

Foundations of Software Architecture

Lab Manual

Autumn 2006

Authors

John Reekie, University of Technology, Sydney

Contributors and reviewers

Lian Loke, University of Technology, Sydney

Rohan McAdam, Honeywell Inc.

Terms of Use

This document is Copyright © H. John Reekie 2004–2005. It is distributed under the terms of the Creative Commons **Attribution-ShareAlike 2.0** license. See below.



Attribution-ShareAlike 2.0

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:



Attribution. You must give the original author credit.



Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

Creative Commons
<http://www.creativecommons.org>

The legal code of this license
<http://creativecommons.org/licenses/by-sa/2.0/legalcode/>

Contents

1	Lab 1: Execution architecture	1
1.1	Checking out your repository	1
1.2	Interactive command-line servers	2
1.3	A multi-process system	3
1.4	Startup scripts	4
2	Lab 2: Implementation architecture	5
2.1	Preliminaries	5
2.2	Initialize the database	6
2.3	Build and run the web server	7
2.4	The user management components	8
3	Lab 3: Realtime Audio Prototype	9
3.1	Database review	9
3.2	Setting up audio playback	11
3.3	Using a database	12
3.4	Completing the prototype (post-lab work)	12
4	Lab 4: Web systems and services	13
4.1	Serving HTTP	13
4.2	Web services	14
5	Lab A: .NET	19
A	Using a CVS repository	20
A.1	Checking out a CVS module	20
A.2	Basic CVS operations	21
A.3	Advanced CVS operations	21
B	Using a subversion repository	22
B.1	Checking out a subversion module	22
B.2	Basic subversion operations	22
B.3	Advanced subversion operations	22

1 Lab 1: Execution architecture

The goal of this lab session is to explore some of the basic concepts of execution architecture. You will also learn how to access the source code repository that you will be using throughout the semester.

1.1 Checking out your repository

The first step is to check out your code from the source code repository server. For this purpose, you will be using either CVS or subversion.

In the following examples, the username 'bbaggins' and the server 'mithril.middleearth.edu' is assumed. Your instructor will provide you with your actual username and the server name at the start of the lab session.

1. Open a shell. On Windows, Go to Program Files, then Cygwin, then select "Cygwin Bash Shell." (On Unix, you should already have a shell window open.)
2. Navigate to a directory where you can work. The best choice for this is usually your networked home directory. You may wish to create a new working directory-there. For example:

```
$ cd h:/
$ mkdir java-projects
$ cd java-projects
```

3. Check out a fresh copy of your team's code module.
 - If you are using CVS, follow the instructions in section A.1.
 - If you are using subversion, follow the instructions in section B.1.

You are now ready to proceed with the rest of the lab.

In your own time

Verify that you can perform a basic set of operations on the code repository, such as modifying a file, checking a modified file back in, and examining the modification logs. Change to the directory containing your working code:

```
$ cd h:/java-projects/myteam
```

Then:

- If you are using CVS, follow the instructions in section A.2.
- If you are using subversion, follow the instructions in section B.2.

1.2 Interactive command-line servers

This exercise gives you the opportunity to examine and play around with two simple client-server architectures. Each server is a simple Java program that you compile and run on your computer, and the “client” is any implementation of *telnet*.

1. Compile and run the “SimpleServer” Java program. (We assume that you know by now how to compile and run Java programs; if not, see an instructor for help during the lab session.)
2. In a shell (or MS-DOS) window, use *telnet* to connect to the server, which should be listening on port 10,000:

```
> telnet localhost 10000
```

Verify that the server is operating correctly by typing at it.

3. Open another shell (or MS-DOS) window and try to connect to the server in the same way. What happens, and why?
4. The SimpleServer class has an obvious defect: it can serve only one request at a time. The “ThreadedServer” Java program is a much improved program.

Compile and run ThreadedServer. Telnet to it again and type at it. Without quitting the first connection, create another shell window and connect to the server.

- Read the source code of ThreadedServer. What is the key architectural difference of this server to SimpleServer?
- What, in your opinion, makes it an *architectural* difference (if anything)?

1.3 A multi-process system

The simple system you ran in the previous section consists of just a single process. In terms of the architectural viewpoint presented in this course, this is:

- A single concurrent subsystem with internal concurrency (since it creates threads dynamically), that is
- stereotyped as a *service* (since its main mode of operation is to respond to requests from other subsystems)

In this section of the lab, you will be working with a slightly more complex system, which has two services. By default, one of these simply forwards requests to the other, and returns its response.

