Sistemas distribuidos – Memoria de la práctica

Curso 2018-2019

Héctor Manuel Luaces Novo | hector@luaces-novo.es | 33343779N

Sumario

Introducción

El presente documento conforma la memoria solicitada por el enunciado de la práctica de la asignatura. En ella, explicaré los detalles de la implementación de la aplicación comparándola con los requisitos que se piden en el enunciado.

Previo a entrar en detalles me gustaría aclarar la aproximación que se siguió para realizar la programación, la estructura de directorios de la práctica, así como describir qué contenidos se entregan en la misma.

Cómo se realizó la codificación

Para realizar la práctica he utilizado el IDE Eclipse 4.8.0 en una máquina con Ubuntu 18.04. El JDK utilizado fue OpenJDK en su versión 11, con lo que confío que no haya problemas derivados de la versión de Java al estar la programación realizada acorde a las especificaciones iniciales.

He intentado seguir buenas prácticas a la hora de programar, asegurándome de que el código estaba desacoplado, de que implementaba patrones diseños para problemas comunes y de que seguía un conjunto de buenas prácticas, tales como:

- Documentar todos las clases y métodos usando *Javadoc.*
- Programar la práctica de forma secuencial y construyendo código sobre código testeado.
- Utilizar baterías de pruebas unitarias para comprobar que los cambios no introdujesen regresiones de errores o cambios que rompiesen la funcionalidad.
- Seguir todos los cambios realizados en un sistema de control de versiones (git). Con la práctica entrego el repositorio en formato *bundle*, por si se quisiera ver la evolución de la codificación en su log.

Adicionalmente, me aseguré en todo momento de que lo que implementaba seguía no solo las especificaciones de la práctica, si no también los comentarios del equipo docente en el foro que pudiesen sugerir recomendaciones, fallos habituales o prácticas a evitar.

No obstante, mi vida personal me impidió dedicarle todo el tiempo que me gustaría a la práctica, lo que se traduce en que algunas cosas no están tan *finas* como me gustaría que estuviesen y en que no he tenido tiempo para implementar la parte opcional de la práctica que solicitaba borrar los *trinos* todavía no leídos, si bien la programación actual hace que esa implementación sea sencilla.

Nótese que esta memoria no tratará la documentación específica de la implementación (variables, métodos, etc.) puesto que **el código se entrega completamente documentado con Javadoc**.

Estructura de directorios de la práctica

A continuación se ilustra el árbol de directorios de la práctica, así como sus contenidos:

- Directorio Raíz
  - **basededatos**: directorio que contiene el código fuente del módulo de base de datos.
  - **bin**: aquí pueden encontrarse los ficheros *.jar* ya compilados de los distintos módulos que conforman la práctica.
  - **common**: en este directorio se encuentra código común que es compartido por las tres aplicaciones. Dicho código se usa como librería y no genera ningún fichero *.jar*.
  - **doc**:
    - **git**: directorio en el que puede encontrarse un bundle con el repositorio de *git* que contiene los detalles de la implementación de la práctica.
    - **assets**: imágenes y otros ficheros que son usados en esta memoria.
      - **funcionamiento.webm**: vídeo que ilustra el funcionamiento de la aplicación.
    - **diagramas**: imágenes *.png* de los diagramas de clases de esta memoria.
  - **servidor**: directorio que contiene el código fuente del módulo servidor.
  - **usuario**: directorio que contiene el código fuente del módulo de usuario
  - **hector_luaces_novo.pdf**: el presente documento.
  - **README.md**: documento que existe sencillamente para documentar el repositorio de *git* en el que se guardó el código de esta aplicación.
  - **basededatos.sh**: Script *bash* que invoca a java para ejecutar el *.jar “./dist/basededatos.jar”*.
  - **servidor.sh**: Script bash que invoca a java para ejecutar el .jar “./dist/servidor.jar”.
  - **usuario.sh**: Script bash que invoca a java para ejecutar el .jar “./dist/usuario.jar”.

Notas sobre la ejecución de la aplicación

La práctica exigía poder levantar los servicios usando los comandos “*basededatos*”, “*servidor*” y “*usuario*”. No me quedó muy claro qué es lo que esperaba el equipo docente: si un conjunto de tres aplicaciones con nombres homónimos o una sola que al ejecutar permitía al usuario escoger qué servicio levantar.

Puesto que de las dos la más normal me parecía la primera, es así como se ha desarrollado: en forma de tres aplicaciones relacionadas pero independientes que se entregan en tres ficheros *.jar*.

Los tres servicios pueden levantarse en cualquier orden, si bien se recomienda ejecutar primero la base de datos, después el servidor y por último los usuarios. Cualquiera de las aplicaciones, sin importar el orden en el que se ejecuten, se encargará de levantar el registro RMI en el caso de que éste no esté funcionando.

En la práctica se usa el puerto 1111 para el registro RMI, el 1112 para el controlador de base de datos, el 1113 para el controlador “servidor” y los puertos superiores al 1114 para las conexiones utilizadas por los clientes.

Esto implica que solo puede haber una instancia de la base de datos y del controlador “servidor” ejecutándose a la vez. Sin embargo, como es de esperar, puede haber tantos clientes conectados como sea necesario (siempre que haya puertos disponibles a los que enlazarse).

