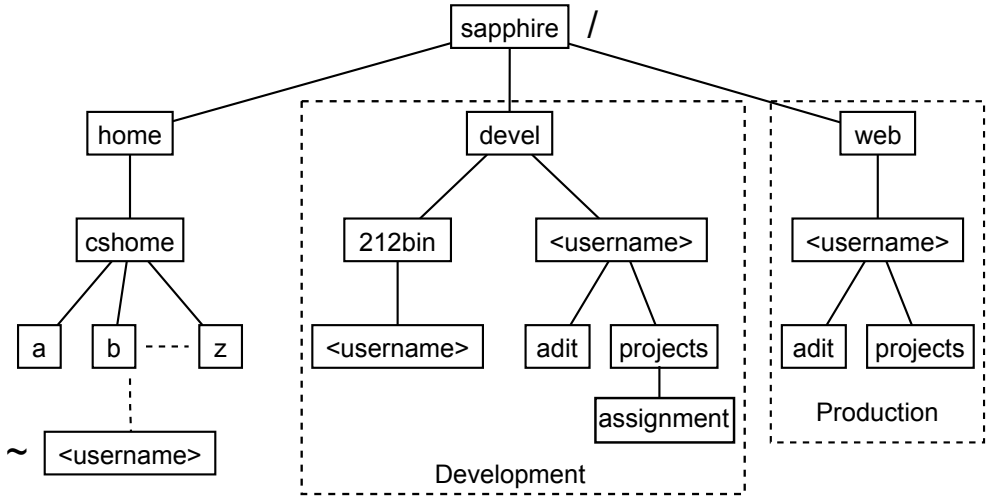


Figure 2.1: The filesystem on `sapphire`.

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 `http://dev212.otago.ac.nz:8080/~<username>/file.html`.

There are a few important details here. Firstly, the `:8080` part of the address tells the browser to connect to port 8080 rather than the standard HTTP port (which is port 80). Port 80 is used for the production server. 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 2.7.

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.

2.3 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 2.2: 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 not be any.

Task 2.3: 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 the `~steven/public` directory. You can copy this file to a file called `index.html` with the command

```
cp ~steven/public/template.html index.html
```

If you then start editing this file with nano, it should appear as shown in Figure 2.2.

Figure 2.2: Editing `index.html` with Nano.

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.

Task 2.4: Edit the default page to include some more interesting HTML.

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

`http://dev212.otago.ac.nz:8080/~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:

`http://dev212.otago.ac.nz:8080/~username/`

2.4 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. We could put all of the files for the course directly in development directory, but things would get very cluttered. A better way is to organise your work with directories. Ordinarily you'd make a directory for the Classic Cinema site:

```
mkdir /devel/username/projects/ClassicCinema/
```

and store your files in there. However, that would not give you a record of what was done in each lab, so you may wish to make a directory for each lab.

Task 2.5: Create a directory in `/devel/username/projects/` to store the files for this lab exercise.

Once that's done we can transfer the files from your CS home storage (that you use on the lab machines) to `sapphire`. This is easily done by (secure) FTP, and for this example we'll demonstrate the process using PhpStorm's FTP functionality, and the FileZilla program that is installed on the lab machines. YummyFTP, which you may be familiar with from COMP112 is also available. You can also use FTP from the command line, or with tools like WinSCP from Windows machines.

Using PhpStorm's FTP capabilities

There is a short video demonstrating the use of PhpStorm's FTP setup. This video is available on UniTube at `https://unitube.otago.ac.nz/view?m=aPu58_RWYzI`.

Using FileZilla

There is also a short video demonstrating the use of FileZilla. This video is available on UniTube at `http://unitube.otago.ac.nz/view?m=ssH75lDaqzk`.

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 2.3.

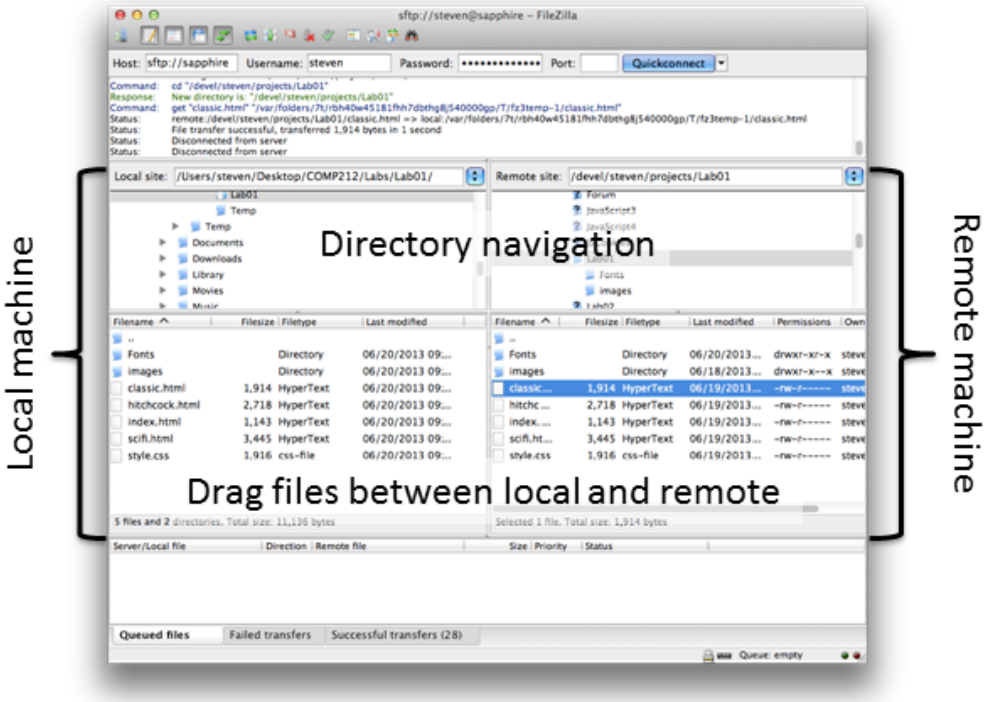


Figure 2.3: The FileZilla FTP client.

