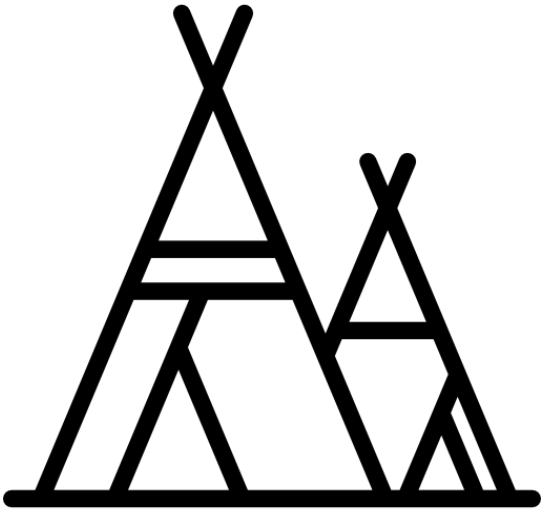
Chat seguro: Shelter



Trabajo realizado por Rafael Soria Diez y Manuel Mínguez Carretero.

Índice de contenidos

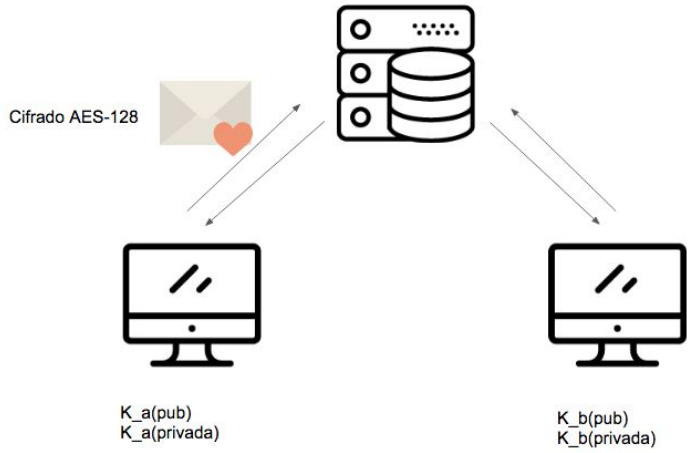
1. Introducción

Shelter es un chat de escritorio escrito en Java tanto en el lado de los clientes como en el lado del servidor. Su principal fortaleza es la seguridad ya que se aplican tres sistemas de seguridad los cuales son las funciones hash, el sistema de cifrado simétrico y asimétrico.

¿Por qué la gente debería utilizar Shelter para comunicarse con otras personas?

Hoy en día las grandes empresas hacen un uso ilícito de los datos que produce la gente, recopilan toda clase de información para usarla en su propio beneficio. En Shelter, aseguramos que las conversaciones entre dos usuarios no puedan ser escuchadas por ninguna persona u organización asegurándose con distintos sistemas de cifrado que hoy en día, se consideran seguros en su totalidad.

Shelter se basa en la siguiente idea:



Dos usuarios quieren mantener una conversación privada. Para ello necesitan establecer una contraseña AES para cifrar sus mensajes y asegurarse de tal forma, que tanto el servidor como posibles sniffers, no sean capaces de saber nada de la conversación.
El problema que se genera es que para poder intercambiar la clave AES por un canal inseguro es que dicha clave puede ser interceptada por lo que se debe aplicar un sistema de cifrado asimétrico para el intercambio de la clave de la conversación.

Para ello, uno de los usuarios debe generar una contraseña AES aleatoria en caso de que esos dos usuarios nunca han mantenido una conversación previamente. Le envía dicha clave cifrada con la clave pública del receptor para que solo él sea capaz de obtener la contraseña. En caso de que ya se haya realizado una conversación entre los dos usuarios, el servidor tiene almacenada la contraseña que ya se creó cuando se inició la conversación

por primera vez. Para asegurarse que el servidor no sabe dicha clave AES de la conversación, esta se encuentra cifrada con las claves públicas de los participantes del chat.

Una vez ambos usuarios tienen la clave de la conversación, ya son capaces de comunicarse de forma segura en un canal inseguro.

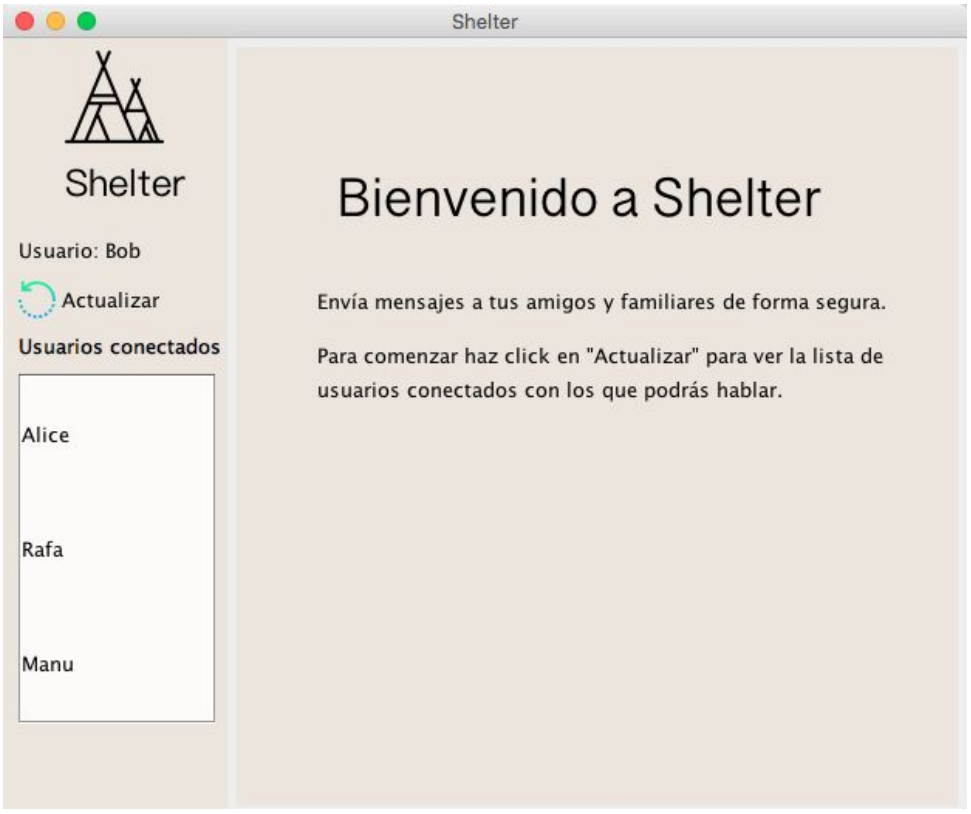
¿Por qué no usar simplemente cifrado asimétrico para cifrar todo?