La aplicación de cliente se encargará, gracias a la abstracción del registro RMI incluida en la práctica, de localizar puertos libres en los que poder enlazarse.

Arquitectura de la aplicación

En el capítulo anterior ya adelanté parte de la arquitectura, pero aprovecho esta sección para expandir eso y entrar en detalles.

La práctica desarrollada está compuesta de 4 proyectos de Eclipse que se encargan de gestionar cosas diferentes:

- El proyecto “***basededatos***” se encarga de implementar la programación de la base de datos.
- El proyecto “***servidor***” implementa el controlador que encapsula el servicio gestor y de autenticación.
- El proyecto “***usuario***” (al que a veces me refiero como cliente) implementa la programación usada por la aplicación que será usada por el usuario final.
- Por último, el proyecto “***común***” no es más que un proyecto usado para compartir código entre los otros tres.

Todos los proyectos de la aplicación han sido nombrados con el prefijo *es.uned*, pues es el que se usaba en uno de los videos que se facilitaba en la práctica con fines docentes y me pareció apropiado.

Los paquetes usados separan el código de las distintas partes de la aplicación en secciones lógicas para hacerlo mas fácil de mantener y, en algunas ocasiones, para evitar que otros paquetes recurran a las implementaciones de clases concretas y tengan que limitarse a usar sus interfaces.

Dichas interfaces, por norma general, están definidas en el proyecto “común” (paquetes “*es.uned.comun.**”) con el fin de que el resto de proyectos use únicamente éstas interfaces para las declaraciones y manejo de variables con esos tipos.

El resto de proyectos implementan, de forma privada cuando procede, las distintas interfaces usando para ello los tipos correspondientes. Sirva como ejemplo la interfaz “*BasededatosInterface*” que declara la funcionalidad que ha de cumplir el controlador homónimo y que es implementada únicamente por el paquete al que da nombre.

Semejante aproximación garantiza el encapsulamiento y que el código esté desacoplado y que, si fuese necesario, pudiesen cambiarse las implementaciones el día de mañana si fuese necesario (p.ej.: podríamos cambiar la implementación del *ServicioDatos* para que en lugar de trabajar con información en memoria trabajase con alguna base de datos en particular.

Entidades y servicios

El enunciado de la práctica habla de tres entidades: base de datos, servidor y usuario. Cada una de estas entidades proporciona una API exportada por sus interfaces que les permite dar distintos servicios.

Estas operaciones dependen de servicios diferentes, es decir, cada una de las entidades usa uno o más servicios y se encarga de utilizar estos para poder dar la funcionalidad que prometen en sus interfaces.

En mi caso he seguido una aproximación similar en la que he llamado a estas entidades c*ontroladores*. Así, tenemos tres controladores “*Basededatos*”, “*Servidor*” y “*Usuario*”.

Todos mis controladores pueden tener una relación de *usa* con un conjunto de 0 o más servicios. Estos servicios son instancias de objetos que proporcionan servicios para que los controladores puedan manejarlos.

En algunos casos las operaciones que ofrecen estos servicios son muy similares a las del controlador. Por ejemplo: la base de datos posee un servicio llamado *ServicioDatos* para el que hace poco más que actuar como un *wrapper*.

Sin embargo, el controlador *Servidor* tiene que coordinar el *ServicioAutenticacion* y el *ServicioGestor* para poder cumplir con las funcionalidades que promete en su interfaz. Sirva como ejemplo que un proceso de *login* requiere el uso de ambos servicios: el de autenticación para comprobar las credenciales y generar una sesión y el gestor para verificar si ese *login* tiene trinos pendientes que ha de leer.

La estructura que siguen cada uno de los controladores se detallará a continuación en los distintos diagramas de clases.

Otras clases secundarias

Dejando de lado los actores principales de nuestra aplicación (controladores y servicios) el aplicativo se apoya en varias clases auxiliares para realizar su trabajo, las cuales pueden ser más o menos independiente o, a su vez, formar otro pequeño sistema con sus propias dependencias.

Un ejemplo de clase independiente usada por el sistema es el *ControladorRegistro*, que no es más que una clase común (y por ende declarada en el paquete homónimo) que se encarga de ofrecer una capa de abstracción sobre las operaciones que trabajan con el registro RMI.

Así, esta clase se encarga de llevar a cabo operaciones tales como: iniciar el registro RMI, exportar clases, realizar la búsqueda en el registro o buscar puertos libres en la aplicación.

Además, se encarga también de generar las URLs RMI que serán usadas por la aplicación. En mi caso he optado por utilizad el nombre de las interfaces de cada clase para generar la URL. Esta aproximación, unida al hecho de tener una abstracción del registro, me permite realizar tareas como

la de instanciar el controlador y buscar en él usando únicamente el nombre de una interfaz, haciendo que si el controlador encuentre en el registro una clase que cumpla la interfaz devuelva su una instancia ya exportada de la misma.

Semejante aproximación hace que las clases que tengan que interactuar con el controlador no tengan que preocuparse de cuál es la clase final que implementa la interfaz que necesitan o de cómo formar URLs RMI; con tan sólo saber qué interfaz necesitan el controlador se encargará de proporcionársela.

Otro conjunto de clases de suma importancia en la práctica son los menús. Los menús son servicios que se asocian a un controlador y permiten a un usuario interactuar con los mismos.