Task 2.6: Transfer the pages you made in Lab 1 to the directory you made in Task 2.4.

2.5 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 2.7: 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 2.8: 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 2.1. 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 2.1: 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 2.9: 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 2.10: 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 2.11: 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.

2.6 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'. In this section we'll develop a script to backup your web development directory.

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 2.12: 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 2.13: 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 2.14: 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 2.15: 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.

2.7 Accessing `sapphire` from Off-Campus

The development server on `sapphire` is configured only to accept connections from on-campus. 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` through another computer. 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. This can be viewed on Unitube at `https://unitube.otago.ac.nz/view?m=T0KY5BV3DPz`.

If you are running OS X or some variety of Linux this process is quite simple. Simply enter the command

```
ssh -L 9999:sapphire.otago.ac.nz:8080 <your cs username>@hex.otago.ac.nz
```

This associates port 9999 on your home computer with port 8080 (the development Web server) on `sapphire`, 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 `http://dev212.otago.ac.nz:8080/~<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:

```
5
ssh -L 9999:sapphire.otago.ac.nz:8080 -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 2.4.

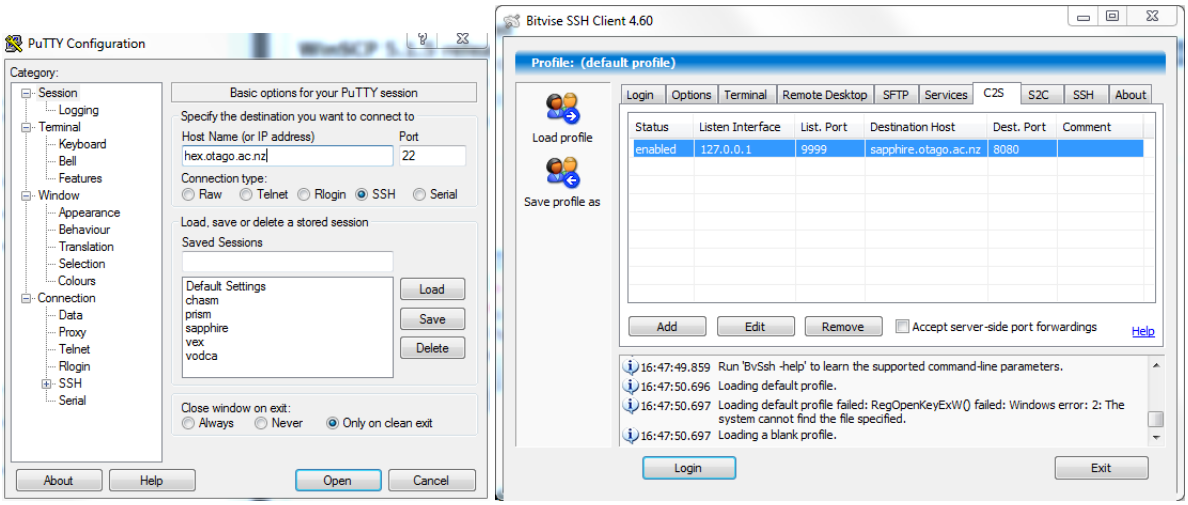


Figure 2.4: Setting up port forwarding with PuTTY (left) and Tunnelier (right).

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 `sapphire.otago.ac.nz:8080`, 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 `sapphire.otago.ac.nz:8080`; and Dest. Port to 8080. You can add other ports (such as a link to port 22 for SSH and SFTP) as required. These details are shown in Figure 2.5

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 `http://dev212.otago.ac.nz:8080/~<username>/file.html`.

2.8 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/`.

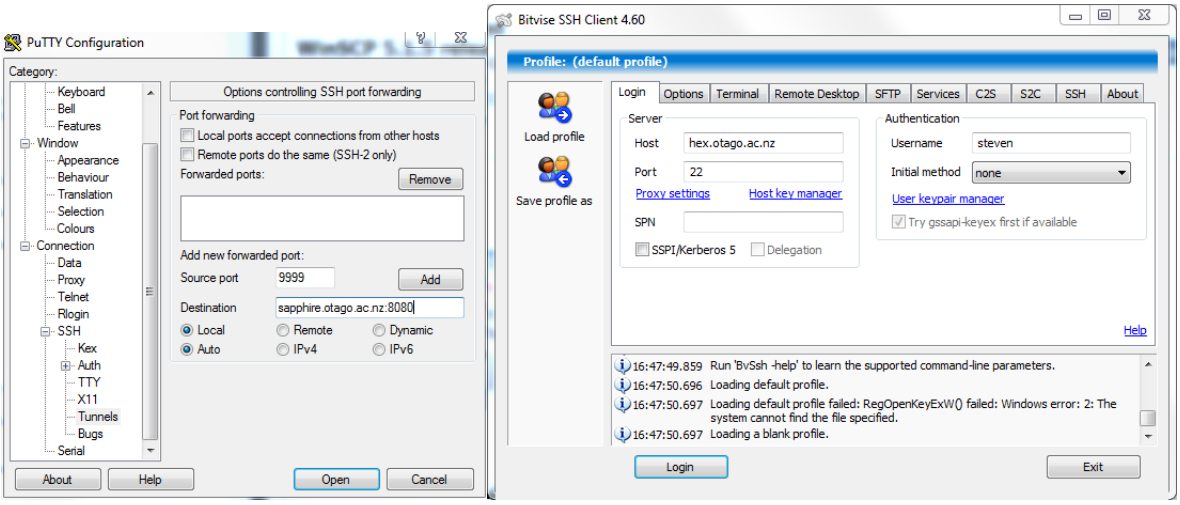


Figure 2.5: Connecting with PuTTY (left) and Tunnelier (right).

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.

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>
```