El concepto del cifrado asimétrico es muy bueno ya que asegura que cualquier persona pueda cifrar un mensaje a un usuario y que solo dicho usuario sea capaz de saber el mensaje descifrado. El problema con este sistema es la lentitud del cifrado y descifrado del mensaje. Las claves tienen una longitud mucho más elevada que los cifrados simétricos, lo cual hace que el cifrado de la información requiera un tiempo mucho más elevado. Además, los algoritmos de cifrado y descifrado de la información, requieren un mayor coste computacional. En el hipotético caso de que se usase solo este sistema de cifrado, los usuarios no podrían mantener una conversación fluida porque los mensajes necesitarían más tiempo para ser procesados.

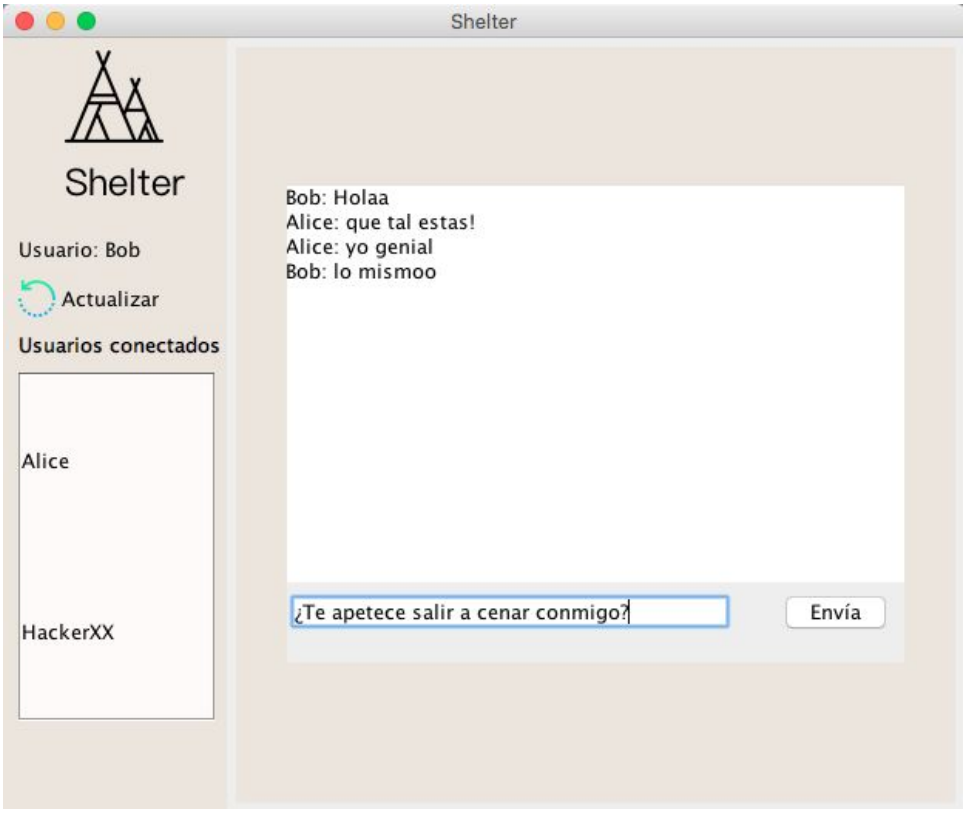
2. Recorrido por Shelter

Nada más iniciar la aplicación aparece una ventana para iniciar sesión. En caso de que no se disponga de un usuario se ofrece la posibilidad de registrarse. Las ventanas del inicio de sesión y de registro son las siguientes:

La pantalla principal nada más iniciar sesión, muestra una pequeña descripción de uso para comenzar a utilizar Shelter. Si se da click en “Actualizar” se lista todos los usuarios que están actualmente activos en Shelter.

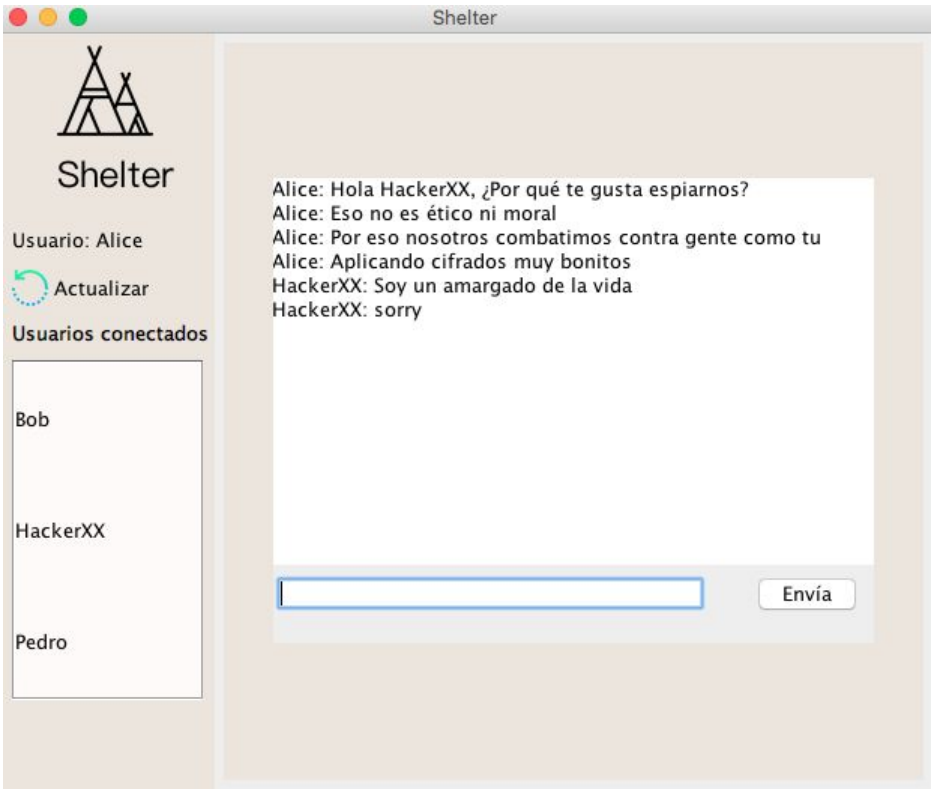


Para iniciar una conversación basta con hacer click a uno de los usuarios conectados. Una vez se hace esto, se abrirá una pantalla con el chat en cuestión.



Esta conversación es privada entre Alice y Bob por lo que si HackerXX intenta saber qué se está comentando entre ellos no sería capaz.

Las conversaciones son individuales, si Alice quiere hablar con HackerXX se iniciaría una conversación paralela entre ellos.



