Sistemas Orientados a Servicios
Grado en Ingeniería Informática - Doble Grado Informática y ADE

Práctica: Definición e implementación de un servicio web JAVA

La práctica consiste en implementar un servicio web, *ETSIINFSocial*, empleando las herramientas de Axis2 para Java. El servicio se deberá poder desplegar en Tomcat.

El servicio *ETSIINFSocial* simula una red social donde los usuarios pueden publicar/consultar sus estados, añadir amigos, consultar la lista de amigos y consultar el estado de los amigos. Para acceder a estas operaciones los usuarios deberán darse de alta en la red social (*addUser*) e iniciar sesión (*login*). El usuario deberá cerrar sesión (*logout*) cuando no acceda a la red y se podrá dar de baja en la red (*removeUser*). Estas operaciones de gestión de usuarios (*addUser, login, logout*) deben ser redirigidas al servicio web *UPMAuthenticationAuthorization,* si bien en el despliegue del servicio *ETSIINFSocial* se debe crear un usuario *admin* en la red social, que habilite el resto de las operaciones; este usuario debe ser gestionado en el servicio *ETSIINFSocial*, no por el servicio*. UPMAuthenticationAuthorization.*

El documento WSDL que define el servicio *ETSIINFSocial* se encuentra disponible en la siguiente dirección:

http://porter.dia.fi.upm.es:8080/practica2223/ETSIINFSocial.wsdl

Las operaciones en Java de este servicio web (*ETSIINFSocial*) generadas desde ese fichero WSDL son:

1. **AddUserResponse addUser (Username username)**

Esta operación añade el usuario *username* a la red social. Solo el usuario *admin* puede añadir usuarios. Esta operación debe usar la operación *addUser* del servicio *UPMAuthenticationAuthorization*. La respuesta (AddUser*Response*) contiene la contraseña que ha generado el servicio *UPMAuthenticationAuthorization* y un *booleano* que indica si la operación se ha realizado correctamente. En caso de que el admin intente añadir un usuario ya registrado o si un usuario distinto al admin llama a esta operación, el booleano será *false*.

2. **Response login (User user)**

Cada llamada a esta operación comienza una nueva sesión para un usuario (*user*). El parámetro *user* tiene dos elementos: nombre de usuario (*name*) y contraseña (*pwd*). La respuesta (*Response*) indica si la operación *login* tiene éxito (la llamada a la operación *login* del servicio *UPMAuthenticationAuthorization*)

Si se llama repetidas veces a la operación *login* en una sesión activa con el mismo usuario ya autenticado, ésta devuelve *true* independientemente de la contraseña utilizada. La sesión del usuario en curso sigue activa.

Si se llama a la operación *login* en una sesión activa del usuario U1 con un usuario U2 (misma instancia del *stub*), distinto al usuario autenticado (U2 distinto de U1), el método devuelve *false* independientemente de si la contraseña utilizada es correcta o no. El usuario U2 no tendrá sesión valida y, por tanto, no podrá acceder a ninguna otra operación. La sesión del usuario en curso (U1) sigue activa.

Un mismo usuario puede tener varias sesiones activas simultáneamente. Si ese usuario decide cerrar una de sus sesiones, solo se cerrará la sesión elegida, dejando activas todas las demás.

El *login* del usuario admin no se debe gestionar a través del servicio *UPMAuthenticationAuthorization*.

3. **void logout()**

Esta operación cierra la sesión del usuario que la invoca. Si esta operación es llamada sin que el usuario haya iniciado sesión (*login* correcto) la llamada no hace nada.

Si un usuario tiene varias sesiones abiertas (ha hecho varias llamadas a la operación *login* con éxito), una llamada a la operación *logout* solo cerrará una sesión. Para cerrar todas las sesiones deberá llamar a la operación *logout* tantas veces como sesiones hay abiertas.

4. **Response removeUser (Username username)**

Esta operación elimina al usuario *username* invocando la operación *removeUser* del servicio *UPMAuthenticationAuthorization*. Solamente el usuario admin y el propio usuario que se quiere eliminar puede llamar esta operación con éxito, habiéndose autenticado el usuario previamente. La respuesta (*Response*) devuelve *true* si la operación elimina al usuario correctamente, en caso contrario se devuelve *false* (un usuario llama a la operación con otro usuario como parámetro o el admin se intenta eliminarse a sí mismo, por ejemplo). Al borrar un usuario se borra toda la información relativa a él.

Se puede eliminar a usuarios con sesiones activas; en este caso, cualquier petición posterior deberá comportarse como si el usuario no existiera. El usuario admin no puede eliminarse.