1. Make a copy of `ThreadedServer.java`, and name it `NameServer.java`. Modify `NameServer.java` so that:
 - (a) It doesn't echo each received character. (This is useful when a person connects, but not for a computer.)
 - (b) It listens on a different port, say, 8001.
 - (c) It contains an `ArrayList` (or `HashMap`), where each key is the first name of one of your team members, and the value is that team member's full name.
 - (d) When it receives a command such as "name Frank," it looks up "Frank" in the `ArrayList` and returns the result.
2. Run `NameServer` and test it.
3. Compile the program `ProxyServer.java`, and run it. `ProxyServer`, when started, needs three parameters:
 - (a) The port number that it listens on.
 - (b) The address of the other server.
 - (c) The port number that the other server listens on.

`ProxyServer` has been written to expect these parameters on the command line; here is an example:

```
> java ProxyServer 8000 localhost 8001
```

4. Connect to `ProxyServer.java` using `telnet`, and test it.
5. Modify `ProxyServer` so that it forwards only "name" commands to `NameServer`, and returns an error message otherwise. Restart it and test.

Starting the two services separately, in two separate shell windows, is quite a pain. In the next part, we will create a *shell script* to start up the system. These scripts can become quite complicated.

1.4 Startup scripts

We will use a very simple script to start the two-server system created in the previous part.

1. Open a text editor and create a new file called *run.sh*. This script will be used to start the two processes, and should look something like this:

```
#!/bin/sh echo Starting up our two services... java NameServer 8001
& java ProxyServer 8000 localhost 8001 &
```

Each line that invokes Java starts a new process that runs independently; this is indicated by the ampersand at the end of the line.

2. Execute this script, using the Cygwin Unix Shell, as follows:

```
$ ./run.sh
```

(If you are on a Unix machine, you may need to change the permissions of the script. Use `chmod a+x run.sh`.)

3. Use telnet to connect to ProxyServer and verify that the system is working as before.
4. Use the *ps* command to see which processes are running.

```
$ ps -f
```

You will recognize the two services because they will show up as Java processes. For example, you may get a listing that looks like this:

UID	PID	PPID	TTY	STIME	COMMAND
JohnR	300	1	con	15:21:32	/usr/bin/bash
JohnR	992	1	con	16:06:17	/cygdrive/c/WINNT/system32/java
JohnR	944	1	con	16:06:17	/cygdrive/c/WINNT/system32/java
JohnR	740	300	con	16:13:33	/usr/bin/ps

The PID column is the process ID. You can kill the processes by:

```
$ kill -9 992
```

```
$ kill -9 944
```

Instructor Verification

Instructor name

Date/time

2 Lab 2: Implementation architecture

This lab provides you with some exposure to the use of off-the-shelf components to quickly build the infrastructure of a software system. In this lab, you will be exploring a technical prototype for a web application.

2.1 Preliminaries

This lab uses a number of off-the-shelf components:

- The `hsqldb` Java database
<http://www.hsqldb.org/>
- The Jetty web server and servlet container
<http://www.mortbay.com/jetty/>
- The FreeMarker template engine
<http://freemarker.sourceforge.net/>
- The ant build tool, used to compile the Java code
<http://ant.apache.org/>

These components are used to create a web application in what is known as a “logical 3-tier architecture.” (We will be looking at this type of architecture more in Module 6.) Figure 1 illustrates the implementation architecture of the system we will be using in this lab.

There are two concurrent subsystems in this system: the web server, and the database server. In the first two parts of the lab (pages 6 and 7) we will exercise these two subsystems individually. In the third part (page 8), we will explore some very simple prototype implementation components built on the combined infrastructure.

To get started:

1. Update your working copy of your CVS (or subversion) module, to make sure that you have the latest version of the lab files.
2. Browse the directory tree. Draw a diagram of the directory structure so that you understand where all the different types of file are:
 - Application-specific Java source files in the *src* directory.
 - Precompiled off-the-shelf modules (supplied as binary files) in the *lib* directory.
 - Space for the application-specific compiled Java files, HTML files, and so on, in the *WEB-INF* directory.
 - System setup and configuration files in the *etc* directory.

