Extensión de la plataforma de sistemas multiagente JADE para aplicaciones empresariales web

Alejandro Reyes Bautista

Extensión de la plataforma de sistemas multiagente JADE para aplicaciones empresariales web

Alejandro Reyes Bautista

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERÍA DEL SOFTWARE

EXTENSIÓN DE LA PLATAFORMA DE SISTEMAS MULTIAGENTE JADE PARA APLICACIONES EMPRESARIALES WEB

EXTENSION OF JADE MULTIAGENT SYSTEMS PLATFORM FOR WEB ENTERPRISE APPLICATIONS

Realizado por
ALEJANDRO REYES BAUTISTA
Tutorizado por
EDUARDO GUZMÁN DE LOS RISCOS
Departamento
LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JULIO 2018

Fecha defensa:
El Secretario del Tribunal

Extensión de la plataforma de sistemas multiagente JADE para aplicaciones empresariales web

Alejandro Reyes Bautista

Extensión de la plataforma de sistemas multiagente JADE para aplicaciones empresariales web

Alejandro Reyes Bautista

Resumen: Extensión de la plataforma JADE para facilitar su integración en aplicaciones empresariales web, tanto Java EE como Spring. Se ha creado una capa de abstracción sobre este middleware que permite delegar todas las tareas "*extras*" requeridas en un entorno web para desarrollar estos agentes más allá de su funcionalidad, como la inicialización del contenedor JADE en el servidor de aplicaciones o servidor web (contenedor servlet), la gestión de los agentes, comportamientos y comunicaciones. A través de la herencia e implementación de las clases e interfaces que se han definido se consigue cubrir esta carencia a la vez que añadir funcionalidad a partir de la compleja gestión que JADE ya realiza, como la separación entre las entidades Agente y Comportamiento, con el fin de reutilizar distintos comportamientos en distintos agentes de forma sencilla, intuitiva y extensible. Esta capa de abstracción varía levemente en algunas anotaciones si se tratan los agentes en la capa EJB o se utiliza Spring Boot, que permite desplegar una aplicación completa en un servidor web empotrado. Por último, la consola JADE de escritorio ya no es de utilidad; motivo por el que se ha desarrollado una API REST que permite el control remoto de agentes y comportamientos a través del formato de datos JSON, dando flexibilidad para la creación de distintos clientes que sean capaces de consumir los servicios proporcionados por la API. Se ha desarrollado un cliente web utilizando las tecnologías JSF y Spring Boot, sustituyendo así la consola JADE para aplicaciones de escritorio.

Palabras clave: agentes, JADE, Java, librería, JavaServer Faces, Spring MVC, Spring Boot, API, REST, aplicación web

Extensión de la plataforma de sistemas multiagente JADE para aplicaciones empresariales web

Alejandro Reyes Bautista

Extensión de la plataforma de sistemas multiagente JADE para aplicaciones empresariales web

Alejandro Reyes Bautista

Abstract: Extension of JADE Platform to ease the integration with web enterprise applications Java EE and Spring. An abstract layer has been introduced wrapping this middleware in order to delegate all the extra-task other than implementation features required by a web environment to develop these agents, as JADE container initialization on the web server (servlet container), agents' management, behaviours' management and communications. Through inheritance and both classes and interfaces implementation which have been defined, this gap is supplied at same time as new features are added using the complex management JADE supports, as detachment between agents and behaviours, in order to reuse distinct behaviours on different agents in an easy, intuitive and extensible manner. This abstraction layer varies slightly in some annotations depending on the use of EJB or Spring Boot, which allows the application to be deployed in an embedded web server. Finally, the JADE desktop console is not useful anymore; reason why an API REST, which allow remote control over agents and behaviours using JSON data format, have been built, providing flexibility to create distinct kind of clients able to consume the API services. A web client has been developed using JSF and Spring Boot technologies, replacing the JADE desktop console.

Keywords: agents, JADE, Java, library, JavaServer Faces, Spring MVC, Spring Boot, API, REST, web application

ÍNDICE

ÍNDICE 1

1. INTRODUCCIÓN 5

1.1 Antecedentes y motivación 5

1.2 Objetivos 6

1.3 Tecnologías y herramientas utilizadas 7

1.4 Metodología 12

1.5 Estructura de la memoria 15

2. REQUISITOS Y CASOS DE USO 17

2.1 Contexto y consideraciones previas 17

2.2 Requisitos funcionales 18

2.3 Casos de uso 19

3. DISEÑO 29

3.1 Arquitectura y componentes del framework 29

3.2 Arquitectura con Aplicaciones Java EE 30

3.3 Arquitectura con Aplicaciones Spring 31

3.4 Arquitectura de la consola web 32

3.5 Arquitectura de la API REST 33

3.6 Arquitectura del cliente web 34

4 IMPLEMENTACIÓN DEL FRAMEWORK 35

4.1 Componentes del framework 35

4.2 Diferencias del framework entre EJB y Spring 38

4.3 Componente manager 39

4.4 Componente agent 44

4.5 Componente behaviour 56

4.6 Ejemplo de comportamiento y agente 68

5. IMPLEMENTACION DE LA API REST 71

5.1 Capas de la API REST 72

5.2 MicroserviceApplication 72

5.3 Dependencias 73

5.4 API REST 74

5.5 Formato JSON 76

5.6 Capa Model: agentModel y responseMessageModel 76

5.7 Capa Service: APIAgentService 80

5.8 Capa Service: APIDescriptionService 81

5.9 Capa Controller: JadeRestController 83

6. IMPLEMENTACION DEL CLIENTE WEB 93

6.1 Capas del cliente web 94

6.2 ConsoleApplication 95

6.3 Dependencias 97

6.4 Capa Service: AgentServiceInterface y AgentService 100

6.5 Capa Model: AgentModel, Entity y Exceptions 101

6.6 Capa Model: AgentDaoInterface y APIAgentDao 103

6.7 Capa Controller: estructura de controladores, vistas y funcionalidades 110

6.8 Capa Controller: AgentListController y agent-list.xhtml 110

6.9 Capa Controller: AgentController y agent-form.xhtml 117

6.10 Capa Controller: AgentBehaviourListController, agent-behaviour-list.xhtml y behaviour-list.xhtml 122

7. CONCLUSIONES Y LÍNEAS FUTURAS 135

REFERENCIAS BIBLIOGRÁFICAS 139

1. INTRODUCCIÓN

A continuación expondremos los antecedentes, motivación y objetivos de este trabajo, las tecnologías utilizadas con este fin y la estructura de la memoria.

1.1 Antecedentes y motivación

El proyecto se enmarca en el área de investigación de los Sistemas Multiagente (SMA), una de las ramas de la Inteligencia Artificial. Más concretamente, se centra en la plataforma de SMA, JADE, desarrollada por Telecom Italia, que es quizás la más extendida a día de hoy. Las aplicaciones que incluyen SMA están formadas por diversos componentes inteligentes que interactúan entre sí con el objetivo de solucionar problemas complejos que un sistema individual o bien no podría resolver, o bien resolvería con dificultad. JADE es un middleware desarrollado completamente en el lenguaje de programación Java y compatible tanto para aplicaciones de escritorio (Java SE) como aplicaciones web (Java EE) y entorno móvil (Java ME CLCD/MIDP1.0); adhiriéndose en todo momento a las especificaciones FIPA (*The Foundation for Intelligent Physical Agents*), que es el estándar de SMA establecido por la IEEE *Computer Society*. JADE es una plataforma desarrollada con licencia abierta, que permite su extensión. No obstante, su compatibilidad con Java EE no implica que exista un entorno de integración eficiente e intuitivo para el desarrollador. Asimismo, desde hace años Spring es utilizado como la principal alternativa fuera del estándar, y como la primera opción a la hora del desarrollo de microservicios con JAVA. JADE es también adaptable a este framework, pero las dificultades siguen siendo muy similares. Esta complejidad reside en el despliegue del contenedor JADE en los contenedores EJB (Enterprise Java Beans), que se ejecutan en el servidor Java EE o en los contenedores servlet en el caso de Spring. Es en el desarrollador en quien recae gestionar todo su ciclo de vida, así como la inicialización y finalización de los distintos agentes, comportamientos y comunicaciones entre agentes y contenedores. El resultado es un gran hándicap en el desarrollo de aplicaciones empresariales web: se alarga el ciclo de vida de desarrollo a la vez que se añade complejidad al mantenimiento de estructuras software que podrían abstraerse en base a estructuras agrupadas en un framework.

Este framework serviría para delegar todas las tareas que se adhieran a ciertos patrones comunes, aliviando la curva de aprendizaje y permitiendo el uso de agentes ya desarrollados en la capa empresarial (Business Layer) desde la capa web con la única preocupación del tratamiento de datos y la presentación al usuario final de la aplicación. Todo esto sin perder flexibilidad a la hora de creación de nuevos agentes y comportamientos desde la capa empresarial para su uso en la capa web. En el presente ya se ha realizado un estudio previo y se ha desarrollado código como parte de una beca de colaboración con el departamento de Lenguaje y Ciencias de la Computación (LCC) de la Universidad de Málaga. Concretamente se ha estudiado la plataforma de Sistemas Multiagente JADE en entorno de escritorio, la tecnología y framework JSF, el mecanismo de metaanotaciones en Java y anotaciones de la librería CDI, además de desarrollarse algunas anotaciones Java para la plataforma JADE.

1.2 Objetivos

El objetivo principal de este proyecto es extender JADE para facilitar su integración en aplicaciones web Java EE de una forma relativamente sencilla. Aunque en la actualidad existen algunas extensiones, tienen el denominador común de ser complejas y haberse quedado obsoletas. Como primer paso ya se han introducido, en un trabajo previo, anotaciones Java en los componentes JADE para facilitar su integración en aplicaciones Java EE, a través de la inyección de dependencias o "filosofía Hollywood" (un patrón para la Inversión de Control) que utilizan frameworks como Spring o la API CDI de Java. A partir de esto, el siguiente paso es desarrollar los mecanismos necesarios para facilitar la integración de agentes JADE en la lógica de negocios de aplicaciones web empresariales. Posteriormente adaptaremos esta capa de abstracción al framework SpringBoot, teniendo en consideración el despliegue en un contenedor servlet (mucho más ligero). Con esta base desarrollaremos un microservicio consistente en una API HTTP para SpringBoot en el que a través de información suministrada mediante formato JSON podamos administrar y monitorizar los agentes y comportamientos de nuestra aplicación. Finalmente desarrollaremos una consola que suplirá a la aplicación de escritorio que actualmente ofrece JADE. Esta consola será un cliente de la API desarrollada, lo

que pone de manifiesto la flexibilidad de construir distintos clientes, tanto web como móvil, que consuman datos de la misma API.

1.3 Tecnologías y herramientas utilizadas

En este apartado justificaremos las tecnologías y herramientas que hemos utilizado en los distintos aspectos del trabajo.

JADE (Java Agent DEvelopment Framework)

Middleware desarrollado completamente en el lenguaje de programación Java y compatible tanto para aplicaciones de escritorio (Java SE) como aplicaciones web (Java EE) y entorno móvil (Java ME CLCD/MIDP1.0); adhiriéndose en todo momento a las especificaciones FIPA (*The Foundation for Intelligent Physical Agents*), que es el estándar de SMA establecido por la IEEE *Computer Society*. JADE es una plataforma desarrollada con licencia abierta, que permite su extensión. Es la base de nuestro proyecto, sobre el que construiremos un adaptador para aplicaciones empresariales web, tal y como discutimos en el apartado anterior.

Figura 1.1 Logo JADE

Java Enterprise Edition (Java EE)

Estándar de plataforma que extiende Java SE (Java Standard Edition Platform) suministrando librerías para despliegues tolerante a fallos, distribuidos, software Java multi-capa, y que se basa principalmente en componentes modulares que de despliegan y ejecutan en un servidor de aplicaciones. Uno de los objetivos principales es mejorar la compatibilidad de JADE con las implementaciones de este estándar. La implementación elegida es **Glassfish**, aunque esto no es una limitación a otros servidores de aplicaciones como Tomcat EE o Websphere.

Figura 1.2 Logo Java EE *Figura 1.3 Logo Glassfish*

Spring Framework 5.0 (Servlet-based Spring MVC framework y Spring Boot)

Framework de desarrollo de aplicaciones web que no recoge todas las especificaciones del estándar JavaEE, sino que integra de forma cuidadosa diversas especificaciones del conjunto, como *Servlet API* (JSR 340), *WebSocket API* (JSR 356), *Bean Validation* (JSR 303) o *JPA* (JSR 338). Spring Framework se divide en módulos, incluyendo aspectos concretos para seguridad como *Spring Security* o para computación en la nube como *Spring Cloud*, los cuales evolucionan de forma continua para facilitar las tareas al desarrollador. Con la ayuda de Spring Boot las aplicaciones son creadas, según el equipo técnico, "*in a devops – and cloud-friendly way"*, lo que significa que el contenedor Servlet se encuentra empotrado con la aplicación y se puede modificar de forma trivial, además de añadir configuraciones y mecanismos de inyección a través de un uso extenso de anotaciones. Usaremos **Apache Tomcat** como contenedor.

Figura 2.4 Logo Spring *Figura 1.5 Logos Apache Tomcat y Spring Boot*

Java Server Faces 2.3 y Primefaces

Especificación Java para construir interfaces de usuarios para aplicaciones web basadas en componentes. En definitiva es un MVC web framework (framework web con la arquitectura Modelo-Vista-Controlador) que simplifica la construcción de interfaces de usuario (UI) mediante la reutilización de los componentes de una página. JSF 2.x utiliza *Facelets* como lenguaje declarativo por defecto en vez de JSP (Java Server Pages). Permite desacoplar la vista del controlador y nos obliga a desarrollar código *xhtml* en vez de *html* para que el controlador frontal ejecute todas las peticiones y renderice los resultados de la manera esperada. Hemos decidido utilizar la implementación **Mojarra 2.3.0**.

Spring ya tiene su propio sistema de vistas (*template engines*) con *Thymeleaf, Freemarker* o *Mustache*, pero es una tecnología que tendríamos que investigar y la solidez, seguridad y experiencia previa con JSF nos ha permitido descartarla.

Hemos desarrollado nuestro cliente API con esta tecnología junto a Spring Boot. Por defecto el manejo de los Beans, del controlador frontal y distintos aspectos se gestionan y funcionan de manera distinta al de JSF dentro de una aplicación Java EE; a través de diversos ajustes y ficheros de configuración hemos podemos delegar funcionalidades para que ambas tecnologías funcionen de forma conjunta, dando como resultado una consola web con el contenedor servlet Tomcat empotrado.

Finalmente, Primefaces es un framework de código abierto para Java Server Faces con más de 100 componentes de interfaces de usuario compatibles con HTML5, validación del lado de cliente, componentes móviles optimizados, motor de temas, *widgets* basados en jQuery y mucho más. Lo utilizamos para dar un aspecto más profesional a nuestra consola con componentes optimizados sin necesidad de especificar multitud de líneas de CSS3 o utilizar otras librerías como Bootstrap.

Figura 3.6 Logo Mojarra JavaServer Faces

Figura 1.8 Logo Primefaces

Maven (Gestión de dependencias)

Utilizamos Maven como sistema de gestión de dependencias. Nos permite descargar de un repositorio central las librerías que necesitamos en nuestro proyecto, con todas sus dependencias y las versiones correctas para evitar compatibilidades, a través de la creación de un fichero *.pom* en formato XML. La decisión de utilizar Maven ha sido arbitraría, cualquier otra alternativa como Gradle hubiera sido igual de válida.

Figura 1.9 Logo Maven

***Git* (*Control de versiones*)**

Control de versiones distribuido con el que registramos los cambios realizados sobre el conjunto de archivos con los que estamos trabajando a lo largo del tiempo, de modo que podemos recuperar las versiones específicas más adelantes, trabajar en distintas ramas según la funcionalidad a implementar y tener a su vez una copia de seguridad en un servidor remoto. Usamos **github** como repositorio remoto de control de versiones, aunque hay otras alternativas como bitbucket que también hemos considerado.

Figura 1.10 Logo Git

IDE (Entorno Integrado de Desarrollo)

Hemos utilizados dos IDE: para Java EE nos hemos decidido por NetBeans, ya que nos ofrece facilidad para la gestión de los EJB y los Servlet y el uso de la librería CDI para inyección de dependencias, además de incorporar las herramientas y recursos necesarios para establecer una conexión y desplegar una aplicación web en Glassfish.

Por otro lado, para Spring nos hemos decantado por IntelliJ Idea. Es cierto que existe *Spring Tool Suite*, un entorno basado en Eclipse para desarrollar aplicaciones Spring, pero en las guías oficiales te ofrecen IntelliJ Idea como alternativa, siendo éste un framework altamente utilizado, con mucha funcionalidad, soporte y disponible para casi cualquier lenguaje de programación. Además de muy cómodo de utilizar con Git y Maven.

Figura 1.11 Logo NetBeans

Figura 1.12 Logo IntelliJ Idea

Otra herramienta utilizada para el desarrollo de la aplicación ha sido Postman, un cliente HTTP para probar nuestra API web. Nos permite utilizar distintos verbos (GET, POST, DELETE, UPDATE…) así como editar el *header* y añadir campos JSON al *body* de la aplicación.

Figura 1.13 Logo Postman

1.4 Metodología

Hemos aplicado un proceso iterativo, ágil e incremental. Se han llevado a cabo pequeñas iteraciones en las que se han ido añadiendo funcionalidades a la versión anterior o se han corregido errores detectados. El beneficio de este sistema es la reducción del tiempo de retroalimentación y el poder detectar cambios o errores en la especificación de requisitos más rápidamente. Las tareas llevadas a cabo siempre han estado ordenadas según prioridad e interdependencia.

Debido al carácter de investigación de este TFG (en primera instancia estamos desarrollando un framework para adaptar una librería existente, no una aplicación para un cliente en sí) la estimación del tiempo de la implementación de las tareas es complicado. Hemos aumentado la funcionalidad sin modificar el código fuente de la librería original, a través de mecanismos de "*wrapper*" que garantizasen la compatibilidad con las futuras versiones de JADE (teniendo en cuenta que poseen retro compatibilidad de versiones). El inconveniente de este factor es el aumento de la incertidumbre en distintos aspectos de la implementación, tanto positiva como negativamente, lo que nos ha obligado a posponer algunos detalles para las siguientes iteraciones. No obstante, es importante dejar constancia que en etapas posteriores como el desarrollo de la API o de la consola web las estimaciones han tenido una entropía bastante menor.

A continuación, comentaremos desde una perspectiva general las iteraciones realizadas en el proceso de desarrollo, teniendo en consideración que las tres primeras fases han sido parte de un estudio previo desarrollado como parte de una beca de colaboración el departamento de Lenguaje y Ciencias de la Computación (LCC) de la Universidad de Málaga.

1. La primera iteración ha consistido en un estudio genérico de la plataforma de Sistemas Multiagente JADE en entorno de escritorio: funcionamiento de los agentes y ciclos de vida, principales clases de comportamientos, comunicaciones, ontologías y gestión de contenedores, además de la consola de escritorio.

2. Una vez alcanzado un conocimiento suficiente empezamos a investigar el mecanismo de meta-anotaciones existente en Java para crear nuestras propias

anotaciones con las que integrar JADE con aplicaciones Java EE. Posteriormente se analizan las anotaciones mediante la librería CDI que permite mediante un gran objeto factoría (*factory)* asegurar que grandes organizaciones puedan construir sistemas de software complejo y distribuido a la vez que compartir datos.

3. Con un conocimiento básico de ambas tecnologías comenzamos a desarrollar anotaciones como @*AgentQualifier* y @*AgentBehaviour* para la inyección de agentes y comportamientos desde la capa empresarial (EJB) a la capa web,

4. Conseguido el hito anterior ya pudimos comenzar el desarrollo de mecanismos de integración de JADE con la capa empresarial (EJB) para que la creación y la gestión de agentes, comportamientos y comunicaciones por parte del desarrollador fuera realizable de manera similar a la programación en el entorno de escritorio Java SE. Se ha añadido esa capa de abstracción que ha permitido obviar que los agentes corren en un contenedor JADE desplegado a su vez en un contenedor EJB, siendo dependiente del ciclo de vida de este último. Como resultado obtuvimos una primera versión del "*core"* de nuestra librería con clases e interfaces tanto para Agentes como Comportamientos y un *Manager* necesario para cohesionar todos los elementos.

5. Para finalizar el framework hemos desarrollado mecanismos de integración entre JADE, EJB y la capa web basada en servlets (JSP o Managed Beans, por ejemplo). Hemos generado la posibilidad de inyectar agentes y comportamientos desde la capa empresarial, asignar comportamientos a agentes (desacoplando así ambas entidades) y poder inicializarlos en el contenedor JADE de forma transparente y sin necesidad del desarrollo de ninguna lógica añadida en esta capa. No se limita la flexibilidad al posibilitar la creación de nuevos comportamientos a través de un comportamiento "factoría" que permite crear instancias de comportamientos sencillos (clases que extienden de *SimpleBehaviour*) especificando el código a ejecutar mediante la implementación de interfaces funcionales y el uso de variables compartidas que determinen el estado actual del comportamiento y la condición de finalización de ser necesario. Finalmente, como la capa web también utiliza la librería JADE, el desarrollador podrá crear comportamientos desde cero y vincularlos a cualquier agente inyectado.

6. Con esto hemos completado la compatibilidad de la librería JADE con la capa empresarial EJB junto a su uso y gestión desde una capa web basada en servlets, a través de la inyección de dependencias (usando CDI) gracias a las anotaciones que hemos desarrollado. La siguiente iteración ha consistido en tanto estudiar el funcionamiento del *core* de la plataforma Spring 5.0 y Spring Boot como adaptar las clases e interfaces desarrolladas que forman nuestro *core* a Spring y así conseguir que el desarrollo tenga las mismas facilidades en ambas plataformas. Esto se ha conseguido con pequeñas modificaciones dependientes de cada entorno de desarrollo (gestión de los beans de EJB respecto a gestión de los beans de Spring, por ejemplo), pero el código java desarrollado es totalmente análogo.

Llegados a este punto, el desarrollador JADE ya podría trabajar con los agentes y comportamientos, disfrutando de las ventajas de uso que hemos desarrollado de forma simultánea a la compatibilidad (desacoplamiento de las entidades *Agent* y *Behaviour*, gestión total de todos los comportamientos del agente en cualquier momento, mayor control en su ciclo de vida en tiempo de ejecución sin necesidad de la consola, etc.), en aplicaciones empresariales web del mismo modo que en aplicaciones de escritorio.

7. Posteriormente comenzamos con el desarrollo de una plataforma web para el control y monitorización de agentes JADE desplegados en un servidor web (contenedor servlet Tomcat empotrado con Spring Boot). El enfoque adoptado ha sido el de separar completamente el frontend del backend de la aplicación, es decir, que ambas aplicaciones se ejecuten en servidores Tomcat distintos y se comuniquen vía HTTP. El backend consiste en una API REST que coexiste en el mismo servidor donde se ejecutan los agentes JADE. A través de distintas rutas el controlador de la API recibe y devuelve información en formato JSON, la guía de funcionalidad de la API se puede consular desde la propia API a través de la dirección "*<direcciónDelServidor>:<puerto de escucha>/api".* La comprobación de las distintas funcionalidades de la API se llevó a cabo utilizando el cliente HTTP Postman, pudiendo aislar cualquier fallo concreto de implementación en el backend del desarrollo de cualquier cliente futuro.

8. Finalmente desarrollamos el frontend para el control y monitorización de los agentes, es decir, la plataforma web que suplirá la aplicación de escritorio que actualmente ofrece JADE. En esta iteración tuvimos que investigar la forma de conectar la tecnología JSF con Spring Boot para crear el cliente que consumiese datos de la API desarrollada. Este es solo uno de los posibles clientes que son posibles desarrollar al haber desacoplado el backend en un API REST dentro de un contenedor servlet Tomcat empotrado (un microservicio que se ejecuta de forma autónoma y con el que nos comunicamos a través de peticiones HTTP a su interfaz API).

9. Por último nos encontramos con la escritura de la memoria del proyecto con todas las especificaciones y documentación necesaria.

Mencionar que en todas las iteraciones han surgido finalmente necesidades que no habían sido recogidas anteriormente y que formaban parte de iteraciones anteriores, por lo que se han ido añadiendo en función de su necesidad para implementar los requisitos del sistema. También añadir que, aunque la memoria se incluya como la última fase, se ha llevado a cabo de forma transversal a lo largo de todo el proyecto, solo que su revisión, corrección de errores y finalización se ha llevado a cabo cuando el software ha alcanzado la etapa estimada para su presentación como trabajo de fin de grado.

1.5 Estructura de la memoria

Una vez conocido los objetivos, tecnologías y metodología del TFG vamos a concretar el contenido de los distintos apartados de este documento

En el siguiente capítulo, el segundo, haremos una descripción de los requisitos funcionales de la consola web (cliente web), además de los casos de uso necesarios para que sea útil y así alcanzar un mayor entendimiento del alcance la implementación realizada. La API (backend) suministra los datos necesarios al cliente web, por lo que los requisitos funcionales del cliente serán un subconjunto de ésta, la cual no especificaremos.

En el tercer capítulo empezaremos detallando la arquitectura de nuestra consola web, tanto backend (API) como frontend (cliente web); las razones por la que hemos elegido dicha arquitectura y sus beneficios. A continuación explicaremos tanto el diseño de la API, como de la consola web y la del framework para desarrolladores (EJB y Spring)

En el cuarto capítulo especificaremos los detalles de implementación del framework en EJB y en Spring: inicialización del contenedor y gestión de comportamientos y agentes.

En el quinto capítulo se expondrán todos los detalles de la API REST, capas internas, rutas especificadas y gestión de la plataforma JADE.

El sexto capítulo dará paso a la consola web; comentaremos lo distintos casos de uso que se dan, descripción de las funcionalidades de los modelos, vistas y controladores y decisiones en el diseño de la interfaz de usuario.

En el séptimo y último capítulo expondremos las conclusiones del trabajo realizado: beneficios obtenidos, limitaciones, ámbitos de uso de la aplicación y posibles líneas de evolución y mejora.

2. REQUISITOS Y CASOS DE USO

En este capítulo indicaremos qué requisitos fundamentales hemos considerado tras analizar el contexto de la aplicación.

2.1 Contexto y consideraciones previas

Podemos decir que resultado de este TFG obtenemos tres productos diferenciables. Por un lado, el framework de desarrollo que posibilita a los desarrolladores un mayor control sobre los agentes y comportamientos JADE en una aplicación web empresarial. Obtenemos dos frameworks análogos, uno para EJB y otro para Spring. La conclusión es que los requisitos funcionales ya fueron modelados para la gestión de sistemas multi-agentes por parte de la librería JADE y nuestro objetivo es solo añadirle esa capa de compatibilidad de la que carece, por lo que, aunque hemos añadido mejoras, no modelaremos esta parte.

En cuanto a la consola web, tal y como explicaremos en el siguiente capítulo, se encuentra dividida en dos partes: una API REST y un cliente web (la interfaz de la consola). Los requisitos funcionales del cliente web serán similares a los de la API, ya que en última instancia es ésta la que suministra el servicio (de hecho, la API es mucho más extensa para dar flexibilidad a los distintos tipos de clientes que quieran consumir de ella), no obstante, no tiene sentido especificar los “casos de uso” de la API, ya que simplemente enviamos una petición y nos devuelve la respuesta con un estado HTTP informando de si ésta ha podido resolverse o habido algún error.

Respecto a la consola web, mencionar que los requisitos funcionales han sido extraídos a partir de la exploración del funcionamiento de la consola JADE de escritorio, añadiendo el aspecto de gestión de comportamientos y su asignación a distintos agentes.

Finalmente, añadir que solo tenemos un único perfil de usuario, el del interesado en gestionar los agentes ejecutándose en el contenedor JADE. El perfil natural sería el de un desarrollador, pero con la modificación de la consola es posible que alguien con menos conocimientos técnicos pudiese llegar a gestionarlos. Nos referiremos a él como “*Usuario*”.

2.2 Requisitos funcionales

Expuesto el contexto y las consideraciones nos disponemos a establecer los requisitos necesarios para alcanzar el objetivo establecido en este TFG, tanto de la consola web (cliente web) como de la API.

Los requisitos de la consola web poseerán el identificador “CW_<identificador del requisito>”.

ID	NOMBRE	DESCRIPCIÓN
CW_1	Listar Agentes	El sistema debe permitir visualizar la lista de agentes disponibles.
CW_2	Visualizar Detalles Agente	El sistema debe permitir visualizar los detalles de un agente disponible.
CW_3	Crear Agente	El sistema debe permitir la creación de un nuevo agente proporcionando la clase y el nombre como parámetros obligatorios.
CW_4	Iniciar Agente	El sistema debe permitir la inicialización de un agente existente que no esté ejecutándose.
CW_5	Detener Agente	El sistema debe permitir la detención de un agente existente que esté ejecutándose.
CW_6	Reiniciar Agente	El sistema debe permitir el reinicio de un agente existente que esté ejecutándose.
CW_7	Eliminar Agente	El sistema debe permitir la eliminación de un agente concreto del sistema.
CW_8	Listar Comportamientos de Agente	El sistema debe permitir visualizar la lista de comportamientos vinculados a un agente.

CW_9	Visualizar Detalles Comportamiento de Agente	El sistema debe permitir visualizar los detalles de un comportamiento vinculado a un agente.
CW_10	Añadir Comportamiento a Agente	El sistema debe permitir añadir un nuevo comportamiento a un agente.
CW_11	Eliminar Comportamiento de Agente	El sistema debe permitir eliminar un comportamiento de un agente.
CW_12	Reiniciar Comportamiento de Agente	El sistema debe permitir reiniciar un comportamiento de un agente.

2.3 Casos de uso

En este apartado, vamos a detallar las interacciones del usuario con la aplicación y los posibles casos adversos a la hora de realizar algunas de las acciones.

Suponemos que las acciones realizadas dentro de la aplicación, el usuario siempre ha iniciado correctamente sesión (SR_2).

