Simple is a framework, not a standalone server such as Tomcat or Jetty. Instead of a ready to run application, you are provided with a framework that will enable a server instance to be created and embedded within an application. An ideal platform for this is Spring, which will allow you to define exactly how your application is to be assembeled. Here you will be provided with a basic introduction to the framework and how it can be used to either embed Simple into your application or create your own application server based on the framework. Each section illustrates a specific feature of the framework, and examples are provided when required.

1. Getting started
2. Dealing with form data
3. Uploading files
4. Maintaining state with sessions
5. Examining the request
6. Asynchronous services

Getting started

This tutorial begins with an overview of how to implement and start a very simple server. In order to implement your own server you will need to provide header information to the client describing the content, and a message representing that content. I will assume that most readers are familiar with the HTTP protocol or are at least aware of it. Below is an implementation of a server that provides a single "Hello World" plain text response. In this implementation we also provide a "Content-Type" header as well as some other optional HTTP headers.

```
import org.simpleframework.http.core.Container;
import org.simpleframework.transport.connect.Connection;
import org.simpleframework.transport.connect.SocketConnection;
import org.simpleframework.http.Response;
import org.simpleframework.http.Request;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.io.PrintStream;

public class HelloWorld implements Container {

   public void handle(Request request, Response response) {
      PrintStream body = response.getPrintStream();
      long time = System.currentTimeMillis();

      response.set("Content-Type", "text/plain");
      response.set("Server", "HelloWorld/1.0 (Simple 4.0)");
      response.setDate("Date", time);
      response.setDate("Last-Modified", time);

      body.println("Hello World");
      body.close();
   }

   public static void main(String[] list) throws Exception {
      Container container = new HelloWorld();
      Connection connection = new SocketConnection(container);
      SocketAddress address = new InetSocketAddress(8080);

      connection.connect(address);
   }
}
```

The above class implements a standalone HTTP server that will provide a single static response to every request, regardless of the URL used. For example connecting to the following address http://localhost:8080/ when running the server on your desktop will provide you with the generated response. Thats it, a HTTP web server implemented as a single Java object.

Dealing with form data

In this section an example of how to deal with form data is provided. Form data is typically sent in a HTTP POST request as a result of submitting a HTML form, and contains name value pairs which provides details on what actions to take and what data to use. It is provided in a Form object, which is essentially a Map of string key value pairs. This form can be acquired from the Request object which represents each HTTP request event handled. Below is an example of how to acquire the form and extract a string value using an arbitrary key

acquire the form and extract a string value using an arbitrary key.

```
Form form = request.getForm();
String value = form.get(key);
```

As well as extracting single values associated with a given key, it is also possible to acquire multiple values for a given key. When multiple values are mapped to a single key value they are provided as a list of strings. The code snippet below illustrates how to acquire a list of values for a given key.

```
List<String> list = form.getAll(key);
String first = list.get(0);
String second = list.get(1);
```

Query data can also be acquired from the request, this provides the parameters sent as part of the request URI only. For example, take the URI `http://www.domain.com/index.html?name=value` which contains a single key value pair. In order to acquire these parameters the Query can be used. Typically it is best to deal with only the Form object, as it contains both the query parameters and any parameters sent within the request messsage body. For convenience the values can be acquired as integers, floats, or boolean values. For example, take the code snippet below.

```
int real = form.getInteger(key);
int decimal = form.getFloat(key);
```

For more extensive information on how form data can be acquired, take a look at the PostTest unit test.

Uploading files

In this section an example of how the integrated file upload functionality can be used is provided. Unlike the Servlet framework Simple supports multipart form encoding out of the box. This allows large files to be uploaded from HTML forms, and also facilitates SOAP with Attachments without any additional libraries. Each part of the multipart message is represented using a Part object, which contains all the headers associated with the part as well as an InputStream that can be used to read the contents of the part. Below is a code snippet which shows how to acquire a part from the request and determine its file name and type.

```
Part part = form.getPart(name);
String path = part.getFileName();
boolean file = part.isFile();
```