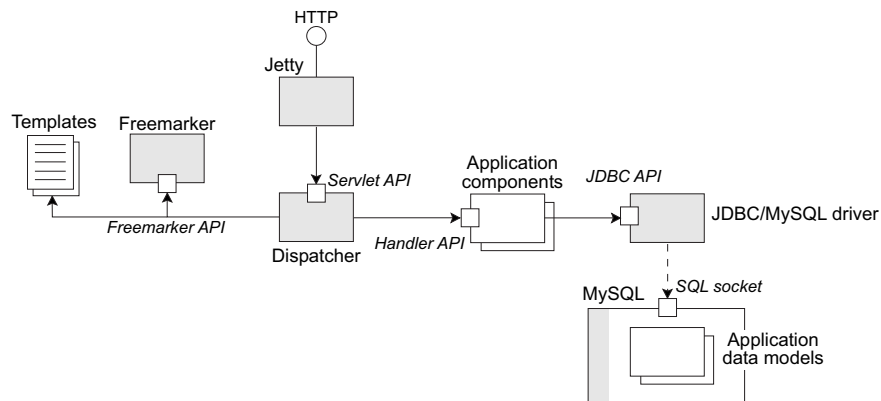


Figure 1: The implementation architecture of this system

2.2 Initialize the database

In this section, you will use connect to a database. We will use the hsqldb database, which is provided in the Jar file *lib/hsqldb.jar*. (Hsqldb is also written in Java, which is handy because it means that we don't have to install a proper database on the lab machines.)

1. Start the hsqldb server and run it in the background:

```
$ java -classpath lib/hsqldb.jar org.hsqldb.Server -database main &
```

2. Populate the database using the supplied *init.sql* script. This short script creates a single table and inserts a couple of rows into it:

```
java -classpath lib/hsqldb.jar org.hsqldb.util.ScriptTool \
  -url "jdbc:hsqldb:hsql://localhost" -database "" -script proto/init.sql
```

(Open the script file and read it, to see what it is doing.)

3. Compile and run the test program *HsqlQuery.java* to verify that the database was initialized correctly:

```
$ javac -classpath ".;lib/hsqldb.jar" proto/HsqlQuery.java
$ java -classpath ".;lib/hsqldb.jar" proto.HsqlQuery
```

(Open and read the file *HsqlQuery.java*, in order to understand what it is doing.)

4. Stop the database by using Ctrl-C in the window in which you started it.
5. You may wish to create an *init.sh* script, to (re)initialize the database.

2.3 Build and run the web server

In this section, you will be compiling code that runs in the web server, and exercising it. Referring again to Figure 1, this part exercises everything except for the database and the components in it.

Note: typically, another component called an *application framework* is used with a setup like this, to provide the infrastructure that “glues” together the web server and the template engine. Since this adds complexity and tends to obscure what is really going on, Dispatcher is used instead to provide a simple version of the same functionality. (This also has the advantage that you can read the source code in order to understand how the off-the-shelf components are “glued together.”)

1. Compile the application-specific Java source files. Because this application is more complex than just a few Java source files, we use a java build tool called *ant* to perform the build. To compile with ant:

```
$ ant compile
```

Ant is controlled by a configuration file called *build.xml*. Read this file and see if you can figure out what the various parts of it mean.

Note: if ant complains about being unable to find *tools.jar*, then execute the following command (changing the path to the proper path to Java on your system) and try again:

```
$ set JAVA_HOME=c:\j2sdk1.4.2_03
```

2. Start the web server. We have prepared a shell script for you to do this; to run it:

```
$ ./run.sh
```

3. Verify that the server has started by using a browser to view localhost on port 8080.

```
http://localhost:<port>
```

To stop the server, follow the link to the Admin interface and click on the “Exit All Servers” button. (The username and password are both “admin.”)

Note: if you don’t get the admin interface, check for errors in the console log when you started up the server. You will most likely find an error indicating that port 8080 is already in use. Locate the XML file that *configures* the server to use port 8080, and modify it to use a different port (try 8088). Then restart the server.

4. Modify *run.sh* to start the database as well as the web server.

2.4 The user management components

Let's assume that your application needs facilities to manage information about its users. We have gotten you started by creating a partial implementation in the files `user.ftl`, `User.java`, and in the user tables loaded into the database earlier in this lab. These prototype components:

- Display a list of usernames, which are fetched from a database table. Each username is hyperlinked.
- When the hyperlinked username is clicked on, display a new page with information about that user.