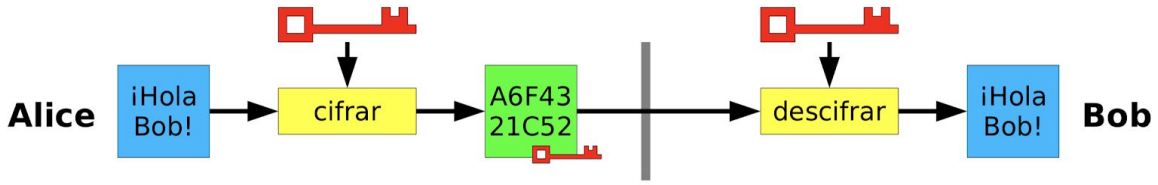
3. Sistemas de seguridad

En los sistemas de seguridad empleado como hemos comentado anteriormente hemos implementado 3 cifrados de seguridad, un cifrado asimétrico, otro simétrico y por último una función hash para cifrar el password de un usuario.

Cifrados

Cifrado Simétrico

El cifrado simétrico se caracteriza por tener una sola clave privada para el cifrado y descifrado entre usuarios. Un ejemplo gráfico es el siguiente:



Tenemos al usuario Alice que le quiere enviar el mensaje "'Hola bob" al usuario Bob, entonces cifra el mensaje con una clave y se lo pasa por un canal de comunicación al usuario Bob dónde está lo descifra con la misma clave.

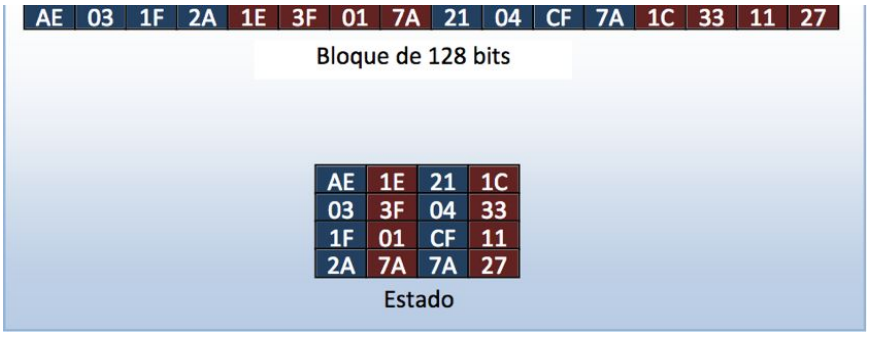
Hay que tener en cuenta que la incorporación de un tercer usuario, significaría la creación de pares de claves, ya que si se envía todo con la misma clave es imposible distinguir el emisor del mensaje.

La ventaja de usar este tipo de cifrado es la rapidez de encriptación y desencriptación, sin embargo, hay que asegurarse que cuando se esté intercambiando la información se encuentre en un canal seguro, ya que un tercero puede interceptar la clave.

Los algoritmos más empleados son DES, triple DES, IDEA y AES. Nosotros para la implementación de este cifrado hemos implementado AES-128 bits.

AES-128

AES son las siglas de Advanced Encryption Standard que es un esquema de cifrado por bloques de longitud de 128 bits, por lo tanto los datos a ser encriptados se dividen en segmentos de 16 bytes y cada segmento se lo puede ver como un bloque o matriz de 4x4 bytes al que se lo llama estado.

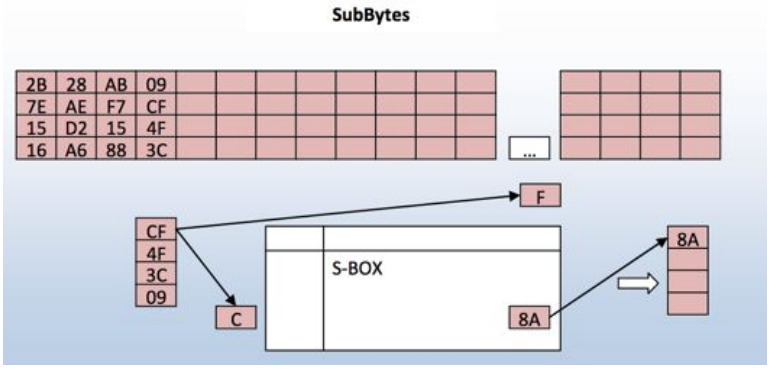


Por ser simétrico, se utiliza la misma clave para encriptar como para desencriptar, El estándar AES-128 se le puede ver como un bloque o matriz de 4x4 bytes, se generan 10 claves, las cuales son denominadas subclaves. Al obtener estas subclaves, se aplica en una series de rondas de operaciones, las cuales se dividen en:

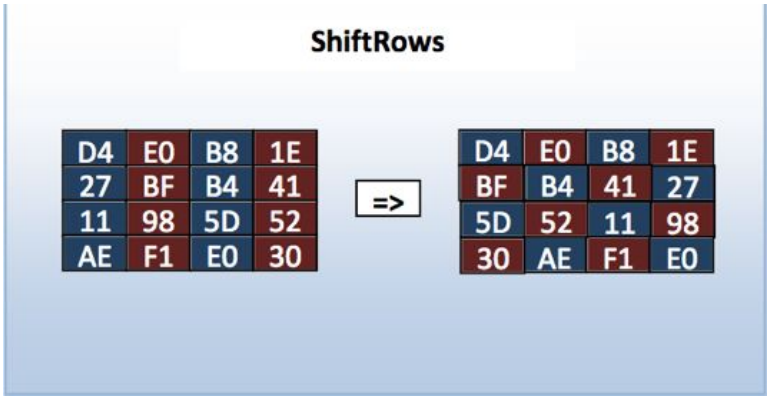
1. SubBytes
2. ShitRows
3. MixColumns
4. AddRoundKey

Una breve explicación de AES es la siguiente:

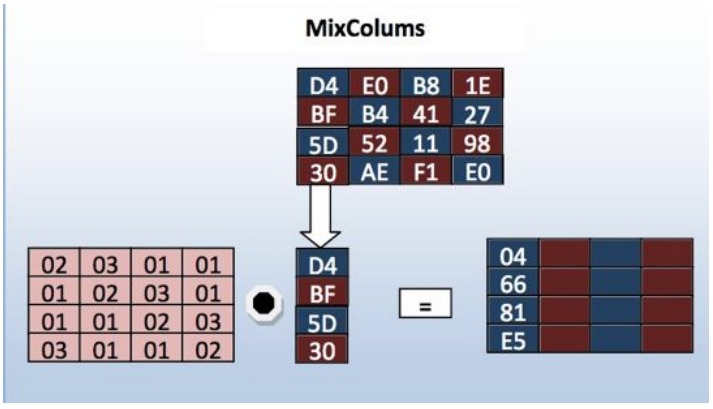
- **SubBytes**: En este paso se realiza una sustitución no lineal donde cada byte es reemplazado con otro de acuerdo a una tabla de búsqueda.



- **ShitRows:** en este paso se realiza una transposición donde cada sila es rotada de anera ciclica un número determinada de veces.



- **MixColumns:** A cada columna del estado se le aplica una transformación lineal, esto es multiplicarlo por una matriz predeterminada en el campo GF



- **AddRounkey**: Se le aplica la operación byte a byte entre el bloque a cifrar y la clave inicial.

