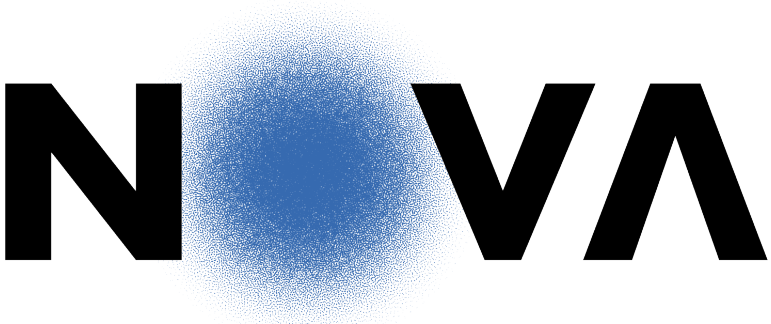


NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Departamento de
Engenharia Eletrotécnica e de Computadores

Serviços e Aplicações em Redes

2024/ 2025

Mestrado em Engenharia Eletrotécnica
e de Computadores

1st Project:
HTTP server with RESTful API

http://tele1.dee.fct.unl.pt/rit2 **Pedro Amaral**

. PROJECT OVERVIEW

This project implements a HTTP/HTTPS server that serves static files and provides a RESTful API for managing group information using MongoDB as the database. The project follows modern software design patterns and principles to create a maintainable and extensible codebase and is implemented using TCP sockets in JAVA. The server must handle GET and POST requests for both static files and REST calls that return dynamic HTML pages and update database contents. The key requirements are:

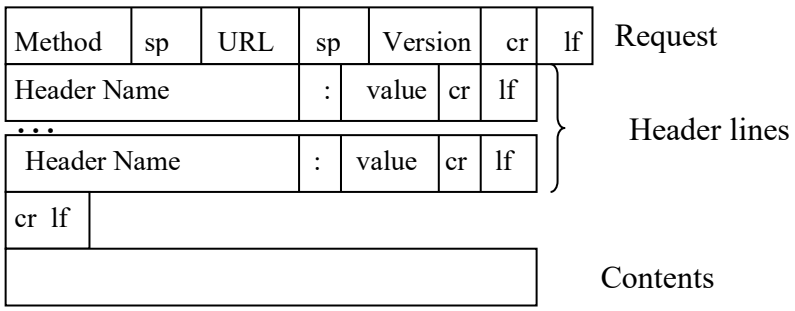
- Support for both IPv6 and IPv4
- HTTPS with HTTP redirecting to HTTPS
- HTTP 1.1 compatibility
- Configurable keep-alive connection management
- Dynamic HTML responses using group data in the MongoDB database
- HTTP header interpretation
- Form data processing
- Optional basic HTPP based user authentication

Section 2 has the basic information about HTTP relevant for this project. Section 3 presents a description of the project that is given as a starting point and outlines the project implementation.

2. HTTP PROTOCOL INFORMATION

The HTTP protocol defines a request-response interaction between a client and a server, where the exchanged messages contain readable text. There are three protocol versions: HTTP 1.0, HTTP 1.1, and HTTP 2.0.

The request message from client to server has the following structure (where sp = space and cr lf = line break). The URL corresponds to a local file name (on a server) or identifies a method to execute when used as a communication protocol for a REST API.



GET and POST methods have a similar structure. A request can include several optional header lines. Examples include:

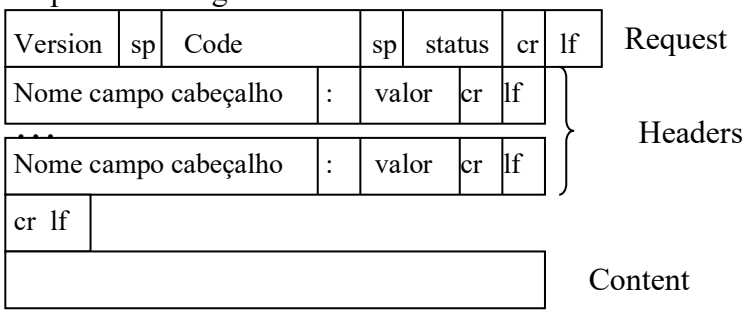
- Date header: date of the access to the server;
- Host header: the accessed server name.
- User-Agent: info on the browser type and operating system used
- Accept header: supported data formats
- Accept-Language: preferred language
- If-Modified-Since and If-None-Match headers: indicate if the file was modified since the last access,

- Connection: to indicate, in HTTP 1.1 , if the brwoser wants to keep the connection open
- Keep-Alive: for how long the connection should be maintained.
- Content-type: the type of data in the message body
- Content-Length: the number of bytes in the message
- Cookie: session data
- etc.