Task 3.1: Make a new HTML file on `sapphire`, 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) which 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.

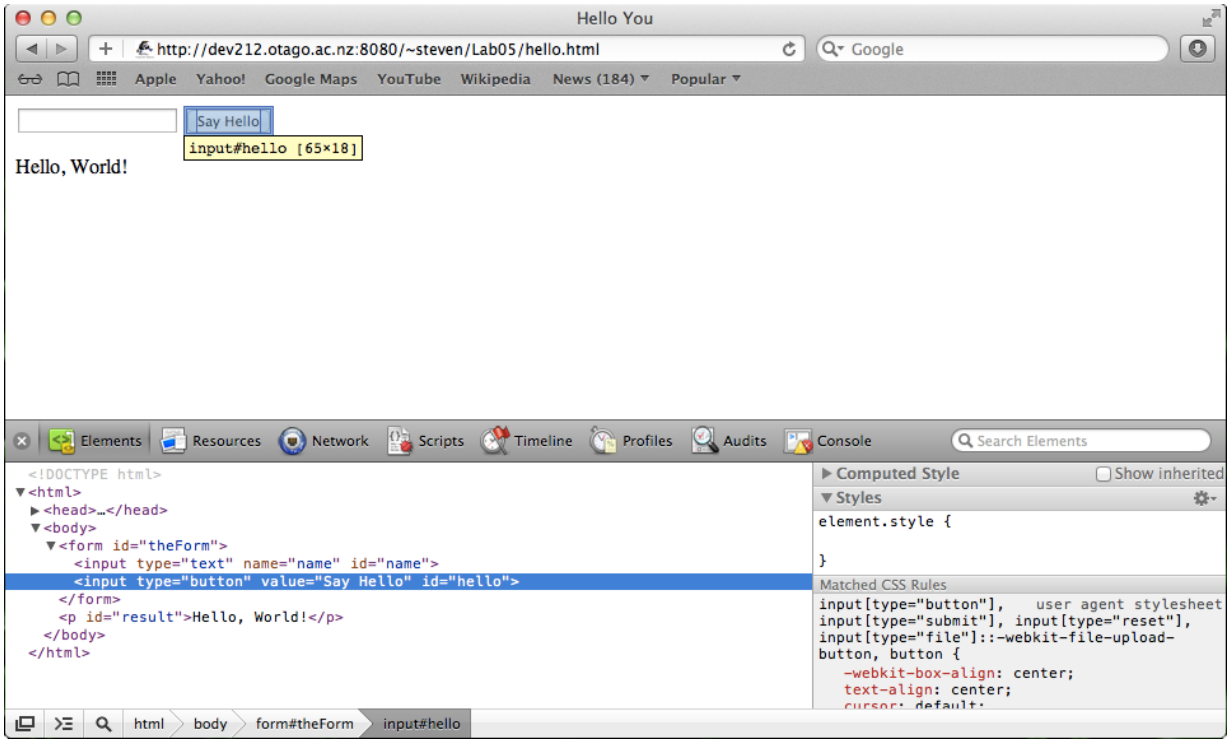


Figure 3.1: Developer Tools in Safari under OS X

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");
```

- 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.

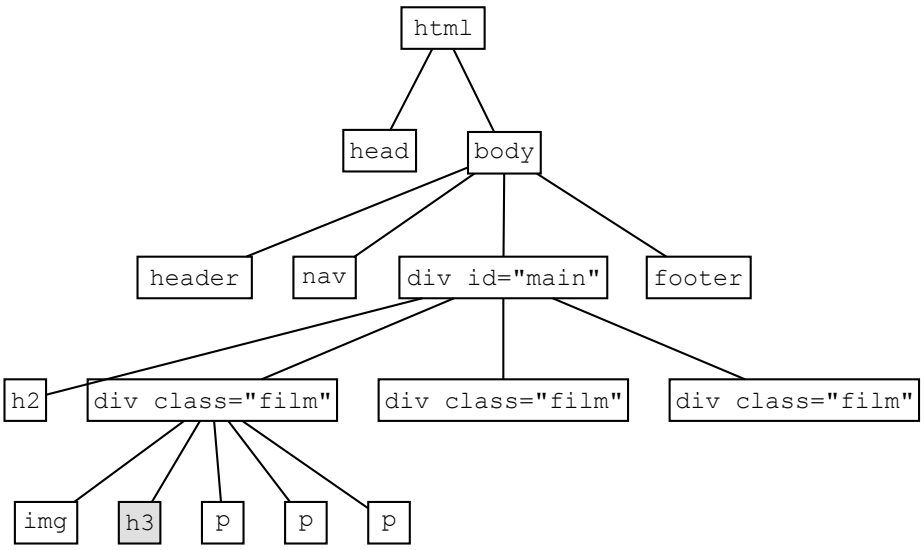


Figure 3.2: Part of the DOM tree structure for the Classic Films page. Given a reference to a level 3 heading element (shaded), we want to reach the related paragraph elements.

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.

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

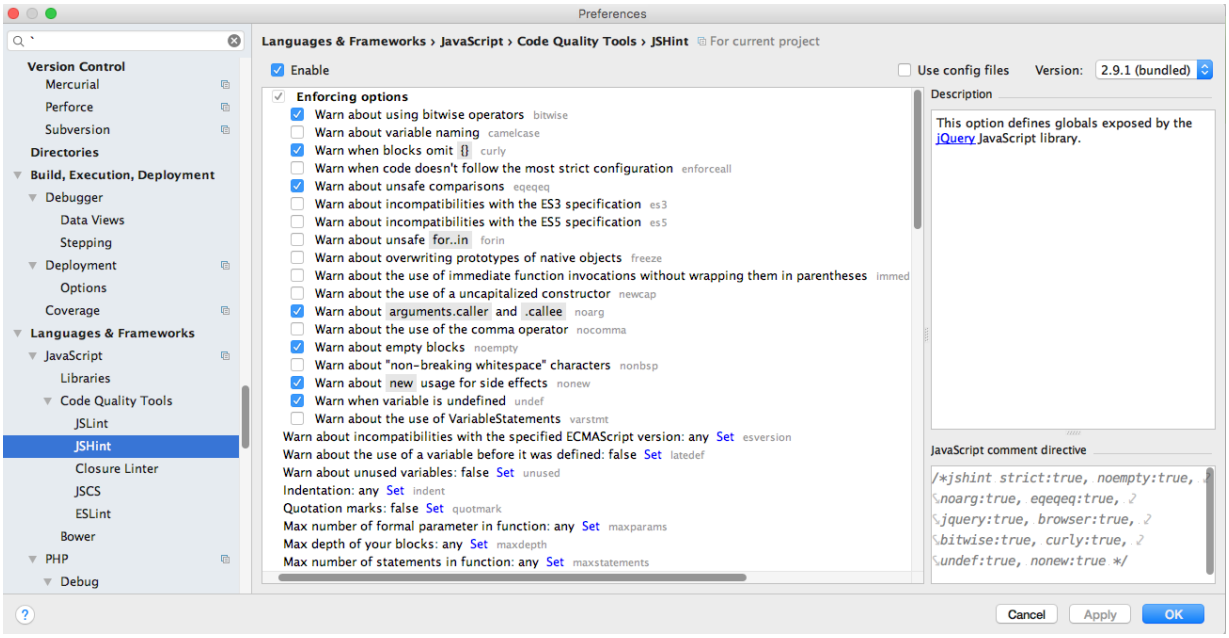


Figure 3.3: Setting up JSHint in PHPStorm.

```
"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- 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:

```
<a href='classic.html'><figure>
  <img src='images/Metropolis.jpg'><figcaption>Classic Movies</figcaption>
</figure></a>
```

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 on `sapphire`. 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.

For the assignment: Cookies are used to maintain state. While it's possible to complete the assignment without cookies, they might be useful. Think about where you might want to maintain state between pages in your application.

Lab 7

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 7.1: 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 `sapphire` from `~steven/public/sampleForm.html` – 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 7.2: 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 7.3: 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 `sapphire` in `~steven/public/`. 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 7.4: 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.

7.1 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 7.5: 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.

7.2 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 7.6: 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 7.7: 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.

7.3 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 7.8: 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 8

Catch Up

There are no new exercises for this lab. You should use this time to catch up on any of the previous labs that you have not completed. You should pay particular attention to the assessed labs, since they will no longer be accepted after this lab.

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 add functionality to the Classic Cinema website, for example:
  - Recognise when multiple copies of the same item are in the cart, and displaying them as a single entry in the checkout.
  - Adding the ability to remove items from the shopping cart.
  - Adding additional interactive form checking
- 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.

At the end of this week, a sample solution to the Classic Cinema development up to Lab 7 will be released. You can use this to continue on with future labs, even if you have not completed all of the labs to date.

Lab 9

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.

9.1 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, (3.1.0 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 9.1: Download the latest compressed (minified) and uncompressed versions of jQuery's 3.x branch from `http://jquery.com/download/`.

The minified version of 3.1.0 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 9.2: 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.

9.2 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 9.3: 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 10

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.

10.1 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 10.1: 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.

10.2 Classic Cinema Reviews

Let's return to the Classic Cinema site. There is a folder on `sapphire` at `~steven/public/reviews/` 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 10.2: 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 10.3: 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 10.4: 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 10.5: 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 11

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.

11.1 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 11.1: 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.

11.2 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 11.2: 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 11.3: 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`.

11.3 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 11.4: 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 12

Managing Deployment

In this lab you will learn how to use shell scripts to automate simple tasks. In particular, you will write a script to move a website from the development server on `sapphire` to the production server.

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 you will be writing scripts that copy, move, delete, and can overwrite files and directories on `sapphire`. Caution is advised, and you should be sure to have an up-to-date backup of any files stored on `sapphire`.