En la práctica para implementarlo en Java hemos utilizado biblioteca donde ya está diseñado las operaciones necesarias para la encriptación y desencriptación de mensajes. Las bibliotecas utilizadas han sido: **javax cryto** y **java.security**.

La clase implementada dota de tres métodos, uno para encriptar, otro para desepcritar y una última para conseguir una key de bytes.

```
private static byte[] getKeyBytes(String key) {
    byte[] keyBytes = new byte[16];
    try {
        byte[] parameterKeyBytes = key.getBytes(ENCODING);
        System.arraycopy(parameterKeyBytes, 0, keyBytes, 0, Math.min(parameterKeyBytes.length, keyBytes.length));
    } catch (UnsupportedEncodingException e) {
        System.out.println("[Error][AES][getKeyBytes][0]: " + e.getMessage());
    }
    return keyBytes;
}
```

El método de encriptar contiene como argumentos el mensaje a encriptar y una key inicial que es generada aleatoriamente a través de la clase **PasswordGenerator**. El funcionamiento de encriptar básicamente consiste en convertir el mensaje original en Base 64.

```
public static String doEncryptedAES(String msj, String key) {
    String msjEncrypted = "error_encrypted";
    byte[] msjEncryptedbyte = null;
    byte[] keyByte = null;
    Cipher cp;
    SecretKeySpec sks = null;
    IvParameterSpec ips = null;
    try {
        msjEncryptedbyte = msj.getBytes(ENCODING);
        keyByte = getKeyBytes(key);
    } catch (NullPointerException | UnsupportedEncodingException e) {
        System.out.println(e.getMessage());
        return msjEncrypted;
    }

    sks = new SecretKeySpec(keyByte, AES);
    ips = new IvParameterSpec(keyByte);

    try {
        cp = Cipher.getInstance(TRANSFORMATION);
        cp.init(Cipher.ENCRYPT_MODE, sks, ips);
        msjEncryptedbyte = cp.doFinal(msjEncryptedbyte);
        msjEncrypted = new String(Base64.encodeBase64(msjEncryptedbyte));
        return msjEncrypted;
    } catch (NoSuchAlgorithmException | NoSuchPaddingException | InvalidKeyException |
            InvalidAlgorithmParameterException | IllegalBlockSizeException | BadPaddingException e) {
        System.out.println(e.getMessage());
        return msjEncrypted;
    }
```

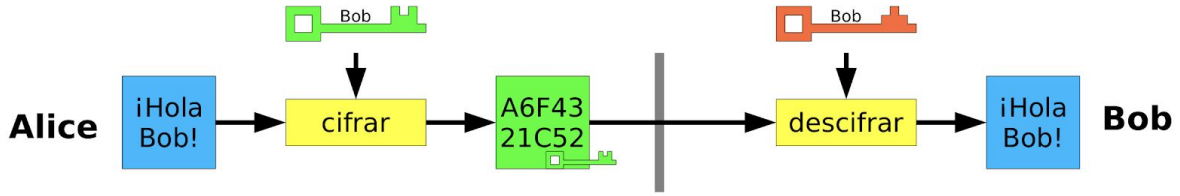
```java
public class PasswordGenerator {

        public static String NUMEROS = "0123456789";

        public static String MAYUSCULAS = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

        public static String MINUSCULAS = "abcdefghijklmnopqrstuvwxyz";

        public static String ESPECIALES = "ñÑ";

        //
        public static String getPinNumber() {
                return getPassword(NUMEROS, 4);
        }

        public static String getPassword() {
                return getPassword(8);
        }

        public static String getPassword(int length) {
                return getPassword(NUMEROS + MAYUSCULAS + MINUSCULAS, length);
        }

        public static String getPassword(String key, int length) {
                String pswd = "";

                for (int i = 0; i < length; i++) {
                        pswd+=(key.charAt((int)(Math.random() * key.length())));
                }

                return pswd;
        }
}
```

El método para desencriptar contiene como argumento el mensaje encriptado y la key para desencriptarla. Su funcionamiento es parecido al método encriptar pero con el efecto inverso, es decir, la cadena en base 64 es revertida en el mensaje original.

```java
public static String doDecryptedAES(String msjEncrypted, String key) {
    String msjDecrypted = "error_decrypted";
    byte[] msjEncryptedByte;
    byte[] keyByte;
    try {
        msjEncryptedByte = Base64.decodeBase64(msjEncrypted.getBytes("UTF8"));
        keyByte = getKeyBytes(key);
    } catch (NullPointerException | UnsupportedEncodingException e) {
        System.out.println(e.getMessage());
        return msjDecrypted;
    }
    SecretKeySpec sks = new SecretKeySpec(keyByte, AES);
    IvParameterSpec ips = new IvParameterSpec(keyByte);
    try {
        Cipher cp = Cipher.getInstance(TRANSFORMATION);
        cp.init(Cipher.DECRYPT_MODE, sks, ips);
        msjEncryptedByte = cp.doFinal(msjEncryptedByte);
        msjDecrypted = new String(msjEncryptedByte, ENCODING);
        return msjDecrypted;
    } catch (UnsupportedEncodingException | InvalidAlgorithmParameterException | InvalidKeyException |
            NoSuchAlgorithmException | BadPaddingException | IllegalBlockSizeException | NoSuchPaddingException e) {
        System.out.println(e.getMessage());
        return msjDecrypted;
    }
}
```

Cifrado asimetrico

El cifrado asimétrico se caracteriza por tener dos tipos de clave, una clave pública y otra privada. Un ejemplo gráfico es el siguiente.



Nota: la llave verde es la clave pública, la naranja clave privada.

En este caso el usuario Bob genera un par de clave haciendo accesible a todo el mundo su clave pública y guardando para él su clave privada. Cuando Alice quiere enviar un mensaje a Bob cifra el mensaje con la clave pública de Bob,por último cuando le llega a bob éste lo descifra con su clave privada.

Alguno de los algoritmos de clave asimétrico son Diffie-Hellman o RSA. En nuestro caso hemos implementado RSA.

RSA

Denominado como Rivest, Shamir y Adleman, es el cifrado asimétrico más utilizado hoy en dia. Los mensaje que se envían en esté cifrado son representados mediante números , y el funcionamiento se basa en el producto de dos números primos elegidos al azar en un rango de 10^{200} los cuales se mantienen en secreto. Se puede decir que la seguridad de este cifrado consiste en la factorización de números enteros.

Un ejemplo del cifrado es el siguiente:

En primer lugar se genera las claves:

1. Se eligen dos números primos, por ejemplo, p=3 y q=11
2. Calcula n=p*q, en este caso, n=3*11=33
3. Calcula z=(p-1)*(q-1), en nuestro caso: z=(3 – 1) * (11 – 1) = 20
4. A esta función se la denomina Función Phi o Fi de Euler.
5. Elige un número primo k, tal que k sea co-primo a z, por ejemplo, z no es divisible por k.
6. Tenemos varias opciones aquí, valores de k como pueden ser 7, 11, 13, 17 o 19 son válidos. 5 es primo, pero no es co-primo de k puesto que 20 (z) es divisible por 5.
7. Elegimos k=7 para simplificar los cálculos con un número pequeño.
8. La clave pública va a ser el conjunto de los números (n,k), es decir, *(33,7)*.