Campos	CASO DE USO 1
Caso de uso:	Listar Agentes
Descripción:	El usuario podrá visualizar los agentes disponibles en la plataforma
Precondición:	-

Escenario principal:	1- El usuario selecciona la opción de gestionar los agentes 2- El sistema muestra la lista de agentes disponibles en la plataforma
Escenario alternativo:	-
Requisitos asociados:	CW_1

Campos	CASO DE USO 2
Caso de uso:	Visualizar Detalles Agente
Descripción:	El usuario podrá visualizar los detalles de un agente disponible
Precondición:	El usuario está visualizando la lista de agentes disponibles (Caso de uso 1 – Listar Agentes)
Escenario principal:	1- El usuario selecciona un agente 2- El sistema muestra los detalles del agente
Escenario alternativo:	-
Requisitos asociados:	CW_1, CW_2

Campos	CASO DE USO 3
Caso de uso:	Crear Agente
Descripción:	El usuario podrá crear un nuevo agente en la plataforma
Precondición:	-
Escenario principal:	1- El usuario selecciona la opción de crear un nuevo agente 2- El sistema muestra los campos a rellenar 3- El usuario rellena los campos obligatorios "*name*" y "*class*" correctamente 4- El sistema redirige al usuario a la página principal dentro de la aplicación (página de visualización de los agentes disponibles)
Escenario alternativo:	1- El usuario introduce un valor incorrecto o no introduce alguno de los campos "*name*" y/o "*class*" 2- El sistema muestra la misma página con una advertencia informando por qué alguno o ambos valores no son válidos
Requisitos asociados:	CW_3

Campos	CASO DE USO 4
Caso de uso:	Iniciar Agente
Descripción:	El usuario podrá iniciar un agente que se encuentre disponible en la plataforma y no esté ejecutándose
Precondición:	El usuario está visualizando la lista de agentes disponibles

	(Caso de uso 1 – Listar Agentes)
Escenario principal:	1- El usuario selecciona iniciar uno de los agentes disponibles 2- El sistema muestra la lista de agentes disponibles con el estado del agente iniciado actualizado
Escenario alternativo:	2.b- El sistema muestra la misma página, con una advertencia informando que el agente no ha podido iniciarse
Requisitos asociados:	CW_1, CW_4

Campos	CASO DE USO 5
Caso de uso:	Detener Agente
Descripción:	El usuario podrá detener un agente que se encuentre disponible en la plataforma y esté ejecutándose
Precondición:	El usuario está visualizando la lista de agentes disponibles (Caso de uso 1 – Listar Agentes)
Escenario principal:	1- El usuario selecciona detener uno de los agentes disponibles que está ejecutándose 2- El sistema muestra la lista de agentes disponibles con el estado del agente detenido actualizado
Escenario alternativo:	2.b- El sistema muestra la misma página, con una advertencia informando que el agente no ha podido detenerse
Requisitos asociados:	CW_1, CW_5

Campos	CASO DE USO 6
Caso de uso:	Reiniciar Agente
Descripción:	El usuario podrá reiniciar un agente que se encuentre disponible en la plataforma y esté ejecutándose
Precondición:	El usuario está visualizando la lista de agentes disponibles (Caso de uso 1 – Listar Agentes)
Escenario principal:	1- El usuario selecciona reiniciar uno de los agentes disponibles que está ejecutándose 2- El sistema muestra la lista de agentes disponibles con el estado del agente reiniciado actualizado
Escenario alternativo:	2.b- El sistema muestra la misma página, con una advertencia informando que el agente no ha podido reiniciarse
Requisitos asociados:	CW_1, CW_6

Campos	CASO DE USO 7
Caso de uso:	Eliminar Agente
Descripción:	El usuario podrá eliminar un agente que se encuentre disponible en la plataforma
Precondición:	El usuario está visualizando la lista de agentes disponibles (Caso de uso 1 – Listar Agentes)
Escenario principal:	1- El usuario selecciona eliminar uno de los agentes disponibles 2- El sistema muestra la lista de agentes disponibles actualizada, sin el agente que se acaba de eliminar

Escenario alternativo:	2.b- El sistema muestra la misma página, con una advertencia informando que el agente no ha podido eliminarse
Requisitos asociados:	CW_1, CW_7

Campos	CASO DE USO 8
Caso de uso:	Listar Comportamientos de Agente
Descripción:	El usuario podrá visualizar la lista de comportamientos vinculados a un agente
Precondición:	-
Escenario principal:	1- El usuario selecciona la opción de visualizar los comportamientos de los agentes 2- El sistema muestra la lista de agentes con los comportamientos asociados
Escenario alternativo:	-
Requisitos asociados:	CW_8

Campos	CASO DE USO 9
Caso de uso:	Visualizar Detalles Comportamiento de Agente
Descripción:	El usuario podrá visualizar los detalles de un comportamiento vinculado a un agente.
Precondición:	El usuario está visualizando la lista de comportamientos vinculados a un agente (Caso de uso 8 – Listar

	Comportamientos de Agente)
Escenario principal:	1- El usuario selecciona un agente 2- El sistema muestra los detalles de los comportamientos asociados
Escenario alternativo:	-
Requisitos asociados:	CW_8, CW_9

Campos	CASO DE USO 10
Caso de uso:	Añadir Comportamiento a Agente
Descripción:	El usuario podrá añadir un nuevo comportamiento a un agente.
Precondición:	El usuario está visualizando la lista de comportamientos vinculados a un agente (Caso de uso 8 – Listar Comportamientos de Agente)
Escenario principal:	1- El usuario selecciona gestionar los comportamientos del agente. 2- El sistema muestra los comportamientos vinculados al agente y los comportamientos disponibles que pueden ser vinculados. 3- El usuario selecciona el comportamiento que quiere añadir al agente y confirma el cambio 4- El sistema redirige al usuario a la página dónde se visualizan los agentes con los comportamientos

	asociados actualizados
Escenario alternativo:	-
Requisitos asociados:	CW_8, CW_10

Campos	**CASO DE USO 11**
Caso de uso:	Eliminar Comportamiento de Agente
Descripción:	El usuario podrá eliminar un comportamiento de un agente
Precondición:	El usuario está visualizando la lista de comportamientos vinculados a un agente (Caso de uso 8 – Listar Comportamientos de Agente)
Escenario principal:	1- El usuario selecciona gestionar los comportamientos del agente. 2- El sistema muestra los comportamientos vinculados al agente que pueden ser eliminados 3- El usuario selecciona el comportamiento que quiere eliminar del agente y confirma el cambio 4- El sistema redirige al usuario a la página donde se visualizan los agentes con los comportamientos asociados actualizados
Escenario alternativo:	-
Requisitos asociados:	CW_8, CW_11

Campos	CASO DE USO 12
Caso de uso:	Resetear Comportamiento de Agente
Descripción:	El usuario podrá resetear un comportamiento de un agente
Precondición:	El usuario está visualizando la lista de comportamientos vinculados a un agente (Caso de uso 8 – Listar Comportamientos de Agente)
Escenario principal:	1- El usuario selecciona gestionar los comportamientos del agente. 2- El sistema muestra los comportamientos vinculados al agente que pueden ser reseteados 3- El usuario selecciona el comportamiento que quiere resetear y confirma el cambio 4- El sistema redirige al usuario a la página donde se visualizan los agentes con el estado del comportamiento actualizado
Escenario alternativo:	
Requisitos asociados:	CW_8, CW_12

3. DISEÑO

3.1 Arquitectura y componentes del framework

Vamos a empezar explicando el diseño de nuestro framework para integrar JADE con aplicaciones empresariales web. Lo hemos dividido en tres componentes con varios subelementos: *Agent, Behaviour* y *Manager*. A través de estos componentes podemos interactuar con la plataforma JADE para conseguir el despliegue del contenedor JADE de forma sincronizada con el contenedor servlet donde está ejecutándose y la gestión de agentes y comportamientos de forma desacoplada.

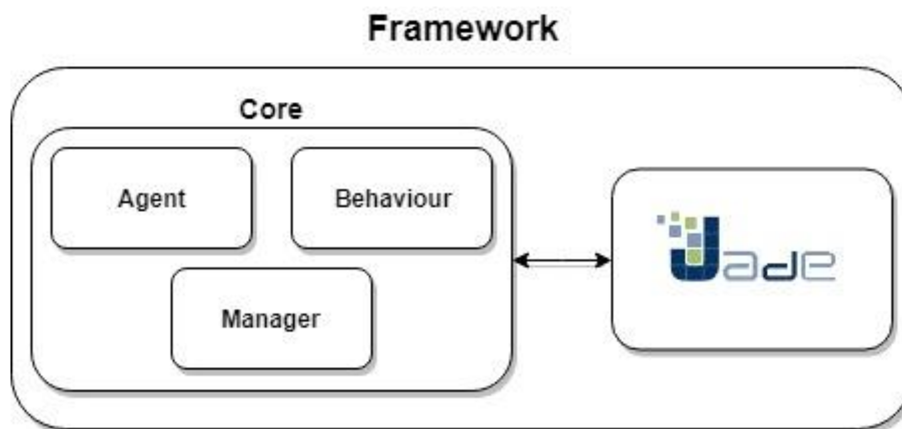

Figura 3.1 Arquitectura del framework

Ahora bien, como explicaremos en el apartado de implementación, aunque algunos subelementos pueden variar dependiendo de si nos encontramos en Java EE o Spring, la arquitectura es la misma. La única diferencia es el lugar en el que se encuentra localizado.

3.2 Arquitectura con Aplicaciones Java EE

Las aplicaciones empresariales web con Java EE se despliegan en un servidor de aplicaciones web como un *.ear*, un paquete que contiene tanto el componente de presentación/vista (*.war*) como el componente de lógica de negocio (*.ejb).* Los agentes y la plataforma Jade se encontrarán localizados en esta capa EJB (Enterprise Java Bean), que a su vez podrá tener distintos componentes como una capa de datos *facade* para gestionar datos persistentes en una base de datos.

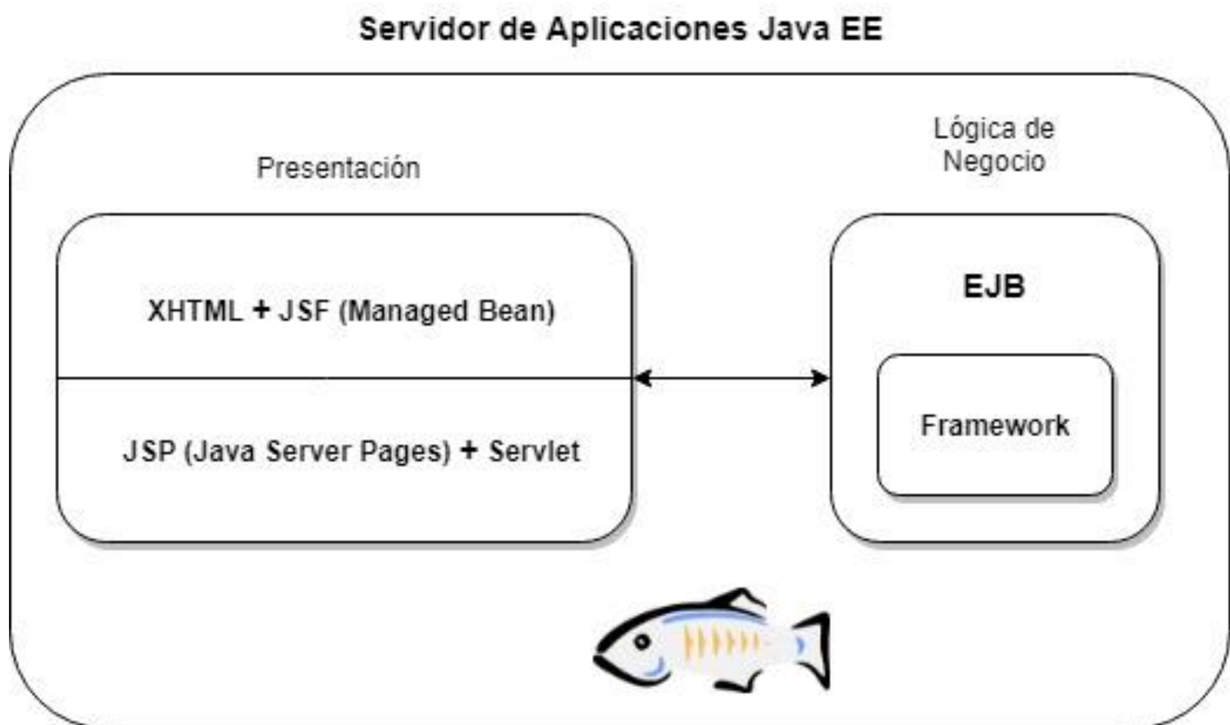

Figura 3.2 Localización del framework en arquitectura genérica Java EE

3.3 Arquitectura con Aplicaciones Spring

Con aplicaciones Spring los agentes y la plataforma Jade también se encontrarán localizados en la capa de lógica de negocio. La diferencia es que con Spring no necesitamos desplegar la aplicación en un servidor de aplicaciones, sino que podemos desplegarla como un único componente *.war* en un servidor web (contenedor java servlet).Teniendo en consideración que de forma genérica se suele dividir la capa de lógica de negocio en una capa de servicios y una capa de datos, nuestros agentes y plataforma Jade estarían situados en esta capa de servicios.

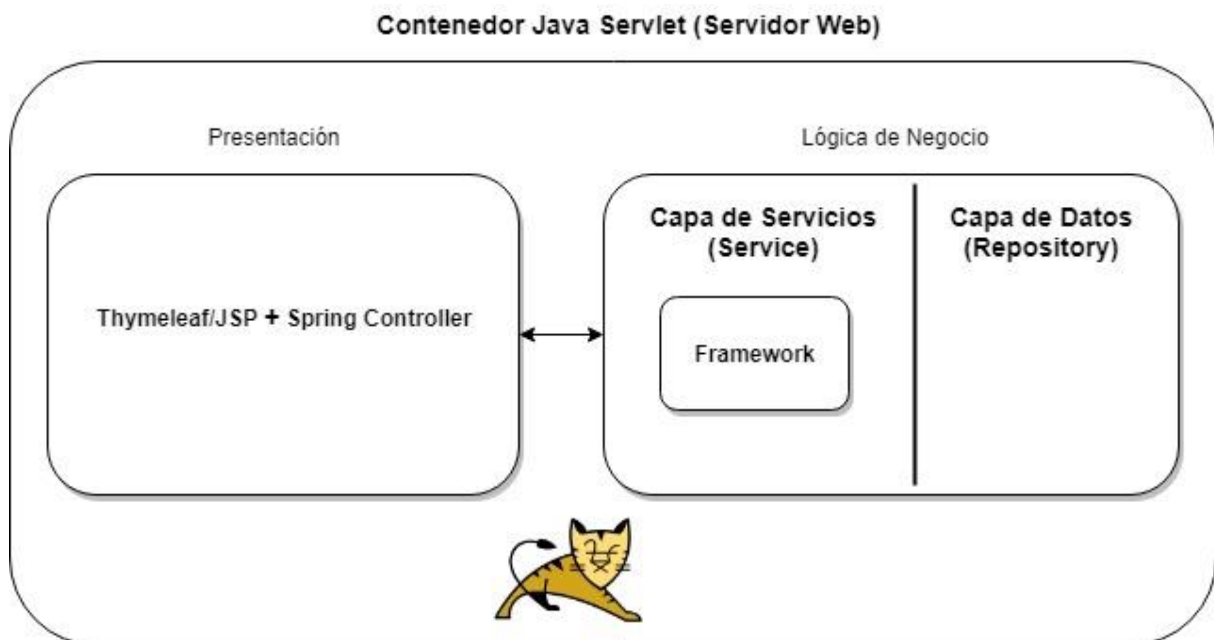

Figura 3.3 Localización del framework en arquitectura genérica Spring

3.4 Arquitectura de la consola web

Ahora necesitamos una consola web para monitorizar y gestionar los agentes que se ejecutan en la plataforma JADE. Con Spring MVC y Spring Boot tenemos la posibilidad de desplegar y ejecutar las aplicaciones en contenedores servlet empotrados, lo que mejora mucho la portabilidad y permite que el desarrollador se enfoque más en la aplicación que en la arquitectura de despliegue. Del mismo modo, Spring Boot facilita mucho la creación de APIs REST mediante anotaciones como *@RestController* que junto a la librería *Jackson* permite la creación de modelos JSON de forma transparente utilizando las clases creadas en nuestra aplicación.

Con estas herramientas y en línea con el objetivo principal de poder monitorizar agentes que estén ejecutándose gracias a nuestro framework en un servidor web hemos decidido dividir la gestión de los agentes en dos partes. Por un lado hemos desarrollado una API REST desde la que poder controlarlo todo a través de peticiones HTTP e información utilizando el formato de dato JSON. Un usuario técnico utilizando un cliente HTTP como Postman es capaz de trabajar con los agentes sin necesidad de ningún requisito extra. La interfaz de la consola web será pues, otra aplicación Spring totalmente autónoma, que consume datos de los agentes a través de esta API REST y las presenta de forma cómoda e intuitiva al usuario final, gestionando todos los problemas para que sin necesidad de un alto nivel técnico específico en la materia sea capaz de llevar a cabo su cometido.

Entre la multitud de ventajas de esta arquitectura destacamos la flexibilidad de crear distintos clientes, tanto web como móvil, con distintos enfoques y tecnologías utilizando una interfaz uniforme. Además de la capacidad de aislar los errores propios de la API y nuestro framework a los problemas del lado del cliente.

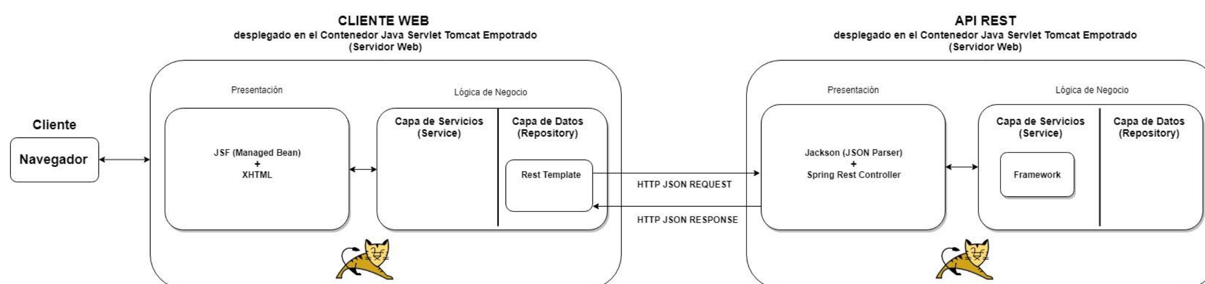

Figura 3.4 Arquitectura de la consola web

3.5 Arquitectura de la API REST

La arquitectura es similar a la genérica de nuestro framework con aplicaciones Spring. En la capa de presentación utilizamos *Spring Rest Controller* para escuchar, y devolver una respuesta adecuada a las peticiones HTTP JSON. Spring hace uso de la librería Jackson para convertir las instancias de las distintas clases en objetos JSON y viceversa. No creamos ninguna vista HTML, solo devolvemos datos JSON.

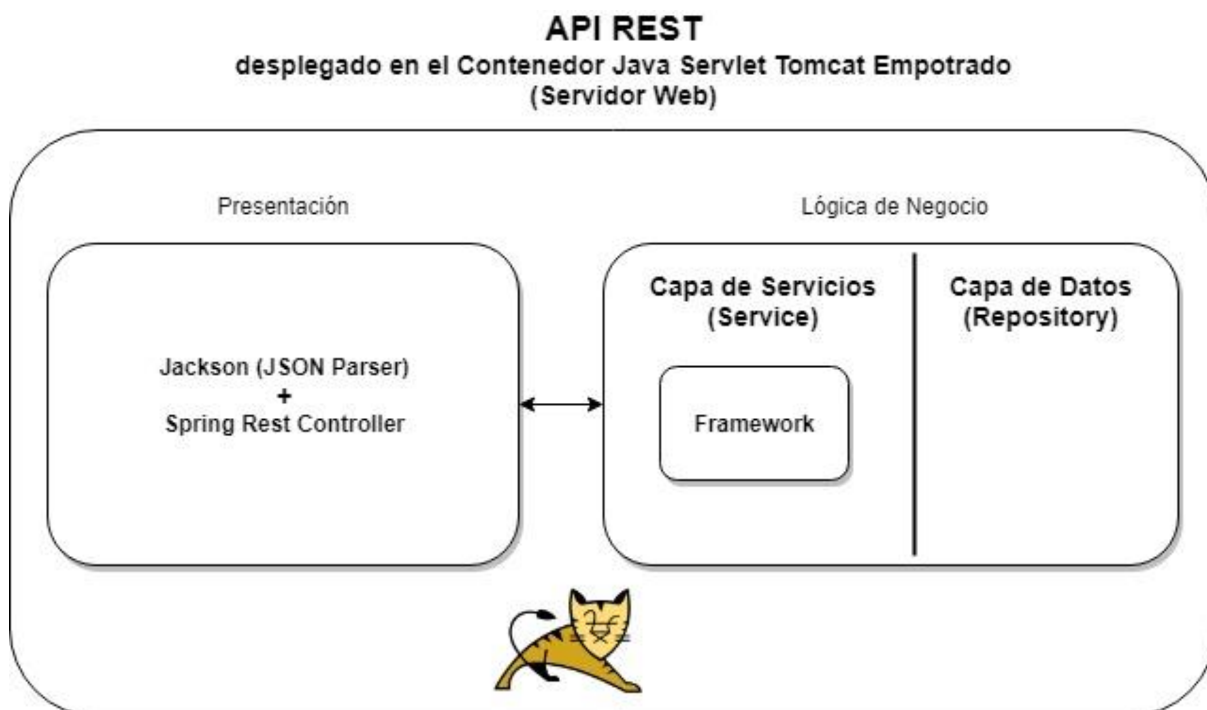

Figura 3.5 Arquitectura de la API REST

3.6 Arquitectura del cliente web

Una vez abierta la posibilidad de crear distintos clientes web, tuvimos que decidirnos por una tecnología y enfoque concreto. Podríamos haber utilizado React, Angular 5, VueJS o VanillaJS con JQuery para el lado del cliente; también tenemos la posibilidad de utilizar otros lenguajes como Ruby (Ruby on Rails), Python (Django) o PHP (Laravel), además de clientes móviles (Android y IOS). La flexibilidad a la hora de utilizar una API REST como soporte de la gestión de agentes nos permite todo esto. Decidimos seguir utilizando Spring por la capacidad de despliegue en un contenedor java servlet empotrado y por cierta homogeneidad en el desarrollo del TFG. No obstante, para la capa de presentación no hemos utilizado ni *Thymeleaf* ni *JSP*, sino que hemos integrado JSF con Spring Boot de manera que podemos gestionar toda la capa de presentación de la forma habitual que nos encontramos en aplicaciones Java EE, pero con la ventaja de no necesitar un servidor de aplicaciones Java EE como glassfish. La capa de lógica de negocio, al igual que en las aplicaciones genéricas con Spring, también la dividimos en dos capas. La capa de datos permite abstraer la gestión de los agentes: los datos referentes a ellos podrían estar en la propia aplicación, en una base de datos o en cualquier otro sitio. Nosotros obtenemos toda la información a través de la interfaz de la API, y para ellos usamos otra tecnología de Spring, *Rest Template*, que permite hacer peticiones a una API REST en formato JSON de forma cómoda e intuitiva.

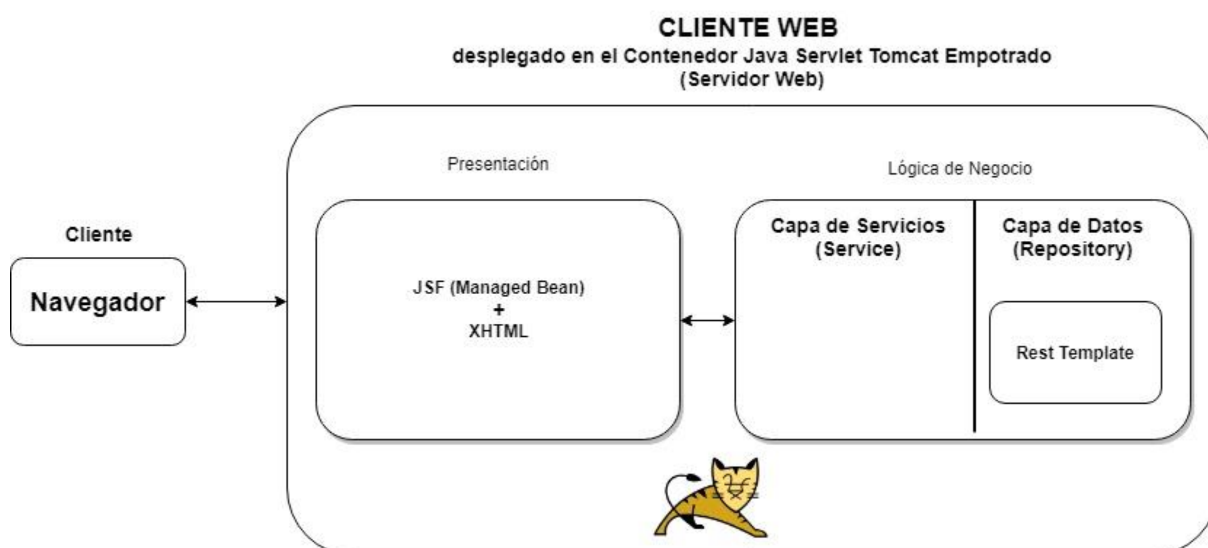

Figura 3.6 Arquitectura del cliente web

4 IMPLEMENTACIÓN DEL FRAMEWORK

En este capítulo especificaremos y justificaremos todos los detalles a nivel de diseño e implementación de los distintos aspectos de este TFG.

Por un lado describiremos los módulos que conforman nuestro framework y cómo actúan a modo de *wrapper* (envoltorio) sobre la plataforma Jade para suministrar esa compatibilidad con aplicaciones web. Explicaremos cómo debe usarse, posibilidades, limitaciones y mejoras respecto a la plataforma original.

Una vez finalizada esta parte tanto para Java EE como para Spring explicaremos el funcionamiento de nuestra API REST y su uso de nuestro framework dentro de la capa de servicios.

Por último, explicaremos el funcionamiento de nuestra consola web, los distintos casos de uso implementados, comunicación con la API y gestión de errores.

4.1 Componentes del framework

En el capítulo anterior explicamos e ilustramos la arquitectura de nuestro framework, localización y componentes. A continuación especificaremos los elementos de los que están compuestos junto a una breve descripción.

Componente	Nombre de ficheros	Descripción
core.manager	**Clases:** AgentManager	Gestión del contenedor JADE. Inicialización automática cuando el Bean es desplegado en el servidor de aplicaciones o servidor web, finalización automática, añadir agentes, eliminar agentes y controlar la lista de agentes en el contenedor

core.agent	**Clases:** JadeAgentException **Clases abstractas:** SpringAgent/ EJBAgent **Interfaces:** AgentInterface **Anotaciones:** AgentQualifier	Interfaz e implementación de las funcionalidades necesarias para gestionar los agentes y sus comportamientos en el contenedor JADE de forma transparente. Los agentes heredan de nuestra clase abstracta implementando los métodos necesarios. AgentQualifier es una anotación para la librería CDI que me permite inyectar agentes desde la capa EJB, no es necesario para Spring.
core.behaviour	**Clases:** SimpleBehaviourFactory, SharedVariable, SharedVariableInteger **Interfaces:** BehaviourWithFactoryInterface, SharedVariableInterface **Anotaciones:** BehaviourQualifier	Interfaces y clases para gestionar comportamientos que no son creados directamente en el EJB o la capa servicios. SimpleBehaviourFactory es una factoría que combinando los distintos elementos permite la creación de estos comportamientos. BehaviourWithFactoryInterface es la interfaz que deben implementar los comportamientos que van a ser gestionados vía API. BehaviourQualifier es una anotación para la librería CDI que permite inyectar comportamientos desde la capa EJB; no es necesario para Spring.

example.agent	**Clases:** HelloAgent, ByeAgent, PlainAgent, ReceiveMessageAgent y SendMessageAgent	Ejemplos de agentes que utilizando nuestro framework se pueden inyectar y gestionar tanto por código como a través de la API.
example.behaviour	**Clases:** SimplePrintMessageBehaviour, SimpleCyclicBehaviour, ReceiveACLMessageBlockBehaviour y SendACLMessageBlockBehaviour	Ejemplos de comportamientos que utilizando nuestro framework se pueden inyectar y gestionar tanto por código como a través de la API.

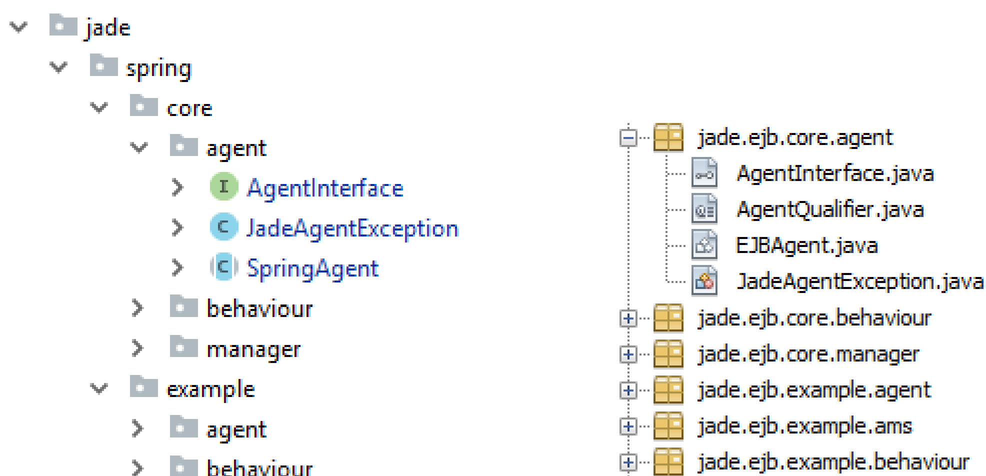

Figura 4.1 Estructura de los componentes del framework. Spring izquierda y EJB derecha

4.2 Diferencias del framework entre EJB y Spring

Nos encontramos ante diferencias menores, por lo que son prácticamente análogos:

-El nombre del paquete por cuestiones de semántica: *jade.spring.core/example* y *jade.ejb.core/example.*

-El par de clases *SpringAgent / EJBAgent* solo se diferencian en el nombre, ante la posibilidad de escoger un nombre común como *WebBehaviour* hemos creído más conveniente utilizar el nombre de cada plataforma queriendo dar la semántica de que es la abstracción del agente JADE para que pueda ser utilizado en dicha plataforma. Del mismo modo, al ser un proyecto abierto con muchas posibilidades de extensión, creemos bastante posible que estas clases puedan evolucionar de manera distinta, por lo que ha sido una decisión de diseño.