Nótese que no solo el ejecutable de “usuario” tiene menús, los otros dos controladores ofrecen también menús destinados a ser usados por el personal administrativo del entorno.

Dichos menús son personalizables y poseen un sistema de opciones, callbacks de opciones, parámetros y validadores de parámetros que permiten a los distintos controladores usar el sistema de forma uniforme, pero no perdiendo así ni ápice de funcionalidad o flexibilidad.

Si bien son mejorables (como toda la aplicación) considero que cumplen muy bien con su labor y semejante abstracción me permitió despreocuparme de tareas banales, comparativamente hablando, para centrar mi tiempo en la lógica de la aplicación y el desarrollo de esta práctica.

Por último, la información de la aplicación se engloba en las clases *Trino* (de la cual no hablaré pues es la misma que proporciona el equipo docente) y la clase *DatosUsuario*, que es una clase que engloba los datos de cada usuario del sistema (*nick*, nombre, contraseña y *callback* de usuario).

Diagramas de clases

Como continuación de la explicación de arquitectura, facilito a continuación varios diagramas de clases que describen partes concretas de la aplicación.

Nótese que debido a la complejidad de la misma no puedo hacer un único diagrama que sea legible, por lo que he optado por esta opción, aprovechando así para explicar partes del sistema que no hayan podido quedar claras en párrafos anteriores.

A continuación mostraré los diagramas agrupados en varios sistemas y abstracciones.

Sistema de controladores

La piedra angular de mi práctica es, sin duda, el sistema de controladores y servicios del cual ya he hablado con anterioridad.

La imagen 1 ilustra la figura del controlador, representada por la interfaz *ControladorInterface*, la cual mantiene una asociación con la interfaz S*ervicioInterface*.

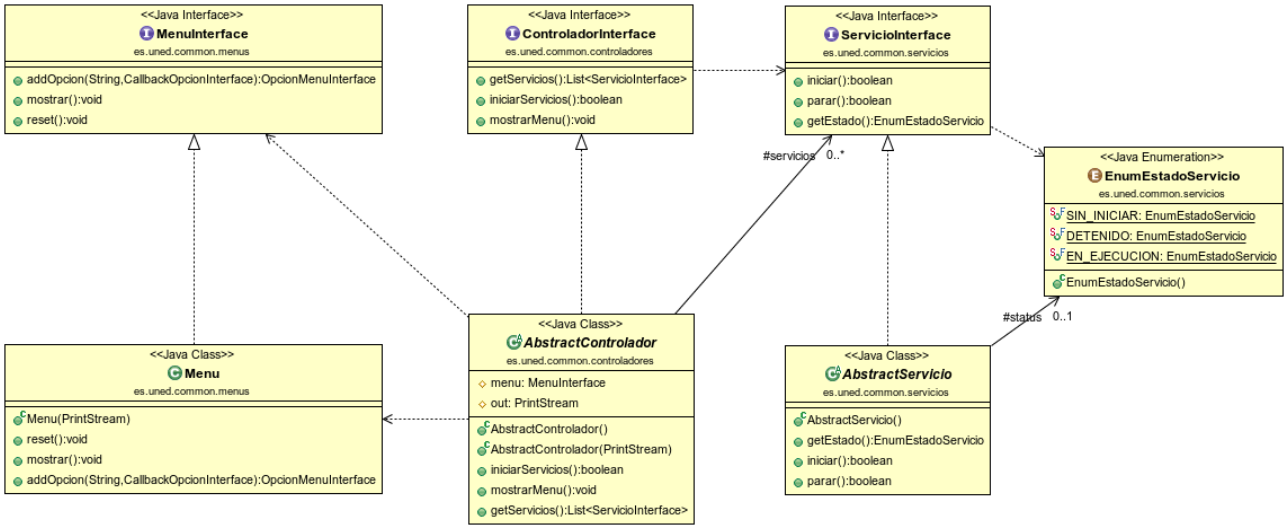


Imagen 1 - diagrama de clases del sistema de controladores

Esto es así porque un controlador agrega a 0 o más servicios y los utilice para ofrecer la funcionalidad que declara su interfaz.

Nótese que el diagrama muestra dos implementaciones a clases abstractas tanto de servicio como de controlador. Esto es así porque ambas clases encapsulan la funcionalidad básica y común que serán usadas por cualquier descendiente. Tareas como la gestión de servicioso la definición de menús asociados están definidas en estas clases.

Adicionalmente puede verse en la imagen la enumeración *EnumEstadoServicio* (que aglutina los distintos estados en los que se puede encontrar un servicio) y la relación de *AbstractControlador* con *Menu*. Nótese que el sistema de menús es más complejo de lo que aquí parece y se hablará de él más adelante.

Datos de usuario

La imagen 2 muestra el diagrama de clases que ilustra cómo trata al usuario la aplicación a nivel de representación de datos.