9. Ahora se calcula la clave privada. Para ello, se elige un número j que verifique la siguiente ecuación: k*j=1 (mod z)
10. En este caso *j=1 (mod 20) que da como resultado j = 3

En segundo lugar, ciframos el mensaje con P^k = E (mod n), donde:

- P es el mensaje
- N y k son la clave publica
- E es el mensaje cifrado

Un ejemplo sería el siguiente:

1. 14^7 = E (mod 33)
2. 14^7 = 105413504
3. 105413504 / 33 = 3194348.606
4. 3194348.606 - 3194348 = 0.606
5. 0.606 * 33 = 19.998 ~ 20

Donde 20 sería el texto cifrado.

Por último para descifrar el mensaje seria empleando la fórmula: E^j = P (mod n), donde:

- E es el mensaje cifrado
- J la clave privada
- P el mensaje en texto plano
- N es parte de la clave pública del destinatario

1. 20^3 = P (mod 33)
2. 8000/33 = 242,242242
3. 242*33 = 7986
4. 8000-7986 = 14

En la práctica para la implementación del algoritmo RSA en Java hemos utilizado la biblioteca java.security, aparte de bibliotecas auxiliares como Integer para hacer correcto su funcionamiento. Su implementación es la siguiente:

```
  // generate an N-bit (roughly) public and private key
  public RSA(int bits) {
      int bitlen = bits;
      SecureRandom r = new SecureRandom();
      BigInteger p = new BigInteger(bitlen / 2, 100, r);
      BigInteger q = new BigInteger(bitlen / 2, 100, r);
      modulus = p.multiply(q);
      BigInteger m = (p.subtract(BigInteger.ONE)).multiply(q
          .subtract(BigInteger.ONE));
      publicKey = new BigInteger("3");
      while (m.gcd(publicKey).intValue() > 1) {
        publicKey = publicKey.add(new BigInteger("2"));
      }
      privateKey = publicKey.modInverse(m);
}

  public BigInteger encrypt(String s, BigInteger publicKey, BigInteger modulus) {
      byte[] bytes = s.getBytes();
      BigInteger message = new BigInteger(bytes);
      return message.modPow(publicKey, modulus);
  }

  public String decrypt(BigInteger encrypted) {
      BigInteger decrypt = encrypted.modPow(privateKey, modulus);
      String resultado = new String(decrypt.toByteArray());
      return resultado;
  }
```

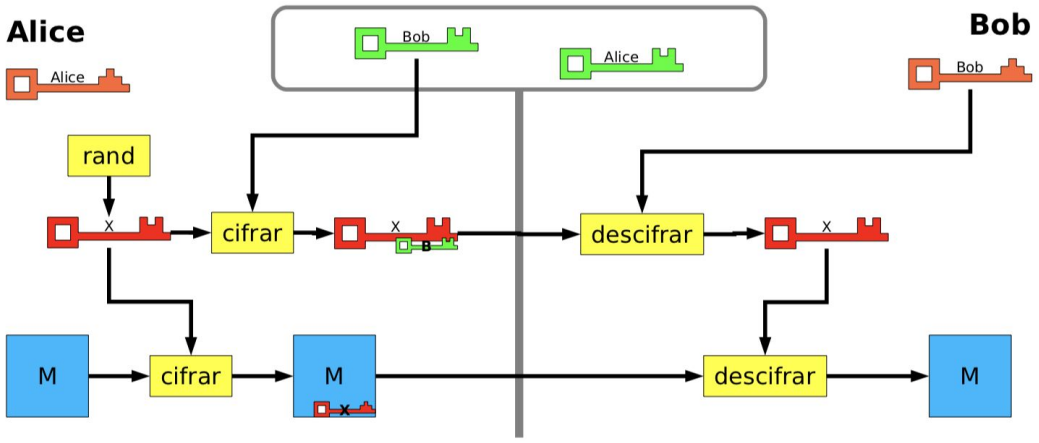
En el constructor básicamente se calcula las claves públicas, privada y el módulos, los módulos se obtiene a partir de una operación con las variables p y que que son números primos. La clave pública se obtiene a partir de un número aleatorio de números primos. Por último la clave privada se obtiene a través de la inversa de la clave pública.

Para encriptar un mensaje, se necesita el mensaje a encriptar, la cual se encripta con la clave privada (en este caso la del receptor) y el modulus.

Por último para desencriptar el mensaje se hace a través de su clave privada.

Cifrado híbrido

Para solucionar las desventajas de ambos cifrados hemos optado por la implementación de fusionar los cifrados asimétricos y simétricos. El procedimiento es el siguiente:



Nota: X es clave simétrica creada por Alice.

En este caso cuando Alice le quiere enviar un mensaje a Bob lo primero que hace es generar una clave simétrica (clave de sesión). Una vez generada la clave enviaremos el mensaje, este mensaje estaría cifrado por la clave simétrica generada. Asimismo este mensaje es cifrado por la clave pública de Bob. Una vez llegado el mensaje, Bob lo descifra con su clave privada para obtener la clave sesión para descifrar el mensaje.

Una vez explicado los cifrados anteriores es hora de ver cómo llamamos a todos los métodos para obtener la seguridad planteada.

1. Existen dos situaciones iniciales para la creación de claves públicas, cuando el usuario se loguea en la aplicación por primera vez y cuando el usuario ya está creado y guardado en la base de datos.
    a. En la primera situación cuando el usuario se va a registrar en la aplicación se llama al constructor de RSA para crear las claves y pasarlas al servidor para guardarla en la base de datos. Hay que destacar que a la hora de enviar la clave privada al servidor se encripta con AES usando la clave maestra del usuario.

```
RSA objetoRSA = new RSA(1024);
getPrivada(objetoRSA,objeto);
getPublica(objetoRSA,objeto);
getModulus(objetoRSA,objeto);
escribirSocket(objeto);
```

Donde el método **getPrivada**() encriptamos la clave privada que es enviada al servidor con AES.

```
public void getPrivada(RSA rsa,ObjetoEnvio objeto){
    BigInteger privada= rsa.getPrivateKey();

    String stringPrivada = rsa.toString(privada);
    String password = usuario.getPassword();
    stringPrivada = doEncryptedAES(stringPrivada,password);
    objeto.setPrivada(stringPrivada);


}
```

b. En la segunda situación cuando el usuario está creado en la base de datos, cuando se logea el usuario le pide al servidor que le envie sus claves públicas y privadas.

```
clave = new Clave();
String modulus = clave.getModulus(usuario);
String privada = clave.getPrivada(usuario);
String publica = clave.getPublica(usuario);
objeto.setModulus(modulus);
objeto.setPrivada(privada);
objeto.setPublicaEmisor(publica);
```

```
public String getPrivada(String usuario){
    String clave = "";
    try {
        String consulta = "SELECT clave_privada FROM ROOT.Clave WHERE Usuario = ?";
```

A la hora de cuando el cliente recibe el mensaje del servidor con los datos de las claves, el cliente desencripta el cifrado AES la clave privada con su clave maestra.

```
String privadaRSA = doDecryptedAES(objeto.getPrivada(), usuario.getPassword());
privada = rsa.toBigInteger(privadaRSA);
```

2. Una vez logueado el usuario con sus claves pública y privada, es hora de ver cómo está construida la seguridad a la hora de iniciar una conversación.