-Las anotaciones *@AgentQualifier* y *@BehaviourQualifier* solo se encuentran presentes en el framework EJB. En Java EE los beans empresariales (*enterprise java beans*) son inyectados en la capa *web* a través de anotaciones de la librería CDI. Nosotros hemos usado esa librería para inyectar nuestros agentes y comportamientos de forma fácil e intuitiva, basta con anotar un bean sin estado (*@Stateless*) con nuestra anotación y el nombre del agente/comportamiento a inyectar (*@AgentQualifier("<nombre del agente>"))* para después en el managed bean inyectarlo ya sea en la variable de clase, un método *set* o el constructor, usando *@Inject* junto a la misma anotación (*@AgentQualifier("<nombre del agente>"))*. De forma análoga conseguimos el mismo resultado para los comportamientos. Por otro lado, aunque Spring admite inyecciones a través de la librería CDI, lo normal es utilizar el sistema de inyección de dependencias que trae consigo. En Spring todos los beans son componentes, y estos componentes se anotan con *@Component* o de forma más concreta con *@Service* o *@Repository*, por ejemplo. Junto a ella anotamos el agente o comportamiento con *@Qualifier("<nombre del agente/comportamiento>")*, y en el lugar donde vayamos a utilizar la instancia inyectada basta con sustituir *@Inyect* por *@Autowired* y añadir la anotación *@Qualifier("<nombre del agente/comportamiento>")* junto a la variable de clase, un método *set* o el constructor.

-Las anotaciones concretas de los beans empresariales y los componentes de Spring son distintas. Para el manager en EJB tenemos que anotar el bean con *@Singleton @Startup* para que se garantice que el bean tenga únicamente una instancia y se inicie automáticamente cuando el servidor de aplicaciones haya sido desplegado de forma satisfactoria. Por otro lado, en Spring la anotación *@Component* y su derivado concreto *@Service* ya te garantiza estas condiciones. Los métodos anotados con *@PostConstruct* y *@PreDestroy*, son válidos para ambos.

4.3 Componente manager

A continuación vamos a explicar los detalles de implementación este componente:

AgentsManager

Esta es la clase utilizada para la gestión del contenedor JADE. Desde JADE 2.3 existe una interfaz que permite que distintas aplicaciones Java usen JADE como una especie de librería y sea posible arrancar una instancia de ejecución de JADE desde dentro. Esta instancia *singleton* se obtiene a través del método estático *jade.core.Runtime.instance(),* que provee dos métodos para crear un contenedor principal de JADE o un contenedor remoto (mecanismo que posibilita que un contenedor se una a un contenedor principal existente para formar una plataforma de agentes distribuida). Ambos métodos requieren como parámetro un objeto *jade.core.Profile* que guarde las opciones de configuración (*hostname* y número de puerto del contenedor principal) necesarios para empezar la ejecución de JADE.

Ambos métodos de *Runtime* devuelven un objeto *wrapper* (envoltorio), que pertenece al paquete *jade.wrapper*, envolviendo la funcionalidad de más alto nivel de los contenedores de los agentes, como instalar y desinstalar distintos *MTPs (Message Transport Protocol* o Protocolo de Transporte de Mensajes), destruir/apagar (*kill*) el contenedor (donde el apagado del contenedor no conlleva acabar con la ejecución de la aplicación Java) y, por supuesto, crear nuevos agentes. El método *createNewAgent* de este envoltorio del contenedor (*AgentContainer*) devuelve a su vez otro objeto envoltorio (*AgentController*) que

encapsula algunas funcionalidades del agente, pero intentando preservar la autonomía de éstos en la medida de lo posible. De forma concreta, la aplicación puede controlar el ciclo de vida de los agentes, pero no puede obtener una referencia al objeto Agente, y por consecuente no puede invocar sus métodos. Además hay que tener en cuenta que aunque el agente haya sido creado aún hay que iniciarlo invocando el método *start()* de *AgentController*.

Teniendo en cuenta todo esto bastaría con especificar el nombre del agente, la ruta hasta la clase del agente y los argumentos de entrada. No obstante, conocer la ruta de la clase de forma dinámica es más complicado, y realmente no tener acceso a la instancia del Agente (*Agent)* y solo al envoltorio *AgentController* nos limita mucho en temas de flexibilidad. Por ello decidimos crear el Agente antes e invocar al método *acceptNewAgent(String nickname, Agent anAgent)* de *AgentContainer*, guardando una lista de los agentes en el contenedor dentro de la clase *AgentsManager*.

Vamos a especificar las anotaciones y variables de clase: tal y como hemos mencionado en las diferencias entre el framework para EJB y Spring, tenemos que anotar el bean/componente para que se garantice que la instancia sea única y se inicie automáticamente cuando el servidor de aplicaciones o el servidor web haya sido desplegado de forma satisfactoria. Guardamos en una variable la única instancia de nuestro contenedor (actualmente trabajamos sólo con un contenedor, pero cambiando la variable a una lista podríamos gestionar un número variable de ellos con agentes concretos ejecutándose en contenedores aislados, pero de momento para la funcionalidad de este TFG la utilización de un contenedor era suficiente. Simplemente hemos tenido en cuenta que el diseño pueda extenderse con facilidad con pequeños cambios intuitivos a cualquiera con deseo de modificar algún aspecto del framework).

Además, gestionamos directamente las instancias de los agentes, por lo que tenemos una lista con todos los agentes presentes en el contenedor. Podríamos haber utilizado el patrón *Singleton* para obtener una única instancia de la clase, pero con la inyección de dependencias no era necesario (es más, la inyección de dependencias no utilizaría el método *getInstance()* para obtener la instancia

singleton); teniendo en cuenta que se usa principalmente dentro de los agentes compatibles con EJB y Spring esto podría llevar a la existencia de más de una instancia. El enfoque adoptado ha sido el de utilizar variables y métodos estáticos, garantizando que podamos tanto inyectar una instancia de *AgentManager* como invocar directamente los métodos con el nombre de la clase desde los agentes compatibles sin riesgo de caer en problemas de consistencia.

```
//@Singleton @Startup
@Service
public class AgentsManager {
    private static AgentContainer mainContainer;
    private static Map<String,Agent> agentsOnContainer = new HashMap<>();
```

Figura 4.2 Anotaciones y variables de la clase AgentsManager

El método *startup* anotado con *@PostConstruct* garantiza que el contenedor JADE se inicie justo después del despliegue del servidor web/de aplicaciones. Con la anotación garantizamos que el bean esté completamente iniciado y, aunque no haya inyecciones, es siempre mejor opción que inicializar los elementos en el constructor. Podemos comprobar cómo se realizan los pasos explicados anteriormente.

```
@PostConstruct
public void startup(){
    if (mainContainer == null){
        createAndInitMainContainer();
    }else{
        try {
            mainContainer.start();
        } catch (ControllerException ex) {
            throw new JadeAgentException("ERROR: ControllerException - " +
                    " main container failed when it was being started -> "
            + ex.getMessage());
        }
    }
}

private static void createAndInitMainContainer(){
    Runtime runtime = Runtime.instance();
    ProfileImpl profile = new ProfileImpl( isMain: false);
    mainContainer = runtime.createMainContainer(profile);
}
```

Figura 4.3 Inicio del contenedor JADE vinculado al despliegue del servidor web/de aplicaciones

El método *shutdown* anotado con *@PreDestroy* es invocado cuando la instancia está en proceso de ser eliminada por el contenedor servlet. De esta manera garantizamos que se liberan los recursos necesarios, y en el caso de un despliegue automático de la aplicación ante pequeños cambios evitamos que el contenedor alcance un estado inconsistente.

```
@PreDestroy
public void shutdown(){
    if (mainContainer != null){
        try {
            mainContainer.kill();
        } catch (StaleProxyException ex) {
            throw new JadeAgentException("ERROR: StaleProxyException - " +
                    "main container failed when it was being finished -> "
            + ex.getMessage());
        }
    }
}
```

Figura 4.4 Apagado del contenedor JADE vinculado a la finalización del servidor web/de aplicaciones

Para la adición de agentes al contenedor *AgentsManager* solo necesita el nombre y la instancia del agente. Comprobamos que tanto el nombre del agente como la instancia sean únicos y entonces añadimos el agente al contenedor. Posteriormente iniciamos el agente (lo que significa que un agente creado con el estado *Initiated (1)* y ya suscrito a los distintos servicios del contenedor, comienza a ejecutar sus comportamientos) y añadimos el nuevo agente al mapa de agentes.

```
public static synchronized void addAgentToMainContainer(String nickname, Agent agent){
    try {
        if (!agentsOnContainer.containsKey(nickname) && agent != null
                && !agentsOnContainer.values().contains(agent)){
            AgentController agentController = mainContainer.acceptNewAgent(nickname, agent);
            agentController.start();
            //add agents to container after it is started, just in case it fails, so we do not fall
            //in an error state where agent is not initiated and it is already added to the list
            agentsOnContainer.put(nickname, agent);
        }else{
            throw new JadeAgentException("ERROR: problem adding an agent to the container:" +
                    " an agent with nickname " + nickname + " already exist in the system" +
                    " or the agent is null or is already added");
        }
    } catch (StaleProxyException ex) {
        throw new JadeAgentException("ERROR: StaleProxyException - exception trying to init again" +
                " the agent " + nickname + " ! -> " + ex.getMessage());
    }
}
```

Figura 4.5 Adición de agentes al contenedor JADE

Para eliminar un agente del contenedor comprobamos que no sea *null*, ni su variable *localName* ni la instancia *AgentController* obtenida con ese nombre tampoco. Si pasa el filtro invocamos el método *doDelete()* sobre la variable agente y lo eliminamos de la lista de agentes. De forma interna esto invocará al método *shutdown* de la clase *Agent* y cambiará el estado del agente en el contenedor a eliminado.

No podemos olvidar que tanto en este método como en el anterior nos encontramos con el modificador *synchronized*. En un entorno multi-hebra cuando una hebra está ejecutando el método sincronizado, el resto de hebras que traten de invocarlo para el mismo objeto se bloquean (suspenden su ejecución) hasta que la primera hebra termine la ejecución. Esto es importante porque garantiza que la lista de comportamientos permanezca actualizada y visible para todas las hebras y no haya problemas de consistencia.

```
public static synchronized void takeDownAgent(String nickname, Agent agent) {
    try {
        if (agent == null) {
            throw new JadeAgentException("ERROR: The agent you are trying to shut down is null");
            //local name is set by contained when agent is added to it.
            //then, if it hasn't been added yet or has been deleted already we do not call
            //agent.doDelete although following API guide it wouldn't have any side effect
        } else if (agent.getLocalName() != null && mainContainer.getAgent(agent.getLocalName()) != null) {
            agent.doDelete();
            deleteAgentFromList(nickname, agent);
        } else {
            throw new JadeAgentException("ERROR: The agent " + nickname + " that you are trying to shut" +
                    " down is not initiated or has been already shutted down");
        }
    } catch (ControllerException ex) {
        throw new JadeAgentException("ERROR: ControllerException - exception shutting down the agent " +
                nickname + " ! -> " + ex.getMessage());
    }
}
```

Figura 4.6 Eliminación de agentes del contenedor JADE

Finalmente tenemos el método que permite que mediante la inyección de una instancia de esta clase en la capa web o capa servicios sea posible obtener todos los agentes activos en el contenedor con el fin de gestionarlos.

```
public Map<String,Agent> getAgentsOnContainer (){
    return agentsOnContainer;
}
```

Figura 4.7 Obtención de los agentes activos en el contenedor

4.4 Componente agent

A continuación vamos a explicar los detalles de implementación este los elementos de este componente:

JadeAgentException

Es simplemente una excepción no controlada que utilizamos para lanzar excepciones en distintos puntos de nuestro framework. Encapsulamos las excepciones lanzadas por la propia librería JADE para poder así tener un control más exacto de los puntos de error.

```
public class JadeAgentException extends RuntimeException{

    public JadeAgentException() { super(); }

    public JadeAgentException(String msg) { super(msg); }

}
```

Figura 4.8 Implementación de JadeAgentException

AgentQualifier

Esta anotación solo está disponible para el framework adaptado a EJB. Hacemos uso de la librería CDI para crear nuestras propias anotaciones y poder inyectar nuestros agentes en la capa web.

La anotación *@Qualifier* de la librería CDI es la que hace posible que nuestra anotación sirva para inyectar instancias java desde los enterprise java beans.

@Target especifica los sitios sintácticos donde puede aplicarse la anotación, nosotros hemos determinado los campos *FIELD* (declaración de variables de clase, incluyendo enumerados), *METHOD* (declaración de métodos), *PARAMETER* (declaración formal de parámetros) y *TYPE* (clases, interfaces incluyendo las de anotaciones y declaracion de enumerados).

@Retention determina el tiempo de retención de las anotaciones, *RetentionPolicy.RUNTIME* especifica que las anotaciones tienen que ser guardadas en los ficheros de clase por el compilador y manterse disponible por la máquina

virtual de Java en tiempo de ejecución, para que así podamos leerlas mediante reflexión (*reflection*). Por defecto su valor es *RetentionPolicy.CLASS*, que incluye las anotaciones en los ficheros de clase pero no se mantienen en tiempo de ejecución, y por último tenemos la opción *RetentionPolicy.SOURCE* que descartan las anotaciones en tiempo de compilacion.

@Documented indica que la anotación será documentada por javadoc y herramientas similares por defecto.

```
@Qualifier
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
```

Figura 4.9 Anotaciones de AgentQualifier

Las anotaciones se definen con el modificador *@interface* y dentro se especifican los valores que permitirán la resolución de la inyección. Nosotros no hemos desarrollado un consumidor de anotaciones para analizarlas y devolver una instancia en función de estos parámetros, pues no es necesario para inyecciones básicas como ésta. Si creásemos una anotación para cada clase agente, simplemente con anotar el lugar donde se va a producir la inyección es suficiente, ya que el sistema busca la clase de la variable y crea una instancia usando el constructor vacio por defecto. Sin embargo la autentica flexibilidad y potencial reside en el uso orientado a interfaces, en el que inyectamos clases que implementan interfaces, y en esta situación si solo tenemos una clase la resolución de la inyección será satisfactoria, pero en caso contrario tendremos problemas en tiempo de compilación ya que no se podrá resolver la inyección. La forma de hacerlo escalable a la creación de un número indeterminado de agentes es la de usar parámetros para resolverlo, ya sea un valor enumerado o un valor String, por ejemplo. Con esta forma de resolver anotaciones basta con anotar nuestra clase agente que implementa nuestra interfaz *AgentInterface* con nuestra anotación y un valor String, para posteriormente utilizar la misma anotación con el mismo nombre en el lugar donde queramos inyectarlo.

```
public @interface AgentQualifier {
    String value() default "";
}
```

Figura 4.10 Anotación AgentQualifier

En Spring no utilizamos esta anotación, sino *@Qualifier*, una anotación propia del ecosistema. Ya no utilizamos la librería CDI porque, aunque existe la posibilidad, creemos que esta opción es la más adecuada con el fin de integrarnos en mayor medida con la forma de desarrollar de Spring. El modo de uso es totalmente análogo a *@AgentQualifier* con la diferencia de que también lo utilizamos de forma indistinta para *@BehaviourQualifier*.

```
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER, ElementType.TYPE, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Qualifier {
    String value() default "";
}
```

Figura 4.11 Anotación Qualifier de org.springframework.beans.factory.annotation

AgentInterface

AgentInterface es la interfaz que determina los métodos necesarios para permitir la compatibilidad con la clase *AgentsManager*. Del mismo modo esto nos permite que más adelante podamos inyectar los agentes utilizando su interfaz en vez de instancias concretas, pudiendo modificar el agente inyectado a través del valor suministrado a *AgentQualifier* o *Qualifier* desde la capa EJB o de servicios sin tener que modificar nada más a nivel de código. La implementación de estos métodos será común a todos los agentes, para lo que implementamos la clase abstracta *EJBAgent/SpringAgent*.

Lo métodos *init()* y *init(String nickname)* me permiten añadir el agente al contenedor jade e iniciarlo.

El método *shutDownAgent()* me permite finalizar la ejecución del agente, devolviendo una instancia de *AgentInterface* que sustituirá a la que estábamos usando hasta el momento. Esto ha sido otra decisión de diseño que hemos tenido

que tomar: a lo largo de todo el desarrollo hemos partido de la premisa de no modificar la librería JADE, sino de desarrollar código que la envolviese añadiendo dicha compatibilidad. Al ser un proyecto de código abierto hemos analizado la implementación de la evolución entre los estados del ciclo de vida de los agentes y hemos comprobado que evolucionar desde el estado *DELETED* (agente que ha sido desuscrito de todos los servicios del contenedor y que no está ejecutándose) a *INITIATED* (instancia de agente creado y listo para ser añadido al contenedor) no es posible. Esta implementación es totalmente lógica, pero implica que si detenemos el agente (eliminándose del contenedor) y queremos volver a iniciarlo, éste ya tendrá el estado *DELETED*, por lo que no se iniciará cuando lo añadamos al contenedor JADE (estará en el contenedor pero con el estado *DELETED*, el método *setup* de la clase *Agent* no será invocado y el agente no podrá ejecutar ningún comportamiento, además necesitamos que el método *setup* sea invocado para otras funcionalidades como el registro en las páginas amarillas). El mejor enfoque que hemos encontrado para resolver este problema es devolver una instancia del mismo tipo de agente con la misma lista de comportamientos después de haya terminado la ejecución del agente y haya sido eliminado del contenedor.

Por otro lado, el método *getAgentInstance()* devuelve la instancia de tipo *Agent* de una instancia de una clase que hereda de ella, ya que multitud de métodos de la librería JADE requieren una instancia de tipo *Agent*.

En cuanto a los comportamientos, la interfaz recoge que nuestros agentes tendrán una colección de comportamientos, y que podrán añadir nuevos comportamientos, reiniciarlos y eliminarlos. Tenemos la opción de añadir comportamientos que empezarán a ejecutarse en el momento o que esperarán al siguiente reinicio del agente, al igual que podemos eliminar comportamientos de la ejecución actual o eliminar comportamientos de forma definitiva.

También podemos gestionar el nombre/*nickname* del agente de forma previa a su iniciación y obtener el nombre de la clase a la que pertenece una instancia concreta de un agente. Como trabajamos orientado a interfaces, dado un momento concreto de la ejecución y con visión de futuro hacia el desarrollo de la API y consola web este método nos devuelve este dato. La implementación es idéntica para cualquier

clase que implemente la interfaz, sea un *Agent* o no y sin necesidad de utilizar variables de clase, por lo que lo declaramos como *default* y la implementamos en la propia interfaz.

```
public interface AgentInterface {
    void init();
    void init(String nickname);
    AgentInterface shutDownAgent();
    Agent getAgentInstance();
    //the user can create it owns jade behaviour if he does not want to use our framework
    //we also use this method to attach behaviour when they are created via SpringBehaviour
    boolean addBehaviourToAgent(Behaviour behaviour);
    //addBehaviourToAgentAndInit is used to add a behaviour to an Agent that is already running
    //and we want to execute it in the current execution instead of waiting to the next restart
    boolean addBehaviourToAgentAndInit(Behaviour behaviour);
    boolean resetBehaviourFromAgent(Behaviour behaviour);
    boolean resetBehaviourFromAgent(String behaviourName);
    boolean removeBehaviourFromAgent(Behaviour behaviour);
    boolean removeBehaviourFromAgent(String behaviourName);
    boolean removeBehaviourFromAgentForever(Behaviour behaviour);
    boolean removeBehaviourFromAgentForever(String behaviourName);
    Collection<Behaviour> getBehavioursFromAgent();

    void setNickname(String nickname);
    String getNickname();

    //useful to identify which class of agent we have (we can have several agents from same class)
    default String getAgentClassName(){
        return this.getClass().getSimpleName();
    }
}
```

Figura 4.12 Implementación de AgentInterface

EJBAgent/SpringAgent

Nos encontramos ante una clase abstracta que extiende de la clase *Agent* de la librería JADE e implementa nuestra interfaz *AgentInterface*. La finalidad es cubrir la funcionalidad necesaria que consiga abstraer a las clases concretas de implementar de forma duplicada la iniciación, apagado y gestión de los comportamientos.

Las variables de clase son el nombre que el agente tendrá cuando se añada al contenedor, la lista de comportamientos que tiene vinculado y si se ha iniciado.

```
public abstract class SpringAgent extends Agent implements AgentInterface{
    private String nickname = "";
    private List<Behaviour> behaviourList = new ArrayList<>();
    private boolean initiated = false;
```

Figura 4.13 Variables de clase de EJBAgent/SpringAgent

Sobrescribimos los métodos *setup()* y *takeDown()* de la clase Agent con la funcionalidad mínima a implementar. Estos métodos son invocados por el contenedor JADE cuando el agente comienza y termina su ejecución, en ellos tenemos que inicializar variables, añadir comportamientos, registrarnos y derregistrarnos de las páginas amarillas, etc. Las subclases que necesiten inicializar variables concretas o registrarse en las páginas amarillas entre otras tareas deben invocar al método de la súper clase (*super.setup () y super.takeDown())* para garantizar el funcionamiento esperado. Para ello usamos la anotación *@OverridingMethodsMustInvokeSuper* (de la librería *com.google.code.findbug .jsr305)* que avisa a las subclases de esta necesidad.

```
@OverridingMethodsMustInvokeSuper
@Override
protected void setup(){
    for (Behaviour b : behaviourList){
        addBehaviour(b);
    }
    initiated = true;
}

@OverridingMethodsMustInvokeSuper
@Override
protected void takeDown(){
    initiated = false;
}
```

Figura 4.14 Métodos setup y takeDown de EJBAgent/SpringAgent

El método *init()* invoca el método estático *addAgentToMainContainer* de la clase AgentsManager con el nombre del agente y la instancia. Aquí hemos optado por una sobrecarga del método *init* para el caso en el que no se suministre el nombre del agente. Si no se especifica el nombre del agente y este no ha sido establecido con anterioridad, el nombre final será el nombre de la clase a la que pertenece la instancia. En el caso de que el agente ya esté iniciado lanzamos una excepción.

```java
@Override
public void init() {
    if (nickname.equals("")){
        init(this.getClass().getSimpleName());
    }else{
        init(nickname);
    }
}

@Override
public void init(String nickname) {
    try {
        if (!isInitiated()) {
            this.nickname = nickname;
            AgentsManager.addAgentToMainContainer(nickname,  agent: this);
        }else{
            throw new JadeAgentException("ERROR: The agent you are" +
                    " trying to init is already initiated");
        }
    } catch (Exception ex) {
        throw new JadeAgentException("ERROR: exception adding the" +
                " agent to the container " + ex.getMessage());
    }
}
```

Figura 4.15 Métodos init de EJBAgent/SpringAgent

Para obtener una instancia de tipo *Agent* (superclase de la instancia actual) solo tenemos que hacer un *cast* de la instancia actual. Así evitamos una encapsulación constante que dificulte la legibilidad del código que haga uso de esta clase.

```java
@Override
public Agent getAgentInstance(){
    return (Agent) this;
}
```

Figura 4.16 Método getAgentInstance de EJBAgent/SpringAgent

En el caso de parar/finalizar el agente, no solo basta con invocar el método *takeDownAgent()* de *AgentsManager*, sino que hay que crear una nueva instancia, eliminar los comportamientos de la instancia original, reiniciarlos y añadírselos a la nueva instancia, sin olvidar especificar el *nickname* que le habíamos asignado. En el caso de que el agente no haya sido iniciado devolvemos la instancia actual sin ninguna modificación.

```
@Override
public AgentInterface shutDownAgent() {
    AgentInterface newInstance = this;
    if (isInitiated()) {
        AgentsManager.takeDownAgent(nickname,  agent: this);
        newInstance = getNewInstance();
        for (Behaviour b : behaviourList) {
            removeBehaviour(b);
            b.reset();
            newInstance.addBehaviourToAgent(b);
        }
        newInstance.setNickname(this.nickname);
    }
    return newInstance;
}
```

Figura 4.17 Métodos shutDownAgent de EJBAgent/SpringAgent

La nueva instancia la obtenemos a través del método *getNewInstance()* declarado en *AgentInterface*. Este método es abstracto y es necesario que cada clase de agente creado lo implemente devolviendo la instancia concreta. Hemos intentado utilizar reflexión para determinar la clase del agente en tiempo de ejecución y devolver una instancia, pero como implementamos una interfaz a la vez que heredamos, la forma en la que está implementada la reflexión no garantiza que los resultados sean los esperados, por lo que hemos elegido esta solución.

```
public abstract AgentInterface getNewInstance();
```

Figura 4.18 Métodos abstracto getNewInstance() de EJBAgent/SpringAgent

A continuación explicaremos el proceso de añadir, reiniciar y eliminar comportamientos del agente. De forma general, los comportamientos se declaraban como clases internas o instancias anónimas y se añadían al agente en el método *setup()* de la clase *Agent.* Sin embargo, al haber desacoplado totalmente los agentes de los comportamientos e inyectarse por separado, tenemos que obligar a que se vinculen mediante el método *addBehaviourToAgent.* Estos comportamientos no tienen que heredar de ninguna clase en concreto, son comportamientos originales que funcionarían con el uso de JADE en entorno de escritorio (Java SE).

A través de este método añadimos el comportamiento a la lista de comportamientos del agente, no al agente directamente, ya que esto ocurre en el *setup* que hemos implementado. Hacemos comprobaciones de que la instancia no sea *null* y que el comportamiento no haya sido añadido previamente. También vinculamos de forma anticipada el agente al comportamiento. De forma anticipada porque ya se hace de manera automática una vez que el comportamiento es añadido al agente (no a la lista de comportamientos), pero de cara a poder suministrar este dato en la API que explicaremos más adelante. Si el agente estaba ejecutándose este comportamiento no pasará a la cola para ejecutarse hasta el siguiente reinicio del agente. Para que empiece a ejecutarse debemos invocar el método *addBehaviourToAgentAndInit* que hace una sobrecarga con el método anterior, añadiendo el comportamiento a la ejecución actual si el agente se encuentra activo.

```
@Override
public boolean addBehaviourToAgent(Behaviour behaviour){
    boolean isBehaviourAdded = false;
    if (behaviour != null && !checkBehaviourExists(behaviour)) {
        behaviourList.add(behaviour);
        behaviour.setAgent(this);
        isBehaviourAdded = true;
    }
    return isBehaviourAdded;
}

@Override
public boolean addBehaviourToAgentAndInit(Behaviour behaviour){
    boolean isBehaviourAdded = this.addBehaviourToAgent(behaviour);
    if (isBehaviourAdded && isInitiated()){
        addBehaviour(behaviour);
    }
    return isBehaviourAdded;
}
```

Figura 4.19 Métodos para añadir un comportamiento a un EJBAgent/SpringAgent

El método *checkBehaviourExists* devuelve si un comportamiento con el mismo nombre (por defecto es el nombre de la clase del comportamiento) existe ya en la lista de comportamientos, no comprueba si es la misma instancia exacta. La implementación es una búsqueda trivial en una lista.

```java
private boolean checkBehaviourExists(Behaviour behaviour) {
    boolean behaviourExists = false;
    if (behaviour != null) {
        behaviourExists = checkBehaviourExistsByName(behaviour.getBehaviourName());
    }
    return behaviourExists;
}


private boolean checkBehaviourExistsByName(String behaviourName){
    boolean behaviourExists = false;
    int index = 0;
    String behaviourNameInList;

    if (behaviourName != null){
        while (!behaviourExists && index < behaviourList.size()){
            behaviourNameInList = behaviourList.get(index).getBehaviourName();
            if (behaviourNameInList != null && behaviourName.equals(behaviourNameInList)){
                behaviourExists = true;
            }
            index++;
        }
    }
    return behaviourExists;
}
```

Figura 4.20 Métodos para comprobar si un comportamiento ya existe en la lista

Con el fin de reiniciar un comportamiento solo tenemos que invocar al método *reset*() de la clase *Behaviour*. Cada comportamiento concreto debe sobrescribir este método con el fin de liberar y reiniciar las variables concretas que utiliza en la lógica de la acción de su comportamiento. No obstante, *OneShotBehaviour* es una excepción, ya que siempre que se invoca al método *done()* que te notifica si ha terminado su ejecución te devuelve *true*, lo que implica que una vez invocado su método *action()* nunca más volverá a ejecutarse. Para este caso no queda otra opción que eliminar el comportamiento y volverlo a añadir.

Si no tenemos acceso a la instancia concreta del comportamiento pero sabemos su nombre, podemos utilizar el método *removeBehaviourFromAgentByName.*

```java
@Override
public boolean resetBehaviourFromAgent(Behaviour behaviour){
    boolean isBehaviourReset = false;
    if (behaviour != null && behaviourList.contains(behaviour)){
        behaviour.reset();
        if (behaviour instanceof OneShotBehaviour){
            removeBehaviour(behaviour);
            addBehaviour(behaviour);
        }
        isBehaviourReset = true;
    }
    return isBehaviourReset;
}

@Override
public boolean resetBehaviourFromAgentByName(String behaviourName){
    Behaviour behaviour = getBehaviourByName(behaviourName);
    return resetBehaviourFromAgent(behaviour);
}
```

Figura 4.20 Métodos para reiniciar un comportamiento

El segundo método, al igual que otros dos que explicaremos a continuación, hace uso del método privado *getBehaviourByName*, que nos permite obtener el comportamiento concreto de la lista de comportamientos usando su nombre.

```java
private Behaviour getBehaviourByName(String behaviourName){
    Behaviour behaviourFound = null;
    int index = 0;
    String behaviourNameInList;

    if (behaviourName != null){
        while (behaviourFound == null && index < behaviourList.size()){
            behaviourNameInList = behaviourList.get(index).getBehaviourName();
            if (behaviourNameInList != null && behaviourName.equals(behaviourNameInList)){
                behaviourFound = behaviourList.get(index);
            }
            index++;
        }
    }
    return behaviourFound;
}
```

Figura 4.20 Método que nos permite obtener un comportamiento a partir del nombre

La forma de eliminar un comportamiento es bastante trivial haciendo uso del método *removeBehaviour* de la clase *Agent*. Los únicos matices son los consistentes en diferenciar si utilizamos una instancia concreta del comportamiento o si solo tenemos el nombre, además de ofrecer la posibilidad de eliminar el comportamiento solo para la ejecución actual o para siempre (lo que implica eliminarlo de la lista de comportamientos).

```
@Override
public boolean removeBehaviourFromAgent(Behaviour behaviour){
    boolean isBehaviourRemoved = false;
    if(behaviour != null && behaviourList.contains(behaviour)){
        this.removeBehaviour(behaviour);
        isBehaviourRemoved = true;
    }
    return isBehaviourRemoved;
}

@Override
public boolean removeBehaviourFromAgent(String behaviourName){
    Behaviour behaviour = getBehaviourByName(behaviourName);
    return removeBehaviourFromAgent(behaviour);
}

@Override
public boolean removeBehaviourFromAgentForever(Behaviour behaviour){
    boolean isBehaviourRemoved = removeBehaviourFromAgent(behaviour);
    if (isBehaviourRemoved){
        behaviourList.remove(behaviour);
    }
    return isBehaviourRemoved;
}

@Override
public boolean removeBehaviourFromAgentForever(String behaviourName){
    Behaviour behaviour = getBehaviourByName(behaviourName);
    return removeBehaviourFromAgentForever(behaviour);
}
```

Figura 4.21 Métodos para eliminar un comportamiento

Finalmente, solo queda obtener la lista de comportamientos del agente, si ha sido iniciado y su *nickname* (ofreciendo también la posibilidad de cambiarlo).

```java
@Override
public List<Behaviour> getBehavioursFromAgent(){
    return behaviourList;
}

@Override
public void setNickname(String nicknameIn){
    nickname = nicknameIn;
}

@Override
public String getNickname(){
    String res = getAgentClassName();
    if (!nickname.equals("")){
        res = nickname;
    }
    return res;
}

public boolean isInitiated (){
    return initiated;
}
```

Figura 4.21 Métodos sobre las variables de clase

4.5 Componente behaviour

En primer lugar, debemos tener en cuenta que, a diferencia de los agentes, que deben implementar la clase *AgentInterface* y, para asegurar la compatibilidad y la implementación genérica de sus métodos, deben extender de *EJBAgent/SpringAgent*, los comportamientos no tienen ningún requisito extra para ser compatibles en un entorno web. Cualquier instancia de comportamiento creada a partir de una clase heredera de *Behaviour* es apta para ser añadida a nuestros agentes a través del método *addBehaviourToAgent* de la interfaz. No obstante, es posible también que haya funcionalidad que no deba ser estática (implementada en una clase) sino que pueda definirse en tiempo de ejecución de forma más flexible. También, en el caso de los EJB nos encontramos con que podríamos estar accediendo a una interfaz remota y no tuviésemos acceso a esta capa para crear nuevos comportamientos (podríamos crear estos comportamientos en la capa *web* sin problemas, puede haber múltiples justificaciones para esto, pero de forma

general rompería con la estructura y diseño en capas creado con el fin de una mayor cohesión y menor acoplamiento).