En él podemos ver una clase principal llamada

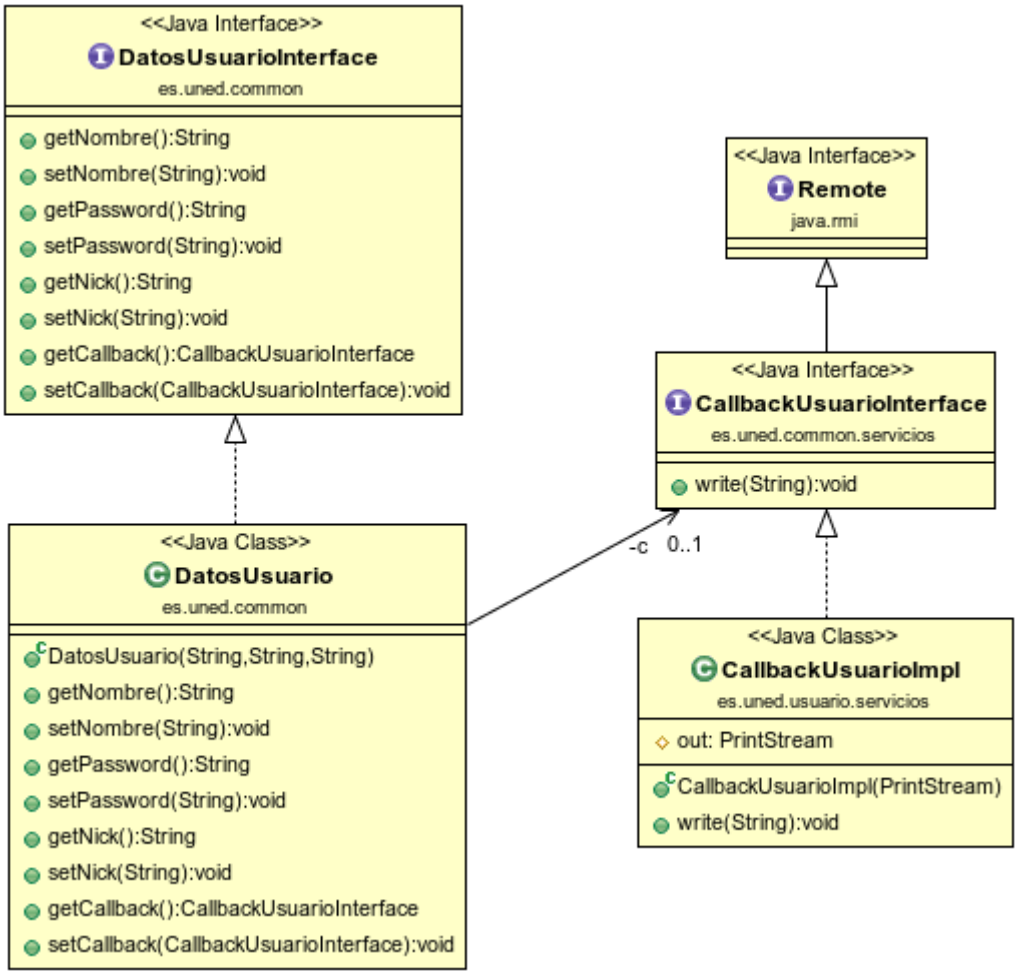


Imagen 2 - sistema de usuarios

DatosUsuario que implementa una interfaz (con el fin de desacoplar el código de las implementaciones) que ofrece mecanismos para obtener y cambiar los datos de un usuario.

Además, se implementa la interfaz *CallbackUsuarioInterface*, esto es porque la aplicación asume que un usuario identificado va a tener siempre un *callback* permitiendo así al servicio gestor localizar a dónde ha de enviar las notificaciones de nuevos trinos, entre otras. Cabe destacar que la interfaz *CallbackUsuarioInterface* es remota pues va a ser usada y exportada a RMI.

Nótese que en este ejemplo se muestra la implementación de *CallbackUsuarioInterface* y que ésta está en un paquete distinto al resto de clases e interfaces.

Sistema de menús

El sistema de menús mostrado en la imagen 3 es el que se usado por los distintos controladores a lo largo de toda la aplicación