Then, separating the headers from the optional content there is an empty line (“\r\n”). FORM values, if they exist are sent in the content of a POST method. An example (GET) request is:

```
GET /page.html HTTP/1.1
Host: tele1.dee.fct.unl.pt
User-Agent: SAR Proxy Demo
Accept-language: pt-pt;pt
Connection: Keep-Alive
```

Response messages have a similar structure:



The most important field of the response is the status code, which defines what happened with the request. Status codes include:

- 200 OK: Success - information returned in message body
- 301 Moved Permanently: Moved to URL contained in ‘Location:’ header field
- 304 Not Modified: File was not modified
- 400 Bad Request: Request not understood by the server
- 403 Forbidden: The requested file cannot be accessed
- 404 Not Found: The requested file does not exist or cannot be accessed
- 501 Not Implemented: Request not supported by the server

Responses can include some headers that are common to requests like Content-Type, Content-Length, Connection, Date, etc. Other headers are specific to responses and serve to things like indicating the file's last modification (Last-Modified), the file's hash value (ETag), defining a state on the client (Set-Cookie), and cache control (Cache-Control in HTTP 1.1 or Pragma in HTTP 1.0). Note that the Content-Type header is mandatory whenever data is returned, indicating the MIME data type, so the browser knows how to interpret the data. HTML files are encoded in the “text/html” format, while image files can be encoded in “image/gif” or “image/jpeg”, and for the purposes of this work, the other data can be sent as “application/octet-stream”. In the case of HTML files, the Content-Encoding header is also important, with the encoding format. It is recommended that the "ISO-8859-1" format is ALWAYS used for all text files on the server. An example server response would be:

```
HTTP/1.1 200 OK
Date: Wed, 27 Feb 2012 12:00:00 GMT
Server: Apache/2.2.8 (Fedora)
Last-Modified: Thu, 28 Set 2003 19:00:00 GMT
ETag: "405990-4ac-4478e4405c6c0"
Content-Length: 6821
Connection: close
Content-Type: text/html; charset=ISO-8859-1

… { data html } …
```

There are differences between HTTP 1.0, HTTP 1.1, and HTTP 2.0 in controlling the TCP connection. In the first case (1.0), the connection is closed after each request. In the second (1.1), the header fields (Connection and Keep-Alive) can be used in the request and response to define whether to keep the TCP connection open and for how long. If both the request and the response contain a Connection header field with a value other than close, the TCP connection is not disconnected, and multiple requests can be sent over that connection. If the request includes the Keep-Alive field, it must be sent in the response. An example of values in a response header maintaining the connection for 60 seconds would be:

```
Connection: keep-alive
Keep-Alive: 60
```

Controlling the TCP connection in HTTP 2.0 will be covered in the theoretical part of the course and is outside the scope of this project.

2.2 *Cookies*

In order to have states associated with an HTTP connection (which by design is stateless), two additional header fields have been defined in a standard external [5] to HTTP. Whenever a server adds a Set-Cookie field to the message, the browser remembers the value received and returns it on subsequent requests to the same server. The syntax of this field is:

`Set-Cookie: `*`NAME=VALUE`*`; expires=DATE; path=PATH; domain=DOMAIN_NAME; secure`

1. *NAME=VALUE* – Sets the name and value of the *cookie*. It is the only required field, and you can use the simplified form "*Set-Cookie: VALUE"* for an empty name.
2. *expires=DATE* – sets the expiration date of the *cookie*. If not set, the *cookie* disappears when you close the browser. You can delete a *cookie* in your browser by sending a date earlier than the current date.
3. *domain=DOMAIN_NAME* – Sets the name of the server. The same Web server can support multiple virtual servers, distinguishing by the value of the Host header field. If not set, the *cookie* is sent based on the IP address.
4. *path=PATH* – Root document from which all requested documents must carry the same *cookie*. If not set, the *cookie* is only sent to the requested page.
5. *Secure* – if selected, the *cookie* is only sent for secure connections.

If a server sends the field:

```
Set-Cookie: CUSTOMER=WILE_E_COYOTE; path=/; expires=Wednesday, 09-Nov-
99 23:12:40 GMT
```

You will receive in the following requests for all documents below "/":

```
Cookie: CUSTOMER=WILE_E_COYOTE
```

The server can set multiple *cookies in separate* Set-Cookie *header fields*, which are returned concatenated (separated by ';') in a single Cookie header field on future requests.