En resumen, hablamos del uso de clases anónimas que hereden de la clase *Behaviour* para implementar los métodos comunes a todos ellos: *action()* y *done()*.

El método *action* recoge la funcionalidad del comportamiento, el código a ejecutar, mientras que *done* determina si el comportamiento ha finalizado y puede ser retirado de la cola de comportamientos activos *(scheduling task)* ya que no volverá a ser llamado. El contenedor de JADE invoca siempre el método *action* la primera vez, para posteriormente invocar el método *done* y verificar si ya ha terminado su ejecución. Esto determina que los requisitos mínimos que necesitamos son el código para *action* y una variable compartida que pueda ser modificada dentro del código de acción para que cuando se vuelva a invocar el método *done* este devuelva el estado actualizado.

Spring tiene diversas implementaciones de clases con distinto nivel de complejidad en el paquete *jade.core.behaviours.* La clase de más alto nivel es *Behaviour* que abstrae el funcionamiento general y de las que heredan clases como *CompositeBehaviour*, *LoaderBehaviour* o *SimpleBehaviour*. A su vez, *ParallelBehaviour* y *SerialBehaviour* heredan de *CompositeBehaviour y FMSBehaviour* de *SerialBehaviour.* Esta estructura implica que el funcionamiento común se abstrae en las superclases dejando en última instancia únicamente las características particulares, además de que el desarrollo de un nuevo comportamiento puede simplificarse mucho si te adaptas a alguno de estos patrones comunes. Nosotros nos hemos centrado en la clase *SimpleBehaviour* de la que heredan *OneShotBehaviour, CyclicBehaviour, TickerBehaviour* y *WakerBehaviour.*

Volviendo a la línea central, estos últimos comportamientos mencionados son los mejores candidatos para que, como hemos mencionado, sin tener que crear una nueva clase podamos generar distintos comportamientos.

BehaviourQualifier

Esta anotación solo está disponible para el framework adaptado a EJB. Hacemos uso de la librería CDI para crear nuestras propias anotaciones y poder inyectar nuestros agentes en la capa web.

El funcionamiento de eta anotación es idéntico al explicado en el apartado de *AgentQualifier*, la única diferencia es el nombre de la anotación y que será utilizada únicamente por los comportamientos y no los agentes en el ecosistema EJB, sin tener validez en Spring.

```
@Qualifier
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface BehaviourQualifier {
    String value() default "";
}
```

Figura 4.22 Anotación BehaviourQualifier para EJB

SimpleBehaviourFactory

Esta clase factoría es la que posibilita la creación de esta serie de comportamientos de manera dinámica. Permitimos tanto la creación de instancias de las subclases como de un nuevo *SimpleBehaviour.* Mencionar también que la complejidad que pueden alcanzar los comportamientos implementados de esta manera es limitada. Al igual que todas las clases de la capa EJB/servicios inyectamos la instancia mediante *@Service* con el valor de *@Qualifier/@BehaviourQualifier* adecuado.

```
//@Stateless
@Service
@Qualifier("SimpleBehaviourFactory")
public class SimpleBehaviourFactory{
```

Figura 4.23 Anotaciones de la clase SimpleBehaviourFactory

Para implementar la acción utilizamos una interfaz funcional; una interfaz con solo un método abstracto (*Figura 4.24*) que puede ser utilizada junto a expresiones lambda o clases anónimas para especificar el comportamiento (*Figura 4.25).*

```
@FunctionalInterface
public interface ActionInterface {
    void action();
}
```

Figura 4.24 Interfaz funcional para las acciones implementadas en SimpleBehaviourFactory

```
SimpleBehaviourFactory.ActionInterface actionInterfaceLambda =
        () -> System.out.println("injecting behaviour and setting one shot");

SimpleBehaviourFactory.ActionInterface actionInterfaceAnonymous =
        new SimpleBehaviourFactory.ActionInterface() {
            @Override
            public void action() {
                System.out.println("injecting behaviour and setting one shot");
            }
        };
```

Figura 4.25 Implementación de la interfaz funcional con expresiones lambdas y clases anónimas

Una vez implementado el método *action* de nuestra interfaz funcional, solo es necesario invocar el método correspondiente con la mencionada implementación y la instancia concreta de *AgentInterface* a la que quiero añadirle el comportamiento. En la figura 4.26 podemos ver la implementación del método *addOneShotBehaviour*. Este crea una nueva instancia de *FactoryOneShotBehaviour*, añade el comportamiento creado al agente pasado como parámetro y devuelve la instancia generada.

```
public Behaviour addOneShotBehaviour(ActionInterface actionInterface, AgentInterface agentInterface) {
    currentBehaviour = new FactoryOneShotBehaviour(actionInterface);
    agentInterface.addBehaviourToAgent(currentBehaviour);
    return currentBehaviour;
}
```

Figura 4.26 Implementación del método addOneShotBehaviour de SimpleBehaviourFactory

La clase privada *FactoryOneShotBehaviour* hereda de *OneShotBehaviour* e implementa el método *action* utilizando la implementación concreta de *AgentInterface (Figura 4.27*).

```
private class FactoryOneShotBehaviour extends OneShotBehaviour {
    ActionInterface actionInterface;

    FactoryOneShotBehaviour(ActionInterface actionInterfaceIn){
        super();
        actionInterface = actionInterfaceIn;
    }

    @Override
    public void action() {
        actionInterface.action();
    }
```

Figura 4.27 Implementación de la clase privada FactoryOneShotBehaviour

Este funcionamiento descrito es idéntico para el método *addCyclicBehaviour, addTickerBehaviour* y *addWakerBehaviour (Figura 4.28).* En el caso de *addTickerBehaviour* debemos suministrar el periodo de ejecución como parámetro adicional mientras que *addWakerBehaviour* necesita el periodo de tiempo tras el cual va a ejecutarse o la fecha específica en la que va a despertar.

```
public Behaviour addCyclicBehaviour(ActionInterface actionInterface, AgentInterface agentInterface) {
    currentBehaviour = new FactoryCyclicBehaviour(actionInterface);
    agentInterface.addBehaviourToAgent(currentBehaviour);
    return currentBehaviour;
}

public Behaviour addTickerBehaviour(ActionInterface actionInterface, AgentInterface agentInterface, long period) {
    currentBehaviour = new FactoryTickerBehaviour(agentInterface.getAgentInstance(), period, actionInterface);
    agentInterface.addBehaviourToAgent(currentBehaviour);
    return currentBehaviour;
}

public Behaviour addWakerBehaviour(ActionInterface actionInterface, AgentInterface agentInterface, long timeout) {
    currentBehaviour = new FactoryWakerBehaviour(agentInterface.getAgentInstance(), timeout, actionInterface);
    agentInterface.addBehaviourToAgent(currentBehaviour);
    return currentBehaviour;
}

public Behaviour addWakerBehaviour(ActionInterface actionInterface, AgentInterface agentInterface, Date wakeupDate) {
    currentBehaviour = new FactoryWakerBehaviour(agentInterface.getAgentInstance(), wakeupDate, actionInterface);
    agentInterface.addBehaviourToAgent(currentBehaviour);
    return currentBehaviour;
}
```

4.28 Implementación de los métodos addCyclicBehaviour, addTickerBehaviour y addWakerBehaviour de SimpleBehaviourFactory

Las clases privadas *FactoryCyclicBehaviour, FactoryTickerBehaviour* y *FactoryWakerBehaviour* siguen la misma lógica que *FactoryOneShotBehaviour*. Cada una hereda de su correspondiente clase (*CyclicBehaviour*, *TickerBehaviour* y *WakerBehaviour* respectivamente) e implementan el método *action*, *onTick* o *onWake* utilizando la implementación concreta de *AgentInterface (Figuras 4.29, 4.30 y 4.31*).

```
private class FactoryCyclicBehaviour extends CyclicBehaviour{
    ActionInterface actionInterface;

    FactoryCyclicBehaviour(ActionInterface actionInterfaceIn){
        super();
        actionInterface = actionInterfaceIn;
    }

    @Override
    public void action() {
        actionInterface.action();
    }
```

Figura 4.29 Implementación de la clase privada FactoryCyclicBehaviour

```
private class FactoryTickerBehaviour extends TickerBehaviour {
    ActionInterface actionInterface;

    FactoryTickerBehaviour(Agent a, long period, ActionInterface actionInterfaceIn) {
        super(a, period);
        actionInterface = actionInterfaceIn;
    }

    @Override
    protected void onTick() {
        actionInterface.action();
    }
```

Figura 4.30 Implementación de la clase privada FactoryTickerBehaviour

```
private class FactoryWakerBehaviour extends WakerBehaviour {
    ActionInterface actionInterface;

    FactoryWakerBehaviour(Agent a, Date wakeupDate, ActionInterface actionInterfaceIn) {
        super(a, wakeupDate);
        actionInterface = actionInterfaceIn;
    }

    FactoryWakerBehaviour(Agent a, long timeout, ActionInterface actionInterfaceIn) {
        super(a, timeout);
        actionInterface = actionInterfaceIn;
    }

    @Override
    protected void onWake() {
        actionInterface.action();
    }
```

Figura 4.31 Implementación de la clase privada FactoryWakerBehaviour

Este proceso de clasificación y abstracción corresponde a su vez con el *Principio de Sustitución de Liskov*, (la *L* de los principios *SOLID*), que indica que todas las clases derivadas deben poder tratarse como la clase base.

Finalmente ofrecemos la posibilidad de crear una instancia de *SimpleBehaviour* en la que se especifica tanto la implementación del método *action* como la implementación del método *done*. No obstante, estamos ante un caso bastante más complejo en el que el código ejecutado en el método *action* debe afectar al resultado del código que hemos implementado para el método *done*. El enfoque empleado para resolver este problema es el de no utilizar nuestra interfaz funcional *ActionInterface* sino la interfaz funcional *Consumer<T>* (acepta un parámetro de entrada y no produce ningún resultado) para el método *action* y la interfaz funcional *Function<T, R>* (acepta un parámetro de entrada y produce un resultado) para el método *done*. El parámetro de entrada que compartirán ambas interfaces funcionales será una variable compartida que implemente nuestra interfaz *SharedVariableInterface* (*Figura 4.32*).

Inciso: SharedVariableInterface, SharedVariable y SharedVariableInteger

Esta interfaz define los métodos *isFinished* y *setFinished* que permiten que desde el método *action* se determine si la ejecución del comportamiento ya ha terminado y desde el método *done* comprobarlo.

```
public interface SharedVariableInterface {

    boolean isFinished();

    void setFinished(boolean finished);

}
```

Figura 4.32 Interfaz SharedVariableInterface

Dado el funcionamiento general de los comportamientos, hemos desarrollado dos clases que implementan esta interfaz: *SharedVariable* y *SharedVariableInteger.*

SharedVariable simplemente hace uso de una variable booleana y su cambio de valor para determinar si un comportamiento ha terminado (*Figura 4.33)*. Es la forma más usada cuando un comportamiento solo puede darse por finalizado cuando se haya cumplido una acción; puede darse situaciones en la que no se den las condiciones adecuadas para terminar la acción (fallo de conexión a una API, que aún no se haya registrado el servicio del agente que se quiere consumir en las páginas amarillas, por ejemplo) y debe volver a intentarlo o que se necesite repetir la ejecución un número indefinido de iteraciones, no adaptándose ni al modelo *OneShotBehaviour* ni *CyclicBehaviour*.

```
public class SharedVariable implements SharedVariableInterface {

    private boolean finished;

    public static SharedVariable getSharedVariableInstance(){
        return new SharedVariable();
    }

    @Override
    public boolean isFinished() {
        return finished;
    }

    @Override
    public void setFinished(boolean finished) {
        this.finished = finished;
    }
}
```

Figura 4.33 Clase SharedVariable

La otra implementación es *SharedVariableInteger (Figura 4.34)*, con la que intentamos cubrir la necesidad de un comportamiento que albergue distintos estados en el que no todos sean válidos como condición de finalización. Enumeramos los estados como valores enteros y el desarrollador determina el valor del estado considerado de terminación. A medida que se ejecuta el método *action* se modifica iterativamente el valor del estado actual, afectando al trozo de código que se ejecutará en la siguiente activación y al resultado devuelto en el método *done* que internamente debe invocar el método *isFinished* de esta clase. En este caso el método *setFinished* es útil si puedes alcanzar el estado de terminación desde distintos nodos, además de poder reiniciar el valor de la variable compartida con previsión de reiniciar el comportamiento completo.

```java
public class SharedVariableInteger implements SharedVariableInterface {
    private static final int MIN_VALUE = 0;

    private int currentValue; //current value used to choose the current state of the behaviour
    private int valueToFinish; //value to check if last state of behaviour has been reached

    public static SharedVariableInteger getSharedVariableInstance(int valueToFinish){
        if(valueToFinish < MIN_VALUE){
            valueToFinish = MIN_VALUE;
        }
        return new SharedVariableInteger(valueToFinish);
    }

    private SharedVariableInteger(int valueToFinishIn){
        currentValue = MIN_VALUE;
        valueToFinish = valueToFinishIn;
    }

    @Override
    public boolean isFinished() {
        return currentValue == valueToFinish;
    }

    @Override
    public void setFinished(boolean finished) {
        if (finished){
            currentValue = valueToFinish;
        }else{
            currentValue = MIN_VALUE;
        }
    }

    public void setCurrentValue(int currentValueIn) {
        currentValue = currentValueIn;
    }

    public int getCurrentValue() { return currentValue; }

}
```

Figura 4.34 Clase SharedVariableInteger

Como podemos observar en la *Figura 4.35* el método *addSimpleBehaviour* es análogo al resto. Solo varían los parámetros de entrada mencionados anteriormente. No obstante, una diferencia evidente es el uso de tipos genéricos: para evitar la duplicidad de código solo exigimos que se utilice una instancia que herede nuestra interfaz *SharedVariableInterface*. El uso de la interfaz en vez de una implementación concreta como parámetro de *Consumer* (*Consumer<SharedVariableInterface>* en vez de *Consumer<SharedVariable>* por ejemplo), *Function* y la variable compartida ya nos permite usar cualquiera de nuestras implementaciones con el mismo método.

Sin embargo, esto no es suficiente ya que podríamos suministrar distintas implementaciones para cada parámetro del método y el comportamiento no sería el esperado. Este es el motivo por el que hacemos uso de los tipos genéricos, para exigir la misma implementación de la interfaz para todos los parámetros.

```
public <T extends SharedVariableInterface> Behaviour addSimpleBehaviour(Consumer<T> actionIn,
                                    Function<T, Boolean> doneIn, T sharedVariable,
                                    AgentInterface agentInterface) {

    currentBehaviour = new FactorySimpleBehaviour<T>(actionIn, doneIn, sharedVariable);
    agentInterface.addBehaviourToAgent(currentBehaviour);
    return currentBehaviour;
}
```

Figura 4.35 Método addSimpleBehaviour de SimpleBehaviourFactory

Con este método se abre la posibilidad de crear nuevas implementaciones sin tener que modificar ni añadir nuevos métodos, verificando el principio *Open/Closed* de los principios *SOLID* que sostiene que el diseño debe ser abierto para poder extenderse, pero cerrado para poder modificarse.

```
private class FactorySimpleBehaviour<T extends SharedVariableInterface> extends SimpleBehaviour{
    Consumer<T> action;
    Function<T, Boolean> done;
    T sharedVariable;

    FactorySimpleBehaviour(Consumer<T> actionIn, Function<T, Boolean> doneIn,
                           T sharedVariableIn) {
        action = actionIn;
        done = doneIn;
        sharedVariable = sharedVariableIn;
    }

    @Override
    public void action() {
        action.accept(sharedVariable);
    }

    @Override
    public boolean done() {
        return done.apply(sharedVariable);
    }

    @Override
    public void reset() {
        super.reset();
        sharedVariable.setFinished(false);
    }
```

Figura 4.35 Clase privada FactorySimpleBehaviour

La forma de utilizar el método *addNewSimpleBehaviour* es un poco más compleja, por lo que ilustramos un ejemplo (*Figura 4.36)* en el que se determina un comportamiento con distintos estados a modo demostración.

```
int valueToFinish = 4;
SharedVariableInteger sharedVariable = SharedVariableInteger.getSharedVariableInstance(valueToFinish);
Consumer<SharedVariableInteger> action = (x) ->
{
    int step = x.getCurrentValue();
    switch (step){
        case 0:
            System.out.println("ARE YOU READY?");
            step++;
            x.setCurrentValue(step);
            break;
        case 1:
            System.out.println("SURE?");
            step++;
            x.setCurrentValue(step);
            break;
        case 2:
            System.out.println("3,2,1...");
            step++;
            x.setCurrentValue(step);
            break;
        case 3:
            System.out.println("GO GO GO!");
            step++;
            x.setCurrentValue(step);
            break;
    }

};

Function<SharedVariableInteger, Boolean> done = SharedVariableInteger::isFinished;
Behaviour simpleSharedVariableBehaviour =
        simpleBehaviourFactory.addSimpleBehaviour(action, done, sharedVariable, secondHelloAgent);
```

Figura 4.36 Ejemplo de comportamiento creado con SimpleBehaviourFactory

BehaviourWithFactoryInterface

Por último, tenemos la interfaz *BehaviourWithFactoryInterface (Figura 4.37)* que permite implementar comportamientos que devuelvan nuevas instancias del mismo tipo que podrán ser vinculadas a otros agentes. Es útil para las clases creadas a través de *SimpleBehaviourFactory* pero sobre todo para comportamientos que ya tienen una configuración determinada, son inyectados y que queremos replicar de manera automática a través de la *API* que hemos implementado. Hay

comportamientos que en el momento de crearse necesitan la referencia a un agente concreto (como *TickerBehaviour*) por lo que hemos definido dos métodos.

```
public interface BehaviourWithFactoryInterface {

    Behaviour getInstance ();

    Behaviour getInstance (Agent agent);
```

Figura 4.37 Interfaz BehaviourWithAgentInterface

4.6 Ejemplo de comportamiento y agente

En la *Figura 4.38* puede apreciarse un ejemplo de agente que implementa nuestra interfaz *AgentInterface*. La única diferencia con un agente que herede de la clase *Agent* de forma directa es la llamada a *super* en los métodos *setup* y *takeDown* y la implementación del método *getNewInstance*. La capa de abstracción se encuentra implementada en *EJBAgent/SpringAgent* como hemos mencionado anteriormente

```
@Service
@Qualifier("PlainAgent")
public class PlainAgent extends SpringAgent implements AgentInterface{

    @Override
    protected void setup(){
        super.setup();
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat( pattern: "dd-MM-yyyy-hh:m:ss");
        System.out.println (simpleDateFormat.format(new Date()) + ": Plain Agent " +
                this.getLocalName() + " is running! :D  -> Welcome!");
    }

    @Override
    protected void takeDown(){
        super.takeDown();
        System.out.println ("Plain Agent " + this.getLocalName() + " has stopped! D: -> Goodbye!");
    }

    @Override
    public AgentInterface getNewInstance() { return new PlainAgent(); }

}
```

Figura 4.38 Clase PlainAgent

Ya se ha anotado con *@Stateless @AgentQualifier("PlainAgent")* o *@Service @Qualifier("PlainAgent")* el resultado de la inyección es el mismo, una instancia del agente totalmente disponible para su uso en la capa *web* o en otra capa *Component* (*Figura 4.39).*

```
@Autowired
@Qualifier("PlainAgent")
private AgentInterface plainAgent;
```

Figura 4.39 Inyección de una instancia del agente PlainAgent

Por otro lado, en la *Figura 4.40* puede apreciarse un ejemplo de comportamiento estándar de la librería JADE. Implementamos la interfaz *BehaviourWithFactoryInterface* para que la instancia inyectada sea replicable y no tengamos que inyectar un número indefinido de ellas. Omitiendo este detalle, esto significa que podríamos migrar cualquier tipo de comportamiento simplemente añadiendo las anotaciones necesarias para que sea inyectable y vincularlo a un agente concreto a posteriori (*Figura 4.41).*

```
@Service
@Qualifier("SimplePrintMessageBehaviour")
public class SimplePrintMessageBehaviour extends OneShotBehaviour implements BehaviourWithFactoryInterface {

    @Override
    public Behaviour getInstance() {
        return new SimplePrintMessageBehaviour();
    }

    @Override
    public Behaviour getInstance(Agent agent) {
        return getInstance();
    }

    @Override
    public void action() {
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat( pattern: "dd-MM-yyyy-hh:m:ss");
        System.out.println(simpleDateFormat.format(new Date())
                + " SimplePrintMessageBehaviour is running. This is the only message it will display :D");
    }
```

Figura 4.40 Clase SimplePrintMessageBehaviour

```
@Autowired
@Qualifier("SimplePrintMessageBehaviour")
SimplePrintMessageBehaviour simplePrintMessageBehaviour;
```

Figura 4.41 Inyección de una instancia del comportamiento SimplePrintMessageBehaviour

Con ambas instancias inyectadas ya podemos trabajar de forma directa e intuitiva utilizando los métodos declarado en *AgentInterface*. En la *Figura 4.42* vemos el proceso de vincular comportamientos inyectados a un agente inyectado, iniciar el agente, eliminar un comportamiento y apagar el agente, recuperando la nueva instancia devuelta (requisito explicado anteriormente).

```
@PostConstruct
public void startup(){
    plainAgent.setNickname("plainAgentTFG");
    plainAgent.addBehaviourToAgent(simplePrintMessageBehaviour);
    plainAgent.addBehaviourToAgent(simpleCyclicBehaviour);
    plainAgent.init();
    plainAgent.removeBehaviourFromAgent(simpleCyclicBehaviour);
    plainAgent = plainAgent.shutDownAgent();
```

Figura 4.42 Ejemplo de uso de agentes y comportamientos inyectados con nuestro framework

5. IMPLEMENTACION DE LA API REST

Nuestra API REST es el *backend* de nuestra consola web. Tal y como especificamos en la fase de diseño, el controlador estará situado en la capa de presentación mientras que la plataforma JADE se encontrará en la capa de servicios/*service* (*Figura 5.1)*. Los agentes y comportamientos creados en esta capa se inyectarán en la capa controlador desde el que los gestionaremos. Como proporcionan un servicio y no son modelo de la lógica de negocio de la aplicación (la aplicación en definitiva es la API REST), no los hemos situado en la capa del modelo.

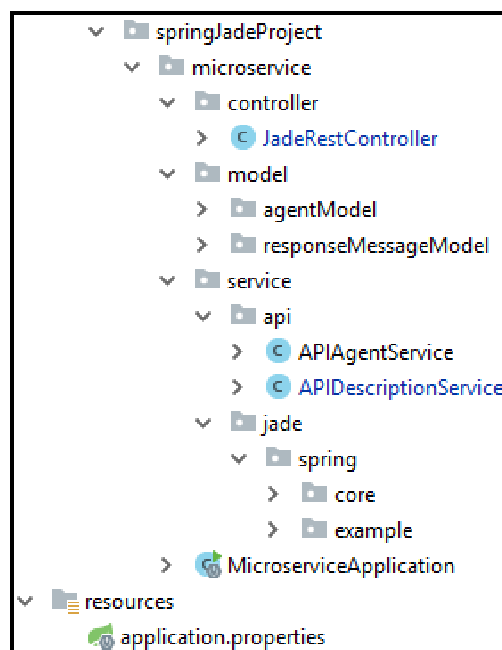

Figura 5.1 Estructura de la API REST en Spring

5.1 Capas de la API REST

Capa	Componente	Nombre ficheros	Descripción
Presentación	Controller	**Clases:** JadeRestController	Escucha las peticiones, gestiona la lista de agentes y comportamientos y devuelve resultados en formato JSON
Lógica de Negocio	Service	**Clases:** Las del framework jade.spring, APIAgentService y APIDescriptionService	Servicios para la gestión de la plataforma JADE y descripción de la API
Lógica de Negocio	Model	**Clases:** JsonAgentBehaviourModel, JsonBehavioursModel, JsonStateAgentModel, APIActionDescription, ResponseErrorMessage, ResponseNotificationMessage **Interfaces:** ResponseMessageInterface	Modelos utilizados por el controlador para recoger e interpretar el texto JSON y devolver respuestas en formato JSON adecuadas.

5.2 MicroserviceApplication

Esta es la clase que anotada con *@SpringBootApplication* posibilita el despliegue de la aplicación en un contenedor servlet (servidor web) empotrado. No necesitamos especificar ningún parámetro extra y por defecto utilizará el contenedor servlet Tomcat en el puerto 8080. Nosotros hemos cambiado el puerto por defecto en el fichero *application.properties (Figura 5.2)* por el puerto 9090.

```
server.port = 9090
```

Figura 5.2 Contenido del fichero application.properties

5.3 Dependencias

Gestionamos las dependencias del proyecto con Maven. Spring, a través de la página https://start.spring.io/, permite que configures el tipo de proyecto que deseas iniciar y te genera una plantilla básica para que puedas empezar a desarrollar sin preocuparte por los detalles de configuración iniciales. En la versión simple puedes determinar el nombre del grupo, del artefacto, si quieres crear un proyecto *maven* o *gradle*, si quieres utilizar *Java, Kotlin* o *Groovy*, la versión de *Spring Boot* y las dependencias respectos a los módulos que soporta Spring (*Figura 5.3)*. La versión completa permite determinar de manera más granular tus necesidades, pero para la mayoría de proyectos es suficiente con la configuración básica.

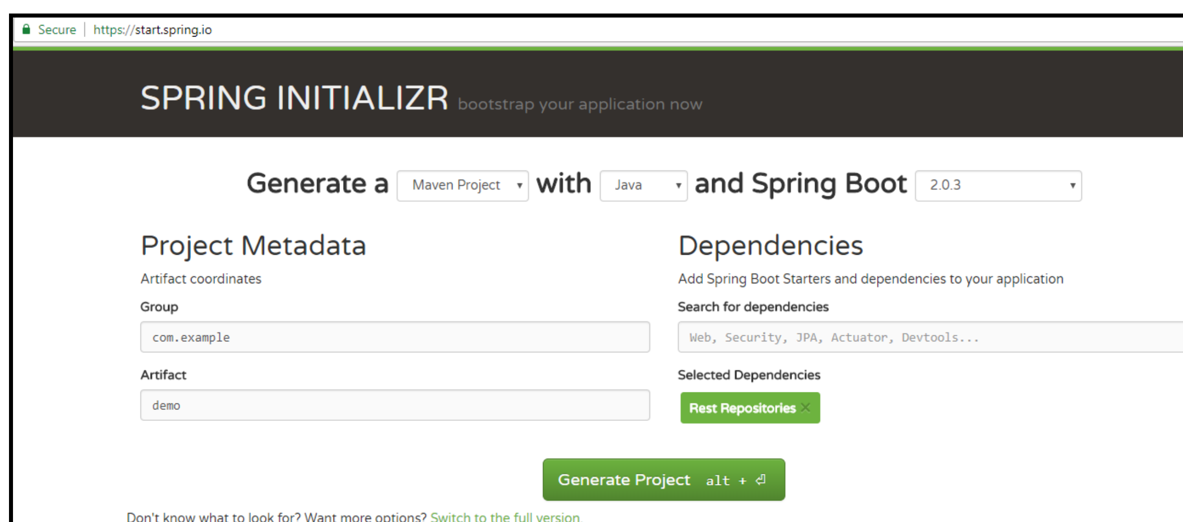

Figura 5.3 Creación de Repositorio REST utilizando Spring Initializr

El fichero *pom (maven)* generado contiene las dependencias necesarias para desarrollar un Repositorio REST con Spring Boot en Java. Además, incluye el *plugin* que te permite desplegar y ejecutar la aplicación web del mismo modo que se ejecuta una aplicación de escritorio (*Figura 5.4).*

```
▼<dependencies>
  ▼<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-data-rest</artifactId>
   </dependency>
  ▼<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-test</artifactId>
     <scope>test</scope>
   </dependency>
 </dependencies>
▼<build>
  ▼<plugins>
    ▼<plugin>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-maven-plugin</artifactId>
     </plugin>
   </plugins>
 </build>
```

Figura 5.4 Dependencias y plugin para un Repositorio REST con Spring Boot

Ahora solo nos falta incorporar la librería JADE. Basta con ir al repositorio central de maven (https://mvnrepository.com/), buscar la última versión de JADE y copiar la dependencia en nuestro fichero pom (*Figura 5.5).*

```
<dependency>
    <groupId>com.tilab.jade</groupId>
    <artifactId>jade</artifactId>
    <version>4.5.0</version>
</dependency>
```

Figura 5.5 Dependencia de la librería JADE para maven

5.4 API REST

REST significa *REpresentational State Transfer* (Transferencia de Estado Representacional), y es un tipo de arquitectura de desarrollo web que se adhiere al estándar HTTP. Se utiliza para crear servicios y aplicaciones que puedan ser usadas por cualquier cliente que entienda HTTP, simplificando el desarrollo respecto a otras alternativas como SOAP y XML-RPC. Es decir, nuestra API ofrece un servicio que está construido siguiendo la arquitectura REST.