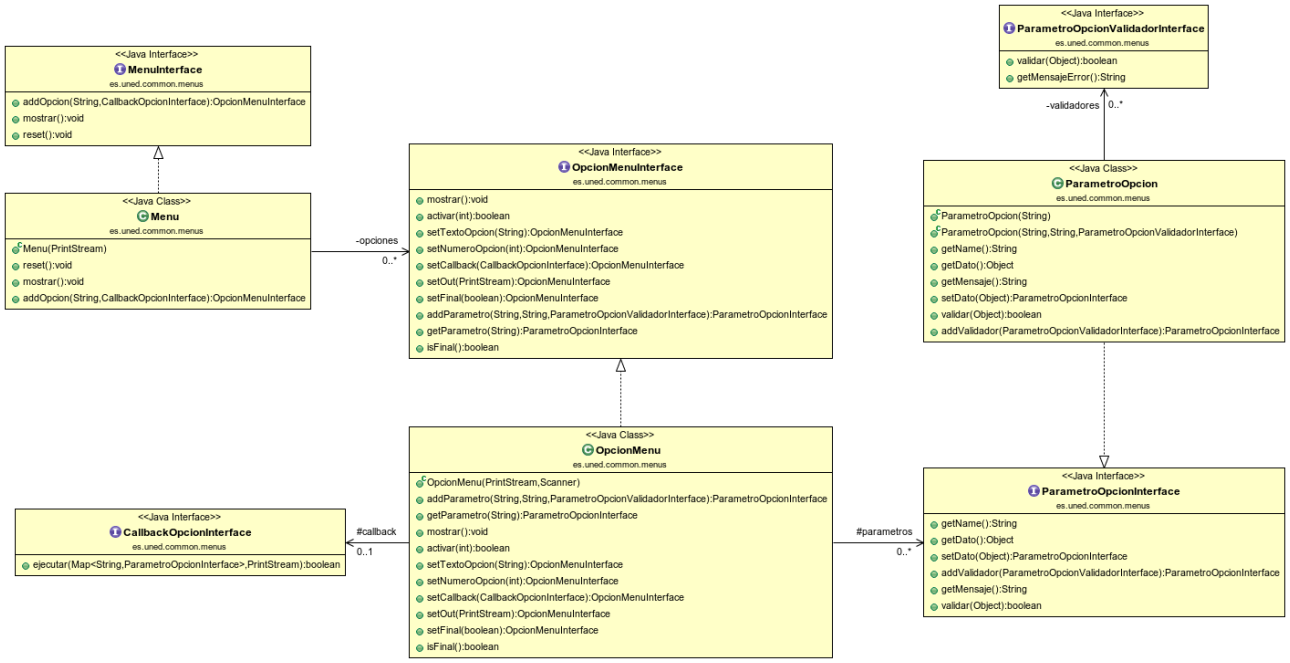


Imagen 3 - sistema de menús

Todas las declaraciones e implementaciones del sistema están en el paquete *es.uned.common* pues tanto implementaciones como interfaces son usadas a lo largo de todo el aplicativo.

La idea general es que los controladores tienen un menú asociado que muestran al usuario para que éste pueda interactuar con ellos.

Los menús tienen varias opciones, que también tienen una representación de clase, las cuales tienen un texto que muestran por pantalla, un *callback* –o acción a ejecutar al activarse– y un conjunto de parámetros que pueden solicitar al usuario.

Dichos parámetros poseen un mensaje que se muestra al usuario para solicitar que introduzca datos y un validador que hace lo que se espera de él para garantizar que los datos son válidos.

Así, un controlador que necesite crear un menú tendrá tan solo que instanciar uno, añadirle opciones y asociar callbacks a éstas. Adicionalmente, puede modificar las opciones para que éstas soliciten parámetros al usuario (con la posibilidad de usar validadores para asegurarse de que los datos son válidos).

Nótese que el diagrama no muestra una implementación de *CallbackOpcionInterface* y no se trata de una errata. El sistema está pensado para que se use dicha interfaz de forma anónima sin necesidad de realizar implementaciones específicas. El razonamiento detrás de esta decisión es que los *callbacks* en el sistema son completamente heterogéneos en controladores y no tiene sentido mantener clases específicas.

Por último, añadir que todas las implementaciones de operaciones en menús implementan mecanismos para hacer que sean *fluent*, es decir, que se puedan encadenar las distintas operaciones

(añadir opción, añadir *callback*, añadir parámetro, etc.) para garantizar que trabajar con su API interna sea fácil e intuitivo para el programador.

Controlador de base de datos

La imagen 4 muestra la implementación de nuestro primer controlador: el de base de datos.