Do the following:

1. Add an entry for `user.ftl` and `User.java` to `Dispatcher.java`.
2. Locate the file that generates the index page of your web application. Add a link to this page, to the `user.ftl` component. Reload in the web browser to verify that the link works correctly.
3. Modify `user.ftl` so that each username on the listing is a hyperlink of the form (the `userid` parameter is of course different for each user):

```
user.ftl?action=showinfo&userid=3
```

Reload the page and verify that the links function correctly.

4. Read `User.java` in order to figure out what is it doing. Try the following:
 - (a) Change the HTTP request parameters to `user.ftl` (in the browser location field) and see if you can make the `User` component "break."
 - (b) Have `User.java` check for an HTTP parameter called "debug," and have it print out debugging information to the console if it is present.
5. Read `user.ftl` and see how it displays the information provided by `User.java`. Note the use of conditionals in the template file, of the form:

```
<#if action = "userinfo">
  <!-- HTML and FreeMarker code to display a single user -->
<#else>
  <!-- HTML and FreeMarker code to display list of users -->
</#if>
```

Instructor Verification

Instructor name

Date/time

3 Lab 3: Realtime Audio Prototype

In this lab, you will be developing an exploratory prototype for a real-time audio processing system. The system being developed is intended for sounds-effects track construction, in which a series of effects “clips” are played at specified times, in order to create the whole effects sound-track.

A track thus consists in essence of a set of pairs, each of which groups a clip with a time at which that clip starts in the effects sound-track. There may be any number of clips playing simultaneously—that is, a clip may start playing before the previous one finishes.

The concurrency view of the prototype is shown in Figure 2.

3.1 Database review

In this section you will review some basic concepts of relational databases. To do so, you will create tables to store the simple data model for student enrolments shown in Figure 3. This simple data model that shows that a major consists of some number of courses, and that students are enrolled in multiple subjects (and that each subject has multiple students enrolled in it).

1. In a browser window, open the database administration interface:

- <http://www.softwarepractice.org/lab3/>

2. Using the Query window (click on the “SQL” in the left frame), create a table to hold majors. **Note:** In order to avoid confusion with other students doing the lab, give the table a name beginning with your **own** name.

```
CREATE TABLE johnr_major (
  major_name varchar(255),
  major_id int(11) not null auto_increment,
  primary key (major_id)
);
```

3. Again using the Query window, create a table to hold subjects. **Note:** In order to avoid confusion with other students doing the lab, give the table a name beginning with your **own** name.

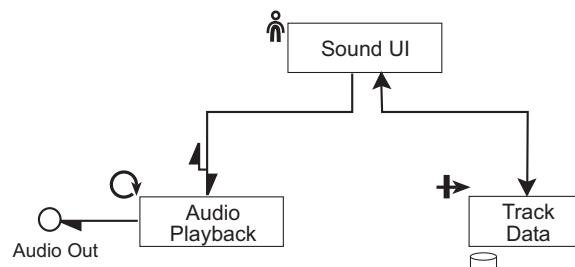


Figure 2: The concurrency view of the prototype

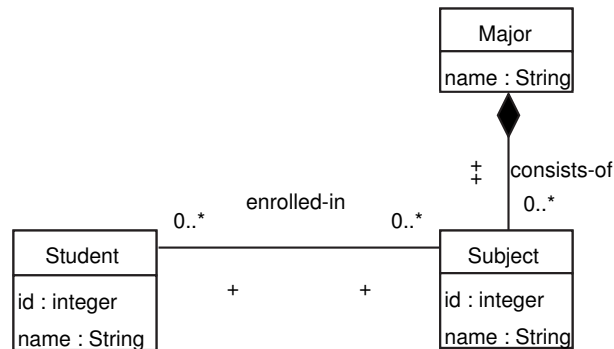


Figure 3: A simple data model

```

CREATE TABLE johnr_subject (
  subject_name varchar(255),
  subject_code varchar(5),
  major_id int(11),
  subject_id int(11) not null auto_increment,
  primary key (subject_id)
);
  
```

4. Use the web interface (the Insert tab) to add a new row to the majors table. For example, B.E. DipEngPrac. View the modified table in the Browse tab. Note the ID of this row.
5. Add some subjects that belong to this major. Assume that the ID of this major was 7—that is the value that will be set in the major_id field:

```

INSERT INTO johnr_subject (subject_name,subject_code,major_id)
VALUES ('Software Architecture','48433',7);
INSERT INTO johnr_subject (subject_name,subject_code,major_id)
VALUES ('Object-Oriented Design','48024',7);
  
```

Browse the table to see the data.