- Nuestra API nunca guarda el estado (la sesión) en el servidor, todas las peticiones se encuentran aisladas y son independientes. La información necesaria para mostrar la información que se solicita siempre se envía con la petición. Como consecuencia, tenemos un sistema más escalable.
- Los nombres de las URIs no implican acción (no utilizamos verbos) y son únicos.
- Nuestras URIs son independientes de todo tipo de formato (como *.jpg* o *.pdf*). Usamos el header *Accept* para especificar el uso de formato JSON.
- Mantenemos una jerarquía y estructura lógica en el diseño de las URIs. Por ejemplo, la ruta */api/agents* obtendrá información de todos los agentes mientras que */api/agent/{localName}* obtendrá la información de un agente concreto. Del mismo modo para referirnos a todos los comportamientos de un agente los comportamientos de un agente utilizamos */api/agent/behaviours* y no */api/behaviours/agent*.
- Uso de consultas sobre la URI utilizando parámetros HTTP en vez de incluirlos en la propia URI.
- Uso de distintos verbos HTTP para distintas acciones:
    - GET: para consultar información
    - POST: para modificar/editar recursos (iniciar un agente, detenerlo o añadirle un nuevo comportamiento, por ejemplo)
    - PUT: para añadir recursos (crear un nuevo agente)
    - DELETE: para eliminar recursos (eliminar un agente)
- Uso de códigos de estado (*200 – OK, 400 – Bad Request, 500 – Internal Server Error, 409 – Conflict* y *422 – Unproccessable Entity).*

5.5 Formato JSON

Hemos elegido el formato de datos JSON como medio de comunicación con nuestra API REST. Es un formato ligero, totalmente independiente del lenguaje y fácil de leer y escribir a la vez que de interpretar y generar. Utiliza convenciones de lenguajes similares a Javascript, Java, Python y C# entre otros lenguajes de programación. Además, existen multitud de librerías que facilitan el tratamiento de este tipo de datos, aliviando la carga de trabajo del desarrollador. De forma concreta, Spring utiliza la librería ***Jackson***, la librería estándar de código abierto para la plataforma de la máquina virtual de Java (*Java Virtual Machine Platform*). De forma general podemos definirlo como un conjunto de herramientas de procesamiento de datos que permite generar texto JSON a partir de POJOs (*Plain Old Java Objects)* y viceversa, aunque posee multitud de herramientas extras para casos más concretos.

Con todo esto, cuando utilizamos un controlador REST (*@RestController*), Spring gestiona de manera transparente el uso de la librería *Jackson* para transformar el objeto o la lista de objetos devueltos en texto JSON, lo que significa que prácticamente no tenemos que preocuparnos de gestionar esta funcionalidad a bajo nivel.

5.6 Capa Model: agentModel y responseMessageModel

Como hemos comentado en el punto anterior, no tenemos que preocuparnos traducir a texto JSON ni los agentes ni los comportamientos que devolvamos como respuesta a una petición (jackson convierte en texto JSON todos los métodos *get* del objeto a no ser que se encuentren anotado con *@JsonIgnore).* No obstante, sí tenemos que gestionar el texto JSON que nos envían en las peticiones que no son *GET*, las respuestas tanto de carácter informativo como de error y respuestas que necesitamos formatear de un modo más concreto como, por ejemplo, la lista de agentes con todos sus comportamientos vinculados: en esta situación no queremos toda la información referente al agente, sino su identificador (*nickname*) asociado a los comportamientos. Como jackson trabaja con POJOs, solo necesitamos crear modelos de clase con los valores que necesitamos (*Figura 5.6)*

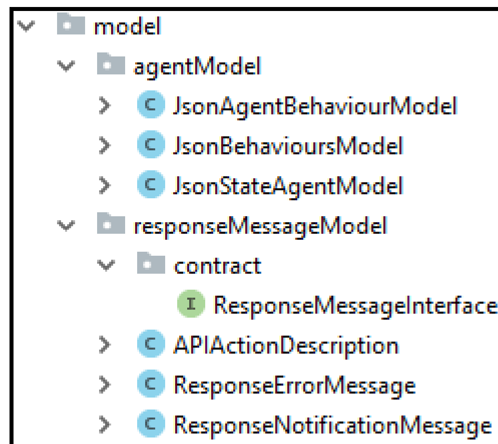

Figura 5.6 Modelos POJOs que utilizamos con Jackson en la API REST

La clase más importante es *JsonAgentBehaviourModel* que permite crear, eliminar, iniciar, detener y reiniciar agentes, además de añadir, eliminar y reiniciar comportamientos (*Figura 5.7)*.

Con la anotación *@JsonIgnoreProperties(ignoreUnknown = true)* conseguimos que si la petición recibida no tiene todos los campos simplemente convierta los campos encontrados sin lanzar ningún tipo de error. Usando *@RestController* es el comportamiento por defecto, por lo que no necesitamos que nuestros modelos lo especifiquen.

```java
public class JsonAgentBehaviourModel {
    public static final String INIT = "init";
    public static final String STOP = "stop";
    public static final String RESTART = "restart";
    public static final String ADD = "add";
    public static final String RESET = "reset";
    public static final String REMOVE = "remove";

    private String className;
    private String agentName;
    private String behaviourName;
    private String action;
    //added to start the new behaviour instantly or wait until next restart
    private boolean startNow;
    //added to remove the behaviour from the current runtime
    //(next restart it will be attached again) or forever
    private boolean forever;

    public String getClassName() { return className; }

    public void setClassName(String className) { this.className = className; }

    public String getAgentName() { return agentName; }

    public void setAgentName(String agentName) { this.agentName = agentName; }

    public String getBehaviourName() { return behaviourName; }

    public void setBehaviourName(String behaviourName) { this.behaviourName = behaviourName; }

    public String getAction() { return action; }

    public void setAction(String action) { this.action = action; }

    public boolean getStartNow() { return startNow; }

    public void setStartNow(boolean startNow) { this.startNow = startNow; }

    public boolean getForever() { return forever; }

    public void setForever(boolean forever) { this.forever = forever; }
```

Figura 5.7 Clase JsonAgentBehaviourModel

Otros modelos implementados son *JsonBehavioursModel* (*Figura 5.8)* que devuelve una lista de con el identificador de cada agente junto a lista de sus comportamientos vinculados, *JsonStateAgentModel (Figura 5.9)* que devuelve el estado del agente, *ResponseNotificationMessage (Figura 5.10)* para informar al cliente del resultado de la petición y *ResponseErrorMessage (Figura 5.11)* para notificar que ha habido algún error.

```
public class JsonBehavioursModel {
    private String agentName;
    private List<Behaviour> behaviourList;

    public JsonBehavioursModel(String agentName, List<Behaviour> behaviourList){
        this.agentName = agentName;
        this.behaviourList = behaviourList;
    }

    public JsonBehavioursModel(){
    }
```

Figura 5.8 Clase JsonBehavioursModel

```
public class JsonStateAgentModel {
    private String name;
    private int code;

    public JsonStateAgentModel(String name, int value) {
        this.name = name;
        this.code = value;
    }

    public JsonStateAgentModel() {
    }
```

Figura 5.9 Clase JsonStateAgentModel

```
public class ResponseNotificationMessage implements ResponseMessageInterface {
    private String message;

    public ResponseNotificationMessage(String message) { this.message = message; }

    public String getMessage() { return message; }

    public void setMessage(String message) { this.message = message; }
}
```

Figura 5.10 Clase ResponseNotificiationMessage

```
public class ResponseErrorMessage implements ResponseMessageInterface{
    private String error;

    public ResponseErrorMessage(String error) { this.error = error; }

    public String getError() { return error; }

    public void setError(String error) { this.error = error; }
}
```

Figura 5.11 Clase ResponseErrorMessage

El último modelo implementado es la clase *APIActionDescription*, que utilizamos para informar a cualquier cliente de la funcionalidad de la API (*Figura 5.12)*. Nos permite especificar la ruta de la funcionalidad, el método/verbo que acepta, la descripción de la funcionalidad, los parámetros que solicita y el tipo de resultado devuelto.

```
public class APIActionDescription {
    private String path;
    private String method;
    private String description;
    private String parameters;
    private String result;
```

Figura 5.12 Clase APIActionDescription

5.7 Capa Service: APIAgentService

Esta clase funciona de nexo de unión entre nuestro controlador y el framework que gestiona la plataforma JADE, que se encuentra en la misma capa *service*.

Al igual que nuestros agentes y comportamientos, estamos ante un @*Component* (Bean de Spring) de tipo *@Service.* En él solo inyectamos otro componente: la instancia de la clase *AgentsManager* que permita obtener información actualizada de los contenedores JADE (*Figura 5.13)*.

```
@Service
public class APIAgentService {

    @Autowired
    private AgentsManager agentsManager;

    public APIAgentService() {
    }
```

Figura 5.13 Declaración de la clase APIAgentService

Este servicio es útil ante numerosos casos, pero tiene el inconveniente de no poder obtener datos de agentes que no estén ejecutándose. Por otro lado, puede obtener información tanto de los agentes gestionados por la API como aquellos que no lo están, o que por algún motivo no están disponibles para quien realiza la consulta.

El fin del servicio es ser consumido por el controlador, que a través de los métodos implementados podrá obtener información de los agentes que están ejecutándose y de sus comportamientos. De forma genérica podemos obtener la lista de agentes activos, tanto las instancias como los nombres de manera aislada, así como los comportamientos de un agente concreto dado su nombre único (*localName/nickname)* filtrados por estado de ejecución (todos los estados, *ready, blocked* o *running*) (*Figura 5.14).* Es importante conocer qué agentes están ejecutándose en el contenedor para un correcto funcionamiento de las funciones de inicio, parada y reinicio de los agentes desde la API.

No necesitamos utilizar la instancia de *AgentsManager* para ningún caso más: la inicialización y finalización de agentes se encuentra dentro de los propios agentes que heredan de *SpringAgent*, como hemos explicado en el apartado 4.

```
public List<Agent> findActiveAgents(){
    Collection<Agent> agentCollection = agentsManager.getAgentsOnContainer().values();
    Agent AMSAgent = agentsManager.getAgentsOnContainer().get("AMSAgent");
    agentCollection.remove(AMSAgent);
    return new ArrayList<>(agentCollection);
}

public List<String> findActiveAgentsName(){
    return new ArrayList<>(agentsManager.getAgentsOnContainer().keySet());
}

public Agent findActiveAgentByLocalName(String localName){
    return agentsManager.getAgentsOnContainer().get(localName);
}

public List<Behaviour> findBehavioursFromAgentByLocalName(String localName){
    return this.getBehavioursFromAgentWithAgentNullAndStateByLocalName(localName,  state: null);
}

public List<String> findBehavioursNameFromAgentByLocalName(String localName){
    return this.getBehavioursNameFromAgentWithStateByLocalName(localName,  state: null);
}

public List<Behaviour> findReadyBehavioursFromAgentByLocalName(String localName){
    return this.getBehavioursFromAgentWithAgentNullAndStateByLocalName(localName, Behaviour.STATE_READY);
}

public List<String> findReadyBehavioursNameFromAgentByLocalName(String localName){
    return this.getBehavioursNameFromAgentWithStateByLocalName(localName, Behaviour.STATE_READY);
}

public List<Behaviour> findBlockedBehavioursFromAgentByLocalName(String localName){
    return this.getBehavioursFromAgentWithAgentNullAndStateByLocalName(localName, Behaviour.STATE_BLOCKED);
}

public List<String> findBlockedBehavioursNameFromAgentByLocalName(String localName){
    return this.getBehavioursNameFromAgentWithStateByLocalName(localName, Behaviour.STATE_BLOCKED);
}
```

Figura 5.14 Algunos métodos públicos de la clase APIAgentService

5.8 Capa Service: APIDescriptionService

Este servicio permite obtener una lista con las descripciones de las distintas funcionalidades de la API. Utilizamos la clase *APIActionDescription* como modelo para definir una funcionalidad concreta a través de su ruta, el método/verbo que utiliza, una descripción, los parámetros que necesita y el tipo de resultado devuelto. Utilizamos un fichero de texto *APIDescription.txt* en el que guardamos en un formato similar a CSV (*Comma-Separated Values*) las distintas funcionalidades con los valores correspondientes a cada parámetro en líneas separadas (*Figura 5.15)*.

```
/api/agent;POST;init the specified agent in the Jade Container;{agentName:String, action:'init'};{message:String} or {error:String} with the result of the request
/api/agent;POST;stop the specified agent in the Jade Container;{agentName:String, action:'stop'};{message:String} or {error:String} with the result of the request
/api/agent;POST;restart the specified agent in the Jade Container;{agentName:String, action:'restart'};{message:String} or {error:String} with the result of the request
/api/agent;PUT;create a new agent;{className:String, agentName:String};{message:String} or {error:String} with the result of the request
/api/agent;DELETE;delete an existing agent;{agentName:String};{message:String} or {error:String} with the result of the request
```

Figura 5.15 Extracto del fichero APIDescription.txt con la funcionalidad de la API

La lista de funcionalidades se lee del fichero la primera vez que se solicita almacenándose en una variable de clase, que será devuelta como respuesta a cada nueva petición *(Figura 5.16)*.

```
@Service
public class APIDescriptionService {
    private static final String FILE_NAME = "APIDescription.txt";
    private List<APIActionDescription> actionList;

    public APIDescriptionService() {
    }

    public List<APIActionDescription> getAPIDescription() {
        if (actionList == null){
            actionList = loadAPIDescriptionFromFile(FILE_NAME);
        }
        return actionList;
    }
```

Figura 5.16 Clase APIDescriptionService

La lectura se realiza a través de un *BufferedReader,* que nos permite leer cada línea por separado; luego utilizamos un *StringTokenizer* para obtener cada parámetro de la funcionalidad por separado (siempre se mantiene el mismo orden). Finalmente creamos una nueva instancia de *ApiActionDescription* y la añadimos a la lista que almacenamos en la variable de clase (*Figura 5.17)*.

```
private List<APIActionDescription> loadAPIDescriptionFromFile(String fileName){
    List<APIActionDescription> result = new ArrayList<>();
    BufferedReader bufferedReader;
    String line;
    StringTokenizer stringTokenizer;
    String path, method, description, parameters, res;

    try {
        bufferedReader = new BufferedReader(new FileReader(fileName));
        line = bufferedReader.readLine();
        while (line != null) {
            stringTokenizer = new StringTokenizer(line,  delim: ";");
            path = stringTokenizer.nextToken();
            method = stringTokenizer.nextToken();
            description = stringTokenizer.nextToken();
            parameters = stringTokenizer.nextToken();
            res = stringTokenizer.nextToken();

            result.add(new APIActionDescription(path, method, description, parameters, res));

            line = bufferedReader.readLine();
        }
        bufferedReader.close();
    } catch (IOException e) {
        System.err.println("Caught IOException: " + e.getMessage());
    }

    return result;
}
```

Figura 5.17 Método que lee la lista de funcionalidades de un fichero

5.9 Capa Controller: JadeRestController

JadeRestController es la clase que anotada con *@RestController* escucha las peticiones HTTP de los clientes para cada ruta *(path)* concreta. De manera general, gracias al uso de la anotación *@RequestMapping,* cualquier petición que comience por *"/api"* será tratada por esta clase y, dependiendo de la ruta y los verbos utilizados, las procesará un método u otro. Este es el lugar donde inyectamos los distintos agentes y comportamientos de la capa Servicios *(Figura 5.18).*

```
@RestController
@RequestMapping("/api")
public class JadeRestController {

    @Autowired
    @Qualifier("HelloAgent")
    private AgentInterface helloAgent;

    @Autowired
    @Qualifier("PlainAgent")
    private AgentInterface plainAgent;

    @Autowired
    @Qualifier("SimpleBehaviourFactory")
    private SimpleBehaviourFactory simpleBehaviourFactory;

    @Autowired
    @Qualifier("SendACLMessage")
    SendACLMessageBlockBehaviour sendACLMessageBlockBehaviour;

    @Autowired
    @Qualifier("SimplePrintMessageBehaviour")
    SimplePrintMessageBehaviour simplePrintMessageBehaviour;
```

Figura 5.18 Clase JadeRestController con distintos agentes y comportamientos inyectados

Además de los agentes y comportamientos deseados también tenemos que inyectar los servicios *APIAgentService* y *APIDescriptionService* que permitirán resolver algunas de las peticiones (*Figura 5.19)*

```
@Autowired
private APIAgentService apiAgentService;

@Autowired
private APIDescriptionService apiDescriptionService;
```

Figura 5.19 Inyección de servicios en el controlador

Cada método de la clase sirve para atender la petición a una ruta concreta con un verbo concreto; cuando la misma ruta y verbo coincidan para distintas acciones, el contenido JSON recibido con la petición tendrá un parámetro *action* que determinará la acción específica que se está solicitando (al ser una API REST, como hemos explicado en el apartado correspondiente, las rutas no pueden indicar acciones). Los verbos y rutas concretos se especifican con anotaciones suministradas por Spring: *@GetMapping, @PostMapping, @PutMapping* y *@DeleteMapping.* Estas anotaciones tienen distintos parámetros como *path, headers, consumes o produces,* pero si solo utilizamos un valor éste será el correspondiente al *path*. Por defecto, todas estas anotaciones dentro de *@RequestMapping* determinan que el resultado devuelto tendrá el formato JSON, aunque para consumir los datos de entrada es recomendable especificarlo con *MediaType.APPLICATION_JSON_VALUE.*

En las peticiones *GET* las consultas sobre la URI pueden recogerse de forma muy sencilla en los parámetros del método que gestiona dicha ruta y verbo. En la ruta determinamos el nombre de la variable como último parámetro entre llaves (“{}”) y la variable del método que va a recoger el valor se anota con *@PathVariable* y el valor especificado entre llaves (*Figura 5.20)*

```
@GetMapping("/agent/{localName}")
public Agent getAgentByLocalNameOnContainer(@PathVariable("localName") String localName){
    return apiAgentService.findActiveAgentByLocalName(localName);
}
```

Figura 5.20 Método con @GetMapping que recoge el valor de la consulta en la variable localName

Por otro lado, con el resto de verbos los datos necesarios para resolver la solicitud se envían en el *body* en formato JSON, y tenemos que “traducirlos” a alguno de nuestros modelos. En estos casos solo necesitamos utilizar la anotación *@RequestBody* junto a la variable del método en la que se quiere guardar el valor recibido *(Figura 5.21)*. Este es un proceso automático en el que se utiliza la librería *Jackson* mencionada anteriormente.

```
@DeleteMapping(path="agent", consumes=MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> deleteAgent(@RequestBody JsonAgentBehaviourModel jsonAgentBehaviourModel){
```

Figura 5.21 Método con @DeleteMapping que recoge el cuerpo de la petición en la variable jsonAgentBehaviourModel

La respuesta a las peticiones que no solicitan información es una instancia de la clase *ResponseEntity<T>,* una clase del paquete *org.springframework.http* que permite devolver códigos HTTP concretos con texto JSON. Tiene métodos factoría que facilitan mucho su uso y en los que podemos especificar el *body* de la respuesta de forma directa (*Figura 5.22)*

```
        return ResponseEntity.ok(new ResponseNotificationMessage("Agent " +
                agentName + " from class " + className + " has been created successfully"));
    }else{
        return ResponseEntity.status(409).body(new ResponseNotificationMessage("An agent with" +
                " the name" + jsonAgentBehaviourModel.getAgentName() + " exists already." +
                " You can't create two agents with the same name"));
    }
}else{
    return ResponseEntity.badRequest().body(new ResponseErrorMessage("Agent class "
            + jsonAgentBehaviourModel.getClassName() + " doesn't exist in the system"));
}
```

Figura 5.22 Ejemplo de uso de ResponseEntity en la aplicación

Una vez concretado los detalles de funcionamiento de una API REST con Spring MVC y Spring Boot, vamos a especificar como manipulamos los agentes y comportamientos que pueden ser gestionados a través de la API.

En primer lugar tenemos que controlar los agentes y comportamientos inyectados y disponibles a través de la API desde el mismo controlador, ya que necesitamos tener registrados todos los tipos de agente disponibles, todas las instancias de agente creadas, todos los tipos de comportamiento disponibles y todas las instancias de comportamiento creadas y vinculadas a algún agente concreto.

Es decir, los agentes y comportamientos inyectados, siendo completamente operativos, no los utilizamos nada más que como plantillas para crear los nuevos agentes y comportamientos. La lista de agentes inyectados será la lista de agentes disponibles *availableAgentList* y la lista de comportamientos inyectados será la lista de comportamientos disponibles *availableBehaviourList.* Todos nuestros agentes

son factorías por defecto al poseer en su interfaz el método *getNewInstance,* y todos los comportamientos que implementan *BehaviourWithFactoryInterface* también. Si no implementan esta interfaz no deberían ser accesibles desde la API. Lo único que tendrán en común los agentes creados a través de la factoría con sus modelos son los métodos *setup* y *shutDown,* pero no tendrán la misma lista de comportamientos que tenía el modelo. Las nuevas instancias de agentes se almacenan en *agentList,* mientras que las nuevas instancias de comportamientos se guardan en *behaviourList* (*Figura 5.23).*

```
//agents injected that I can instantiate
private Map<String, AgentInterface> availableAgentList;
//agent instances I have created
private Map<String, AgentInterface> agentList;

//behaviours injected that I can instantiate and attach to agents via API
private Map<String, Behaviour> availableBehaviourList;
//behaviour instances
private List<Behaviour> behaviourList;
```

Figura 5.23 Listas de comportamientos y agentes

Inicializamos la lista de agentes y de comportamientos disponibles en el método anotado con *@PostConstruct*. Esta anotación permite que el método sea invocado de forma automática una vez se haya producido la inyección de dependencias, y así poder llevar a cabo cualquier tipo de inicialización antes de que el bean esté disponible para recibir peticiones (*Figura 5.24).*

```
@PostConstruct
public void startup(){
    availableAgentList = new HashMap<>();
    availableAgentList.put(helloAgent.getAgentClassName(), helloAgent);
    availableAgentList.put(byeAgent.getAgentClassName(), byeAgent);
    availableAgentList.put(receiveMessageAgent.getAgentClassName(), receiveMessageAgent);
    availableAgentList.put(sendMessageAgent.getAgentClassName(), sendMessageAgent);
    availableAgentList.put(plainAgent.getAgentClassName(), plainAgent);

    availableBehaviourList = new HashMap<>();
    availableBehaviourList.put(sendACLMessageBlockBehaviour.getBehaviourName(), sendACLMessageBlockBehaviour);
    availableBehaviourList.put(receiveACLMessageBlockBehaviour.getBehaviourName(), receiveACLMessageBlockBehaviour);
    availableBehaviourList.put(simplePrintMessageBehaviour.getBehaviourName(), simplePrintMessageBehaviour);
    availableBehaviourList.put(simpleCyclicBehaviour.getBehaviourName(), simpleCyclicBehaviour);
```

Figura 5.24 Inicialización de las listas de agentes y comportamientos disponibles

Llegados a este punto nuestro controlador ya se encuentra disponible para recibir peticiones de cualquier cliente. En primer lugar con la ruta base *"/api"* utilizamos el *apiDescriptionService* para mostrar toda la funcionalidad de la API (*Figura 5.25).* Podemos apreciar un extracto del resultado en la *Figura 5.26.*

```
@GetMapping
public List<APIActionDescription> getApiDescription(){
    return apiDescriptionService.getAPIDescription();
}
```

Figura 5.25 Método que devuelve al cliente la funcionalidad de nuestra API en formato JSON


```
{
    "path": "/api/agent",
    "method": "POST",
    "description": "restart the specified agent in the Jade Container",
    "parameters": "{agentName:String, action:'restart'}",
    "result": "{message:String} or {error:String} with the result of the request"
},
{
    "path": "/api/agent",
    "method": "PUT",
    "description": "create a new agent",
    "parameters": "{className:String, agentName:String}",
    "result": "{message:String} or {error:String} with the result of the request"
},
{
    "path": "/api/agent",
    "method": "DELETE",
    "description": "delete an existing agent",
    "parameters": "{agentName:String}",
    "result": "{message:String} or {error:String} with the result of the request"
},
{
    "path": "/api/behaviours",
    "method": "GET",
    "description": "get information of all behaviours instances which are in the Jade Container",
    "parameters": "-",
    "result": "List of behaviours"
},
```


Figura 5.26 Extracto de la funcionalidad de la API

El cliente tiene la posibilidad de obtener información de un agente tanto en el contenedor como en la lista de agentes disponibles. También puede buscar información concreta de los agentes disponibles utilizando el nombre de clase. De forma general *"/api/agent"* hace referencia a agentes en el contenedor, *"/api/agent/available"* a la lista de agentes disponibles y *"/api/agent/created"* a la lista de agentes creados *(Figura 5.27)*

```java
@GetMapping("/agent/{localName}")
public Agent getAgentByLocalNameOnContainer(@PathVariable("localName") String localName){
    return apiAgentService.findActiveAgentByLocalName(localName);
}

@GetMapping("/agent/available/{className}")
public Agent getAvailableAgentByLocalName(@PathVariable("className") String className){
    return availableAgentList.get(className).getAgentInstance();
}

@GetMapping("/agent/created/{localName}")
public Agent getCreatedAgentByLocalName(@PathVariable("localName") String localName){
    return agentList.get(localName).getAgentInstance();
}
```

Figura 5.27 Métodos para obtener información de un agente determinado

Por otro lado, si un agente se encuentra en el contenedor también podemos obtener una descripción del tipo *AMSAgentDescription*. Para ello usamos un agente especial que inyectamos desde la capa servicio y que utilizando la clase *AMSService* de la librería JADE puede obtener está información con las restricciones que querramos especificar.

Aunque podemos obtener la información sobre el estado del agente a partir de la instancia, tenemos otro par de métodos para que sea accesible de manera directa.

Al igual que en el caso de *"/api/agent"* ocurre lo mismo para *"/api/agents"* y la posibilidad de obtener la información de todos los agentes del contenedor o de la lista de agentes que podemos controlar desde la API. Podemos elegir entre obtener las instancias o solo los nombres que utilizan como identificador.

Un método muy interesante es *getBehavioursFromAllAgent* que te permite con solo una petición obtener la lista de los nombres de los distintos agentes existentes con todos los comportamientos que tienen vinculados *(Figura 5.28 y 5.29)*

```java
@GetMapping("/agents/available/behaviours")
public List<JsonBehavioursModel> getBehavioursFromAllAgent(){
    return this.getBehavioursFromAllAgentsInjected();
}
```

Figura 5.28 Método para obtener todos los agentes con sus comportamientos vinculados

```
{
    "agentName": "SecondHelloAgent",
    "behaviourList": [
        {
            "restartCounter": 0,
            "executionState": "READY",
            "agentLocalName": "SecondHelloAgent",
            "behaviourName": "OneShotBehaviour",
            "runnable": true,
            "dataStore": {
                "empty": true
            }
        },
        {
            "restartCounter": 0,
            "executionState": "READY",
            "agentLocalName": "SecondHelloAgent",
            "behaviourName": "ReceiveACLMessageBlockBehaviour",
            "runnable": true,
            "dataStore": {
                "empty": true
            }
        },
```

Figura 5.29 Extracto de la respuesta con el nombre de un agente y sus comportamientos

De forma más concreta, podemos obtener todos los comportamientos (instancias completas o solo el nombre) vinculados a un agente concreto (del contenedor o de la lista de agentes creados). Desde el contenedor, el uso del servicio *APIAgentService* nos permite filtrar los comportamientos por estado de ejecución.

En cuanto a la gestión de los agentes, utilizamos la ruta "/api/*agent*" para crearlos, eliminarlos, iniciarlos, detenerlos y reiniciarlos, la única diferencia es el verbo utilizado y el valor del parámetro *action* enviado en el cuerpo (*body)* de la petición. Estos métodos son los que tienen una mayor lógica de control en el que hay que verificar todas las posibilidades para evitar llevar el sistema a un estado de error.