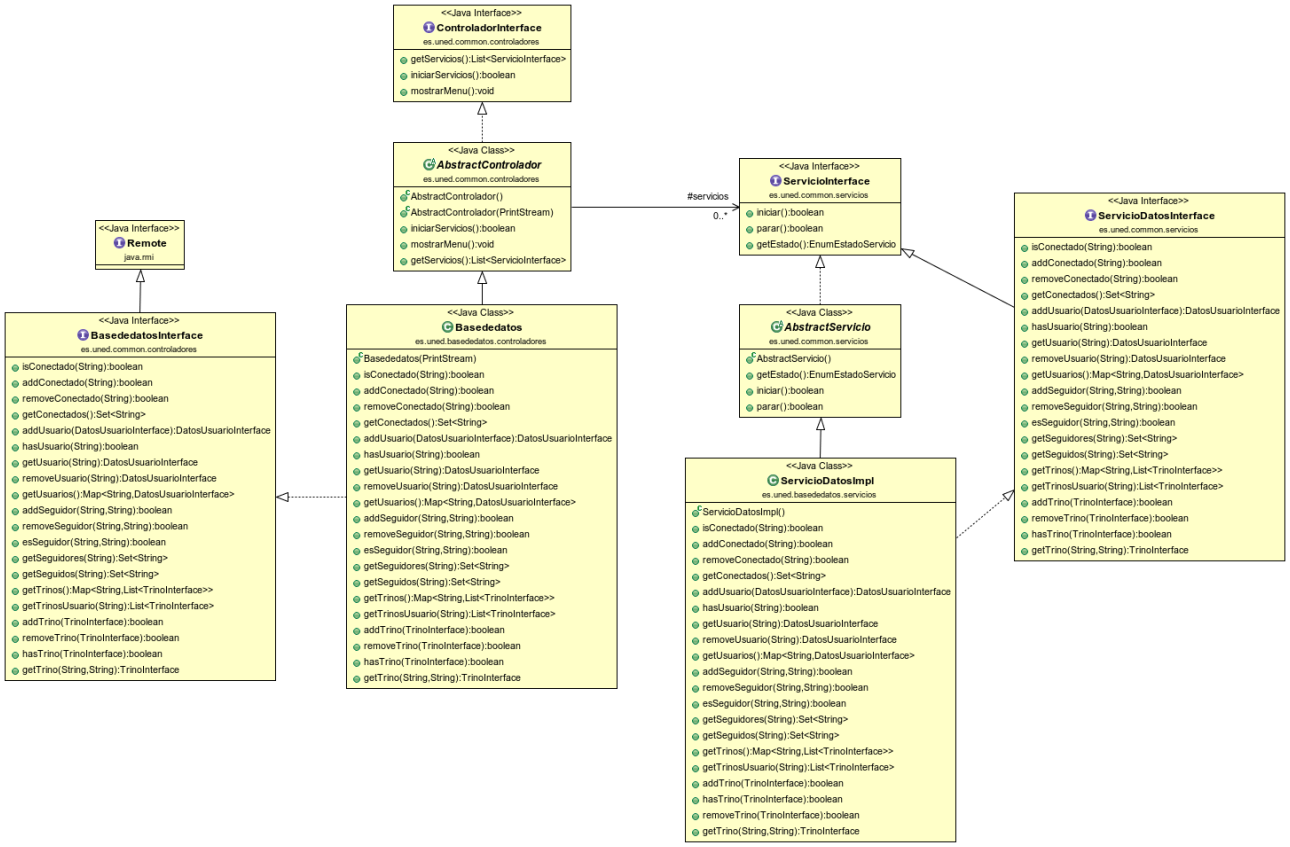


Imagen 4 - controlador y servicios de base de datos

Nótese como la imagen ilustra toda la arquitectura que hemos descrito:

- Los controladores implementan la interfaz y clase abstracta homónima.
- Cada controlador puede tener servicios, siendo en este caso usado el servicio *ServicioDatos*.
- Los controladores tienen asociado un menú que se usa para interactuar con el usuario.

Cabe destacar que el controlador implementa la interfaz *Remote* para hacer que éste pueda ser exportado por RMI y que, posteriormente, otras clases puedan usar el registro para comunicarse con él mediante la API que expone.

La base de datos es un servicio “tonto” en el sentido de que no hace comprobaciones de lógica de negocio y se limita a almacenar los datos que le facilitan, dejando para otras partes de la arquitectura del sistema el realizar las comprobaciones que puedan ser pertinentes con la integridad de los datos o las acciones a realizar cuando la información se lee o se guarda.

Además de lo anterior, el controlador apenas es un *wrapper* de su servicio asociado. Esto es así porque todas las operaciones de datos son mantenidas por el servicio del que depende. Podría hacerse un argumento para defender que el código de ambos podría unificarse en una misma clase, pero al desacoplarlo así garantizamos la homogeneidad con el resto del sistema, separamos responsabilidades (el servicio almacena los datos y el controlador se encarga de publicar su API) y hacemos que el sistema sea fácilmente extensible en el futuro.

A nivel de implementación, el servicio *ServicioDatosImpl* utiliza varias estructuras de datos para guardar la información en memoria:

- Una estructura tipo *Map* para almacenar los usuarios con sus estructuras *DatosUsuarioInterface*.
- Un *Set* para guardar la lista de usuarios conectados.
- Un *Map* que relaciona usuarios con el *Set* de sus seguidores.
- Otro *Map* que relaciona usuarios con una lista (“*List*”) de Trinos.

Nótese que el uso de las distintas colecciones no es arbitrario. Se usan *Sets* en aquellas partes en las que queremos forzar que no existan duplicados y *List* en aquellos donde éstos pueden coexistir de forma normal. Las estructuras tipo *Map*, al ser implementadas normalmente mediante la clase *HashMap*, también se benefician de las características del *Set*, pues su implementación a bajo nivel es idéntica (de hecho *Set* usa internamente un *HashMap).*

Controlador de servidor

El controlador de servidor es el segundo controlador explicado y el más grande de la práctica. La imagen 5 ilustra todo el diagrama de clases de las entidades relacionadas con el mismo.