6. Now run a query to select all subjects in that major (the one with ID 7):

```

SELECT * from johnr_subject where major_id=7
  
```

7. Now run a query that also displays the name of the selected major:

8. (Optional) Add the student table. Then, to represent the many-to-many association between students and subjects, create another table with two columns—one containing student IDs, and one containing subject IDs.

3.2 Setting up audio playback

In this section of the lab, you will create a simple subsystem prototype that plays back audio clips in real time. For the audio processing you will be using the Java Media Framework (JMF). The generic JMF jar file is included with the lab materials, but if you are doing this on your own computer you may want to install one of the optimized versions of JMF. You can obtain the JMF and documentation from:

- <http://java.sun.com/products/java-media/>

The demonstration sound files included with the lab are from the following link. You should be able to find other clips and sound files on the web that you can use without copyright issues. You may even want to record your own.

- <http://www.brandens.net/files/Sounds/FX/sndfx.htm>

To proceed:

1. Update your CVS (or subversion) module to ensure that you have the latest versions of the laboratory files.
2. Browse the directory tree and make sure that you understand its layout (it's very simple).
3. Compile and run `TrackPlayer.java`. This simple program assembles sound clips into a continuous audio stream. (It doesn't do it particularly well—remember that this is the first step of a prototype.)

Note: you will need to include the `jmf.jar` file in the classpath when you compile and run `TrackPlayer`.

4. Read the source code of `TrackPlayer.java`, and make sure that you understand how it works.

3.3 Using a database

TrackPlayer has its playlist hardwired into the source code. In this section you will get the track data from a database.

1. Draw the data model of tracks and clips.
2. Figure out the table structure needed to store this data model in a relational database. Use the database admin interface from the previous part of the lab to help you construct the structure of these tables.
3. Create a test data set and use the database admin interface to help you construct a set of typical queries.
4. Write a script for the hsqldb server, which creates the tables and initializes them with your test dataset. Start up the hsqldb database and load the script.

Note: there are some syntactic differences between MySQL and hsqldb. Refer to the init.sql script from Lab 2 for the proper syntax for hsqldb.

5. Modify TrackPlayer to run the appropriate query to fetch the clip playback data.

3.4 Completing the prototype (post-lab work)

Create a simple user interface with a button that, when pressed, uses a TrackPlayer object to play back a track.

- What problem does this interface have? (For example: what would happen if you tried to play back two tracks at once?)
- Modify the program to fix the problem. Show that the modified program is able to play back two different tracks simultaneously.

Draw the concurrency view of the completed prototype.

Instructor Verification

Instructor name

Date/time

4 Lab 4: Web systems and services

The goal of this lab is to explore the technology used to build web systems.

4.1 Serving HTTP

This exercise illustrates how simple a client-server architecture can be when using the HTTP protocol.

1. Did you know that you can connect to a web server using telnet? Try, for example, the following:

```
> telnet www.google.com.au 80
```

At the prompt, enter the following:

```
> GET / HTTP/1.0
```

Then hit return twice. What this is doing is telling the Google web server that you are performing a “GET” operation on the resource named “/” (i.e. the server’s document root), using version 1.0 of the HTTP protocol.

Scroll back up the page and look at the HTTP headers returned by the server. You will see a status line that says HTTP/1.0 200 OK, followed by several lines starting with keywords such as “Content-length” and “Content-Type” and “Date.”

2. Connect to the server again, and this time request an unknown resource:

```
> GET /foo HTTP/1.0
```

What does the status line say this time?