5. **Response changePassword (PasswordPair password)**

Esta operación accede al método *changePassword* del servicio *UPMAuthenticationAuthorization* para cambiar la contraseña del usuario. El parámetro *password* incluye la contraseña actual (*oldpwd*) y la nueva (*newpwd*). Si hay varias sesiones abiertas de un usuario que llaman a esta operación, solo quedará registrado el último cambio. La respuesta (*ChangePasswordResponse*) devuelve si la operación ha tenido. La operación devuelve *false* si se intenta acceder sin haber hecho previamente *login* con éxito o la contraseña *oldpwd* es incorrecta.

El usuario *admin* puede cambiar su contraseña, si bien esta gestión no se debe realizar llamando al servicio *UPMAuthenticationAuthorization*.

6. **Response addFriend (Username username)**

Esta operación añade un amigo al usuario que la invoca (y viceversa). El parámetro *username* tiene el nombre del usuario a añadir. No se almacenarán amigos repetidos. El resultado *(Response*) incluye un *boolean* (*result*) que es *true* si se añade a un amigo que está registrado en la red social. Devuelve *false* si el usuario no ha hecho *login* con éxito o si el amigo a añadir no existe en la red. Esta relación es recíproca, es decir, si un usuario U1 añade como amigo a U2 eso implica que U2 es amigo de U1.

7. **Response removeFriend (Username username)**

Esta operación elimina un amigo de los amigos del usuario que la invoca (y viceversa). De manera análoga al método anterior, el parámetro *username* tiene el nombre con el usuario del amigo a eliminar. La operación indica si se ha eliminado correctamente al amigo, siendo *false* si el usuario no ha hecho *login* con éxito o si el amigo a eliminar no existe en la red o no está en su lista de amigos.

8. **FriendList getMyFriends()**

Esta operación devuelve la lista de amigos del usuario (*FriendList*) con un *boolean* (*result*) que es *true* si el usuario que llama a la operación ha hecho *login* previo con éxito y un *array* con los nombres de usuario de los amigos (*friends*). Si el usuario no ha llamado con éxito a la operación *login*, devolverá *false.*

9. **Response publishState(State state)**

Esta operación permite añadir un nuevo estado (*state*) al usuario que llama a la operación. El parámetro *state* tiene un campo *message* de tipo *String* con el texto del mensaje del estado del usuario. La operación devuelve *true* si el usuario llama a este método después de haber hecho *login* con éxito, en caso contrario devuelve *false*.

10. **StatesList getMyStates()**

Esta operación devuelve los últimos 10 estados publicados por el usuario actual(*StateList*) y un *boolean* (*result*) que será *true* si se ha hecho *login.* La operación devuelve *false* si el usuario no ha hecho un *login* previo.

11. **StatesList getMyFriendStates(Username username)**

Esta operación devuelve los últimos 10 estados (*StateList*) de un amigo *y* un *boolean* que es *true,* si el usuario que llama a la operación ha hecho *login* previo con éxito y es amigo del usuario *username.* La operación devuelve *false* si el usuario no ha hecho *login* previo o no es amigo del usuario *username*.

El WSDL del servicio ***UPMAuthenticationAuthorization*** se encuentra disponible y funcionando en: http://porter.dia.fi.upm.es:8080/axis2/services/UPMAuthenticationAuthorizationWSSkeleton?wsdl2

Este servicio tiene las siguientes operaciones:

1. **LoginResponseBackEnd login(LoginBackEnd login)**

El parámetro *login* tiene dos elementos: nombre de usuario (*name*) y contraseña (*password*). Esta operación comprueba que el usuario existe y tiene la contraseña suministrada. La respuesta (*LoginResponseBackEnd*) es un *boolean* que indica si la operación tiene éxito.

2. **AddUserResponseBackEnd addUser(UserBackEnd user)**

Esta operación añade un usuario al sistema. El parámetro *user* tiene el nombre de usuario del usuario a añadir. La respuesta (*AddUserResponseBackEnd*) tiene un campo *result* de tipo *boolean* que es *true* si la operación tiene éxito y un campo *password* de tipo *String* con la contraseña autogenerada para el nuevo usuario. La operación devuelve *false* si se intenta añadir un usuario con un nombre de usuario ya registrado.

3. **RemoveUserResponse removeUser(RemoveUser removeUser)**

Esta operación borra un usuario del sistema. El parámetro *removeUser* tiene el nombre de usuario. La respuesta (*RemoveUserResponse*) tiene un campo *result* de tipo *boolean* que es *true* si la operación tiene éxito (el nombre de usuario y la contraseña son correctos). La operación devuelve *false* si se intenta borrar un usuario que no existe.