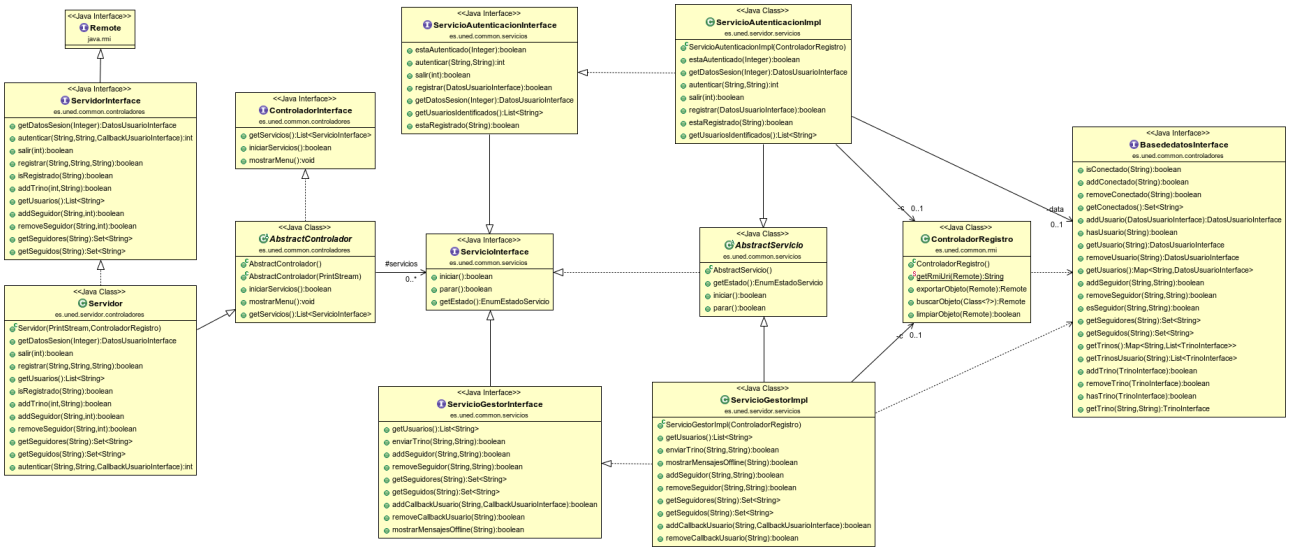


Imagen 5 - controlador de servidor

Fijándonos en el diagrama vemos la misma arquitectura ya descrita con anterioridad: controladores que implementan su interfaz y que se asocian con servicios, que también tienen su interfaz. Implementaciones del controlador que heredan de la clase *AbstractController* y que implementan la interfaz *Remote*.

No obstante, aquí es importante fijarse en que el controlador hace uso de dos servicios:

- *ServicioGestorInterface*.
- *ServicioAutenticacionInterface*.

Eso hace que la programación del servidor no se limite a ser un mero *wrapper* como el que se usaba en el controlador de base de datos. Aquí el controlador tendrá que coordinar las acciones de sendos servicios para cumplir con la API que publica su interfaz.

Por ejemplo: los procesos de *login* o *logout* necesitan coordinar ambos servicios para gestionar la sesión asociada al usuario, informar al motor de base de datos de los cambios y asegurarse de notificar a los *callbacks* de usuario correspondientes de cualquier cambio.

También es importante fijarse en que el diagrama muestra a la clase *ControladorRegistro*, clase que no veíamos en el controlador de base de datos.

¿A qué se debe esto?, pues sencillamente a que los servicios usados por este controlador **necesitan utilizar al controlador de base de datos para realizar su trabajo**. Por ello, a la hora de instanciar al controlador “*Servidor*” será necesario proporcionar una instancia del *ControladorRegistro* el cual, a su vez, será facilitado –mediante inyección de dependencias– a los dos servicios.

Aquí se ve el poder de RMI, pues este *middleware* permite a clases desconocidas interactuar entre ellas de forma remota usando únicamente los contratos declarados en las interfaces.

Considero que RMI es una gran herramienta una vez se ha llegado al punto en el que la aplicación ya ofrece las abstracciones necesarias para interactuar con el registro.

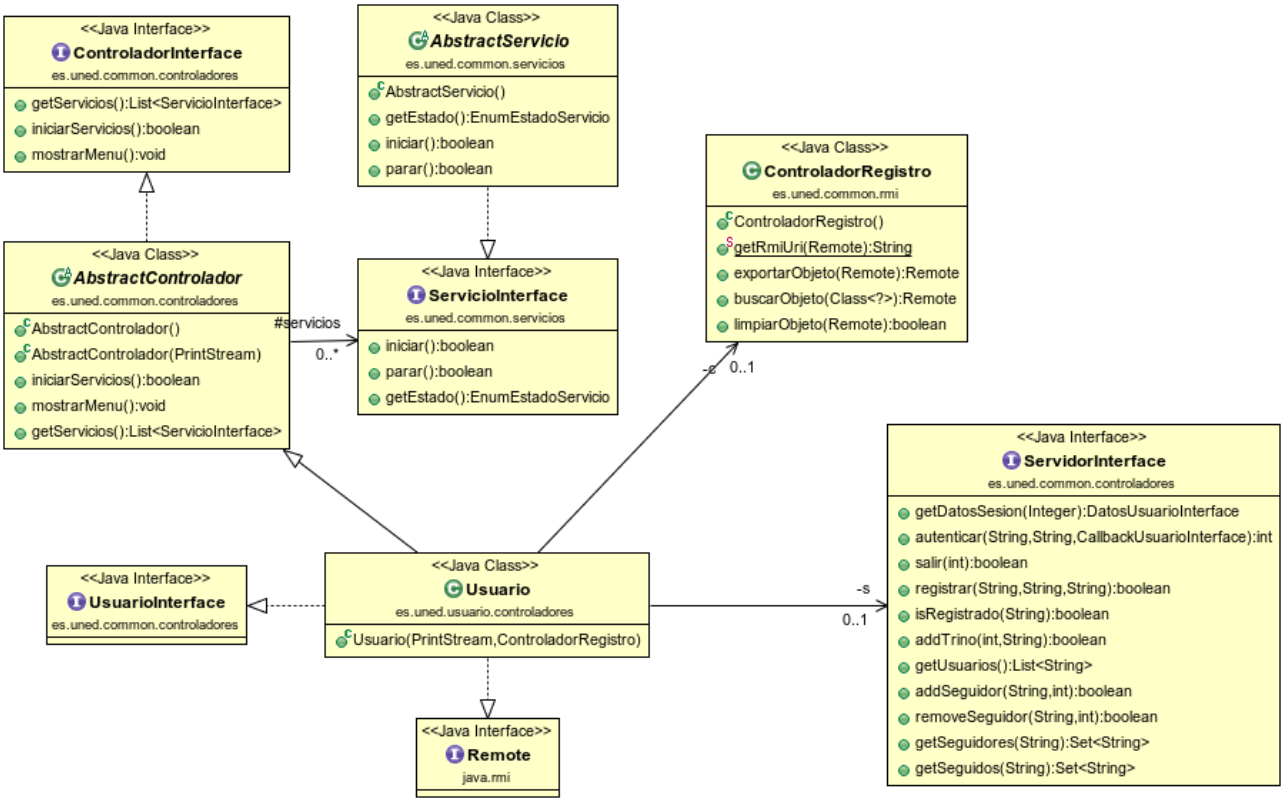
A nivel de implementación mencionar que el *ServicioGestor* gestiona los logins generando *tokens* de sesión con cada acceso válido. Dichos *tokens* podrán ser usados por los clientes para identificarse y realizar operaciones privilegiadas (enviar trinos, seguir, no seguir, etc.).