When you are developing a web application, you will often make changes that break existing behaviour. Sometimes this is intentional, when you are replacing old functions with new versions, but often it is not. In any case, you don't want users of your web application to see these changes until they are tested and working properly. The usual way to manage this is to have multiple web servers. One is used for development, where changes can be made and testing carried out away from public view. A second web server is used for the live site. This should be completely separate from the development server, and changes are only made to the live (or production) server when they have been fully tested.

There are two web servers running on `sapphire` – the development server that we have been using so far, and the production server. The location of files for each webserver, and the basics of Unix shell scripting were covered in Lab 2. You may find it useful to refer back to that lab for this one.

The files for your production site are placed on `sapphire` in `/web/username/projects`, which serves the same purpose as `/devel/username/projects` does for the development server. Your production websites can be accessed via `web212.otago.ac.nz/~username/`.

In order to move a project from development to production, you need to copy the relevant files across. This process is often called *deployment*, and you could do it in the Unix shell with commands like—

```
cd /devel/username/projects
cp -R myProject /web/username/projects/myProject
```

However, this has a number of limitations:

- It does not properly deal with the situation where `/web/username/projects/myProject` already exists. It will overwrite the old site, but what if there is a mistake in the new version? How can you get the old site back?

- In some cases, not all of the files in the development site need to be copied to the production server. As a simple example, if you edit the file `index.html` in emacs, it will often create a backup file named `index.html~`. You probably don't need this file in your production site.

- While the new files are being copied, your new site will be in an inconsistent state. For example, say you have added an updated version of `checkout.html` to the Classic Cinema pages that relies on a new JavaScript file, `new.js`. If `checkout.html` is copied first, someone might see the new version before `new.js` has been copied across.

There are a number of solutions to these problems, and in this lab we'll look at one of them. We will write a simple shell script that will ensure that the new version of the site replaces the old one in a single step, and makes a backup of the old site that we can retrieve if something goes wrong.

12.1 Shell Scripting Again

You wrote some simple shell scripts to update permissions in Lab 2. Remember from that lab, that shell scripts start with the line

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

and have to be executable by the user who is running them.

Rather than making a separate script for each project, we can write one that will deploy a project given its name as a command line parameter. For example, we want to be able to deploy the project for `Lab11` with the command

```
deploy.sh Lab11
```

Since this script is not associated with a particular project, it makes sense to place it in a central location. A common place for scripts and commands is in a directory called `bin` (for binary, which historically indicated executable files) in your home directory.

Task 12.1: Create a script called `deploy.sh` and place it in `~/bin`. For now, the script should just consist of a single line that prints a statement confirming that it has been called. Refer back to Lab 2 for details.

12.2 Shell Script Parameters and Variables

First we need to find out what project we are to deploy. This will be provided as command line parameters, which can be accessed as follows:

- `$0` gives the name of the script that is running;

- `$1`, `$2`, etc. give the first parameter, second parameter, and so forth;

- `$#` gives the number of parameters provided.

For example, we can check to see if there is exactly one command line parameter, and if there is report what it was, with the code—

```
if [ $# -ne 1 ];
then
  # Not exactly one argument
  echo $0 expects exactly one argument:
  echo $0 \<project name\>
  exit
fi

echo Deploying $1
```

This checks to see if the number of arguments (`$#`) is *not* equal (`-ne`) to 1. If this is true, it reports an error message and exits. Otherwise the script continues and reports what project it is deploying.

As well as variables representing the command line parameters, you can declare your own. You can do this like this:

```
MY_VARIABLE=42
echo $MY_VARIABLE
```

Note that shell variables by convention are in upper case, although this is not a requirement. Also note that you use a `$` prefix when using the value of a variable, but not when declaring them. Be careful to ensure that there is no whitespace on either side of the = sign.

In addition there are some other builtin variables and environment variables you can access. One that will be of use to us is `$USER`, which gives the username of the person running the script.

12.3 The Deployment Script

Let's start by thinking about what you would do to deploy a project called `myProject`. You could do a basic version of this with the command

```
cp -R /devel/username/projects/myProject /web/username/projects/myProject
```

This has all of the problems we outlined earlier, but we can make it a bit more general by substituting `$USER` for your username and `$1` for `myProject`.

Task 12.2: Update your deployment script so that it uses this approach. Try to deploy one of your earlier labs, and check that you can view the deployed site through the production server. Update one of the files and deploy the site again, making sure that the changes are deployed.

Next we deal with the issues associated with overwriting the existing site. Assuming we're deploying `myProject`, we can overcome these issues with the following commands:

```
cp -R /devel/username/projects/myProject /web/username/projects/myProject.new
mv /web/username/projects/myProject /web/username/projects/myProject.old
mv /web/username/projects/myProject.new /web/username/projects/myProject
```

This copies the new version to `myProject.new` then shuffles the existing version out to `myProject.old`. Note that moving a file with `mv` is a simple and very quick operation, since it involves just one update to the filesystem. Again, this can be made more general, and we can define some variables to describe what is going on:

```
# $SOURCEDIR is where the files are coming from
SOURCEDIR=/devel/$USER/projects/$1
```

```
# $TARGETDIR is where the files are going to
TARGETDIR=/web/$USER/projects/$1

# $TEMPDIR is the temporary staging location
TEMPDIR=$TARGETDIR.new

# $BACKUPDIR is where we store a copy of the old site
BACKUPDIR=$TARGETDIR.old
```

We only need to do the swapping of the old and new versions if there is, in fact, an old version. We can test for this in a bash `if` statement with `-d`, which checks if a directory exists:

```
cp -R $SOURCEDIR $TEMPDIR

if [ -d $TARGETDIR ];
then
  mv $TARGETDIR $BACKUPDIR
fi

mv $TEMPDIR $TARGETDIR
```

There are a couple of other minor things to add:

- If there is an existing backup, it will need to be removed before copying to `$BACKUPDIR`.
- Permissions should be set so that the backup site cannot be accessed by the web server.

