Case Studies

For the case studies, you will need to study the code in your own time, as he will only go through the highlights of the code.

All code examples will not be quite professional standard, as he's stripped away extra stuff to teach us things.

Case Study 1: User Comments

1A: Comments on a Web Page

Want to be able to leave comments and see other people's comments.

We need a database to hold the comments:

```
    CREATE TABLE comments_table
(
    comment_id INT AUTO_INCREMENT,
    username VARCHAR(255),
    comment TEXT,
    PRIMARY KEY (comment_id)
);
```

Note the TEXT datatype which can be an arbitrary length.

Post

We use a self-processing page in a python program, which will use post in its form.

We use post rather than get because:

- We want to update the database, because issuing the request more than once would do something different.
- We're sending a lot of data.

<article>

HTML 5 recommends using the <article> tag for comments.

<textarea>

Used for multi-line text boxes:

```
<textarea name="new_comment" id="new_comment" rows="5" cols="50"></textarea>
```

Still use `form_data.getfirst` with it, like with standard text inputs.

<legend>

The legend tag adds a title to the box drawn by the <fieldset> tag, used for defining sections in forms sometimes.

1B: A Web Site Comments Facility

We modify what we've made already to work on as many pages as we would like.

We want individual comments on each page.

Database

We need a slightly different database schema:

```
CREATE TABLE comments_table
(
    comment_id INT AUTO_INCREMENT,
    username VARCHAR(255),
    url VARCHAR(255) NOT NULL,
    comment TEXT,
    PRIMARY KEY (comment_id)
);
```

The 'url' field now holds the url of the page the comment was left on.

Many Pages

Each page is a python program, a self-processing page, and then we use `import` to load the code for the comments, which will be written in a separate module `comments.py`.

Comments.py

```
from os import environ
url = environ.get('SCRIPT_NAME')
```

The latter line gets the url of this program (i.e. page1 or page2).

Case Study 2: Sessions and Shopping Carts

Again, the code here is very stripped down, and is ugly and not especially functional, so it can be small enough to show us.

Background

HTTP is **stateless**, it has no memory. We need to be able to remember people for remembering people who've contacted the server in the past, or for a sequence of requests from the same client in a short period of time (a **session**).

For this we use cookies.

Cookies

Two types:

- Persistent
    - These are stored on disk and have an expiry date.
- In-memory
    - These are stored in RAM and have no expiry date, but are removed when the browser closes or your computer shuts down.

For sessions we'll use in-memory cookies.

Session IDs

We give each session an ID, which is sent back and forth in a cookie during the session.

It's created by the server, sent to the client, and sent back every time the client makes a new request.

Hashing the Server Time

```
from hashlib import sha256
from time import time

sid = sha256(repr(time()).encode()).hexdigest()
```

This code produces a unique hex number based on the server time. Even two conversations that occur at the same time will have different IDs.

Session State

This is the data kept about a session.

We store it in a database or a file (databases are often overkill) on the server side, rather than sending it back and forth.

The Shelve Module

In Python, this is a way of writing and reading files that looks as if you're interacting with a dictionary.

```
from shelve import open

session_store = open('sess_' + sid, writeback=True)

# Examples of writing to the file
session_store['firstname'] = 'Hugh'
session_store['surname'] = 'Jeegoh'

# Example of reading from the file
print(session_store.get('firstname'))

session_store.close()
```

Using `writeback=True` allows us to read and write to the file, whereas setting it to `False` only lets us read.

We interact with the file as if with a dictionary and it's written to the disk as we go.

A Simple Shopping Cart

Three python programs, a database to store available wines, and 1 file per session to hold the shopping cart for each user.

add_to_cart.py:

- Check if the user already has a cookie, take the session id if they do, or create a new one.
- Check which bottle of wine they want (using FieldStorage()) and update their shopping cart.

show_cart.py:

- Do the cookie checking.
- Retrieve quantities from the shopping cart, get the relevant names from the database, and put them in a HTML table along with the quantities from the shopping carts.

We still would need some way of purging old session stores, because otherwise the files will just build up forever.

Case Study 3A: User Authentication

You need both cookies and a session store, so that you can flag that the user is logged in, and remove that when the user logs out.

You can't delete the cookie on the user's computer, so you need to make sure they can't see the protected pages after they log out.

Since you have to have the same authentication code (involving the session store) on all protected pages, you could make this cleaner by calling a function and passing in what you want to put inside the if statement.

Case Study 3B: AJAX

Client-side scripts can also send http requests.

XMLHttpRequest Objects

Your script needs to do 5 things:

1. Create the request object.
2. Register a function that will handle the response (assuming asynchronous response handling).
3. Specify the URL and the HTTP method (e.g. GET or POST).
4. Optionally, specify any special headers that are to be sent.
5. Send the request.

Here's the code example:

```
var request = new XMLHttpRequest();
request.addEventListener('readystatechange', handle_response, false);
request.open('GET', url, true);
request.setRequestHeader('User-Agent', 'XMLHttpRequest');
request.setReguestHeader('Accept-Language', 'en');
request.send(null);
```

The 'null' is where you put the data if using a post request.

Once you get the response, it's stored in the same variable the request was stored in.

Handling Responses

```
function handle_response() {
    // Check that the response has fully arrived
    if ( request.readyState === 4 ) {
        // Check the request was successful
        if ( request.status === 200 ) {
            // do whatever you want to do with
            // request.responseText or request.responseXML
        }
    }
}
```

State 4 means the response has fully arrived.

AJAX

AJAX stands for Asynchronous JavaScript and XML, and is not a specific programming language.

It's meant to make the web more responsive and fun to use, rather than reloading the whole page for any change. E.g. when you like something on facebook, it needs to update the server.

AJAX may be slowing things down, as websites send many tiny requests all the time.