3. You may have guessed by now that writing a basic HTTP server is not very difficult. Compile and run the `HttpServer` program; it expects a single argument when you run it, which is the port number to listen on (try using 8000). Connect to it with a web browser, using the URL

```
http://localhost:8000
```

- (a) Read the source code of `HttpServer` to be sure that you understand why it does what it does.
 - (b) Modify `HttpServer` to print out the HTTP headers received from the web browser.
4. An HTTP client is of course similar. Compile and run the `HttpClient` program. Pass it a URL as argument when you run it eg

```
$ java HttpClient http://www.google.com.au/
```

Read the source code of `HttpClient` to be sure that you understand why it does what it does.

4.2 Web services

In the first part of the Lab, you looked at the basic elements of the HTTP protocol underlying the web. In combination with Lab 2, you should now have a pretty good idea of how “Web applications” work.

What if you want your data to be readable by computers instead of people? In that case, you want to implement a “web service.” This term is used with a variety of meanings, but in essence, it generally refers to the use of HTTP and XML for communication between remote computers. Three well-known (competing) forms of web services are:

- REST (Representational State Transfer) uses URIs to represent resources, and the HTTP GET/PUT/POST/DELETE methods to manipulate those resources.
- XML-RPC (XML-Rremote Procedure Call) uses a simple XML package format embedded in HTTP POST messages to provide a general remote calling mechanism.
- SOAP (Simple Object Access Protocol) is a more complex and complete specification that defines things like data types, resource description, and namespaces. SOAP is often carried over HTTP but is not necessarily so.

In this part of the lab you will build a simple REST service. The reason for choosing REST is two-fold: i) it is the simplest of the three to implement for a Lab, and ii) it is arguably the most “Web-compatible” of the three approaches anyway (but the arguments rage on...)

Essential REST

Earlier proposals for REST focussed on several key ideas:

- Resources are represented by URIs. For example, my user database (on the next page) is at the URI *http://www.mysite.com/users/*.
- A *read* of that resource is performed only the HTTP GET method. (GET is what you used above to fetch a web page.) A *modification* of that resource, however, is performed using an HTTP PUT, POST, or DELETE method.
- Parameters sent to the service are encoded as URL strings.
- Results returned from a web service are returned as an XML packet in the body of the response.

In the modern implementation of web services through REST, these key concepts remain. The first two are, however, diluted somewhat from the early (“early” being 2002) vision of REST. Specifically:

- A single URI is used to access a service, rather than individual URIs. For examples, I would access information about the JohnR user with the URI *http://www.mysite.com/users/?username=JohnR* rather than the “more correct” *http://www.mysite.com/users/JohnR/*.
- The PUT and DELETE methods tend not to be used. Rather, POST is used for any operation that changes data. (It is still considered vitally important that a REST service not modify data with the GET method.)

Many of the publically-available web services provide SOAP or XML-RPC implementations as well as REST. See, for example, the Flickr photo-sharing service.¹

¹<http://www.flickr.com/services/api/>

Some experiments

Our web service, for this example, will maintain a list mapping user names to email addresses. In order to keep the code simple, we have written a standalone server in the file `XmlServer.java`. (Normally, the same web server will also both web applications and web services.)

1. Compile and run `XmlServer`. Use the web browser to view the output of the application at the URL:
 - `http://localhost:8000/users/`

Then view the URL:

- `http://localhost:8000/users/?username=JohnR`

Read the code of `XmlServer` to figure out why this works.

2. Make a copy of `HttpClient.java`, and rename it `XmlClient.java`. Use `XmlClient` to fetch and display the data from the URLs given above.
3. Modify `XmlClient` so that it accepts four arguments. If the second argument is the string “update,” then send the third and fourth arguments as a username to update, and an email address which is to replace the current one. To do this, `XmlClient` must:

- (a) Send a POST method instead of a GET. The HTTP request header should therefore look like this:

```
POST /users/ HTTP/1.0
```

- (b) Send parameters in the message body. The message body should look like this:

```
operation=update&username=JohnR&email=johnr@eng.uts.edu.au
```

Run `XmlClient` to perform the update and then do another GET to confirm that the update worked.

4. Modify `XmlClient` again so that if the second argument is “create,” a POST method is again sent to the server. This time, the third and fourth arguments to `XmlClient` are the username and email address of a new user.

Run `XmlClient` to create a new user with username *HenryFord* and email address *model-t@ford.com*. Verify that the update took place.

What have you learnt?

Hopefully, you can now see how the basic mechanisms of HTTP, together with some simple data-formatting help from XML, can be used to implement distributed computing systems.