4. **ChangePasswordResponseBackEnd changePassword(ChangePasswordBackend changePassword)**

Esta operación permite que un usuario ya registrado pueda cambiar su contraseña. El parámetro *changePassword* incluye el nombre del usuario (*name*), la contraseña actual (*oldpwd*) y la nueva (*newpwd*). La respuesta (*ChangePasswordResponse*) tiene un campo *result* de tipo *boolean* que es *true* si la operación tiene éxito, es decir, la contraseña actual coincide con la contraseña actual del usuario y se ha realizado el cambio de contraseña. La operación devuelve *false* en caso contrario.

5. **ExistUserResponse existUser(Username username)**

Esta operación permite averiguar si un usuario está registrado en el sistema. El parámetro *username* incluye un campo *String* (*name*) con el nombre del usuario a buscar. La respuesta (*ExistUserResponse*) tiene un campo *result* de tipo *boolean* que es *true* si el usuario existe en el sistema.

Requisitos del servicio web *ETSIINFSocial*

1. En el momento en que se despliegue el servicio *ETSIINFSocial*, debe tener al usuario *admin* con nombre de usuario ***admin*** y contraseña ***admin***. Solo puede haber un *admin* en el sistema y éste debe ser gestionado en el servicio *ETSIINFSocial* (no en el servicio de *UPMAuthenticationAuthorization*).
2. La información de los usuarios (*username*, *password*) se gestiona en el servicio *UPMAuthenticationAuthorization.*
3. La información de los estados y amigos de un usuario debe ser almacenada en el servicio *ETSIINFSocial* (en memoria) obligatoriamente. Las implementaciones con ficheros o bases de datos no son válidas.
4. Se deberán hacer pruebas con distintos clientes conectados al mismo tiempo.
5. El estado del servicio tiene que persistir en memoria a las sesiones de los clientes. Si se añade un estado en una sesión y se cierra la sesión, cuando el usuario vuelva a acceder, al consultar la lista de estados, se le devolverá una lista que contenga el último estado añadido.
6. Para aprobar la práctica, ésta **deberá funcionar correctamente con el software incluido en la máquina virtual** (disponible en Moodle) y descrito en la guía de instalación de herramientas (con JDK versión 1.7 y axis2 versión 1.6.2).

Se pide:

- Implementar el servicio web *ETSIINFSocial* en Java empleando Axis2 (versión 1.6.2).
- Implementar un programa cliente que acceda al servicio web que pruebe el servicio desarrollado con distintos usuario.
- Hacer pruebas con distintos clientes.

La práctica debe realizarse en grupos de tres alumnos. **Solo se entregará una práctica por grupo** a través de Moodle.

FECHA DE ENTREGA: 4-6-2023 hasta las 23:55.

SE COMPRUEBAN COPIAS Y PLAGIOS

NO UTILIZAR REPOSITORIOS PÚBLICOS DE GITHUB

UTILIZAD EL FORO PARA DUDAS, EVITAD PUBLICAR CÓDIGO

Instrucciones para la entrega de la práctica:

Un alumno de los alumnos del grupo deberá subir a Moodle el fichero comprimido (.tar.gz, .rar, .zip, .7z) con una carpeta llamada *appellido1appellido2apellido3* con el siguiente contenido:

– **Una carpeta llamada "servicio" con todo el código fuente del servicio**, la carpeta *resources* y el fichero build.xml. El formato de la carpeta tiene que estar listo para que ejecutando el comando "ant" dentro de la carpeta "servicio" se cree el fichero con el servicio (extensión .aar).

– **Una carpeta llamada "cliente"** con todo el **código fuente del cliente**.

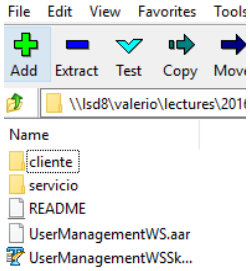
– Una copia de la clase *skeleton* con la implementación del servicio. (es.upm.etsiinf.sos.ETSIINFSocialSkeleton.java)

– El fichero de despliegue .aar para desplegar el servicio en Tomcat.

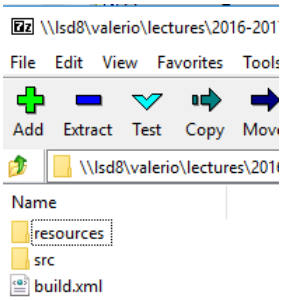
– Un README con los datos de los integrantes del grupo: Nombre Apellidos y número de matrícula de cada uno.

Ejemplo de archivo de entrega

- Grupo formado por: Juan Blanco, Carmen Ruiz y Paco Negro el fichero rar debe llamarse: blancoruiznegro.7z (u otra extensión de archivo comprimido)
- Explorando la carpeta "blancoruiznegro" se ve:



- Explorando la carpeta "servicio" hay:



El nombre completo de la clase *skeleton* debe ser: **es.upm.etsiinf.sos.ETSIINFSocialSkeleton.java**