2.3 Forms

Forms with HTML 2.0 support data entry fields between <form> and </form> tags. Each field has a name, type (e.g., Checkboxes, Radio Buttons, Text boxes, Submit buttons), and initial value. For example:

```
<form ACTION=”http://tele1.dee.fct.unl.pt/create” method=POST>
Name <input type="text" name =”Finger” size=40></form>
```

When you use the POST method through a submit button or indirectly through a GET with a “?” in a URL, the values are sent to the server in a compact format. It concatenates all the form fields into a string, delimited by ‘&’. Each field is encoded as “name=value”, replacing blank spaces with ‘+’ and reserved characters with their corresponding hexadecimal value in the format “%aa” (except for Internet Explorer, which replaces spaces with “%A0”). For instance, the following set of fields (represented as name=value)

```
star = Eastwood
cert = 15
movie = Pale Rider
```

would be encoded in the string "*star=Eastwood&cert=15&movie=Pale+Rider*". The equivalent invocation performed with a GET would be:

http://tele1.dee.fct.unl.pt/api/finger?star=Eastwood&cert=15&movie=Pale+Rider

More information on the use of forms can be found on pages 667-670 of the recommended textbook [4], or on various tutorial pages on the Internet (e.g. [7]).

2.4 Controlo de *caching*

When a browser receives a file, it usually stores it in a local memory, called a *cache*. In order for the file to be saved, it must not have authentication fields or *cookies*, and it must not be a dynamic page. HTTP defines several methods to control whether the file is still current. The most common ones use the "*If-Modified-Since*" and "*If-None-Match" header fields*. In the first case, in the second application, the header with the date of the last modification is added; in the second case, the header with the digital signature of the file (received in the "*ETag*" field). If the file is current, it is answered with the code 304, and with the Date, *Server*, *ETag* fields, and the connection control fields, if any. Otherwise, you should ignore the header field and return the new file.

```
Request 1:
GET /somedir/page.html HTTP/1.0
…
Response 1:
HTTP/1.0 200 OK
Last-Modified: Thu, 23 Oct 2002 12:00:00 GMT
ETag:"a4137-7ff-42068cd3"
…
Request 2:
GET /somedir/page.html HTTP/1.0
If-Modified-Since: Thu, 23 Oct 2002 12:00:00 GMT
If-None-Match: "a4137-7ff-42068cd3"
```

```
…
Response 2:
HTTP/1.0 304 Not Modified
Date: Thu, 23 Oct 2002 12:35:00 GMT
```

Other methods are to use caching control fields. In HTTP 1.0, the header field "*Pragma: no-cache*" can be returned in both requests (do not use cached values as a response) and responses (do not cache). In HTTP 1.1, the "*Cache-control*" header field must be used, which can have a subset of several possible values. The value "*no-cache*" has the same meaning as before. The value "*no-store*" in the request or response means that the response can never be cached. The value "*max-age=n*" defines the maximum lifetime of the response, or the maximum time that the file can be cached to be valid.

3. TCP SOCKETS IN JAVA

The project will be developed using VSCode as the IDE with the Dev Containers extension to allow code developing in containers and Docker Desktop to run the containers.

3.1. IPv6

The Java language supports IPv6 transparently since Java version 1.4. The `InetAddress` class handles both IPv4 and IPv6 addresses. It has the subclasses `Inet4Address` and `Inet6Address` for IPv4 and IPv6 functions, respectively. This architecture lets IPv6 addresses be supported in other java.net classes as `InetAddress` parameters are used.

To get the local IPv4 or IPv6 machine address, use the `getLocalHost` class function.

```
try {
    InetAddress addr = InetAddress.getLocalHost();
} catch (UnknownHostException e) {
    // tratar excepção
}
```

The `getHostAddress` method allows you to get the address in *string format*. The name associated with the address can be obtained with the `getHostName method`.