    - En primer lugar, el usuario que va a iniciar la conversación tiene que generar una clave compartida aleatoria. Dicha clave tiene que guardarse en el servidor cifrada, por lo que el usuario es el encargado de recibir la clave pública del receptor para poder así, cifrar la clave compartida con su clave pública y una nueva con la clave pública del receptor.
    - Una vez el usuario ya ha generado dos claves AES cifradas con cada clave pública, se encarga de enviarlas al servidor para que este las almacene en la base de datos. Al estar las claves cifradas, el servidor es incapaz de adivinar la clave compartida de la conversación.

```
BigInteger publica = rsa.toBigInteger(publicaReceptor);
BigInteger modulus = rsa.toBigInteger(modulusReceptor);

key = PasswordGenerator.getPassword(
        PasswordGenerator.MINUSCULAS+
        PasswordGenerator.MAYUSCULAS+
        PasswordGenerator.ESPECIALES,10);

BigInteger claveSesionReceptor = rsa.encrypt(key, publica, modulus);
BigInteger claveSesionEmisor = rsa.encrypt(key, rsa.getPublicKey(), rsa.getModulus());
```

3. Una vez creada la conversación ya podemos enviar mensajes para poder iniciar la conversación.
    - En primer lugar, cuando un usuario hace click a otro para iniciar una conversación, el servidor le devuelve los mensajes (si los tuviera) de la conversación entre los usuarios, además de la key AES encriptada con la clave pública de RSA guardada anteriormente al cliente. Este lo desencripta con la clave privada del usuario. Una vez obtenido la key de cifrado de la conversación, podremos descifrar los mensajes anteriores para poder mostrarlo en pantalla.

      ```
      resultado = resultado + doDecryptedAES(partes[i], key)
      ```

    - Por último para escribir los mensajes, encriptamos los mensajes con AES utilizando la misma clave de sesión.

```
String mensajeCifrado = doEncryptedAES(user + ": " + tfMensaje.getText(), key);
objeto.setMensaje(mensajeCifrado);
escribirSocket(objeto);
```

Función Hash

Las funciones Hash, también llamadas como función de resumen, tienen como objetivo verificar la identidad de los mensajes o en el caso que nosotros vamos a aplicar, verificar la contraseña de un usuario.

Un mensaje el cual se ha aplicado una función de resumen, tiene como resultado un hash específico con bloques de longitud n a partir de bloques de longitud m, en el caso de que el mensaje se modificase ligeramente, la función de resumen cambiaría.

La función hash aplicada en nuestra práctica es SHA2 ya que hoy en día se sigue considerando segura.

Para hacer el hash de un mensaje en Shelter se hace de la siguiente manera:

```
private void botonEnviarActionPerformed(java.awt.event.ActionEvent evt) {
    usuario = textUsuario.getText();
    ip = textIP.getText();
    puerto = Integer.parseInt(textPuerto.getText());
    password = org.apache.commons.codec.digest.DigestUtils.sha256Hex(textClave.getText() +"salt");

    this.setVisible(false);


}
```

Para iniciar sesión, el cliente aplica SHA2 para generar un hash de su contraseña y se la envía al servidor el cual comprueba si el hash que tiene almacenado de ese usuario coincide con el hash recibido.
Para el registro de un usuario, este es el encargado de generar el hash y enviárselo al servidor para que lo almacene en la BD.

4. Base de datos

La base de datos ha sido uno de los mayores desafíos en la práctica. El objetivo principal era poder tener una persistencia de los datos pero a su vez, garantizar la seguridad de los usuarios de la aplicación. Para ello era necesario asegurarse que el servidor no supiese nada privado de los usuarios y que nada se almacene en la base de datos que pueda comprometer a los usuarios.

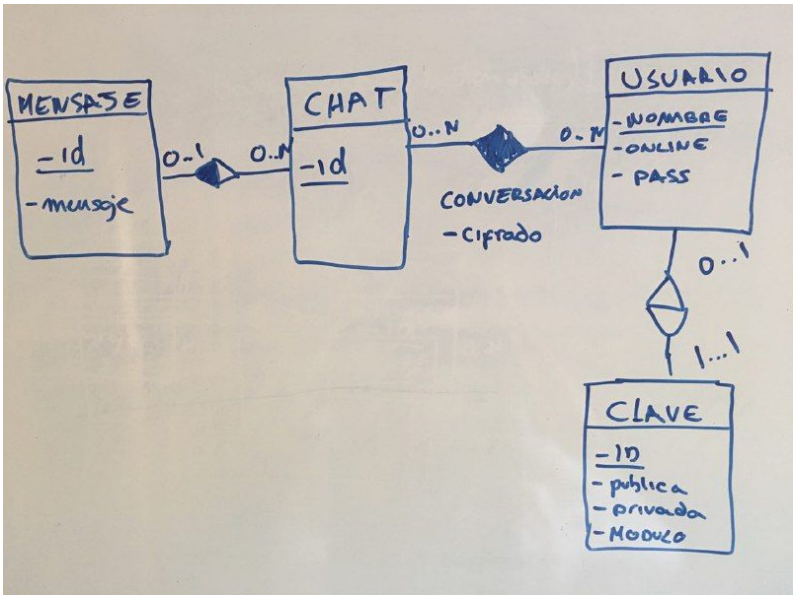
La información de carácter personal y que por tanto el servidor no puede saber es la siguiente:

- Los mensajes enviados
- Contraseña de login
- Contraseña de la conversación
- Clave privada de cada usuario

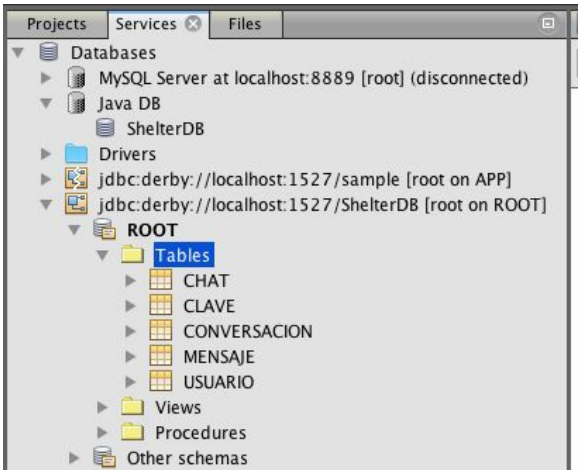
Estos cuatro puntos tienen que estar cifrados en la base de datos. Los tipos de cifrado se explicaran posteriormente uno a uno.