In order to acquire all the parts uploaded as part of the request the list of parts can be acquired. This can be acquired from the form object and can be used to iterate over the parts. Below is a code snippet showing how to acquire the list of parts.

```
List<Part> list = form.getParts();

for(Part part : list) {
   InputStream stream = part.getInputStream();
   String name = part.getName();

   if(part.isFile()) {
      name = part.getFileName();
   }
   handle(name, stream);
}
```

If you are only interested in the simple string values posted as part of the HTML form you can simply determine whether the part is a file or not and acquire the contents of the part as a string. To ensure as much transparency as possible between multipart and conventional form posts the Form object contains those parameters via the typical get method, as shown in a previous tutorial. For example, take the HTML code snippet below.

```
<form action='/upload.html' enctype='multipart/form-data' method='post'>
  <input type='text' name='subject'>
  <input type='text' name='description'>
  <input type='file' name='upload'>
  <input type='submit'>
</form>
```

From the above HTML form the encoding is `multipart/form-data`, so we will receive a file named `upload` and two other

text parameters. Because the encoding of the data is multipart the text parameters can be acquired either as a part object, or as a normal form parameter. However, the file part is accessible only as a part. This is because a file is typically either too large to convert to a string, or it contains binary content and so is best read as a stream of bytes. Below is an example of how you would acquire the values submitted by this form.

```
String subject = form.get("subject");
String description = form.get("description");
Part upload = form.getPart("upload");
String content = form.getPart("upload").getContent();
```

In order to maintain compatibility with other file upload frameworks the Request input stream will still contain the unmodified byte stream sent by the browser, or in the case of SOAP, the client. This allows frameworks such as the Apache File Upload to be used if preferred, although this will not perform as well, and is not capable of dealing with

Apache File Upload to be used if preferred, although this will not perform as well, and is not capable of dealing with arbitrary nesting of multipart bodies as you would find in SOAP protocols that use attachements, such as MM7 for example.

For performance all file uploads are loaded in to memory, however it is possible to strategize this using your own custom Allocator implementation. For an example of multipart uploads take a look at the UploadTest unit test.

Maintaining state with sessions

The HTTP protocol is stateless, which means there is no way to distinguish one client from another using the standard headers. However, an extension to the standard protocol allows state to be maintained across requests using cookies. By default simple makes use of these cookies to track clients connected to the server. This ensures that each time the client sends a HTTP request to the server it contains a unique identifier, which can be used to distinguish clients.

Using session cookies allows a Session object to be maintained on the server for a client if required. The session object created is essentially a Map object that allows clients to build up information as they move through your site. If you have used Servlets then this concept should be familiar to you, and the framework attempts to provide a similar API. The code snippet below shows how to acquire a session from the request and extract a value for an arbitrary key.

```
Session session = request.getSession();
String value = session.get(key);
```

If the session has not been previously created then it is instantiated and leased by the client within the server for a fixed period of time, typically twenty minutes. This Lease is renewed each time the client sends a request to the server where that request results in an access to the session. If the lease period expires then the session is removed and collected by the Java garbage collector. If you wish to remove the session before the expiry period due to some event occuring, such as the user logging out, then you can cancel the session as follows.

```
Session session = request.getSession(false);

if(session != null) {
   session.cancel();
}
```

For flexibility the session infrastructure is exposed via a default implementation, the SessionManager. The session manager allows session objects to be created using an arbitrary key, such as a parameter from the request URI as is done for Servlets. This allows you to write your own custom strategy which can be used to manage the session life cycle for your application. For an example of sessions in action see the SessionTest

Examining the request

The headers provided by the Request object can be acquired through various getter methods. Each method provides a different way to acquire data from the HTTP request. All header values can be acquired as integers, dates, strings, or in the case of comma separated tokens a list of strings. The following headers, which have special signifigance, can be acquired from the request using convenience methods.

```
Content-Type
Content-Length
Cookie
Connection
Accept-Language
```

The `Content-Type` header is used to describe the content sent from the client to the server. It allows the server to determine how to deal with the request body, for example, form data identified by the content type `application/x-www-form-urlencoded` is parsed by the server in order to provide parameters. The code snippet below provides an example of how to acquire the various parts of the content type header. As can be seen, the ContentType interface also provides a means to acquire the character set parameter from the header.

```
ContentType type = request.getContentType();
String primary = type.getPrimary();
String secondary = type.getSecondary();

String charset = type.getCharset();
```