Podemos apreciar el método *createAgent* en la *Figura 5.30.* De forma general, se obtienen los parámetros del cuerpo de la petición, se verifica el nombre de la clase existe y que el nombre del agente no ha sido asignado previamente (identificador único), se crea una nueva instancia y se añade a la lista de agentes creados.

```
@PutMapping(path="agent", consumes=MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> createAgent(@RequestBody JsonAgentBehaviourModel jsonAgentBehaviourModel){
    String className = jsonAgentBehaviourModel.getClassName();
    String agentName = jsonAgentBehaviourModel.getAgentName();
    AgentInterface agentToCreateInstance;
    AgentInterface newAgent;
    if (checkAgentClassExistsByClassName(className)){
        if (!checkAgentInstanceExistsByName(agentName)){
            agentToCreateInstance = availableAgentList.get(className);
            newAgent = ((SpringAgent) agentToCreateInstance).getNewInstance();
            newAgent.setNickname(agentName);
            agentList.put(agentName, newAgent);
            return ResponseEntity.ok(new ResponseNotificationMessage("Agent " +
                    agentName + " from class " + className + " has been created successfully"));
        }else{
            return ResponseEntity.status(409).body(new ResponseNotificationMessage("An agent with" +
                    " the name" + jsonAgentBehaviourModel.getAgentName() + " exists already." +
                    " You can't create two agents with the same name"));
        }
    }else{
        return ResponseEntity.badRequest().body(new ResponseErrorMessage("Agent class "
                + jsonAgentBehaviourModel.getClassName() + " doesn't exist in the system"));
    }
```

Figura 5.30 Método para crear un nuevo agente a través de la API

La lógica del método *deleteAgent (Figura 5.31)* es análoga a la anterior. Obtenemos los parámetros del cuerpo de la petición, verificamos que el nombre del agente existe, obligamos a que finalice su ejecución si había sido iniciado y lo eliminamos de la lista de agentes creados.

```
@DeleteMapping(path="agent", consumes=MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> deleteAgent(@RequestBody JsonAgentBehaviourModel jsonAgentBehaviourModel){
    String agentName = jsonAgentBehaviourModel.getAgentName();
    AgentInterface agentToDelete;
    if (checkAgentInstanceExistsByName(agentName)){
        agentToDelete = agentList.get(agentName);
        if (((SpringAgent)agentToDelete).isInitiated()){
            agentToDelete.shutDownAgent();
        }
        agentList.remove(agentName);
        return ResponseEntity.ok(new ResponseNotificationMessage("Agent " + agentName +
                " has been deleted successfully"));
    }else{
        return ResponseEntity.badRequest().body(new ResponseErrorMessage("Agent " + agentName +
                " doesn't exist in the system"));
    }
}
```

Figura 5.31 Método para eliminar un agente existente a través de la API

Los métodos *init, stop* y *restart* utilizan la misma ruta y el mismo verbo, la diferencia reside en el valor del parámetro *action*, como podemos apreciar en la *Figura 5.32*. Los detalles de implementación son análogos a los de iniciar y eliminar un agente:

se verifican los parámetros de entrada, se controla que el agente exista en la lista de agentes creados y si está ejecutándose o no en el contenedor para que pueda ser iniciado, detenido o reiniciado, añadiendo la nueva instancia devuelta a la lista de agentes creados en los dos últimos casos.

```
@PostMapping(path="agent", consumes=MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> initStopRestartAgent(@RequestBody JsonAgentBehaviourModel jsonAgentBehaviourModel){
    String action = (jsonAgentBehaviourModel.getAction() + "").trim();
    switch (action){
        case JsonAgentBehaviourModel.INIT: return initAgent(jsonAgentBehaviourModel);
        case JsonAgentBehaviourModel.STOP: return stopAgent(jsonAgentBehaviourModel);
        case JsonAgentBehaviourModel.RESTART: return restartAgent(jsonAgentBehaviourModel);
        default: return ResponseEntity.badRequest().body(new ResponseErrorMessage("Action " + action +
                " is not allowed. Try init, stop or restart"));
    }
}
```

Figura 5.32 Método para iniciar, detener y reiniciar un agente

En cuanto al tema de los comportamientos, también ofrecemos la posibilidad de obtener la lista de todas las instancias (o solo nombres) de comportamientos creados y vinculados a algún agente concreto y todos los modelos de comportamiento disponibles (o solo nombres). Los comportamientos no se crean y se eliminan igual que los agentes, sino que se crean en el momento en el que los vinculamos a algún agente y se eliminan cuando los desvinculamos. Así pues, la estructura del método para añadir, reiniciar y eliminar comportamientos (*Figura 5.33)* es la misma encontrada en el método para iniciar, detener y reiniciar un agente.

```
@PostMapping(path="agent/behaviour", consumes=MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> addResetRemoveBehaviourToAgent(@RequestBody JsonAgentBehaviourModel jsonAgentBehaviourModel){
    String action = (jsonAgentBehaviourModel.getAction() + "").trim();
    switch (action){
        case JsonAgentBehaviourModel.ADD: return addBehaviourToAgent(jsonAgentBehaviourModel);
        case JsonAgentBehaviourModel.RESET: return resetBehaviourFromAgent(jsonAgentBehaviourModel);
        case JsonAgentBehaviourModel.REMOVE: return removeBehaviourFromAgent(jsonAgentBehaviourModel);
        default: return ResponseEntity.badRequest().body(new ResponseErrorMessage("Action " + action +
                " is not allowed. Try add, reset or remove"));
    }
}
```

Figura 5.33 Método para añadir, reiniciar y eliminar comportamientos

Remitimos a los métodos explicados anteriormente y al código de la aplicación a la hora de entender el funcionamiento de la gestión de comportamientos. No obstante, es importante remarcar que cuando añadimos un comportamiento podemos especificar mediante el parámetro *startNow* si queremos se añada a la cola de

ejecución en el momento (si el agente concreto está ejecutándose) o si pasará a ser efectivo en el siguiente reinicio del agente. De manera análoga, cuando eliminamos un comportamiento podemos decidir a través del parámetro *forever* si queremos eliminarlo únicamente de la ejecución actual o de la lista de comportamientos del agente. Por último, vamos a ilustrar únicamente el método *addBehaviourToAgent* a modo de demostración de lo explicado anteriormente y de uso de los comportamientos disponibles como plantilla gracias a la interfaz *BehaviourWithFactoryInterface (Figura 5.34)*

```
private ResponseEntity<?> addBehaviourToAgent(JsonAgentBehaviourModel jsonAgentBehaviourModel){
    boolean isBehaviourAdded;
    String agentName = jsonAgentBehaviourModel.getAgentName();
    String behaviourName = jsonAgentBehaviourModel.getBehaviourName();
    boolean startNow = jsonAgentBehaviourModel.getStartNow();

    AgentInterface agent = agentList.get(agentName);
    Behaviour behaviour = availableBehaviourList.get(behaviourName);

    if (agent != null && behaviour != null){
        if (behaviour instanceof BehaviourWithFactoryInterface){
            behaviour = ((BehaviourWithFactoryInterface) behaviour).getInstance(agent.getAgentInstance());
        } else if (behaviour.getAgent() != null){
            return ResponseEntity.badRequest().body(new ResponseErrorMessage("Behaviour " + behaviourName
                    + " is already attached to the Agent " + behaviour.getAgent().getLocalName() + ". You should use behaviours" +
                    " with factory methods to get multiples instances for different agents."));
        }

        if (!startNow) {
            isBehaviourAdded = agent.addBehaviourToAgent(behaviour);
        }else{
            isBehaviourAdded = agent.addBehaviourToAgentAndInit(behaviour);
        }

        if (isBehaviourAdded){
            behaviourList.add(behaviour);
            return ResponseEntity.ok(new ResponseNotificationMessage("Behaviour" + behaviourName
                    + " has been added to the agent " + agentName + " successfully"));
        }else{
            return ResponseEntity.status(500).body(new ResponseErrorMessage("The agent " + agentName
                    + " has already the behaviour " + behaviourName + ". It is not allowed" +
                    " to add a behaviour to the same agent twice"));
        }

    }else if (agentName == null || behaviourName == null){
        return ResponseEntity.unprocessableEntity().body(new ResponseErrorMessage("Values 'agentName' and 'behaviourName' are required"));
    }else{
        return ResponseEntity.badRequest().body(new ResponseErrorMessage("Agent " + agentName + " or behaviour "
                + behaviourName + " don't exist in the system"));
    }
```

Figura 5.34 Método para añadir un comportamiento a un agente a través de la API

6. IMPLEMENTACION DEL CLIENTE WEB

Nuestro cliente web es el frontend de nuestra consola web. Tal y como especificamos en la fase de diseño, los controladores y las vistas *.xhtml estarán* situados en la capa de presentación mientras que la gestión de la plataforma JADE se encontrará en la capa de servicios/*service.* Que la gestión esté en esta capa no implica que dentro el cliente web se encuentre la plataforma JADE, sino que se genera una capa de abstracción en la que solo nos interesan los métodos que nos ofrece el servicio, no importa como los ejecuta (al igual que a lo largo de todo el TFG, remarcamos el trabajo orientado a interfaces y no a implementaciones concretas).

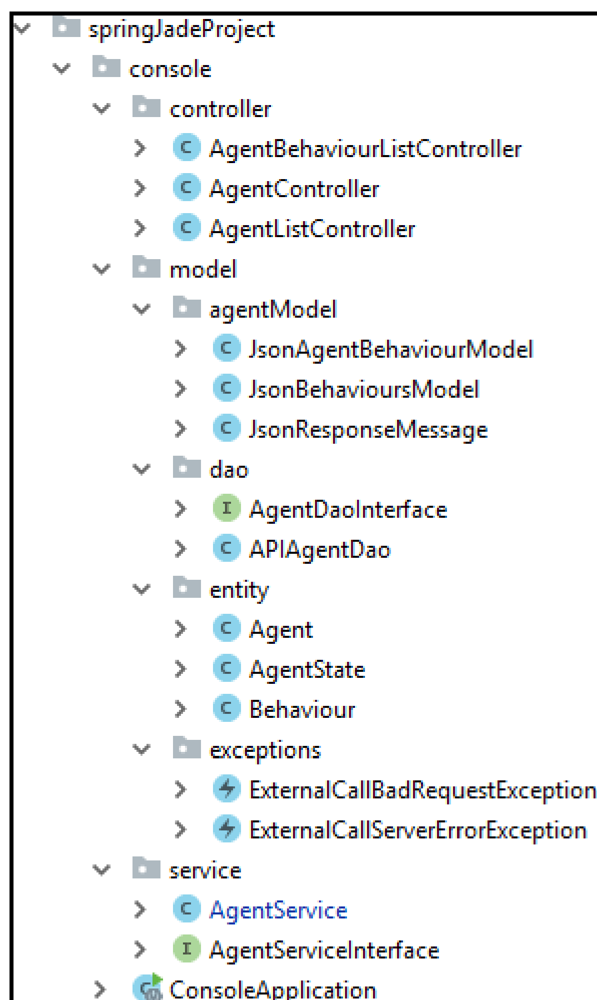

Figura 6.1 Estructura del cliente web en Spring

6.1 Capas del cliente web

Capa	Componente	Nombre ficheros	Descripción
Presentación	Controller y Webapp	**Clases Managed Beans:** AgentBehaviourListController, AgentController y AgentListController **.XHTML:** agent-behaviour-list, agent-form, agent-list, behaviour-list, behaviour-list-copy, toolbar	Implementación de las vistas con componentes XHTML y gestión de las funcionalidades relacionadas con los datos obtenidos de la consola web y las operaciones disponibles en la lógica de negocio
Lógica de Negocio	Service	**Clases:** AgentService **Interfaces:** AgentServiceInterface	Servicio para obtener información y gestionar agentes y comportamientos
Lógica de Negocio	Model	**Clases:** JsonAgentBehaviourModel, JsonBehavioursModel, JsonResponseMessage, APIAgentDao, Agent, AgentState, Behaviour, ExternalCallBadRequesetException, ExternalCallServerErrorException **Interfaces:** AgentDaoInterface	Modelos utilizados para enviar peticiones JSON y recoger y convertir en entidades java el resultado devuelto por la API. Excepciones para controlar la comunicación con la API en caso de error. APIAgentDao implementa la interfaz necesaria para gestionar los agentes a través de los métodos públicos de la API

6.2 ConsoleApplication

Esta es la clase que anotada con *@SpringBootApplication* posibilita el despliegue de la aplicación en un contenedor servlet (servidor web) empotrado. A diferencia de un repositorio de un Repositorio REST, para el cliente web estamos enlazando la tecnología JSF destinada a su uso con implementaciones del estándar Java EE con Spring MVC y Spring Boot, que no implementan todo el estándar.

Los Managed Beans son distintos de los Spring Beans, pero necesitamos los Managed Beans como controladores en JSF, y por usar Spring Boot de forma interna todo se gestiona con los Beans de Spring. Es decir, necesitamos código de configuración que permita envolver los Beans de Spring y que de forma transparente deleguen el control en los Managed Bean *(Figura 6.2)*

```
@SpringBootApplication
public class ConsoleApplication extends SpringBootServletInitializer{

    public static void main(String[] args) {
        SpringApplication.run(ConsoleApplication.class, args);
    }

    //To configure JSF with Spring Boot, we need to create these two more beans.
    @Bean
    public ServletRegistrationBean servletRegistrationBean() {
        FacesServlet servlet = new FacesServlet();
        ServletRegistrationBean registration = new ServletRegistrationBean<>(servlet, ...urlMappings: "*.xhtml");
        registration.setLoadOnStartup(1);
        return registration;
    }

    @Bean
    public FilterRegistrationBean rewriteFilter() {
        FilterRegistrationBean rewriteFilter = new FilterRegistrationBean<>(new RewriteFilter());
        rewriteFilter.setDispatcherTypes(EnumSet.of(DispatcherType.FORWARD, DispatcherType.REQUEST,
                DispatcherType.ASYNC, DispatcherType.ERROR));
        rewriteFilter.addUrlPatterns("/*");

        return rewriteFilter;
    }
}
```

Figura 6.2 Beans de configuración

Mencionar que al igual que en la API REST no necesitamos especificar ningún parámetro extra en el fichero *application*.properties para que aplicación utilice el contenedor servlet Tomcat en el puerto 8080.

No obstante, los *beans* creados no son suficientes y necesitamos añadir numerosas dependencias al proyecto y configurar los ficheros *faces-config.xml (Figura 6.3)* y *web.xml (Figura 6.4)*. Además de adoptar una estructura concreta de carpetas para que todo pudiese funcionar correctamente *(Figura 6.5)*. La configuración del cliente web no es directa y la documentación para complementar ambas tecnologías es escasa y obsoleta. Se especifican configuraciones para versiones antiguas, de forma incompleta y confusa. No vamos a explicar todos los detalles de la configuración final ya que no es el propósito principal de este apartado, pero si vamos a mostrarlas para que sea posible replicar el proyecto con facilidad. Un punto importante es el factor de que configuración es compatible con la librería *Mojarra* pero no con *MyFaces*. Hemos intentado arreglar el problema, pero hay multitud de referencias de desarrolladores con problemas similares que siguen sin solución.

```xml
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.2"
              xmlns="http://xmlns.jcp.org/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
              http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd">

    <application>
        <el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-resolver>
    </application>

</faces-config>
```

Figura 6.3 Fichero faces-config.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="4.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd">
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.xhtml</url-pattern>
    </servlet-mapping>

    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <listener>
        <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
    </listener>
</web-app>
```

Figura 6.4 Fichero web.xml

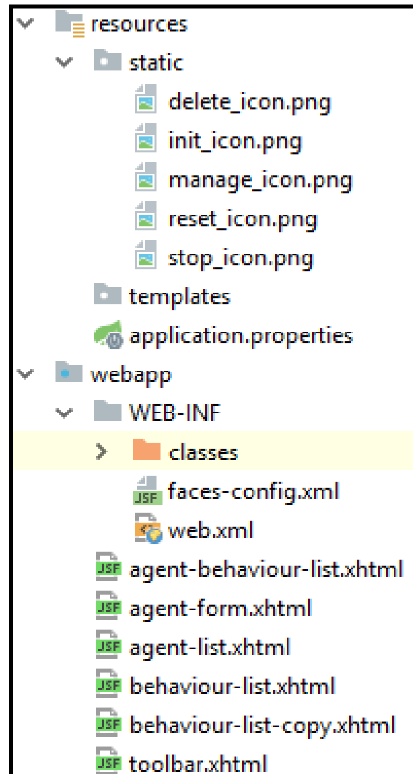

Figura 6.5 Estructura de carpetas necesarias para JSF

6.3 Dependencias

Volvemos a gestionar las dependencias del proyecto con Maven a través de la página https://start.spring.io/ , descartando la dependencia *Rest Repositories* y añadiendo la dependencia *Web (Figura 6.6).*

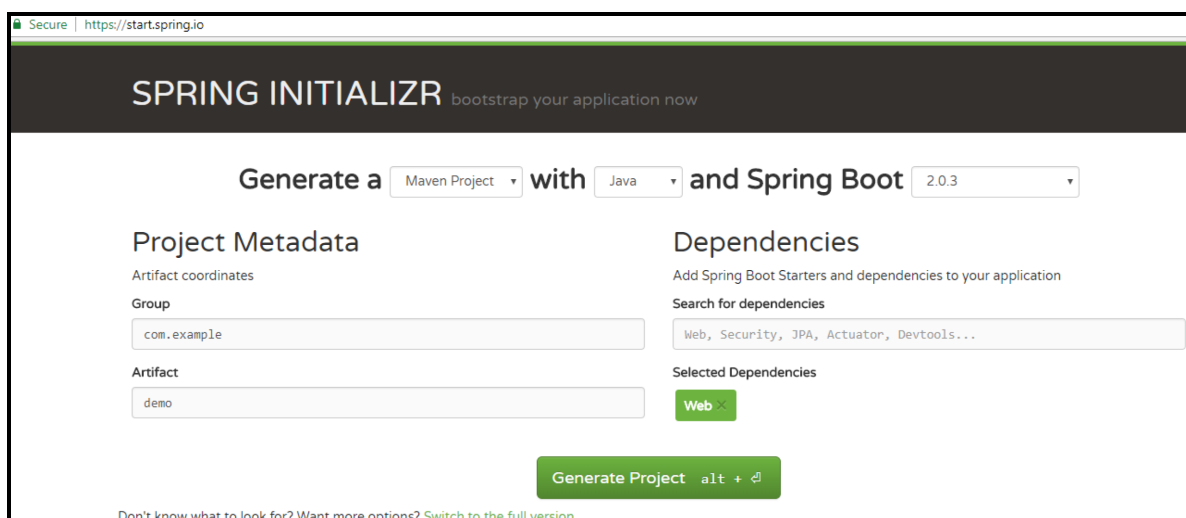

Figura 6.6 Creación de una aplicación web utilizando Spring Initializr

El fichero *pom (maven)* generado contiene las dependencias necesarias para desarrollar una aplicación web con Spring MVC y Spring Boot. Respecto al pom de la API REST desaparece la dependencia de la librería JADE y se modifica el artefacto *spring-boot-starter-data-rest* por *spring-boot-starter-web*. Las dependencias fundamentales que hemos tenido que añadir han sido la implementación *Mojarra* de JSF y *Jasper* de Tomcat, que permite interpretar y mostrar las vistas xhtml en tiempo de ejecución *(Figura 6.7).* En cuanto al tema de las vistas, hemos utilizado *PrimeFaces* como librería de componentes de interfaces de usuario, tal y como explicamos en el capítulo *1.3 Tecnologías y Herramientas utilizadas (Figura 6.8).*

```xml
<!--Mojarra implementation -->
<dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-api</artifactId>
    <version>2.2.17</version>
</dependency>
<dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-impl</artifactId>
    <version>2.2.17</version>
</dependency>

<!--needed so the JVM can parse and execute JSF
 view on runtime-->
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

Figura 6.7 Dependecias Mojarra y Jasper

```xml
<!--open source UI framework for JSF-->
<dependency>
    <groupId>org.primefaces</groupId>
    <artifactId>primefaces</artifactId>
    <version>6.2</version>
</dependency>
```

Figura 6.8 Dependencia org.primefaces

Otra dependencia que hemos incorporado ha sido la librería de código abierto *org.ocpsoft.rewrite (Figura 6.9).* Es una herramienta que permite redirigir y rescribir direcciones URL para entornos Servlet con frameworks de Java web (a partir de Java EE 6 y Servlet 2.5). Soporta integración con CDI, Spring DI, JavaServer Faces, JavaServer Pages, Struts o Spring Web Flow, por nombrar solo algunos ejemplos. El segundo Bean de *Console Application* utiliza una instancia de esta librería para determinar quién va a gestionar la rescritura de las direcciones URLs (hay varias librerías que proporcionan esta funcionalidad). De forma simplificada, nosotros la utilizamos para asociar cada URL a un controlador (*Managed Bean)* y a una vista xhtml a través de las anotaciones *@ELBeanName* y *@Join.*

Por último, aun utilizando el mismo plugin tenemos que especificar que especificar que el lugar donde se generarán las clases será dentro de la carpeta *WEB-INF*, ya que la librería *Rewrite* no es capaz de escanear las configuraciones en aplicaciones web que no siguen una estructura clásica (*Figura 6.10).*

```
<!--open-source routing and URL rewriting solution for
 Servlet and Java Web Frameworks-->
<dependency>
    <groupId>org.ocpsoft.rewrite</groupId>
    <artifactId>rewrite-servlet</artifactId>
    <version>3.4.2.Final</version>
</dependency>

<dependency>
    <groupId>org.ocpsoft.rewrite</groupId>
    <artifactId>rewrite-integration-faces</artifactId>
    <version>3.4.2.Final</version>
</dependency>

<dependency>
    <groupId>org.ocpsoft.rewrite</groupId>
    <artifactId>rewrite-config-prettyfaces</artifactId>
    <version>3.4.2.Final</version>
</dependency>

<dependency>
    <groupId>org.ocpsoft.logging</groupId>
    <artifactId>logging-adapter-slf4j</artifactId>
    <version>1.0.5.Final</version>
</dependency>
```

Figura 6.9 Dependecias org.ocpsof

```
<!--This configuration is important because Rewrite isn't prepared to
 scan for configurations on non-classical web applications-->
<build>
    <outputDirectory>src/main/webapp/WEB-INF/classes</outputDirectory>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Figura 6.10 Ruta de las clases generadas del proyecto

6.4 Capa Service: AgentServiceInterface y AgentService

La estructuración de un sistema en capas como el nuestro nos permite aislar las funcionalidades concretas y desacoplar la presentación de la lógica de negocio. *AgentServiceInterface (Figura 6.11)* proporciona los métodos necesarios para que los controladores *Managed Beans* basados en los datos de entrada suministrados por el usuario puedan gestionar los agentes y comportamientos sin tener que preocuparse por ninguna implementación concreta. Limitamos así la labor de los controladores a la gestión de la presentación de los datos y la navegación a través de las vistas con el fin de crear la mejor experiencia de usuario posible.

```
public interface AgentServiceInterface {
    Collection<Agent> loadAllAvailableAgents();

    Collection<Agent> loadAllCreatedAgents();

    Map<String, Agent> loadAllCreatedAgentsInMap();

    Collection<String> loadAllCreatedAgentsNames();

    boolean createAgent (String className, String agentName);

    boolean deleteAgent (Agent agent);

    boolean initAgent(Agent agent);

    boolean stopAgent(Agent agent);

    boolean restartAgent(Agent agent);

    Collection<Behaviour> loadBehavioursAvailable();

    Collection<JsonBehavioursModel> loadAllAgentsAndItsBehaviours();

    boolean addBehavioursToAgent(Agent agent, Collection<Behaviour> behaviours);

    boolean removeBehavioursFromAgent(Agent agent, Collection<Behaviour> behaviours);

    boolean resetBehaviour(Agent agent, Behaviour behaviour);

    Collection<String> loadAllAgentClassNames();
}
```

Figura 6.11 Métodos públicos de la interfaz AgentServiceInterface

Con AgentService tenemos una implementación concreta basada en que la gestión de la plataforma JADE se realiza a través de los métodos públicos de la API REST desarrollada. Esta implementación utiliza un servicio *dao (data access object)* que permite abstraer al servicio del uso de la API y que finalmente solo se encargue de detalles menores *(Figuras 6.12 y 6.13).*

```java
@Service
public class AgentService implements AgentServiceInterface{
    private AgentDaoInterface agentDao;

    public AgentService(@Autowired AgentDaoInterface agentDaoIn) {
        agentDao = agentDaoIn;
    }

    @Override
    public Collection<Agent> loadAllAvailableAgents() {
        Collection<Agent> result = agentDao.getAllAvailableAgents();
        for (Agent agent: result){
            if (agent.getName() == null){
                agent.setName("-");
            }
        }
        return result;
    }
```

Figura 6.12 Servicio AgentService

```java
@Override
public boolean createAgent(String className, String agentName) {
    return agentDao.createAgent(className, agentName);
}

@Override
public boolean deleteAgent(Agent agent) {
    return agentDao.deleteAgent(agent);
}

//similar to update some parameter from an agent.
// This is why we use a DAO
@Override
public boolean initAgent(Agent agent) {
    return agentDao.initAgent(agent);
}
```

Figura 6.13 Abstracción del uso de la API con el dao

Si hubiésemos realizado un único proyecto con la gestión de la interfaz de la consola web y la plataforma JADE, la implementación de los métodos hubiera sido distinta por diversos motivos. En ese caso, *AgentService* inyectaría una instancia de *AgentsManager* para gestionar la plataforma JADE, y utilizaríamos otra implementación de *AgentDaoInterface* que no utilizara la API sino un mecanismo de persistencia como una base de datos donde almacenásemos los agentes y comportamientos, mientras que, por ejemplo, la lógica para añadir y eliminar un comportamiento de un agente se encontraría en este servicio.

6.5 Capa Model: AgentModel, Entity y Exceptions

En esta capa nos encontramos en primer lugar con los distintos modelos que forman parte de la lógica de negocio del cliente web. Al desacoplar de la consola web de toda la gestión de la plataforma JADE podemos olvidar el uso de la librería: solo necesitamos versiones simplificadas de las clases *Agent* y *Behaviour* con las variables imprescindibles para su gestión por parte del cliente web. Las entidades de agente y comportamientos en texto JSON devueltas por la API no recogen, en primer lugar, todos los parámetros de los agentes y comportamientos de la librería JADE, ya que muchos parámetros corresponden a una gestión interna por parte del contenedor y la librería *jackson* no tiene acceso a través del uso de métodos *get* y *set.* Además, al extender de la clase *SpringAgent* poseen un conjunto de variables extras que aportan información que también podemos utilizar. En definitiva, las instancias de la clase *Agent* se definen por su *nickname, initiated, name* (nombre en

el contenedor), *agentClassName* y *agentState (Figura 6.14)*. A su vez agentState es una clase con los valores *name* y *value*. Hemos definido las variables de clase con el mismo nombre que el devuelto por la API, para que la librería *jackson* interprete el texto JSON y cree la instancia de forma automática. Por otro lado tenemos la clase *Behaviour* que se define por su *restartCounter, executionState, agentLocalName, behaviourName y checked (Figura 6.15).* Mencionar que el valor *checked* no es devuelto por la API, sino que se utiliza para gestionar si un comportamiento ha sido seleccionado desde la interfaz. Todas las clases incorporan la anotación *@JsonIgnoreProperties(ignoreUnknown = true)* para que los parámetros no contemplados en estos modelos sean ignorados sin interrumpir el proceso de interpretación.

```java
@JsonIgnoreProperties(ignoreUnknown = true)
public class Agent {
    private String nickname; //agent name => localName
    private boolean initiated;
    //agent name in container => localName@containerAddress
    private String name;
    private String agentClassName;
    private AgentState agentState;

    public Agent(String nickname, Boolean initiated, String name,
                 String agentClassName, AgentState agentState) {
        this.nickname = nickname;
        this.initiated = initiated;
        this.name = name;
        this.agentClassName = agentClassName;
        this.agentState = agentState;
    }

    protected Agent(){}
```

Figura 6.14 Clase Agent

```java
@JsonIgnoreProperties(ignoreUnknown = true)
public class Behaviour {
    private int restartCounter;
    private String executionState;
    private String agentLocalName;
    private String behaviourName;
    //variable required for interface purposes.
    // I want to know which behaviour is checked
    // to remove or add it to an agent
    private boolean checked;
```

Figura 6.15 Clase Behaviour

Para el resto de comunicaciones con la API tenemos el modelo *JsonResponseMessage* para las notificaciones de éxito y error, *JsonBehavioursModel* para la lista de comportamientos vinculadas al nombre del agente y *JsonAgentBehaviourModel* para construir el cuerpo de las peticiones. Son los mismos modelos que explicamos en el capítulo de la API.

Por último tenemos dos clases para la gestión de excepciones: *ExternalCallBadRequestException* y *ExternalCallServerErrorException.* Las utilizamos simplemente para aislar y tratar mejor las excepciones resultado de la comunicación con la API.

6.6 Capa Model: AgentDaoInterface y APIAgentDao

De forma genérica, el patrón *dao* (*data access object)* se utiliza para separar el acceso a bajo nivel a algún mecanismo de persistencia (como una base de datos) de los servicios de lógica de negocio de más alto nivel. La clase *AgentDaoInterface (Figura 6.16)* define las operaciones estándar que van a ejecutarse sobre los distintos objetos, mientras que *APIAgentDao* es la clase que implementa la interfaz y obtiene realmente los datos de la API, creando las diferentes instancias de los modelos mencionados en el apartado anterior.

```
public interface AgentDaoInterface {
    Collection<Agent> getAllAvailableAgents();

    Collection<Agent> getAllCreatedAgents();

    Collection<String> getAllCreatedAgentsNames();

    boolean createAgent (String className, String agentName);

    boolean deleteAgent (Agent agent);

    boolean initAgent(Agent agent);

    boolean stopAgent(Agent agent);

    boolean restartAgent(Agent agent);

    Collection<Behaviour> loadBehavioursAvailable();

    Collection<JsonBehavioursModel> loadAllAgentsAndItsBehaviours();

    boolean addBehavioursToAgent (Agent agent, Collection<Behaviour> behaviours);

    boolean removeBehavioursFromAgent (Agent agent, Collection<Behaviour> behaviours);

    boolean resetBehaviourFromAgent (Agent agent, Behaviour behaviour);

    Collection <String> loadAllAgentClassNames();
}
```

Figura 6.16 Interfaz AgentDaoInterface

La clase *APIAgentDao* es un componente del tipo *@Repository,* y al ser la única implementación disponible de *AgentDaoInterface* no necesitamos incluir la anotación *@Qualifier* para que sea inyectado.

El primer elemento necesario para la implementación es la dirección/ruta base de la API *(Figura 6.17)*. Para este TFG hemos realizado todas las pruebas en el propio

equipo de trabajo, pero no habría ninguna complejidad extra en desplegar la API en un servidor en la nube como Amazon Web Services o Google Cloud.

```
@Repository
public class APIAgentDao implements AgentDaoInterface{
    private static final String API_URL = "http://localhost:9090/api";
```

Figura 6.17 Clase APIAgentDao con la dirección de la API

Para realizar peticiones a la API hemos utilizado la clase *RestTemplate* del paquete *org.springframwork.web.client* y la clase *HttpHeaders, HttpEntity<T>* y *ResponseEntity<T>* del paquete *org.springframewokr.http.* Esta librería utiliza *jackson* de forma interna por lo que al igual que en el desarrollo de la API podemos abstraer la conversión de texto JSON a bajo nivel. A modo de clasificación podemos definir las peticiones a la API que necesitamos implementar en dos tipos: las que envían información en el cuerpo de la petición y obtienen un mensaje de éxito o de error como respuesta y aquellas que no utilizan parámetros de consulta sobre la URI y obtienen una lista de objetos.

El primer método es *sendJsonRequesToAPI*, con la ruta de la petición, el método/verbo a utilizar y el cuerpo de la petición (una instancia de la clase *JsonAgentBehaviourModel*) como argumentos. El proceso con las clases proporcionadas por Spring es bastante directo: creamos una instancia de la clase *RestTemplate* y otra de la clase *HttpHeaders* (la cabecera) dónde especificamos que el contenido tiene formato JSON para luego crear una instancia de *HttpEntity* con el cuerpo y la cabecera de la petición. Ya solo queda usar el método *exchange* de la clase *RestTemplate* para realizar la petición, recibiendo la respuesta en una instancia de la clase *ResponseEntity*. Esta instancia contiene el código y descripción del error además del cuerpo del mensaje. Controlamos las excepciones tanto por una petición con una estructura incorrecta, como por problemas del servidor o de conexión *(Figura 6.18).*

```java
private boolean sendJsonRequestToAPI(String url, HttpMethod httpMethod, JsonAgentBehaviourModel jsonRequest) {
    RestTemplate restTemplate = new RestTemplate();
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_JSON);

    HttpEntity<JsonAgentBehaviourModel> entity = new HttpEntity<>(jsonRequest, headers);

    try {
        ResponseEntity<JsonResponseMessage> agentResponse =
                restTemplate.exchange(url,
                        httpMethod, entity, JsonResponseMessage.class);

        if (agentResponse.getStatusCode() == HttpStatus.OK) {
            System.out.println(agentResponse.getBody().getMessage());
        }
    } catch (final HttpClientErrorException httpClientErrorException) {
        if (httpClientErrorException.getStatusCode() == HttpStatus.CONFLICT){
            return true;
        }else {
            throw new ExternalCallBadRequestException(httpClientErrorException.getMessage()
                    + " for " + url + " with " + httpClientErrorException.getResponseBodyAsString());
        }
    } catch (HttpServerErrorException httpServerErrorException) {
        throw new ExternalCallServerErrorException(httpServerErrorException.getMessage()
                + " for " + url + " with " + httpServerErrorException.getResponseBodyAsString());
    } catch (Exception ex){
        System.out.println(ex.getMessage());
        //connection refused for example
        return false;
    }
    return true;
}
```

Figura 6.18 Método sendJsonRequestToAPI

Este tipo de petición es utilizada por los métodos *createAgent, deleteAgent (Figura 6.19), initAgent, stopAgent, restartAgent, addBehavioursToAgent, removeBehavioursFromAgent* y *resetBehaviourFromAgent*

```java
@Override
public boolean createAgent(String className, String agentName) {
    String url = API_URL.concat("/agent");
    HttpMethod httpMethod = HttpMethod.PUT;
    JsonAgentBehaviourModel jsonRequest =
            createJsonRequest(agentName);
    jsonRequest.setClassName(className);
    return sendJsonRequestToAPI(url, httpMethod, jsonRequest);
}

@Override
public boolean deleteAgent(Agent agent) {
    String url = API_URL.concat("/agent");
    HttpMethod httpMethod = HttpMethod.DELETE;
    JsonAgentBehaviourModel jsonRequest =
            createJsonRequest(agent.getNickname());
    return sendJsonRequestToAPI(url, httpMethod, jsonRequest);
}
```

Figura 6.19 Métodos createAgent y deleteAgent

Si no recibimos el verbo a utilizar hemos implementado el mismo método con la misma signatura, pero con la ausencia de este argumento. El nuevo método simplemente invoca al método que hemos descrito con el verbo POST por defecto.

Los métodos *initAgent, stopAgent, restartAgent* y *resetBehaviourFromAgent* utilizan esta sobrecarga del método *(Figura 6.20)*.