La base de datos que se ha creado es una DerbyDB. Apache Derby es un sistema gestor de base de datos relacional escrito en Java que está destinado para ser utilizado en aplicaciones Java. Uno de los puntos positivos de este SGBD es que tiene un tamaño de 2 MB de espacio en disco por lo que es bastante ligero.

Es esquema lógico de la base de datos empleada es el siguiente:



Una vez diseñado cómo debería ser la base de datos se ha procedido a la creación de las tablas respetando las claves primarias y ajenas. Teniendo un total de 5 tablas:



Contenido de las tablas

CHAT

La tabla chat nada más tiene un elemento de tipo numérico el cual hará de clave ajena en la tabla *Mensaje* y parte de la clave primaria en la tabla *Conversacion*.

#	ID
1	1

CLAVE

La tabla *Clave* almacena las claves de los usuarios, las claves públicas, privadas y el módulo para RSA.

En cuanto a la clave privada de RSA, se ha cifrado previamente con AES-128 utilizando como clave la clave maestra del usuario.

La clave privada cifrada es generada por el cliente y enviada posteriormente al servidor, para que este no sepa el contenido de esta.

Las columnas de la tabla Clave son las siguientes:

#	ID	USUARIO	CLAVE_PRIVADA	CLAVE_PUBLICA
1	1	Alice	9Cta3uQk8cLisDeGNC/aDdFyNjxQVLPG...	5
2	2	Bob	Rol+1azcEOpFkcNf9cOA60hPzoQJgbJDi...	11
3	3	HackerXX	VDMm8Auvoc3KYvN6Ahxmc9LOxfn0r4C...	7

MODULUS
832593354311126006541822408 07...
120206040224740652632807237 22...
866421737916481267504812566 26...

CONVERSACION

La tabla *Conversacion* tiene como clave primaria la unión entre el identificador de Chat y el Nombre de usuario. Lo que hace es almacenar que usuarios pertenecen a qué chats indicando también, cuál es la contraseña AES asociada a dicha conversación.

Esa contraseña AES esta cifrada con la clave pública del usuario participante en ese chat, haciendo que el servidor no sepa nunca cuál es la clave para descifrar los mensajes de las conversaciones.

Por tanto, los elementos de *Conversacion* son los siguientes:

#	USUARIO	CHAT	CIFRADO
1	Bob	1	21774326953125281515447800182323530 8...
2	Alice	1	21648580614552294288845536541074053 6...

MENSAJE

La tabla Mensaje tiene como clave ajena el chat asociado. Además también están almacenados todos los mensajes almacenados en Shelter cifrados con la clave AES especificada en la tabla *Conversacion*.

Las columnas son las siguientes:

#	ID	MENSAJE	CHAT
1	1	f+Xmv85XD2/bXTjl9bYwRw==	1
2	2	Ku0EGfleuBqGoljUhySv59AUuhwNx0gUnlOpTNoecMA=	1
3	3	L0lEco5TvoF6KqMqMbl7wqL6fiz10c9aQOVRuM0ArL4=	1
4	4	K+/44B2n6X6YH6Jhi4eQew==	1

USUARIO

La tabla Usuario almacena el nombre de usuario y la contraseña del login. Se utiliza para relacionarla con la clave Conversación y Clave así como para comprobar el login de los usuarios.

Para almacenar la contraseña de login del usuario se ha aplicado el hash SHA2 ya que no es necesario recuperar el contenido de la contraseña, simplemente se necesita comparar si el hash generado por el usuario al loguearse es el mismo que el almacenado en la base de datos.

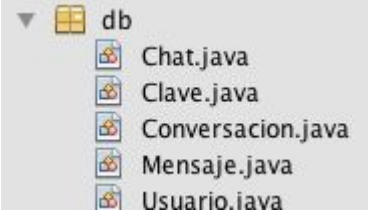
La tabla contiene lo siguiente:

#	NOMBRE	ONLINE	PASSWORD
1	Alice	☐	dd8028c8192aa4aacee2b939212034759480072 9...
2	Bob	☐	91b11e65a0128df751cf0e43b1a10cab81e81128 0...
3	HackerXX	☑	3eea6ecb352af0c391c18f4651ede410cf4da8585 2...

<u>Uso de la BD en Java</u>

Se ha aplicado una especie de modelo MVC, es decir, Model-View-Controller. La clase principal no accede directamente a la base de datos sino que delega las acciones a una clase dedicada al manejo de esta.
No hemos realizado clases controladoras ya que la complejidad requerida no es muy alta, tan solo con una clase Modelo por tabla lo hemos considerado suficiente.

Las clases creadas en Java son las siguientes:



Estas clases son las encargadas de interactuar con la persistencia de datos usando conexiones JDBC.

Todas las clases pueden añadir, eliminar, modificar elementos y algunas de ellas también están preparadas para poder listar elementos de la base de datos en base a unos parámetros.

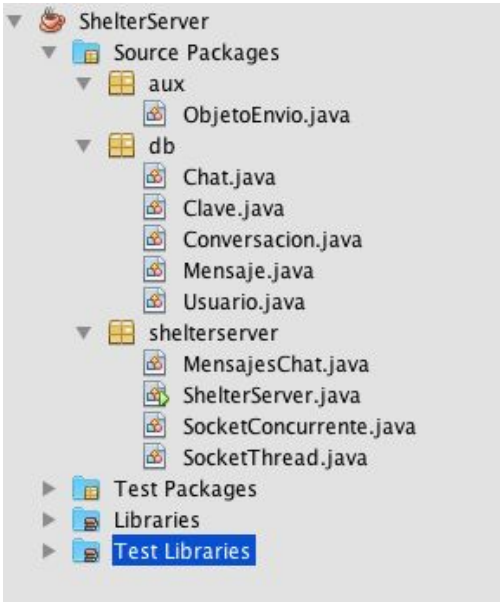
5. Guía de instalación

A continuación, se documentará paso a paso todo lo que se debe de hacer para realizar una correcta instalación tanto de los clientes como del servidor.

Primero de todo debe iniciar el servidor, para ello hay que importar la carpeta del proyecto al programa Netbeans. Necesitamos utilizar este programa ya que la base de datos es de Oracle Derby y el programa Netbeans lo instala por defecto. Es más fácil instalar simplemente el programa Netbeans que instalar el sistema gestor de base de datos por nuestra cuenta.