Task 12.3: Update your script so that it makes a backup of the existing site when deploying a new version. Check that the backup site cannot be accessed in a browser. How could you restore the old version if you realise a mistake has been made?

Finally we can alter our script to only transfer particular files to the production server. Suppose we have a file in `$SOURCEDIR` called `manifest` that contains a list of files we want to copy. For example, to copy all of the HTML files, `style.css`, and all of the files in the `images` directory ending in .jpg `manifest` might look like this:

```
*.html
style.css
images/*.jpg
```

We can copy just these files across with the commands

```
cd $SOURCEDIR
mkdir $TEMPDIR
cp --parents $(cat manifest) $TEMPDIR
```

The `$(...)` syntax inserts the result of one command inside another, the `cat` command just prints the contents of a file, and the `--parents` option to `cp` makes sure that subdirectory structures are created when copying. When deploying a project we want to use this technique if there is a `manifest` file, and a normal recursive copy if not. We can check for the presence of a manifest file in the same way as we checked for the presence of `$TARGETDIR`, except that we use the `-f` flag (for file) rather than `-d` (for directory):

```
cd $SOURCEDIR
if [ -f manifest ];
then
  # Manifest exists, use it
else
  # No manifest, copy all files
fi
```

Task 12.4: Alter your script so that it uses a manifest file, if one exists. If no manifest exists, then copy the entire project directory.

For the assignment: Note that you are *not* required to deploy your work to the production server for Assignment 1, but you are for Assignment 2.

Lab 13

Version Control Systems

In this optional 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 Subversion, and see that such systems can provide an alternative to deployment using shell scripts.

Note: This lab is not assessed, and is optional. If you still have unfinished assessed labs or work to do on your assignment, they should probably take priority.

In this lab you will be running commands that could potentially overwrite files and directories, so caution is advised.

You saw in Lab 12 that it can be useful to automate the task of deploying sets of files onto a live site when working in an environment that provides both a development and a live server. The scripts that you wrote copied across particular versions of your local files to `sapphire`. However it was still your responsibility to manage backups of any past versions of the content that you had created, for example, if you might need to 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 optional lab, we will explore the use of the Subversion version management system. For a quick overview of the history, Subversion is essentially an update of the earlier Concurrent Version System (CVS), which was itself an evolution of the Revision Control System (RCS).

Today, Git is also an extremely popular version management system. Although Git was build 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. Git has a more complicated workflow than Subversion, but it is also more powerful. There are excellent online tutorials for simple use of Git, and you are encouraged to experiment with it. However for this lab, we are only scratching the surface of version management, for which Subversion is a more straightforward tool to use. There are plenty of other excellent version management systems that have different strengths and weaknesses, such as Mercurial.

We will focus on how you can use Subversion to maintain the past versions of website content, and

how to use Subversion to deploy content onto a live server, in a similar manner to the shell scripts in Lab 12. 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.

13.1 Subversion concepts

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

Repository. 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 Subversion, and for you in this lab, the repository is often *remote*. This means that your past versions are stored on a server on the network (or the internet). For COSC212, the server `groke.otago.ac.nz` has been set up to contain an empty repository for every student that has access to `sapphire`.

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 Subversion when you have reached a state of the files that you want it to remember for you. Your act of requesting Subversion to remember a particular version of files is known as *committing* a *revision* to the repository.

Working copy. In Subversion your repository is stored on a different server. However the actual files and directories that you want to commit to your Subversion repository are stored on the computer where you are editing them, such as `sapphire` or your laptop, a workstation in the lab, etc. The term *working copy* is used to describe a set of files and folders that has been set to link to a repository. You can have as many independent working copies as you wish, but you will need to commit changes from them to cause any change to the repository.

In Subversion, a working copy will contain the hidden `.svn` directory, since that is the way that Subversion keeps track of what changes it needs to send to the repository when you commit a revision. (As an aside, Git uses a `.git` directory in its working copies.)

13.2 Using Subversion working copies on the development server

Recall from Lab 12 that there are two web servers running on `sapphire` – the development server that we have been using most of the time, and the production server that you used in Lab 12. You may find it useful to refer back both to that lab, and for Unix introduction to Lab 2.

Also note that in this lab we will continue with the convention in directory names and URLs where "username" should be replaced with your Computer Science username.

We will begin experimenting with Subversion with a working copy that is served up by the development web server. Later in the lab we will checkout another working copy on the live web server.

13.3 Adding content to your Subversion repository

Your repository has already been created, but it contains no content—it is not yet set up to track changes in any files or folders. So your first task will involve adding some content that will be recorded in your Subversion repository.

Task 13.1: Create some test content that is visible from the development server, as instructed below.

Create an empty folder where `dev212.otago.ac.nz/~username/svntest` would show its content, and change into that folder:

```
cd /devel/username/projects
mkdir svntest
cd svntest
```

Create an `index.html` file in that directory and check that you can access it from your web browser. For example:

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

Task 13.2: Instruct Subversion to expect to track changes to your example content.