The `Accept-Language` header is used by the browser to convey the locale the client exists within. It provides a means for the server to localize the content served. For example, someone from Germany may require content in the German language, alternatively someone from another country may require content in another language. It is important for the server to interpret these client preferences. In order to do so a list of Locale objects can be acquired representing the client preferences. Take the code snippet below, this shows how to acquire the client language preferences.

```
List<Locale> list = request.getLocales();

for(Locale locale : list) {
   String language = locale.getLanguage();

   if(isLanguage(Locale.GERMAN, language)) {
      handleGerman(request);
   }
   if(isLanguage(Locale.FRENCH, language)) {
      handleFrench(request);
   }
```

```
   }
   if(isLanguage(Locale.ENGLISH, language)) {
      handleEnglish(request);
   }
}
```

The `Cookie` headers can be acquired as Cookie objects that expose the individuals parts of the cookie. Each cookie can be acquired by the unique cookie name, once acquired the constituent parts of the cookie can be examined. Below is an example of how to acquire a cookie.

```
Cookie cookie = request.getCookie(name);
String path = cookie.getPath();
String value = cookie.getValue();
```

The request URI can be acquired as an Address object, which contains the query, path, and if sent, the domain, port and scheme. Typically the domain, port and scheme will only be sent if the server is used as a HTTP proxy server. For the most part either the relative path is sufficient. This can be acquired as a Path from the request as follows.

```
Path path = request.getPath();
String directory = path.getDirectory();
String name = path.getName();
String[] segments = path.getSegments();
```

If a `Content-Length` header has been sent with the request this can be acquired as an integer value from the request object. The associated method will return the length as a positive integer if it exists, if it does not exist then the method will return a negative value to indicate that the length could not be determined. Also, whether the connection is persistent or not can be acquired through the request. This is an indication of whether the combination of protocol version and `Connection` header results in a persistent connection. The code below illustrates how to determine the length of the content and the persistence of the connection.

```
int length = request.getContentLength();
boolean persistent = request.isKeepAlive();
```

Alternatively the length of the content within the request can be determined using the request input stream. The number of available bytes within the stream represents the size of the content delivered. In a scenario where the request body is sent as a chunked encoded stream then this is typically the only means of determining length.

Asynchronous services

The framework supports asynchronous services and is capable of serving content in a separate thread to the original servicing thread. This allows the server to scale to very high loads, even when the service depends on some high latency resource to become available. For example, consider a scenario where your service is dependant on some remote process to complete a task like completion of a SOAP request or waiting for a JMS message to arrive. Below is a very simple example and rather contrived example of how this can be implemented.

```
import org.simpleframework.http.core.Container;
import org.simpleframework.transport.connect.Connection;
import org.simpleframework.transport.connect.SocketConnection;
import org.simpleframework.http.Response;
import org.simpleframework.http.Request;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.io.PrintStream;

public class AsynchronousService implements Container {

   public static class Task implements Runnable {

      private final Response response;
      private final Request request;

      public Task(Request request, Response response) {
         this.response = response;
         this.request = request;
      }

      public void run() {
         PrintStream body = response.getPrintStream();
         long time = System.currentTimeMillis();

         response.set("Content-Type", "text/plain");
         response.set("Server", "HelloWorld/1.0 (Simple 4.0)");
         response.setDate("Date", time);
         response.setDate("Last-Modified", time);

         body.println("Hello World");
         body.close();
      }
```

```
   }

   public AsynchronousService(Scheduler queue) {
      this.queue = queue;
   }

   public void handle(Request request, Response response) {
      Task task = new Task(request, response);

      queue.offer(task);
   }

   public static void main(String[] list) throws Exception {
      Scheduler scheduler = new Throttle();
      Container container = new AsynchronousService(queue);
      Connection connection = new SocketConnection(container);
      SocketAddress address = new InetSocketAddress(8080);

      connection.connect(address);
   }
}
```

The above class illustrates how the completion of the response is not tied to the servicing thread. Here the response is scheduled for completion after some arbitrary throttle period. Such asynchronous is useful when the completion of the HTTP transaction requires some resource to become available.