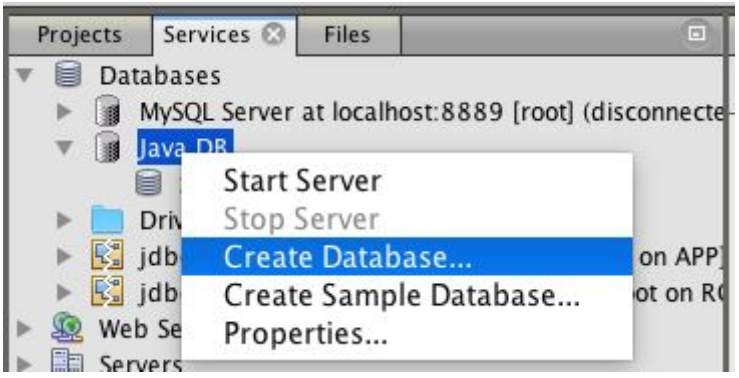
Para importar el proyecto simplemente habrá que clickar en Archivo -> Abrir proyecto.

Una vez abierto se tiene que ver así:

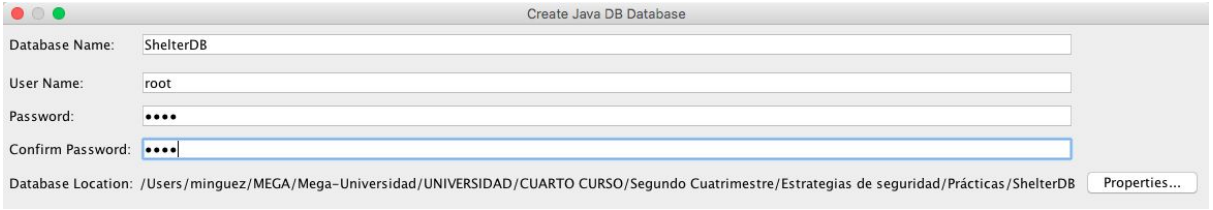


El siguiente paso antes de lanzar el servidor, es iniciar la base de datos. Para ello en el menú de navegación que está encima del proyecto, se deberá clickar en “Servicios” y una vez salga el nuevo menú, se deberá desplegar el icono con el nombre de “Base de datos” y posteriormente hacer click derecho en “Java DB”.

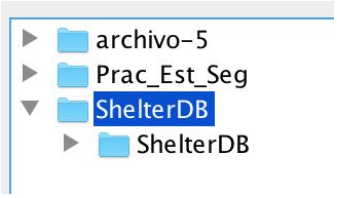
Una vez se haga se deberá clickar a “Crear Base de Datos”:



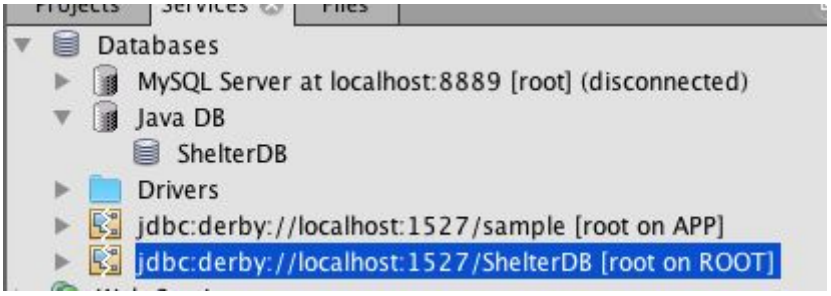
Y poner los siguientes campos:



Donde el usuario y contraseña es “root” y la ubicación de la base de datos tiene que ser donde se ha guardado la base de datos proporcionada. En la ubicación se debe seleccionar la carpeta ShelterDB que está ***dentro*** de la carpeta ShelterDB (hay una dentro de la otra con el mismo nombre). Si no se hace este paso no se va a poder configurar correctamente.

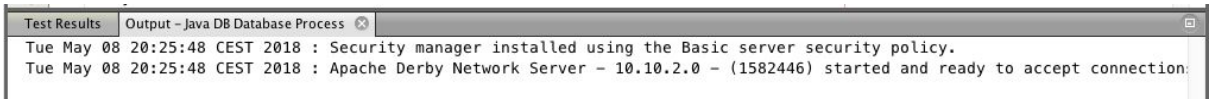


Una vez hecho este paso aparecerá un nuevo elemento en el menú el cual deberemos hacer doble click para iniciar la base de datos:

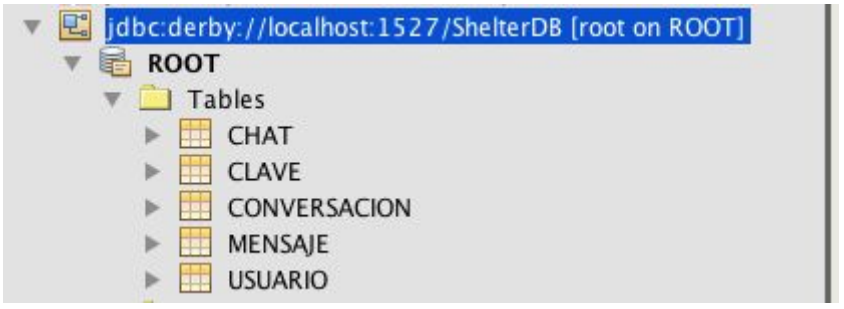


En el caso que nos de un error, hay que detener el servidor en ShelterDB y conectarlo otra vez. En este intento nos pedirá otra vez el nombre y la contraseña (root y root) este fallo se da en algunas versiones de Netbeans debido a que no coge las credenciales desde un primer momento.

Saldrá el siguiente mensaje en la terminal de Netbeans:



Y ya podremos ver el contenido de la base de datos:



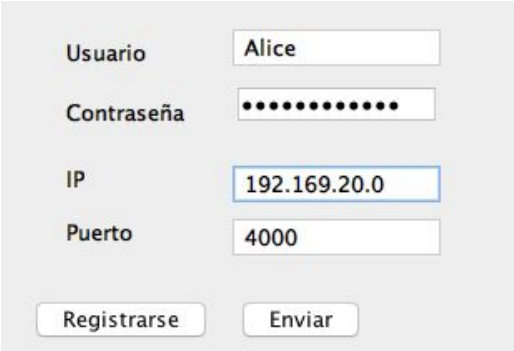
Pudiendo hacer cualquier tipo de sentencia SQL sobre ella.

6. Despliegue

Una vez ya se haya lanzado la base de datos el siguiente paso es iniciar el servidor. Para ello se hace haciendo click derecho sobre el proyecto y pulsando “Ejecutar”:

Cuando el servidor esté corriendo, el siguiente paso será iniciar los clientes en los ordenadores que se desee. Para iniciar los clientes se deberá hacer doble click sobre el archivo shelter.jar o lanzarlo desde la terminal en caso de utilizar Linux.

Ya abierto el programa, para su correcta utilización se deberá poner la dirección IP del servidor. Por ejemplo, si el servidor estuviese en una maquina cuya dirección ip privada es 192.168.20.0 se debería hacer de la siguiente manera.



Teniendo en cuenta la dirección IP del servidor, el cliente será capaz de registrar nuevos usuarios, iniciar sesión y utilizar el chat Shelter sin problemas.