Now let's use our first Subversion command, that will turn your `/devel/username/projects` directory into a working copy. Note the space followed by the full-stop at the end of the `svn checkout` command below—it is part of the command!

```
cd /devel/username/projects
svn checkout https://groke.otago.ac.nz/212/username .
```

This command checks out the repository `https://groke.otago.ac.nz/212/username` as a working copy into the directory that is specified. Here we have specified '`.`' as the directory, which is equivalent to "the current directory". However, there were no files in your repository, so this `checkout` will not make an obvious difference (although it will create the hidden `.svn` directory).

When working copies are visible to a webserver, for security reasons access to the version management system's own folders (e.g., `.svn`) should be denied to visitors to your website.

Now tell Subversion that it should track what is in the `svntest` directory by `adding` it.

```
svn add svntest
```

This action alone will not cause any content to reach the the repository. To achieve that, you need to explicitly commit the content, as we explore in the next section.

13.4 Committing changes to your Subversion repository

Task 13.3: Commit a revision of your example content to your repository.

We are now ready to tell Subversion to remember the current contests of the files in our svntest directory, complete with the current state of the `index.html` file, as a revision.

```
cd /devel/username/projects
svn commit -m "The initial version of my svntest folder." svntest
```

Note that string given to the `-m` flag is the commit's log message. It is a good idea to write informative log messages so that you can scan the history of commits when you need to compare changes or revert to an old version. A message such as "made some changes" will not usually help you remember what those changes were. Good log messages are even more important when working in a team, although we are not discussing that sort of use of Subversion in this lab.

13.5 Checking out another working copy of your repository on the live server

Now let's check out another working copy of your repository, but instead place it so that it shows content on the live server. We are aiming to achieve a similar effect to the deployment scripts discussed in Lab 12. Subversion has the useful feature that we can check out a subtree of our overall repository as a working copy (As an aside, Git does not support this directly).

Task 13.4: Place a working copy of part of your repository so that it is visible through the live web server.

Let's just check out the `svntest` directory as the working copy on the live server.

```
cd /web/username/projects
svn checkout https://groke.otago.ac.nz/212/username/ .
```

Test that you can view the expected content at `web212.otago.ac.nz/~username/svntest`.

Task 13.5: Test updates to your content using the development server working copy, and commit them.

Now, change back to your development server working copy—

```
cd /devel/username/projects/svntest
```

—and make a change to your `index.html` page. For example, change, "Some initial text" to "Some updated text".

Confirm that on the development server you see this change, but that the live server is still displaying the content in your first commit.

When you are satisfied with the updates to your `index.html` file, you can commit a revision to the repository:

```
cd /devel/username/projects/svntest
svn commit -m "Important update" index.html
```

In this instance of the `svn commit` command, we have explicitly indicated that `index.html` should be the only change reflected in the revision.

Task 13.6: Update the live server with your committed content.

Despite having made this commit to the repository, the live server will still be serving up the previously committed content. Since the live server is using a working copy, it is straightforward to update the live server to the currently committed revision with the following command:

```
cd /web/username/projects/svntest
svn update
```

13.6 Other ways to use version management systems for website deployment

In this lab, there was nothing different about the working copies that we checked out into the development and into the live server paths. By convention we made changes to the working copy on the development server, and just ran `svn update`s on the live server. There are many other ways in which version management systems can be used to support the deployment of websites.

For example, Subversion provides a capability to run scripts automatically when commits are made to repositories. Using these scripts, stronger restrictions than the convention that we applied can be enforced. For example, the live server can be set up so that content can only be updated by making a Subversion commit. Content visible by the development server can continue to be edited directly. In a team environment, this way of working ensures that the live server will only ever show content from a revision that an authenticated user has explicitly declared is ready for production use at some particular point in time.

13.7 Support for version management systems within editors and integrated development environments (IDEs)

In this lab we have focused on the use of command line tools to access your Subversion repository. A more convenient option is often provided directly within editors and integrated development environments. For example, Eclipse, Apple's Xcode, Microsoft's Visual Studio and indeed PhpStorm, all have built-in support for Subversion and Git.

This allows commits to be made directly within your editing environment, as well as providing an ability for you to conveniently access and make comparisons with previous revisions' content.

Lab 14

Catch Up

There are no new exercises for this lab. You should use this time to catch up on any of the previous labs that you have not completed. Labs Lab 9 to Lab 12 will no longer be accepted for marking after this week. Also, the first assignment is due at the end of this week.

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 fairly meaningless print statements by using special tags to indicate which parts of a page are PHP. 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.

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 `/devel/username/projects` or `/web/username/projects` like HTML content.
- PHP scripts do not need to be executable by the web server (although they must still be readable). The web server does not run the scripts themselves, rather it runs a PHP interpreter, whcih 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 a rather unhelpful server error message.