```
@Override
public boolean initAgent(Agent agent) {
    String url = API_URL.concat("/agent");
    JsonAgentBehaviourModel jsonRequest =
            createJsonRequest(agent.getNickname());
    jsonRequest.setAction(JsonAgentBehaviourModel.INIT);
    return sendJsonRequestToAPI(url, jsonRequest);
}

@Override
public boolean stopAgent(Agent agent) {
    String url = API_URL.concat("/agent");
    JsonAgentBehaviourModel jsonRequest =
            createJsonRequest(agent.getNickname());
    jsonRequest.setAction(JsonAgentBehaviourModel.STOP);
    return sendJsonRequestToAPI(url, jsonRequest);
}

@Override
public boolean restartAgent(Agent agent) {
    String url = API_URL.concat("/agent");
    JsonAgentBehaviourModel jsonRequest =
            createJsonRequest(agent.getNickname());
    jsonRequest.setAction(JsonAgentBehaviourModel.RESTART);
    return sendJsonRequestToAPI(url, jsonRequest);
}
```

Figura 6.20 Métodos initAgent, stopAgent y restartAgent

Por otro lado, los métodos *addBehavioursToAgent* y *removeBehavioursFromAgent (Figura 6.21)* utilizan otro método intermedio: *manageBehavioursFromAgent (Figura 6.22)*. Este método tiene como argumentos la ruta de la API, la acción a realizar, el nombre del agente y la lista de comportamientos que queremos gestionar. Se crea el contenido de la petición *jsonRequest* de manera general, pero para cada elemento de la lista se modifica con el nombre del comportamiento concreto y se realiza la petición aislada a través de *sendJsonRequesToAPI.*

```
@Override
public boolean addBehavioursToAgent(Agent agent, Collection<Behaviour> behaviours) {
    String url = API_URL.concat("/agent/behaviour");
    return manageBehavioursFromAgent(url, JsonAgentBehaviourModel.ADD,
            agent.getNickname(), behaviours);
}

@Override
public boolean removeBehavioursFromAgent(Agent agent, Collection<Behaviour> behaviours) {
    String url = API_URL.concat("/agent/behaviour");
    return manageBehavioursFromAgent(url, JsonAgentBehaviourModel.REMOVE,
            agent.getNickname(), behaviours);
}
```

Figura 6.21 Métodos addBehavioursToAgent y removeBehavioursFromAgent

```
private boolean manageBehavioursFromAgent(String url, String action, String agentName, Collection<Behaviour> behaviours){
    boolean allBehavioursDone = true; //=>false if server have had problems to add/remove any behaviour
    //On the server, startNow is only read when you add a behaviour. RemoveForever is only read when you remove a behaviour
    JsonAgentBehaviourModel jsonRequest = createJsonRequest(agentName, behaviourName: "", startNow: true, removeForever: true);
    jsonRequest.setAction(action);

    for (Behaviour behaviour : behaviours){
        jsonRequest.setBehaviourName(behaviour.getBehaviourName());
        allBehavioursDone = allBehavioursDone && sendJsonRequestToAPI(url, jsonRequest);
    }
    return allBehavioursDone;
}
```

Figura 6.22 Método intermedio manageBehavioursFromAgent

El segundo método principal que hemos mencionado anteriormente es *requestListToAPI,* con la ruta de la petición y una clase genérica sin variable de tipo (usamos el signo de interrogación *'?'* o *wildcard) ParameterizedTypeReference<T>* para definir el tipo de respuesta como argumento. En este caso no tenemos que configurar la cabecera y debemos invocar el método *exchange* con el tipo de respuesta como último parámetro. La gestión de excepciones es análoga al método anterior *(Figura 6.23).* Este método es bastante "abstracto" y versátil ya que en función del valor genérico de la clase *responseType* podemos obtener listas de cualquier tipo de elemento.

```
private Collection<?> requestListToAPI(String url, ParameterizedTypeReference<?> responseType) {
    RestTemplate restTemplate = new RestTemplate();
    Collection<?> res = new ArrayList<>();

    try {
        ResponseEntity<?> responseEntity = restTemplate.exchange(url,
                HttpMethod.GET,  requestEntity: null, responseType);

        if (responseEntity.getStatusCode() == HttpStatus.OK) {
            res = (Collection<?>) responseEntity.getBody();
        }
    } catch (final HttpClientErrorException httpClientErrorException) {
        throw new ExternalCallBadRequestException(httpClientErrorException.getMessage() +
                " for " + url + " with " + httpClientErrorException.getResponseBodyAsString());
    } catch (HttpServerErrorException httpServerErrorException) {
        throw new ExternalCallServerErrorException(httpServerErrorException.getMessage() +
                " for " + url + " with " + httpServerErrorException.getResponseBodyAsString());
    } catch (Exception ex){
        System.out.println(ex.getMessage());
    }

    return res;
}
```

Figura 6.23 Método requestListToAPI

Este tipo de petición es utilizada por los métodos *loadAllAgentsAndItsBehaviours (Figura 6.24), getAllAvailableAgents, getAllCreatedAgents, loadBehavioursAvailable, getAllCreatedAgentsNames* y *loadAllAgentClassNames.*

```
@Override
@SuppressWarnings("unchecked")
public Collection<JsonBehavioursModel> loadAllAgentsAndItsBehaviours() {
    String url = API_URL.concat("/agents/available/behaviours");
    ParameterizedTypeReference<List<JsonBehavioursModel>> typeResponse =
            new ParameterizedTypeReference<List<JsonBehavioursModel>>() {};

    return (Collection<JsonBehavioursModel>) requestListToAPI(url, typeResponse);
}
```

Figura 6.24 Método loadAllAgentsAndItsBehaviours

Al final, la única diferencia entre los distintos métodos es el tipo genérico de la instancia *ParameterizedTypeReference*, por lo que hemos creado métodos intermedios para obtener una lista de agentes, una lista de comportamientos y una lista de *Strings (Figura 6.25).*

```
@SuppressWarnings("unchecked")
private Collection<Agent> requestAgentListToAPI(String url) {
    ParameterizedTypeReference<List<Agent>> typeResponse =
            new ParameterizedTypeReference<List<Agent>>() {};
    return (Collection<Agent>)requestListToAPI(url, typeResponse);
}

@SuppressWarnings("unchecked")
private Collection<String> requestStringListToAPI(String url) {
    ParameterizedTypeReference<List<String>> typeResponse =
            new ParameterizedTypeReference<List<String>>() {};
    return (Collection<String>)requestListToAPI(url, typeResponse);
}

@SuppressWarnings("unchecked")
private Collection<Behaviour> requestBehaviourListToAPI(String url) {
    ParameterizedTypeReference<List<Behaviour>> typeResponse =
            new ParameterizedTypeReference<List<Behaviour>>() {};
    return (Collection<Behaviour>)requestListToAPI(url, typeResponse);
}
```

Figura 6.25 Métodos intermedios para obtener listas de agentes, comportamientos y Strings

6.7 Capa Controller: estructura de controladores, vistas y funcionalidades

Controlador	Vista	Funcionalidades
AgentListController	agent-list	Listar agentes, visualizar detalles del agente, iniciar agente, detener agente, reiniciar agente y eliminar agente
AgentController	agent-form	Crear agente
AgentBehaviourListController	agent-behaviour-list	Listar comportamientos de agente y visualizar detalles comportamiento de agente.
	behaviour-list	Añadir comportamiento a agente, eliminar comportamiento de agente, y reiniciar comportamiento de agente

6.8 Capa Controller: AgentListController y agent-list.xhtml

Este par controlador-vista se encarga de gestionar la página principal de la aplicación. *AgentListController* es un bean de sesión ya que necesitamos mantener la lista de agentes actualizada y mostrar los mensajes de éxito o de error. Con un ámbito de *request* el bean se destruiría al finalizar la petición (con la acción de iniciar un agente, por ejemplo), por lo que la lista de agentes no podría recuperarse y aparecería vacía. Especificamos cuatro anotaciones de clase:

- *@Scope*: especifica el ámbito (*scope)* del bean

- *@Component*: determina que estamos ante un bean de Spring
- *@ELBeanName*: permite especificar el nombre del bean para que sea utilizado desde las distintas vistas xhtml
- *@Join*: determina la ruta URI gestionada por este controlador y la primera vista que mostrará al ser invocado

Asimismo, solo tenemos cuatro variables de clase: una instancia inyectada de *AgentServiceInterface*, la lista de agentes creados y los mensajes de éxito y error. La lista de agentes se actualiza en cada petición request, ya que la API puede tener multitud de clientes distintos desde los que gestionar los agentes y comportamientos y no queremos mostrar al usuario final un estado inconsistente. Con este fin, y asegurándonos que la instancia del servicio ya ha sido inyectada al ejecutar el método, utilizamos las anotaciones *@Deferred, @RequestAction* y *@IgnorePostBack*, en sustitución a *@PostBack*, que solo se ejecutaría una única vez en el bean de sesión *(Figura 6.26).*

```
@Scope(value = "session")
@Component(value = "agentListController")
@ELBeanName(value = "agentListController")
@Join(path = "/", to = "/agent-list.xhtml")
public class AgentListController {
    @Autowired
    private AgentServiceInterface agentService;
    private List<Agent> agents;
    private String successfulMessage;
    private String errorMessage;

    @Deferred
    @RequestAction
    @IgnorePostback
    public void loadData(){
        agents = new ArrayList<>(agentService.loadAllCreatedAgents());
    }
```

Figura 6.26 Clase AgentListController

Las vistas son capaces de acceder al valor de estas variables y modificarlas a través de los métodos *get* y *set.* Los métodos accesibles desde la vista son init, stop *(Figura 6.27),* restart, delete *(Figura 6.28)* y reloadData. El funcionamiento de los métodos es muy simple: invocan el método correspondiente del servicio *agentService* con el agente seleccionado y almacenan en una variable el mensaje de éxito o de error de la operación, para finalmente devolver la ruta de la vista que va a ser mostrada a continuación (en este caso nos mantenemos en la página principal).

```java
public String init(Agent agent){
    if (agentService.initAgent(agent)){
        displaySuccessfulMessage("Agent "
                + agent.getNickname()
                + " was initiated successfully");
    }else{
        displayErrorMessage("Agent "
                + agent.getNickname()
                + " have had problems to initiate");
    }
    return "/agent-list.xhtml?faces-redirect=true";
}

public String stop(Agent agent){
    if (agentService.stopAgent(agent)){
        displaySuccessfulMessage("Agent "
                + agent.getNickname() +
                " was stopped successfully");
    }else{
        displayErrorMessage("Agent "
                + agent.getNickname() +
                " have had problems to stop");
    }
    return "/agent-list.xhtml?faces-redirect=true";
}
```

Figura 6.27 Métodos init y stop

```java
public String restart(Agent agent){
    if (agentService.restartAgent(agent)){
        displaySuccessfulMessage("Agent "
                + agent.getNickname()
                + " was restarted successfully");
    }else{
        displayErrorMessage("Agent "
                + agent.getNickname()
                + " have had problems to restart");
    }
    return "/agent-list.xhtml?faces-redirect=true";
}

public String delete(Agent agent){
    System.out.println("Delete agent called");
    if (agentService.deleteAgent(agent)){
        displaySuccessfulMessage("Agent "
                + agent.getNickname()
                + " was deleted successfully");
    }else{
        displayErrorMessage("Agent "
                + agent.getNickname()
                + " have had problems to be deleted");
    }
    return "/agent-list.xhtml?faces-redirect=true";
}
```

Figura 6.28 Métodos restart y delete

Antes de explicar la vista *agent-list.xhtml* vamos a comentar el uso de *toolbar.xhtml.* Con *facelets* y *primefaces* podemos crear interfaces como componentes para que sean reutilizables en otras vistas *(Figura 6.29)*. Hemos especificado el título de la página y una barra de tareas *(toolbar)* como elemento común, con lo que especificando que las nuevas vistas son una composición de elementos y utilizan el componente *toolbar.xhtml* conseguimos este resultado. Esta barra de tareas permite navegar a través de las funcionalidades de la consola de forma directa, mejorando la experiencia de usuario *(Figura 6.30)*

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:p="http://primefaces.org/ui">
<f:view>
    <h:head>
        <title>Jade Agent Web Console</title>
    </h:head>
    <h:body>
        <div class="ui-g">
            <p:toolbar>
                <f:facet name="left">
                    <p:button href="/" value="Agent Manager" />
                    <p:button href="/agent" value ="New Agent"/>
                    <p:button href="/agent/behaviour/list" value="Agent Behaviour List" />
                </f:facet>
            </p:toolbar>
        </div>
        <div class="ui-g-12">
            <ui:insert name="content"/>
        </div>
    </h:body>
</f:view>
</html>
```

Figura 6.29 Vista toolbar.xhtml definida para ser utilizada por el resto de vistas

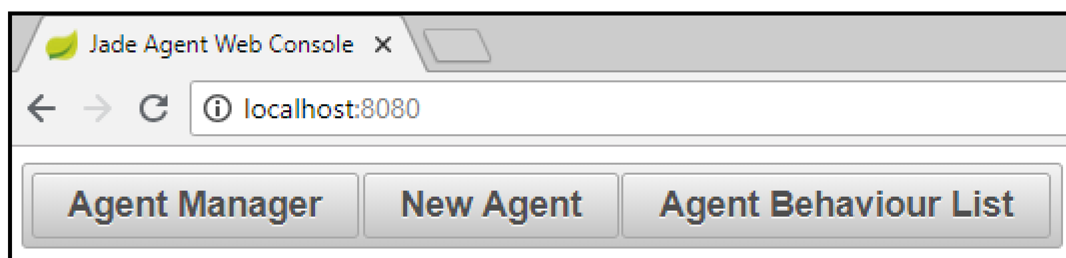

Figura 6.30 Componente toolbar.xhtml

En cuanto a la vista *agent-list.xhtml* es básicamente una composición con *toolbar.xhtml* en la que se define una tabla con la lista de agentes. Cada fila corresponde a la información concreta de un agente. De manera inicial hemos decidido mostrar el nombre del agente, su clase, su nombre en el contenedor, si está ejecutándose en el contenedor y su estado *(Figura 6.31)*.

```
<p:panel header="Agent List" style="text-align: center;">
    <p:dataTable id="table"
                 var="agent" value="#{agentListController.agents}"
                 tableStyle="width:auto" resizableColumns="true"
                 style="text-align: center;"
                 emptyMessage="No agents found">

        <p:column style="width: 60%;">
            <f:facet name="header">Agent name</f:facet>
            <h:outputText value="#{agent.nickname}"/>
        </p:column>

        <p:column style="width: 60%;">
            <f:facet name="header">Agent class</f:facet>
            <h:outputText value="#{agent.agentClassName}"/>
        </p:column>
```

Figura 6.31 Extracto de la tabla de agentes de agent-list.xhtml

Cada fila de la tabla también incluye los botones para iniciar, detener, reiniciar y eliminar el agente. Hemos utilizado un enlace *commandLink* con una imagen *graphicImage* en vez de un botón para hacer la interfaz más atractiva *(Figura 6.32).* Si el agente no ha sido iniciado solo aparece el botón iniciar y eliminar, en caso contrario el botón iniciar se oculta y aparecen los botones detener y reiniciar.

```
<p:column style="width: 50%;">
    <f:facet name="header">Action</f:facet>
    <h:commandLink id="init-button"
                   rendered="#{!agent.initiated}"
                   action="#{agentListController.init(agent)}">
        <p:graphicImage style="width: 30px;"
                        title="Init Agent"
                        value="/init_icon.png"/>
    </h:commandLink>
    <h:commandLink id="stop-button" style="margin-right:5px"
                   rendered="#{agent.initiated}"
                   action="#{agentListController.stop(agent)}">
        <p:graphicImage style="width: 25px;"
                        title="Stop Agent"
                        value="/stop_icon.png"/>
    </h:commandLink>
    <h:commandLink id="restart-button"
                   rendered="#{agent.initiated}"
                   action="#{agentListController.restart(agent)}">
        <p:graphicImage style="width: 25px;"
                        title="Restart Agent"
                        value="/reset_icon.png"/>
    </h:commandLink>
 </p:column>

<p:column style="width: 30%;">
    <f:facet name="header">Delete Agent</f:facet>
    <h:commandLink id="delete-button"
                   onclick="return confirm('Are you sure you want to delete this agent?');"
                   action="#{agentListController.delete(agent)}">
        <p:graphicImage style="width: 25px;"
                        title="Delete Agent"
                        value="/delete_icon.png"/>
    </h:commandLink>
</p:column>
```

Figura 6.32 Botones para iniciar, detener, reiniciar y eliminar un agente

Por último, hemos incluido un botón para recargar la lista de agentes y mensajes de éxito y error ante las distintas acciones disponibles en la página *(Figura 6.33).*

```
    </p:panel>
    <p:commandButton id="reload-data-button"
                     style="width: 100%; background: #85b4b3;"
                     action="#{agentListController.reloadData}"
                     value="Reload"/>
</h:form>
<h:outputText value="#{agentListController.successfulMessage}"
              style="color:green;"
              rendered="#{agentListController.successfulMessage != null}"/>
<h:outputText value="#{agentListController.errorMessage}"
              style="color:red;"
              rendered="#{agentListController.errorMessage != null}"/>
```

Figura 6.33 Botón para recargar la lista de agentes y mensajes de éxito y error

A continuación vamos a mostrar los distintos casos de uso de esta página principal. Mostramos la lista de agentes con todos los detalles de forma simultánea, si no hay agentes creados se muestra un mensaje con la lista vacía. Cuando iniciamos un agente su estado cambia y el botón iniciar desaparece, apareciendo los botones detener y reiniciar en su lugar. Cuando pulsamos detener ocurre el proceso inverso. Podemos eliminar un agente en cualquier momento, la gestión garantiza que si está iniciado se procederá a su detención de forma previa a la eliminación. Para cualquier operación, incluyendo la recarga de la lista de agentes, se muestra un mensaje de éxito o de error.

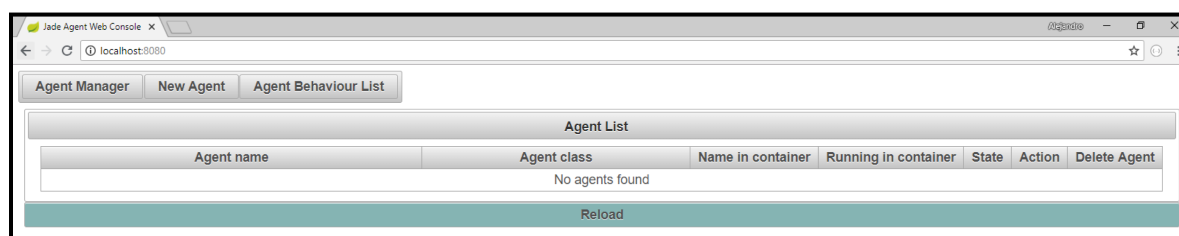

Figura 6.34 Página principal sin agentes creados

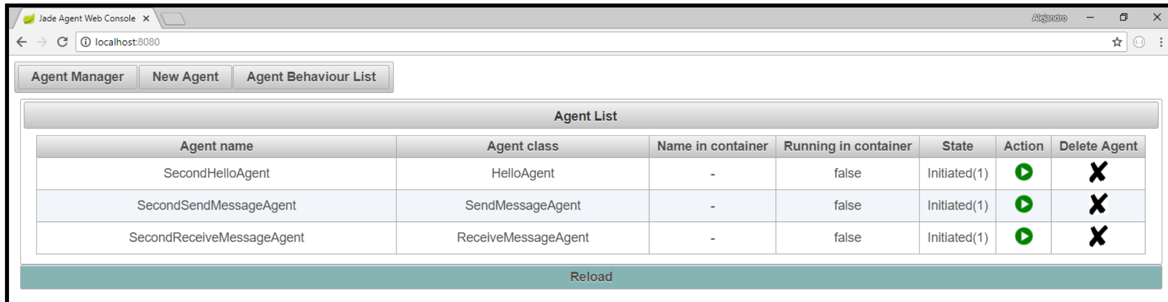

Figura 6.35 Página principal con agentes creados

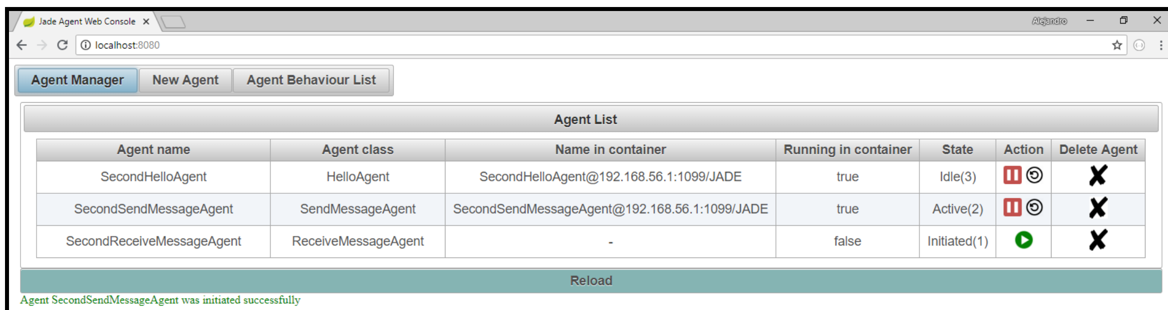

Figura 6.36 Página principal tras haber iniciado el primer agente (SecondHelloAgent). El segundo ya estaba iniciado

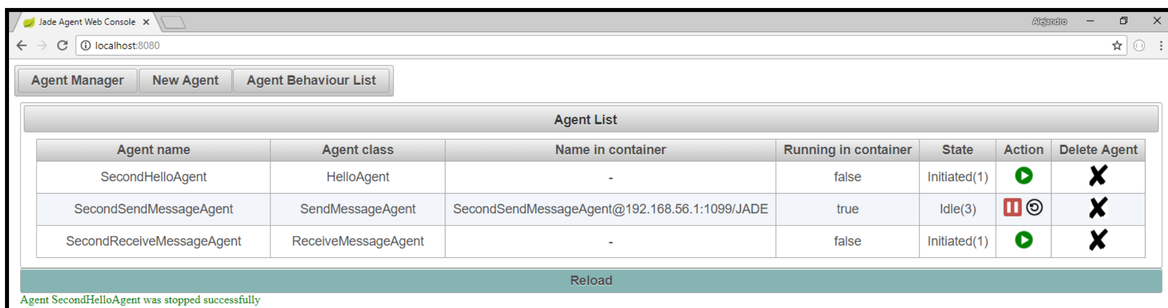

Figura 6.37 Página principal tras detener el primer agente (SecondHelloAgent)

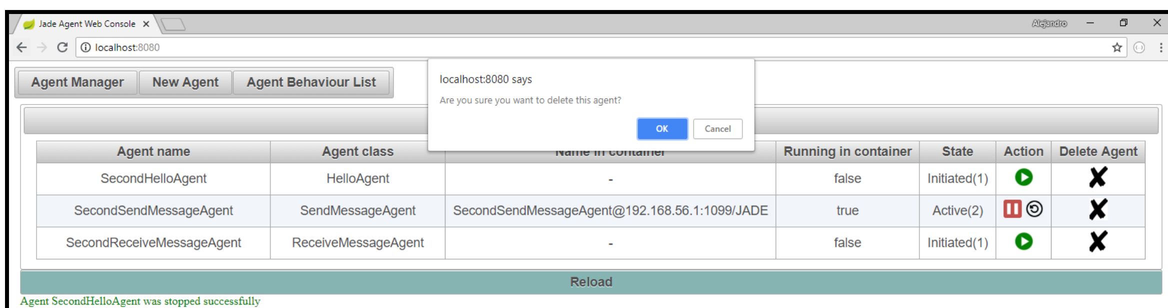

Figura 6.38 Página principal. Panel de confirmación al pulsar el botón eliminar del primer agente

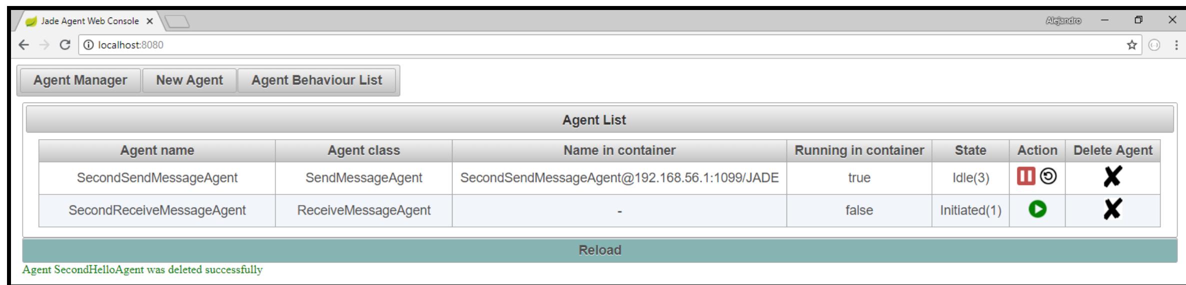

Figura 6.39 Página principal tras eliminar el primer agente (SecondHelloAgent)

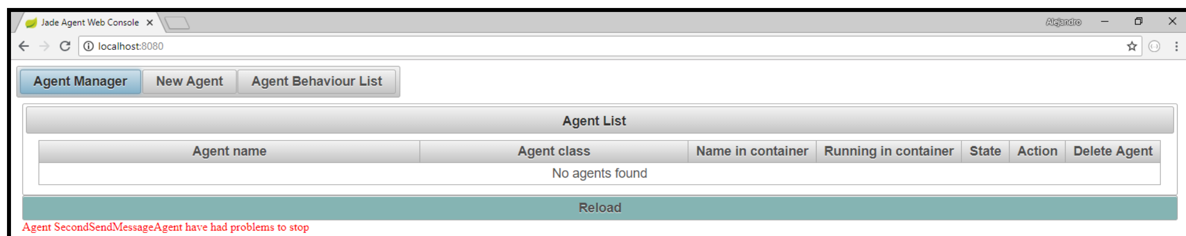

Figura 6.40 Página principal tras apagar la API e intentar detener el primer agente

6.9 Capa Controller: AgentController y agent-form.xhtml

Este par controlador-vista se encarga de gestionar la creación de nuevos agentes. El sistema de anotaciones de la clase es análogo al descrito para *AgentListController.* Tenemos cuatro variables de clase: una instancia inyectada de *AgentServiceInterface*, la lista de nombres de clase de agentes disponibles como modelos, la lista de nombres de agentes que ya existen, el nombre del agente que vamos a crear, el nombre de la clase que queremos utilizar como modelo y una variable para gestionar la lista de nombres ya seleccionados (solo a modo informativo). Asimismo, utilizamos el método *loadData* con las anotaciones explicadas en el apartado anterior para cargar estas dos listas a través del servicio *agentService (Figura 6.41).*

```
@Scope(value = "session")
@Component(value = "agentController")
@ELBeanName(value = "agentController")
@Join(path = "/agent", to = "/agent-form.xhtml")
public class AgentController {
    private AgentServiceInterface agentService;

    private List<String> agentClassNamesList;
    private List<String> agentBannedNamesList;
    private String agentName;
    private String selectedClassName;
    private String firstBannedName;

    @Deferred
    @RequestAction
    @IgnorePostback
    public void loadData() {
        agentClassNamesList = new ArrayList<>(agentService.loadAllAgentClassNames());
        agentBannedNamesList = new ArrayList<>(agentService.loadAllCreatedAgentsNames());
    }

    public AgentController(@Autowired AgentServiceInterface agentServiceIn) {
        this.agentService = agentServiceIn;
    }
```

Figura 6.41 Clase AgentController

El único dato de entrada de libre configuración por parte del usuario es el nombre del agente. El alcance del valor del resto de variables viene determinado por la propia aplicación. Por ello necesitamos asegurar que el nombre del agente cumple ciertos requisitos, como que no sea vacío y tenga una longitud mínima, que no supere una longitud máxima y solo utilice caracteres alfanuméricos. Gestionamos las restricciones a través de las anotaciones disponibles en la librería *javax.validation.constraints*. *@Size* permite determinar la longitud mínima y máxima de la cadena y el mensaje de error correspondiente. Del mismo modo *@Pattern* nos facilita el uso de expresiones regulares *(Figura 6.42)*

```
@Size(min=3, max=25, message="Error: Min 3 and max 25 characters")
@Pattern(regexp ="[a-zA-Z0-9]+",
        message="Error: String is not valid (only characters a-z A-Z 0-9)")
public String getAgentName() {
    return agentName;
}
```

Figura 6.42 Anotaciones para gestionar las restricciones del nombre del agente

Por último, tenemos el método para crear el agente con los datos seleccionados y un método para revisar que el nombre introducido no coincida con uno de los ya existentes. Si coincide el botón crear se desactiva hasta que no se introduzca otro

nombre distinto. Si el nombre pasa todos los filtros, se crea el nuevo agente y volvemos a la página principal *(Figura 6.43).*

```
public String create(){
        agentService.createAgent(selectedClassName, agentName);
        return "/agent-list.xhtml?faces-redirect=true";
}

public boolean checkName() {
    boolean isNameCorrect = true;
    if (agentBannedNamesList.contains(getAgentName())) {
        isNameCorrect = false;
    }
    return isNameCorrect;
}
```

Figura 6.43 Métodos create y checkName

En cuanto a la vista *agent-form.xhtml* es básicamente una composición con *toolbar.xhtml* en la que se define un formulario con tres campos: el campo nombre, del tipo *inputText* y obligatorio, un menú con los nombres que no puedes elegir ya que han sido asignados previamente a otros agentes y la lista de clases que puedes utilizar como modelo para crear tu agente, también obligatorio. También tenemos un mensaje de error concreto para el campo nombre: se mostrarán los errores de campo obligatorio, de longitud incorrecta y de no estar formado únicamente por caracteres alfanuméricos. Además, a través de AJAX (*Asynchronous Javascript* and *XML*) podemos escuchar el evento *keyup* (cuando dejas de presionar una tecla) para actualizar el estado del botón crear (se activa o desactiva dependiendo de si el nombre introducido se encuentra en la lista de nombres ya utilizados), el mensaje de error y el propio campo de forma dinámica *(Figura 6.44).*

```
<h:form id="agentForm">
    <p:panel id="panelForm" header="Create a new agent" style="text-align: center;">
        <h:panelGrid id="panelGrid" columns="3">
            <p:outputLabel for="name" value="Name:" />
            <p:inputText id="name" value="#{agentController.agentName}" required="true"
                         requiredMessage="Please enter the agent name">
                <f:ajax event="keyup" render="createButton @this nameMessage"/>
            </p:inputText>
            <p:message for="name" id="nameMessage" errorStyle="color:red;"/>
            <p:outputLabel for="bannedNames" value="Banned Names (information):" />
            <p:selectOneMenu id="bannedNames" value="#{agentController.firstBannedName}">
                <f:selectItems value="#{agentController.agentBannedNamesList}" var="selectedBannedName"
                               itemLabel="#{selectedBannedName}" itemValue="#{selectedBannedName}" />
            </p:selectOneMenu>
            <p:message for="bannedNames"/>
            <p:outputLabel for="class" value="Class:" />
            <p:selectOneMenu id="class" value="#{agentController.selectedClassName}"
                             required="true" requiredMessage="Please select a class">
                <f:selectItems value="#{agentController.agentClassNamesList}" var="selectedClassName"
                               itemLabel="#{selectedClassName}" itemValue="#{selectedClassName}" />
            </p:selectOneMenu>
            <p:message for="class"/>
        </h:panelGrid>
        <p:commandButton id="createButton" value="Create Agent"
                         disabled="#{!agentController.checkName()}"
                         action="#{agentController.create()}" >
        </p:commandButton>
    </p:panel>
</h:form>
```

Figura 6.44 Vista agent-form.xhtml

A continuación vamos a mostrar los distintos escenarios que pueden ocurrir en el caso de uso *Crear Agente*. Si el campo se encuentra vacío vemos el mensaje de error informándonos que el campo es obligatorio, si la longitud es menor que tres caracteres o mayor que veinticinco nos informa de este error, si incluimos algún carácter no alfanumérico nos lo especifica también y si finalmente elegimos un nombre válido pero ya está siendo utilizado por otro agente, el botón crear se deshabilita de forma automática. Por defecto se selecciona la primera clase de agente disponible, por lo que no hay ningún caso de error correspondiente a este campo. Con todos los campos validados, al pulsar el botón crear se crea un nuevo agente y volvemos a la página principal, donde podemos verlo reflejado. En caso de que hubiera algún problema en este último paso simplemente el agente no aparecería en la lista de agentes, ya que no habría sido registrado con éxito en la API.