- To get much further, though, you will need some XML support—see the Sun `javax.xml` package, the Apache Xerces package, or the list at
 - http://www.xml.com/pub/rg/Java_Parsers
- If you would like to experiment with XML-RPC, download and try out the XML-RPC library from
 - <http://ws.apache.org/xmlrpc/>

Further reading

- <http://www.xfront.com/REST-Web-Services.html>
- <http://webservices.xml.com/pub/a/ws/2002/02/20/rest.html>

Instructor Verification

Instructor name

Date/time

5 Lab A: .NET

This lab provides you with an introduction to the Microsoft .NET platform. .NET is an powerful piece of technology that should be evaluated when choosing infrastructure. It does have some significant constraints, such as being tied to the Windows operating system.

This lab is based on a project performed in the 48433 Software Architecture subject in Autumn 2005, by Nicholas Harris, Manjuman Hossain, Chris Chiu, and Michael Tran.

Instructor Verification

Instructor name	Date/time

A Using a CVS repository

This Appendix gives instructions on how to use CVS. The examples assume that you have a Cygwin bash shell open (on Windows), or are running in a Unix (Linux, FreeBSD, or OS/X) machine.

The examples assume the following parameters. You will of course need to modify the examples for your particular setup.

1. Your user name on the CVS server is *bbaggins*.
2. The CVS server name is *theshire.middleearth.edu*.
3. The CVS repository is located at */export/cvs/sa48433*.
4. You are a member of *myteam*.

A.1 Checking out a CVS module

The following instructions are used to get a fresh copy of your files from the CVS repository.

1. Open a shell. On Windows, Go to Program Files, then Cygwin, then select “Cygwin Bash Shell.” (On Unix, you should already have a shell window open.)
2. Verify that you are able to log into the CVS server:

```
$ ssh bbaggins@theshire.middleearth.edu
```

You will be asked to confirm that you wish to log in, and then asked for your password. After logging in, have a look at your home directory and then exit:

```
bbaggins@theshire$ pwd
bbaggins@theshire$ ls -al
bbaggins@theshire$ exit
```

3. Navigate to the directory where you wish to store your files. For example:

```
$ cd h:/java
```

4. Enter a command similar to the following to check out your code:

```
$ cvs -d :ext:bbaggins@theshire.middleearth.edu:/export/cvs/sa48433 co myteam
```

5. Verify that you have obtained the code:

```
$ cd myteam
$ ls -R
```

A.2 Basic CVS operations

After working through the following instructions, you will have enough knowledge to use the CVS repository at a basic level.

1. Edit the SimpleServer.java file. (Any other file will do, just change the commands below accordingly.) Also, have one of your team partners check out their own copy of your team's working directory. *Do this on a different computer.*
2. Check the changed file into CVS, as follows:

```
$ cvs ci -m "Modified to test checkins for Lab 1" SimpleServer.java
```

3. Have a look at the logs for that file, to verify that your change took effect:

```
$ cvs log SimpleServer.java
```

4. Ask your team partner to update his or her copy of the file to include your changes. On his or her machine, first look at the difference between the local copy of the file and the repository version:

```
$ cvs diff SimpleServer.java
```

5. Update you own copy of the file to get the changes made by your partner:

```
$ cvs update
```

Look at the file in an editor and verify that you can see the changes.

A.3 Advanced CVS operations

This section gives some advice on how to use the CVS repository at a more advanced level.

- To add a text file:

```
$ cvs add MyClass.java  
$ cvs ci -m "Added MyClass" MyClass.java
```

- To add a binary file:

```
$ cvs add SomeLib.jar  
$ cvs ci -m "Added some lib" SomeLib.jar
```

B Using a subversion repository

This Appendix gives instructions on how to use subversion. The examples assume that you have a Cygwin bash shell open (on Windows), or are running in a Unix (Linux, FreeBSD, or OS/X) machine.

The examples assume the following parameters. You will of course need to modify the examples for your particular setup.

1. Your user name on the subversion server is *baggins*.
2. The subversion server name is *theshire.middleearth.edu*.
3. The subversion repository is located at */export/svn/sa48433*.
4. You are a member of *myteam*.

B.1 Checking out a subversion module

...

B.2 Basic subversion operations

...

B.3 Advanced subversion operations

This section gives some advice on how to use the subversion repository at a more advanced level.

...