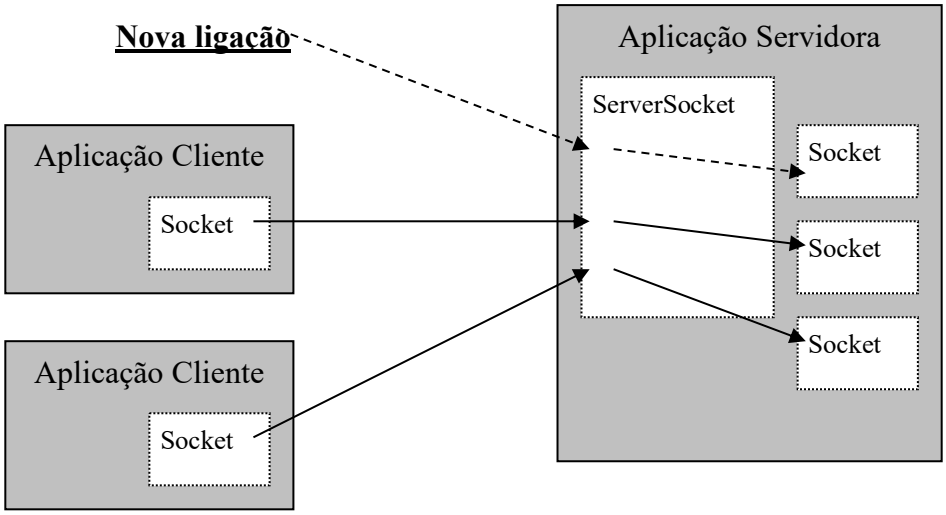
Similarly, the `InetAddress.getByName` or `InetAddress.getAllByName methods` allow you to obtain the addresses (IPv4 or IPv6) associated with a domain name.

3.2 TCP Sockets

In Java, the interface to TCP sockets is done through two classes: *ServerSocket* and *Socket*.

3.2.1 The ServerSocket class

The ServerSocket class defines a server object that can accept multiple connections. When creating an object, you specify the listening port. The `accept()` method blocks the object until a connection is received, that creates a `Socket-class` object for communication with the client.



The constructors of the *ServerSocket class* allow you to define the port, the number of new pending connections that are accepted by the object, and the IP address (the network interface) to which the connection is made. There is also a fourth constructor, which does not initialize the port, allowing you to use the *setReuseAddress()* function, before defining the port with the *bind() function*.

```
// Public Constructors
public ServerSocket(int port) throws IOException;
public ServerSocket(int port, int backlog) throws IOException;
public ServerSocket(int port, int backlog, InetAddress bindAddr) throws
     IOException;
public ServerSocket() throws IOException;
```

The two main functions allow you to receive calls and terminate the server.

```
public Socket accept() throws IOException;
public void close() throws IOException;
```

Other functions allow access to socket parameters and settings:

```
public InetAddress getInetAddress();
public int getLocalPort();
public synchronized int getSoTimeout() throws IOException;
public synchronized void setSoTimeout(int timeout) throws
      SocketException;
```

To allow you to communicate with clients and accept new connections in parallel, it is common to use multiple tasks (thread-type objects) to handle each connection.

3.2.2 The Socket Class

The *Socket* class defines a talkback object in stream mode. It can be created through a constructor or from the *accept operation*. The builder allows you to program clients: you specify the IP address and port to which you want to connect, and the builder establishes the connection.

```
public Socket(String host, int port) throws UnknownHostException,
     IOException;
public Socket(InetAddress address, int port) throws IOException;
public Socket(String host, int port, InetAddress localAddr,
     int localPort) throws IOException;
public Socket(InetAddress address, int port, InetAddress localAddr,
     int localPort) throws IOException;
```

The socket write and read operations are performed through java.io packet objects (*InputStream* and *OutputStream*), described in the next section, returned by two functions of the class. There are also functions to close the socket and to obtain information about the identity of the connection.

```
public InputStream getInputStream() throws IOException;
public OutputStream getOutputStream() throws IOException;
public synchronized void close() throws IOException;
public InetAddress getInetAddress();
public InetAddress getLocalAddress();
public int getLocalPort();
public int getPort();
public isConnected();
public isClosed();
```

Various operations to configure the parameters of the TCP protocol can be performed using methods of this class. The *getReceiveBufferSize*, *setReceiveBufferSize*, *getSendBufferSize,* and *setSendBufferSize* functions allow you to modify the size of the buffers used in the TCP protocol. The *getTCPNoDelay* and *setTCPNoDelay* functions control the use of Nagle's algorithm (*false* = off). The *getSoLinger* and *setSoLinger functions* control what happens when the connection is closed: if it is connected, the value defines the number of seconds that you wait until you try to send the rest of the data that is in the TCP buffer and has not yet been sent. If it is active, it can lead to data loss that is not detectable by the application.

```
public int getReceiveBufferSize() throws SocketException;
public synchronized void setReceiveBufferSize(int size) throws
     SocketException;
public int getSendBufferSize() throws SocketException;
public synchronized void setSendBufferSize(int size) throws
      SocketException;
public boolean getTcpNoDelay() throws SocketException;
public void setTcpNoDelay(boolean on) throws SocketException;
public int getSoLinger() throws SocketException;
public void setSoLinger(boolean on, int val) throws SocketException;
public synchronized int getSoTimeout() throws SocketException;
public synchronized void setSoTimeout (int timeout) throws
     SocketException;
```

The *getSoTimeout* and *setSoTimeout* functions allow you to configure the maximum amount of time that a read operation can be blocked before it is canceled. If the time expires, a *SocketTimeoutException exception is thrown*. These functions also exist for the ServerSocket and DatagramSocket classes.