Fortunately, there is 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 `/devel/log/phperr.log`. However, all of the class's errors will be stored there so you need to search through the log for your error messages. You can do this with the Unix command `grep`:

```
grep <username> /devel/log/phperr.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 <username> /devel/log/phperr.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.

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.

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 which 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 two 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.

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` 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` from

  ```
  pub.setup = function () {
    $("#checkoutForm").submit(validateCheckout);
      $("#cardNumber").keypress(checkKeyIsDigit);
  };
  ```

  to

  ```
  pub.setup = function () {

  }
  ```

  You should also ensure that HTML5 validation is disabled in the form.

  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-1.11.1.min.js', 'cookies.js');
    include("header.php");
  ?>
    <div id="main">
      <p> Placeholder for checkout validation </p>
    </div>
  <?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 `~steven/public/validationFunctions.php`. 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 `\devel` and `\web` 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/2.2/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 XML files on the server – these were used for longer term storage of information that could be shared between users, such as the reviews of movies. We also saw how XML files could be updated using CGI scripts.

In this lab and the next, we'll revisit these two methods of preserving state from the perspective of PHP. We'll access the shopping cart stored in a cookie, 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 which 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 3600s (or 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.

Cookies in the Cart

Returning to the Classic Cinema site, the shopping cart is stored 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 the cookie, 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.

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 17.3: 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` – it is an associative array or 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 is available on `sapphire` from `~steven/public/sessionCounter1.php`.

Task 17.4: 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 17.5: 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 17.6: 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 a cookie 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 *Trantula* for $7.99. After processing, `orders.xml`, might look something like this:

```
<?xml version="1.0"?>
<orders>

  <order>

    <delivery>
      <name>Steven Mills</name>
      <email>steven@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 – `<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 `validateCheckout.php` 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.

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 18.3: 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 18.4: 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 18.5: 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');
```

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 19

Getting Started with MySQL

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.

19.1 Connecting to MySQL

To get started with MySQL, log in to `sapphire`. 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 `-`:

```
mysql -h sapphire -D username_dev -u username -p
```

The options we're using here are as follows:

- `-h` specifies which host you wish to connect to. In this case we want to connect to the database server running on `sapphire`. While we're using `sapphire` for both the web server and database server, often two separate machines are used, so it is important to specify the host.
- `-D` specifies which database you wish to use on that host. The database server on `sapphire` has two databases for each user – one for development (`username_dev`), and a second one for production (`username_prod`). Here we're using the development database.
- `-u` specifies the username you wish to use to connect to the database. In this case it is the same as your CS username.
- `-p` specifies that you will be providing a password, which you will be prompted to provide.

Your initial password will be `CS1.` followed by your student ID. For example, if your studentID was 1234567, your initial password would be `CS1.1234567` (note the '`.`' before your ID). Since this isn't very secure you should change it after first logging on. You can do this with the command

```
SET PASSWORD = PASSWORD('password');
```

Of course, `password` is even less secure than your ID number, so you should use something more sensible.

Task 19.1: Connect to your development database on `sapphire` and set your password to be something more secure (but memorable). When you've done this, give the command `exit` to leave MySQL, and make sure you can log back in.

19.2 Creating a Users Table

MySQL, like most databases 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
Steven	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 (an actual value must be given).

Task 19.2: Create the Users table in MySQL using the 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='steven';
```

The `WHERE` clause specifies a condition, and only those rows that meet the condition are returned.

Task 19.3: Add a few users to the database. 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. The solution 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 WHERE username <> '';
INSERT INTO Users (name, password) VALUES ('me', SHA('secret'));
SELECT * FROM Users;
```

Note that on `sapphire` you cannot just use the command

```
DELETE FROM Users;
```

to remove all entries from the table. A `DELETE` statement that does not place some restriction on the rows to be deleted is usually a mistake, so MySQL on `sapphire` is configured to block it. In this case we 'restrict' the rows to be those with a non-empty username.

Task 19.4: Clear out the Users table and add a few users with different passwords using SHA encryption.

19.3 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 19.5: 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.

19.4 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='steven@cs.otago.ac.nz'
 WHERE username='steven';
```

You can also use `UPDATE` to alter existing values, such as changing a password:

```
UPDATE Users
   SET password = SHA('new_password')
 WHERE username = 'steven';
```

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 19.6: Update the Users table to have an email column, and set email values for the existing users.

Lab 20

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 20.1: 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.

20.1 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
- Free up the resources (the result object and the database connection) that we've used.

The PHP code to do this looks like this:

```
$conn = new mysqli('sapphire', 'username', 'password', 'username_dev');
if ($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 (your CS username) 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 20.2: 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.

20.2 Adding a New User

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 20.3: 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.

20.3 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 simple script to do this in `~steven/public/injection.php`. To get this running you will need to put your username and MySQL password in the two variables at the top of the page. 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 20.4: Take a copy of `injection.php` and provide the appropriate values for the database username and password. 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 20.5: Make suitable changes to `injection.php` and your registration code to protect against SQL injection.

Lab 21

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 21.1: 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.

21.1 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 19), and how to connect to the database from PHP (Lab 20). 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 21.2: 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 21.3: 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 21.4: 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`.

21.2 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 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 the previous page).

21.3 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 21.5: 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 links to 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 thing that may be required on multiple pages, you might want to factor the code for checking authentication out into a separate script.

Lab 22

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.

22.1 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 22.1: 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 22.2: 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 22.3: 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 22.4: 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.

22.2 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 22.5: 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 22.6: 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 22.7: Update `orders.php` so that it has the behaviour described above.

22.3 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 10 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 22.8: 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 23

Catch-Up/Assignment

There are no new exercises for this lab. You should use this time to catch up on any of the previous labs that you have not completed. Labs Lab 15 to Lab 22 will no longer be accepted for marking after this week. Also, the second assignment is due at the end of this week.

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 24

Catch-Up/Assignment

There are no new exercises for this lab. You should use this time to catch up on any of the previous labs that you have not completed. Labs Lab 15 to Lab 22 will no longer be accepted for marking after this week. Also, the second assignment is due at the end of this week.

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.