Para ello, dicho servicio mantiene una tabla volátil de sesiones separada del controlador de base de datos. Dicha decisión viene motivada a que, por seguridad, considero que las sesiones no han de guardarse en algún mecanismo persistente para evitar hacer más fácil ataques de *session hijacking*, si bien en el ámbito de ésta práctica dicha aproximación es completamente irrelevante.

El *ServicioGestor* implementa el núcleo de la lógica de negocio de la aplicación. Añade trinos, permite a los usuarios seguir a otros, informa a los *callbacks* de usuarios cuando hay algún nuevo trino y cuando no han leído trinos desconectados, etc.

Controlador de usuario

El último controlador se ilustra en la imagen 6.



Llegados a este punto el diagrama de clases debería resultar familiar y sin sorpresas para el lector. El diagrama implementa la arquitectura como hasta ahora, incluyendo la inyección de dependencias del *ControladorRegistro* necesaria para poder comunicarse con el *ServidorInterface* remoto.

Nótese que este controlador **no tiene servicios asociados**. Esto es así ya que el controlador servidor se encarga de mantener la funcionalidad de los *callacks* y la clase no es más que un menú glorificado que permite al usuario interactuar de forma remota con el resto de controladores.

Ejemplos de funcionamiento

Inicialmente esta sección se había hecho con capturas de pantalla para ilustrar el funcionamiento de la aplicación.

El resultado fue un montón de texto e imágenes que, en mi opinión, se hacían difíciles de leer y no quedaban muy bien maquetadas.

En su lugar decidí grabar un pequeño vídeo (02:40) en formato WebM que adjunto a la memoria de esta práctica y pueden encontrar en esta página web que he habilitado expresamente para mostrarlo:

https://uned.luaces-novo.es/sistemas_distribuidos/funcionamiento_practica.webm

Cualquier navegador moderno es capaz de reproducir el vídeo, no siendo necesario el uso de programas, *codecs* o herramientas específicas.

El vídeo también se encuentra en la ruta **./PRACTICA/doc/assets/funcionamiento** del directorio de práctica entregado.

Dicho vídeo ilustra el funcionamiento de la aplicación, pues se ejecuta en un mismo terminal con 4 pantallas y realiza tareas tales como:

- Iniciar los controladores principales.
- Demostrar el procedimiento de registro y *login*.
- Mostrar como las opciones de los servicios principales muestran la información de la base de datos.
- Envío de trinos a seguidores, tanto en tiempo real como desconectados.
- Proceso de seguir y no seguir a otros usuarios.

Conclusiones

Aprovecho este capítulo para cerrar esta memoria y añadir comentarios sobre todo el proceso de la práctica.

Comenzando con las conclusiones, indicar que ésta me ha hecho ponerme en contacto con RMI, tecnología que no conocía demasiado y que me ha parecido bastante interesante. Si bien Java no me gusta mucho (ya que requiere dedicarse enteramente a él para hacer las cosas bien), este *middleware* demuestra, una vez más, el alcance de la tecnología.

Con esto la práctica me ha parecido interesante y amena, siendo el material de apoyo entregado más que suficiente para realizarla (a falta de alguna pequeña aclaración aquí y allá).

Por desgracia, no pude dedicarle el tiempo que me gustaría a la misma, lo que ha hecho que no pudiese sumergirme en algunos aspectos de RMI tanto como me gustaría.

Esto mismo ha condicionado también un montón de cosas que me gustaría revisar y de las cuales no estoy muy orgulloso. Entre ellas, las cosas que creo que habría que mejorar en mi práctica son:

- Para empezar, habría añadir la funcionalidad opcional no implementada.
- Revisar la jerarquía de clases, pues la complejidad del código compartido y las interfaces que se heredan entre sí confunde al IDE y hace que marque muchos errores.
- Añadir algún mecanismo para guardar registros de sistema de las operaciones que se realizan.
- Mover todo el tratamiento de la información al controlador de base de datos, pues el controlador de servidor mantiene alguna estructura en memoria por motivos de seguridad.

- Revisar el tratamiento general de errores, la usabilidad y los mensajes que se muestran al usuario.
- Mejorar el tratamiento de RMI y sus URLs.

Con esto y, sin más, cierro la presente memoria que confío vaya en las líneas de lo que espera el equipo docente de la misma.