3.2.3 Communication on TCP sockets

Socket class objects provide methods to get an object from the *InputStream class* (*getInputStream*) to read from the socket, and to get an object from the *OutputStream class*

(*getOutputStream*). However, these classes only support writing byte *arrays*. Thus, it is common to use other classes in the java.io package to wrap these base classes, obtaining greater flexibility:

To read *strings* from a socket it is possible to work with:

1. The *InputStreamReader* class processes the byte string by interpreting it as a character string (converting the character format).
2. The *BufferedReader* class stores characters received from a beam of type *InputStreamReader*, supporting the *readLine()* method to wait for the reception of a full line.

To write strings in a socket it is possible to work with:

1. The *OutputStreamWriter* class processes the string and encodes it to a sequence of bytes.
2. The *PrintWriter* class supports the methods of writing *strings* and variables of other formats (*print* and *println*) for a bundle of type *OutputStreamWriter*.
3. The *PrintStream class* supports both the methods of writing *strings* and *byte []* to a beam of type *OutputStream*.

If you want to send objects in a binary (non-readable) format, you should use the *DataInputStream* and *DataOutputStream classes*.

An example of using these classes would be the following:

```
try {          // soc representa uma variável do tipo Socket inicializada
     // Cria feixe de leitura
  InputStream ins = soc.getInputStream( );
  BufferedReader in = new BufferedReader(
     new InputStreamReader(ins, "8859_1" ));  // Tipo de caracter
     // Cria feixe de escrita
  OutputStream out = soc.getOutputStream( );
  PrintStream pout = new PrintStream(out);
  // Em alternativa poder-se-ia usar PrintWriter:
  // PrintWriter pout = new PrintWriter(
  //        new OutputStreamWriter(out, "8859_1"), true);
     // Lê linha e ecoa-a  para a saída
  String in_string= in.readln();
  pout.writeln("Recebi: "+ in_string);
}
catch (IOException e ) { ... }
```

3.3 Sockets TLS

In Java, the interface to TLS sockets is performed through the *Java Secure Socket Extension* (JSSE) [7], distributed along with JavaSE 1.4.2 and later. For the realization of servers based on blocking interfaces, this extension provides the *SSLServerSocket* and *SSLSocket* classes, with a similar operation to the *ServerSocket* and *Socket* classes presented above. To perform the management of security contexts, including certificates and public keys, it provides the *SSLContext* class, from which it is possible to create socket factories TLS servers (*SSLServerSocketFactory*) or clients (*SSLSocketFactory*). To create an HTTPS-capable TLS server, you must include an X.509 certificate in the context of the socket server. JSSE provides tools to create these certificates.

3.3.1 Creating a TLS Context (Already Done in the Provided Project)

The management of TLS contexts is done through objects of the *javax.net.ssl.SSLContext* class. The Java platform allows you to define at the start of an application, which is the file with the certificates with the local keys, loaded for static parameters of the *SSLContext* class. For

example, the following command line loads the certificates from the *keystore* file, protected with the password *password*:

```
% java -Djavax.net.ssl.keyStore=keystore -Djavax.net.ssl.keyStorePassword=password
       Server;
```

Using the *static getInstance* method on the *SSLContext* class it is possible to create a context that performs a specific security protocol, which can be "SSL", "SSLv2", "SSLv3", "TLS", and "TLSv1". The differences between these various protocols lie in the cryptographic methods they use. In the case of this work we will use TLSv1.2, since the methods used in SSL are no longer considered secure.

```
public static SSLContext getInstance(String protocol)
                                throws NoSuchAlgorithmException;
public static SSLContext getInstance(String protocol, String provider)
           throws NoSuchAlgorithmException, NoSuchProviderException;
```

Subsequently, this context must be initialized with the local private keys (*KeyManager*), and if necessary to perform TLS clients that authenticate the servers, with the public keys of the servers (*TrustManager*). In this second case, it would be necessary to provide the name of the file with the list of public keys on the command line, with the following argument: *-Djavax.net.ssl.trustStore=MyCacertsFile*.

```
public void init(KeyManager[] km, TrustManager[] tm, SecureRandom random);
```

To create a TLS server, you will need to initialize the server's local private key table, and associated X.509 certificates, which will be sent to the client during the secure connection is established. JSSE provides the *java.security.KeyStore* class to store cryptographic keys and certificates, supporting various encryption mechanisms. *KeyStore* objects are created with the *getInstance* method, where the security method used is indicated. By default, the class supports the native JSSE method (JKS), but there are tools to convert files in other formats to JKS.[1]

```
KeyStore ks = KeyStore.getInstance("JKS");
```

De seguida, é necessário carregar os dados do ficheiro com as chaves privadas e os certificados, contidos num ficheiro JKS (no exemplo seguinte, com o nome `"keyStoreName"`):

```
char[] password = getPassword();     // Pede a password ao utilizador
java.io.FileInputStream fis = new java.io.FileInputStream("keyStoreName");
ks.load(fis, password);              // Carrega os dados
fis.close();
```

O JSSE fornece a classe *javax.net.ssl.KeyManagerFactory* para suportar a instanciação de gestores de chaves privadas. Utilizando o método *getInstance* é possível criar uma fábrica de gestores associado a um tipo de certificado (geralmente "SunX509"), que posteriormente, é carregado com os certificados e chaves privadas a partir de um objecto *KeyStore* com o método *init*.

```
KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
kmf.init(ksKeys, passphrase);        // Inicializa as chaves locais
```

Finalmente, é possível criar uma instância da interface *javax.net.ssl KeyManager* utilizando o método *getKeyManagers*, e usá-lo para instanciar um contexto TLS. O código seguinte ilustra um exemplo completo de uma função de inicialização de contexto, guardado na variável pública ***sslContext***, do tipo *SSLContext*.

```
private static void initContext(String keyfilename, String password)
```

[1] e.g. conversão entre OpenSSL ou apache e JKS: http://mark.foster.cc/kb/openssl-keytool.html

```
            throws Exception {
    if (sslContext != null)    // Só corre uma vez
      return;
    try {
      // Create/initialize the SSLContext with key material
      char[] passphrase = password.toCharArray();
      KeyStore ksKeys;
      try {
        ksKeys= KeyStore.getInstance("JKS");
      } catch (Exception e) {
        System.out.println("KeyStore.getInstance: "+e);
        return;
      }
      ksKeys.load(new FileInputStream(keyfilename), passphrase);
      System.out.println("KsKeys tem "+ksKeys.size()+" chaves ");
      // Selecciona a utilização de chaves no formato X.509
      KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
      kmf.init(ksKeys, passphrase);   // Inicializa as chaves locais
      // Cria contexto TLS
      sslContext = SSLContext.getInstance("TLSv1.2");
      sslContext.init(kmf.getKeyManagers(), null, null);
    } catch (Exception e) {
      System.out.println("Falhou criacao de contexto.");
      e.printStackTrace();
      throw e;
    }
  }
```

3.3.2 TLS Server Sockets

A TLS socket server is created with an SSL socket factory object, of the *javax.net.ssl.SSLServerSocketFactory* class. You can create a factory by default, without local certificates, using the static method *SSLSocketFactory.getDefault()*. But when local X.509 certificates need to be managed, a factory should be created by invoking the *getServerSocketFactory() method* on top of a previously initialized object of the *javax.net.ssl.SSLContext* class.

```
SSLServerSocketFactory sslSrvFact = sslContext.getServerSocketFactory();
```

TLS server sockets are created by using a variant of the *createServerSocket* method, which has the same parameters as the constructors of the *ServerSocket* class presented earlier, with the same meaning.

```
public SSLServerSocket createServerSocket();
public SSLServerSocket createServerSocket(int port) throws IOException;
public SSLServerSocket createServerSocket(int port, int backlog)
            throws IOException;
public SSLServerSocket createServerSocket (int port, int backlog,
            InetAddress bindAddr) throws IOException;
```

The ***javax.net.ssl.SSLServerSocket class*** inherits from the *java.net.ServerSocket* class all of the methods presented in section 3.2, behaving like a normal server socket. Thus, the programming of an application is carried out in the same way as before. The difference lies in a set of methods specific to the *SSLServerSocket* class, which allow you to configure aspects of the TLS protocol. There are methods to select which encryption protocols can be used. Other methods control how TLS negotiation is done:

```
// Controla se o cliente tem de se autenticar – isto é enviar o seu certificado para
o servidor. Se não enviar rejeita o estabelecimento de ligação TLS.
   public abstract void setNeedClientAuth(boolean flag);
   public abstract boolean getNeedClientAuth();
// Controla se o cliente deve se autenticar – neste caso permite a ligação mesmo que
o cliente envie o certificado para o servidor.
   public abstract void setWantClientAuth(boolean flag);
   public abstract boolean getWantClientAuth();
// Por omissão funciona no modo servidor – o servidor é o primeiro a enviar o
certificado. Num servidor FTP por vezes é necessário inverter a autenticação, mas num
servidor Web não.
   public abstract void setUseClientMode(boolean flag);
   public abstract boolean getUseClientMode();
```

On the HTTPS Web server that you are going to perform in this job you must turn off client authentication.

3.3.3. Datas compatíveis com o protocolo HTTP

The HTTP protocol defines a specific format for writing dates:

```
   Thu, 23 Oct 2002 12:00:00 GMT
```

You can write and read variables with this format by using an object of the *DateFormat class*.

```
DateFormat httpformat=
    new SimpleDateFormat ("EE, d MMM yyyy HH:mm:ss zz", Locale.UK);
httpformat.setTimeZone(TimeZone.getTimeZone("GMT"));
Date writing
out.println("A data actual é " + httpformat.format(dNow));
Reading dates
try {
    Date dNow= httpformat.parse(str);
} catch (ParseException e) {
    System.out.println("Data inválida: " + e + "\n");
}
```

You can compare dates by using the *compareTo* method of objects of the *Date* class. Adding and subtracting time ranges to dates is also possible by converting the date to *long* using the *getTime method*. Since the value counts the number of milliseconds since 1970, it is enough to add or subtract the value corresponding to the interval. For example, to move forward one day would be:

```
Date d= new Date(dNow.getTime() + (long)24*60*60*1000);
```

Alternatively, you can use the *add* function of the *Calendar class* to perform the operation:

```
Calendar now= Calendar.getInstance();
now.add(Calendar.DAY_OF_YEAR, +1);
Date d= now.getTime();
```

4. SPECIFICATIONS

The goal of this work is to program a web server as an exercise to deepen the knowledge of the HTTP protocol, by implementing a "tailor-made" realization of the protocol supported directly in the use of TCP sockets. The students will have to implement the treatment of the requests, the interpretation of the several headers, in order to produce the corresponding responses to the

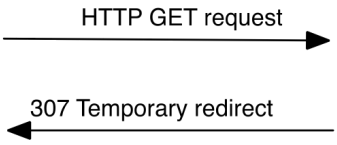
clients. The server is executed in the command line, with the parameters being hardcoded as a set of constants in the main class (Main.java). Parameters include, among others, the ports for the normal TCP server and for the TLS server, the directory of the static HTML files to be served, and the option to request or not a username and password for access.

4.1 Detailed Specification

The HTTP server must be able to support multiple clients in parallel, using individual tasks to respond to each client. To facilitate development, initial code is provided. The provided server is functional for simple requests of static files and also already provides the form to maioulate group information as a response to a GET method for an URL ending in “api”. However the provided code does not interpret nor send any HTTP heeders and is therefore not properly adhering to the HTTP standard in its responses to clients.

The project can be opened and run in the Visual Studio Code IDE and contains the definition of two development containers, one to run the server in Java and the other to run the database. You should install Docker Desktop on your machine together with VSCode, in VSCode you should install the Dev Containers extension, so you don't need to have JAVA on your machine and do not need to install mongodb, because the code will be developed and run inside a container that contains all the necessary dependencies and can connect to the database container.

The server must receive requests on the **HTTP port** or **HTTPS port**. HTTP requests must ALL be **redirected** to the HTTPS service. Thus, when a request for a page http://URL1 is received a redirect response must be sent to the browser with the address https://URL1.



If the boolean Authorization variable is set to true in the main class, HTTP authentication in basic mode (using the Authorization header) should be used in HTTPS requests, this will only be necessary if the browser does NOT send a cookie obtained in response to a previous authorization in the request.

The public final static String StaticFiles = “html” defines the root directory were static files reside in Web server. URLs in POST methods are treated as API requests and result in dynamic HTML code produced based on the parameters received in the POST.

The value of the **KeepAliveTime variable** is only used if you keep a connection open, in HTTP 1.1. Keep-Alive sets the maximum number of seconds of inactivity until the connection is disconnected (if it initializes at **0** it means that it does not disconnect if you want to keep the connection active you must initialize it with the desired value).

4.2 Project structure

The Project has the following structure:

```
project_root/
├── .devcontainer/          # Development container configuration
│   ├── devcontainer.json     # VSCode devcontainer settings
│   ├── docker-compose.yml    # Docker services (Java + MongoDB)
│   └── Dockerfile            # Java development environment
├── proj1/
│   ├── pom.xml                 # Maven project configuration
│   ├── html/                   # Static files directory
│   └── src/
│       ├── main/
│       │   ├── java/com/sar/
│       │   │   ├── config/     # Configuration classes
│       │   │   ├── model/      # Data models
│       │   │   ├── repository/ # Database operations
│       │   │   ├── service/    # Business logic
│       │   │   ├── server/     # Server components
│       │   │   └── web/        # Web handling
│       │   └── resources/      # Configuration files
│       └── test/               # Test files
```

Below is a brief explanation of the main folders and files in the project. This structure is designed to keep the codebase organized, encourage separation of concerns, and make development more maintainable.

- devcontainer/
  - devcontainer.json: Configures VSCode devcontainer settings, including extensions and post-create commands (e.g., Maven installation, log directory creation).
  - docker-compose.yml: Defines how Docker services run together, typically including a Java container and a MongoDB container.
  - Dockerfile: Specifies the Java development environment, including any required tools (e.g., Maven, MongoDB client).
- proj1/
  - pom.xml: The Maven configuration file that manages dependencies, build settings, and plugins for the project.
  - html/: Contains all static files (HTML, CSS, JavaScript, images) that are served directly by the server without additional processing.
  - src/: Holds the core source code and resources for the application.
  - main/:
    - java/com/sar/: Main Java package.
    - config/: Configuration classes (e.g., MongoConfig) that set up external services or environment specifics.
    - model/: Application data models (e.g., Group.java) representing entities stored in MongoDB.
    - repository/: Data access interfaces and implementations (e.g., GroupRepository, MongoGroupRepository) for interacting with MongoDB.

        - service/: Business logic classes (e.g., GroupService, GroupServiceImpl) that coordinate between the repository layer and the server/application logic.
        - server/: Core classes responsible for starting and managing the server (e.g., Main.java, ServerThread, ConnectionThread).
        - web/: HTTP handling logic, including:
            - AbstractRequestHandler: Template logic for handling different HTTP methods.
            - ApiHandler: Handles RESTful API requests (GET and POST) and integrates with the service layer.
            - StaticFileHandler: Serves static files from the html directory.
            - http/: Classes Request, Response, Headers, and ReplyCode for parsing/fabricating HTTP messages.
- resources/: Contains any project-specific configuration files (e.g., logback.xml for logging settings).

This structure allows each part of the application (configuration, data, logic, server, and web handling) to function independently while still interacting cleanly with the others. It helps maintain a clear distinction between concerns, ensures scalability for new features, and makes the project easy to understand for newcomers.

4.5 Testing

The web server must be compatible with any browser, you can use the browser developer tools to inspect the contents of the request and response headers.

At a minimum, the server should redirect to HTTPS and support normal file (GET) requests **and keeping the connection open** whenever the browser tells you to via the `Connection` header. You should also read all header fields and interpret at least the following headers (*`Connection`*, *`Keep-Alive`*, *`If-Modified-Since`*). The server should always return the *`Date`*, *`Server`*, *`Last modified`*, *`Content-Type`*, *`Content-Length`*, and *`Content-Encoding`* fields. You should also support the POST method, and receive and store the POST contents

To aim for a good grade, you should also develop the interaction with the database and return the pages generated accordingly.

4.6 Development of the work

The work will be developed in six weeks. It is proposed that the following goals be defined for the realization of the work:

1. **Before the first class** get the documentation and provided code.
2. **By the end the first week:** You should have set up your developing environment and be able to run the provided code.
3. **At the end of the second week** you should have implemented the reading of the request headers, the preparation of the relevant headers for responses with static files and the sending of the response headers trough the socket.
4. **By the end of the second week** you should have implemented the reuse of connections with HTTP/1.1, and the redirection to HTTPS.
5. **At the end of the third week** you should have implemented the validation of the "*`If-Modified-Since`*" header and the processing of the `Authorization` header when it is requested.

6. **At the end of the fourth week** you should have completed the API interaction to create groups in the database and the sending of an authorization cookie that can serve as an alternative to the `Authorization` header.
7. **By the end of the sixt week,** the reading/sending of cookies to control interactions with the API should be completed and you should do the final tests.

4.7 Students' Posture

Each group should consider the following:

- Program according to the general principles of good coding (use of indentation, presentation of comments, use of variables with names that conform to their functions...) and
- Proceed in such a way that the work to be done is equally distributed among the two members of the group.