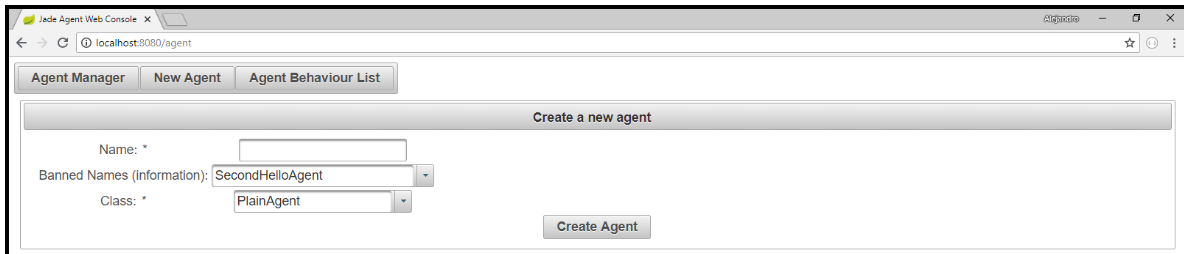

Figura 6.45 Vista agent-form.xhtml inicial

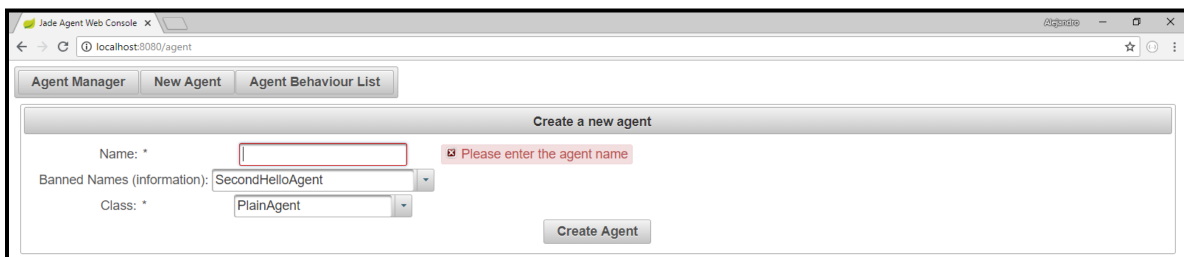

Figura 6.46 Vista agent-form.xhtml con nombre vacío

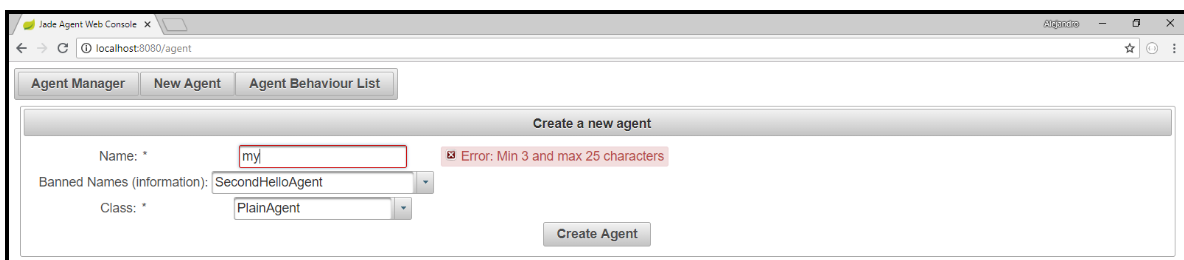

Figura 6.47 Vista agent-form.xhtml con nombre de longitud insuficiente

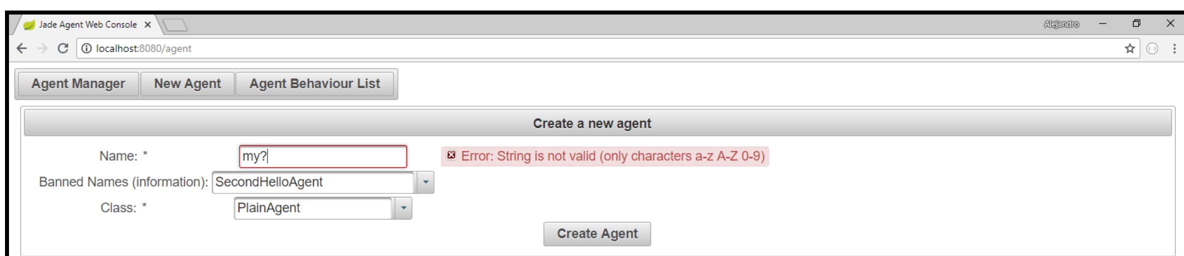

Figura 6.48 Vista agent-form.xhtml con nombre formado por algún carácter no alfanumérico

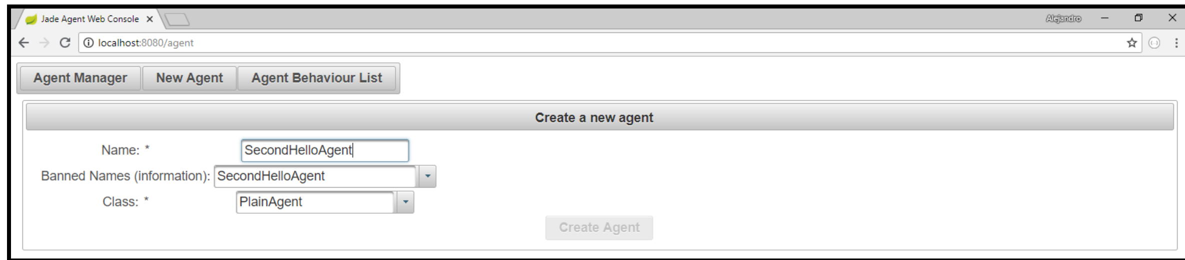

Figura 6.49 Vista agent-form.xhtml con nombre incluido en la lista de nombres ya seleccionados

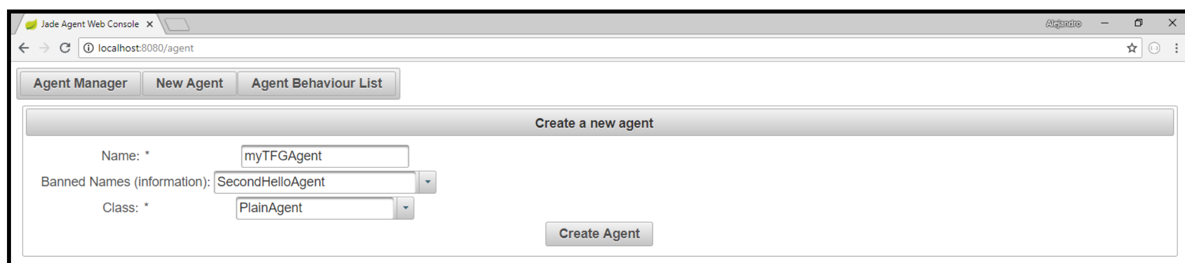

Figura 6.50 Vista agent-form.xhtml con todos los campos validados

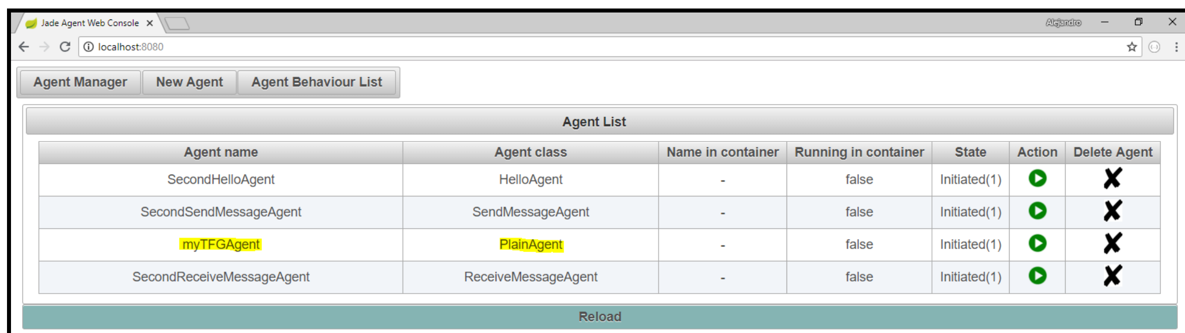

Figura 6.51 Página principal con el nuevo agente creado

6.10 Capa Controller: AgentBehaviourListController, agent-behaviour-list.xhtml y behaviour-list.xhtml

Este último controlador gestiona dos vistas. Por un lado, junto a *agent-behaviour-list.xhtml* se encarga tanto de mostrar la lista de comportamientos vinculados a los distintos agentes como sus detalles más importantes. Por otro, junto a *behaviour-list.xhtml* se encarga de gestionar los comportamientos del agente seleccionado: añadir nuevos comportamientos, eliminarlos y reiniciar los que tiene vinculado en ese preciso momento. Como el agente es seleccionado por el usuario en la anterior

vista, utilizamos el mismo controlador para poder almacenar el agente seleccionado y la lista de comportamientos que tiene vinculado.

El sistema de anotaciones de la clase es análogo al descrito para los dos controladores previos. Además de la instancia inyectada de *AgentServiceinterface* tenemos nueve variables de clase: cuatro utilizadas con la primera vista y cinco utilizadas por la segunda. Para la primera tenemos la lista de agentes, la lista formada por el nombre del agente y los comportamientos que tiene vinculado, el mensaje de éxito y el mensaje de error. Para la segunda tenemos la lista de comportamientos disponibles, el agente seleccionado, la lista de comportamientos vinculados al agente seleccionado (modificable), otra lista para los comportamientos vinculados al agente seleccionado que no es modificable y la lista de comportamientos que están disponibles para ser añadidos al agente seleccionado (un agente solo puede tener vinculado una instancia de cada comportamiento). Utilizamos el método *loadData* con las anotaciones explicadas en el apartado anterior para cargar las listas a través del servicio *agentService (Figura 6.52).*

```
@Scope(value = "session")
@Component(value = "agentBehaviourListController")
@ELBeanName(value = "agentBehaviourListController")
@Join(path = "/agent/behaviour/list", to = "/agent-behaviour-list.xhtml")
public class AgentBehaviourListController {
    @Autowired
    private AgentServiceInterface agentService;

    private Map<String, Agent> agents;
    private List<JsonBehavioursModel> agentAndBehavioursList;
    private String successfulMessage;
    private String errorMessage;

    //all behaviours I can add to agents (factory behaviours in the micro-service)
    private List<Behaviour> behavioursAvailable;
    private Agent agentSelected;
    //behaviours agent selected has attached
    private List<Behaviour> agentSelectedBehaviours;
    //behaviours agent selected has attached. Original, not modified by interface
    //so we know which are the new behaviours we have to add via API
    private List<Behaviour> agentOriginalSelectedBehaviours;
    //behaviours that can be added to selected agent (from behavioursAvailable)
    private List<Behaviour> agentSelectedBehavioursAvailable;

    @Deferred
    @RequestAction
    @IgnorePostback
    public void loadData(){
        agents = agentService.loadAllCreatedAgentsInMap();
        behavioursAvailable = new ArrayList<>(agentService.loadBehavioursAvailable());
        agentAndBehavioursList = new ArrayList<>(agentService.loadAllAgentsAndItsBehaviours());
    }
```

Figura 6.52 Clase AgentBehaviourListController

En definitiva, tenemos un duplicado tanto de la lista de comportamientos vinculados como la de comportamientos disponibles. La lista de comportamientos disponibles para el agente es subconjunto de la lista de comportamientos disponibles en la que se descartan aquellos comportamientos que ya tiene vinculados. En la interfaz podemos ir eliminando comportamientos y se irán añadiendo a esta lista, no a la original con todos los comportamientos que pueden utilizar los distintos agentes. De forma análoga, la lista modificable de comportamientos vinculados al agente irá añadiendo comportamientos a través de la interfaz, mientras que la original no varía. Esto permite ver el flujo de añadir y eliminar comportamientos de manera más realista sin que aparezcan comportamientos duplicados en ningún lado y, además, una vez aceptado los cambios solo tenemos que ver qué comportamientos nuevos no están en la lista de comportamientos vinculados original para vincularlos a través

de la API, y qué comportamientos estaban en la lista original y ya no están, para desvincularlos a través de la API. Esto evita la sobrecarga de añadir y/o eliminar en el momento los distintos comportamientos en vez de esperar a que el usuario realice todo el ajuste y se lleven a cabo las operaciones estrictamente necesarias.

La creación de estas listas se produce en el método *manageBehaviours (Figura 6.53)*, el nexo de unión que da paso a la vista *behaviour-list.xhtml.* Además, al igual que en el controlador *AgentListController* tenemos la posibilidad de recargar la página para visualizar los datos actualizados.

```
public String reloadData(){
    displaySuccessfulMessage("Agents have been reloaded from server successfully");
    return "/agent-behaviour-list.xhtml?faces-redirect=true";
}

public String manageBehaviours (JsonBehavioursModel jsonBehavioursModel){
    agentSelected = agents.get(jsonBehavioursModel.getAgentName());
    agentSelectedBehaviours = jsonBehavioursModel.getBehaviourList();
    agentOriginalSelectedBehaviours = new ArrayList<>(agentSelectedBehaviours);
    agentSelectedBehavioursAvailable = new ArrayList<>();
    for (Behaviour behaviour : behavioursAvailable){
        //contains makes use of our override equals on behaviour class
        if (!agentSelectedBehaviours.contains(behaviour)){
            agentSelectedBehavioursAvailable.add(behaviour);
        }
    }
    resetMessages();
    return "/behaviour-list.xhtml?faces-redirect=true";
}
```

Figura 6.53 Métodos reloadData y manageBehaviours

Antes de continuar con la descripción de los métodos utilizados por el controlador vamos a explicar la vista *agent-behaviour-list.xhtml.* Como en los casos anteriores, volvemos a tener una composición con *toolbar.xhtml*. Se define una tabla general para la lista formada por el nombre de un agente concreto y sus comportamientos vinculados, en la que la primera columna es el nombre del agente, las dos siguientes son tablas con los valores de los distintos comportamientos y la cuarta un acceso para gestionar los comportamientos del agente seleccionado. La segunda columna muestra el nombre de todos los comportamientos y la tercera muestra (en el mismo orden) el estado de ejecución de todos ellos *(Figura 6.54).*

```
<p:dataTable id="table"
             var="agent"
             value="#{agentBehaviourListController.agentAndBehavioursList}"
             emptyMessage="No agents found">

    <p:column style="width: 25%; text-align: center;">
        <f:facet name="header">Agent name</f:facet>
        <h:outputText value="#{agent.agentName}"/>
    </p:column>

    <p:column style="width: 60%; text-align: center;">
        <f:facet name="header">Behaviours</f:facet>
        <h:dataTable value="#{agent.behaviourList}" var="behaviour">
            <h:column>
                <h:outputText value="#{behaviour.behaviourName}"/>
            </h:column>
        </h:dataTable>
    </p:column>

    <p:column style="width: 25%; text-align: center;">
        <f:facet name="header">Behaviours Execution State</f:facet>
        <h:dataTable value="#{agent.behaviourList}" var="behaviour">
            <h:column>
                <h:outputText value="#{behaviour.executionState}"/>
            </h:column>
        </h:dataTable>
    </p:column>

    <p:column style="width: 20%; text-align: center;">
        <f:facet name="header">Behaviours Restart Counter</f:facet>
        <h:dataTable value="#{agent.behaviourList}" var="behaviour">
            <h:column>
                <h:outputText value="#{behaviour.restartCounter}"/>
            </h:column>
        </h:dataTable>
```

Figura 6.54 Extracto de la vista agent-behaviour-list.xhtml

El acceso a la segunda vista es a través de un botón idéntico al utilizado para eliminar un agente en la página principal. Asimismo, la funcionalidad para recargar la página y mostrar los textos informativos es idéntica, por lo que no ahondaremos más en este punto. Podemos visualizar la página que implementa el listado de comportamientos y visualización de detalles en las siguientes figuras.

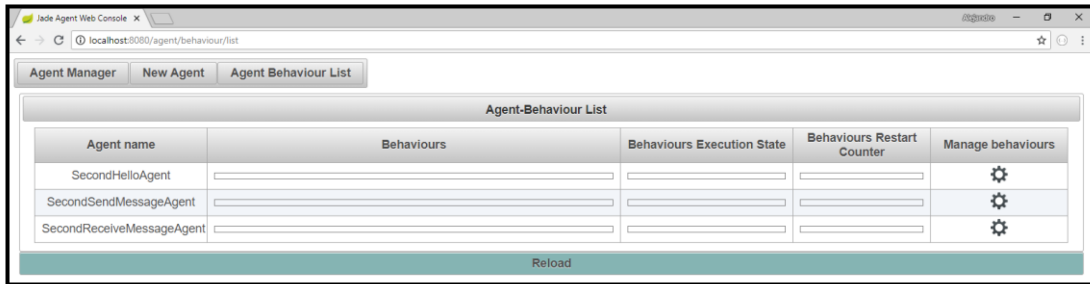

Figura 6.55 Vista agent-behaviour-list.xhtml sin comportamientos

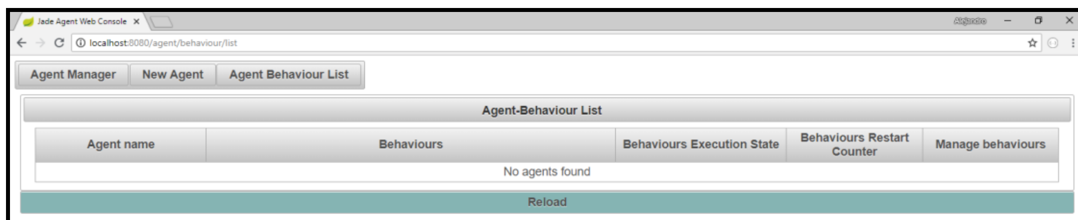

Figura 6.56 Vista agent-behaviour-list.xhtml sin agentes

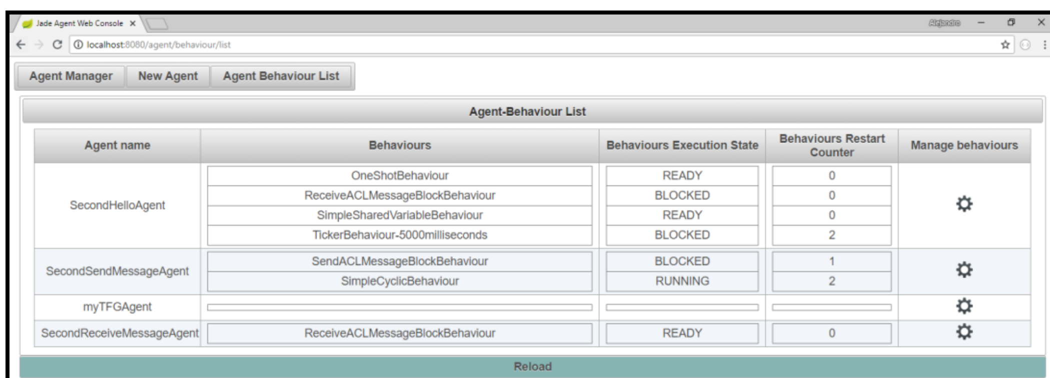

Figura 6.57 Vista agent-behaviour-list.xhtml con distintos agentes y comportamientos

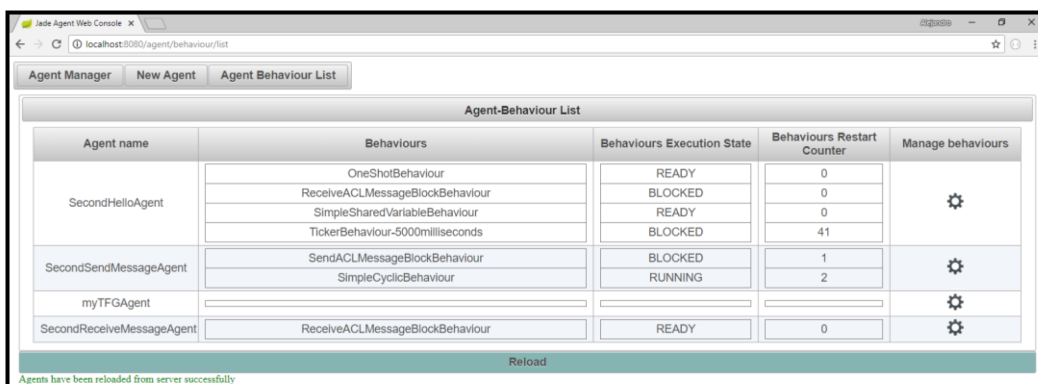

Figura 6.58 Vista recargada. El comportamiento TickerBehaviour se ha ejecutado ya 41 veces

Vamos a continuar con la descripción de los métodos utilizados por el controlador para la vista *behaviour-list.xhtml: addBehaviours, removeBehaviours, resetBehaviours, acceptChanges* e *isBehaviourAttachedToAgent.* Como hemos explicado anteriormente, tenemos por un lado la lista de comportamientos vinculados y por otra la lista de comportamientos disponibles. Podemos seleccionar un número indefinido de comportamientos de cada tabla para pasarlos a la otra a través de los botones *Add* y *Remove (Figura 6.59)*. Este flujo no tiene ningún resultado efectivo hasta que no se pulsa el botón aceptar y se acepta el diálogo de confirmación *(Figura 6.60)*. Asimismo, todos los comportamientos vinculados pueden ser reseteados con el botón que aparece junto a su nombre. Solo aquellos comportamientos que están en la tabla de vinculados y realmente estén vinculados al agente en el contenedor JADE tendrán el botón disponible *(Figura 6.61).*

```
public String addBehaviours(){
    List<Behaviour> iterateList =
            new ArrayList<>(agentSelectedBehavioursAvailable);
    for (Behaviour b : iterateList){
        if (b.getChecked()){
            agentSelectedBehavioursAvailable.remove(b);
            b.setChecked(false);
            agentSelectedBehaviours.add(b);
        }
    }
    displaySuccessfulMessage("Behaviours added successfully");
    return "/behaviour-list.xhtml?faces-redirect=true";
}

public String removeBehaviours(){
    List<Behaviour> iterateList = new ArrayList<>(agentSelectedBehaviours);
    for (Behaviour b : iterateList){
        if (b.getChecked()){
            agentSelectedBehaviours.remove(b);
            if (behavioursAvailable.contains(b)){
                b.setChecked(false);
                agentSelectedBehavioursAvailable.add(b);
            }
        }
    }
    displaySuccessfulMessage("Behaviours removed successfully");
    return "/behaviour-list.xhtml?faces-redirect=true";
}
```

Figura 6.59 Métodos addBehaviours y removeBehaviours

```java
public String acceptChanges(){
    List<Behaviour> behavioursAdded = new ArrayList<>();
    List<Behaviour> behavioursRemoved = new ArrayList<>();

    for (Behaviour b : agentSelectedBehaviours){
        if (!agentOriginalSelectedBehaviours.contains(b)){
            behavioursAdded.add(b);
        }
    }

    for (Behaviour b : agentOriginalSelectedBehaviours){
        if (!agentSelectedBehaviours.contains(b)){
            behavioursRemoved.add(b);
        }
    }
    agentService.addBehavioursToAgent(agentSelected, behavioursAdded);
    agentService.removeBehavioursFromAgent(agentSelected, behavioursRemoved);

    resetMessages();
    return "/agent-behaviour-list.xhtml?faces-redirect=true";
}
```

Figura 6.60 Método acceptChanges

```java
public String resetBehaviour(Behaviour behaviour){
    if (agentService.resetBehaviour(agentSelected, behaviour)){
        displaySuccessfulMessage("Behaviour "
                + behaviour.getBehaviourName()
                + " has been reset successfully");
    }else{
        displayErrorMessage("Behaviour "
                + behaviour.getBehaviourName()
                + " have had problems to be reset");
    }
    return "/behaviour-list.xhtml?faces-redirect=true";
}

//check if the behaviour is attached to agent in JADE container
public boolean isBehaviourAttachedToAgent(Behaviour behaviour){
    return agentOriginalSelectedBehaviours.contains(behaviour);
}
```

Figura 6.61 Métodos resetBehaviour y isBehaviourAttachedToAgent

Ahora pasamos a la vista *behaviour-list.xhtml*, a la que accedemos a través del botón *manage behaviours.* Como todas las vistas del sistema, también incorporamos el componente *toolbar.xhtml.* En el panel tenemos dos tablas en las que cada fila es un comportamiento con un *checkBox* que permite seleccionarlo. Vemos que en la primera tabla tenemos también la opción de reiniciar el comportamiento si éste se encuentra vinculado al agente *(Figura 6.62).*

```
<p:panel style="width: 100%; float:left; text-align: center;" header="'#{agentBehaviourListController.agentSelected.nickname}'">
    <p:dataTable id="behavioursAttachedTable"
                 var="behaviour" value="#{agentBehaviourListController.agentSelectedBehaviours}"
                 emptyMessage="No behaviours attached found"
                 style="width: 50%; float:left;">
        <p:column style="width: 60%; text-align: center;">
            <f:facet name="header">Behaviours Attached</f:facet>
            <h:outputText value="#{behaviour.behaviourName}"/>
            <p:commandLink action="#{agentBehaviourListController.resetBehaviour(behaviour)}"
                           rendered="#{agentBehaviourListController.isBehaviourAttachedToAgent(behaviour)}">
                <p:graphicImage style="width: 25px; float:left"
                                title="Reset Behaviour"
                                value="/reset_icon.png"/>
            </p:commandLink>
            <p:selectBooleanCheckbox  style="width: 20px; float:right" value="#{behaviour.checked}" />
        </p:column>
    </p:dataTable>

    <p:dataTable id="behavioursAvailableTable"
                 var="behaviour" value="#{agentBehaviourListController.agentSelectedBehavioursAvailable}"
                 emptyMessage="No behaviours available found"
                 style="width: 50%; float:left;">
        <p:column style="width: 60%; text-align: center;">
            <f:facet name="header">Behaviours Available</f:facet>
            <h:outputText value="#{behaviour.behaviourName}"/>
            <p:selectBooleanCheckbox style="width: 20px; float:right" value="#{behaviour.checked}" />
        </p:column>
    </p:dataTable>
</p:panel>
```

Figura 6.62 Tablas de comportamientos vinculados y disponibles

Los botones para añadir, eliminar y aceptar los cambios son comunes a todos los comportamientos y se encuentran al final del formulario. Desplegamos los mensajes informativos del mismo modo que el resto de vistas, por lo que no ahondaremos más en el tema *(Figura 6.63).*

```
        <p:commandButton id="remove-behaviour-button"
                         style="width: 49.8%; float:left;"
                         action="#{agentBehaviourListController.removeBehaviours}"
                         value="Remove"/>
        <p:commandButton id="add-behaviour-button"
                         style="width: 49.8%; float:right;"
                         action="#{agentBehaviourListController.addBehaviours}"
                         value="Add"/>
        <p:commandButton id="accept-changes-behaviour-button"
                         style="width: 100%; background: #85b4b3;"
                         onclick="return confirm('Are you sure you want to change the behaviours from this agent?');"
                         action="#{agentBehaviourListController.acceptChanges}"
                         value="Accept Changes"/>
</h:form>
<h:outputText value="#{agentBehaviourListController.successfulMessage}" style="color:green;"
              rendered="#{agentListController.successfulMessage != null}"/>
<h:outputText value="#{agentBehaviourListController.errorMessage}" style="color:red;"
              rendered="#{agentListController.errorMessage != null}"/>
```

Figura 6.63 Botones añadir, eliminar y aceptar junto a los mensajes informativos

A continuación vamos a mostrar los distintos escenarios que pueden darse en los casos de uso *añadir comportamiento a agente, eliminar comportamiento de agente* y *reiniciar comportamiento de agente.* En el caso de no haber agentes en el sistema no podremos acceder a la vista.

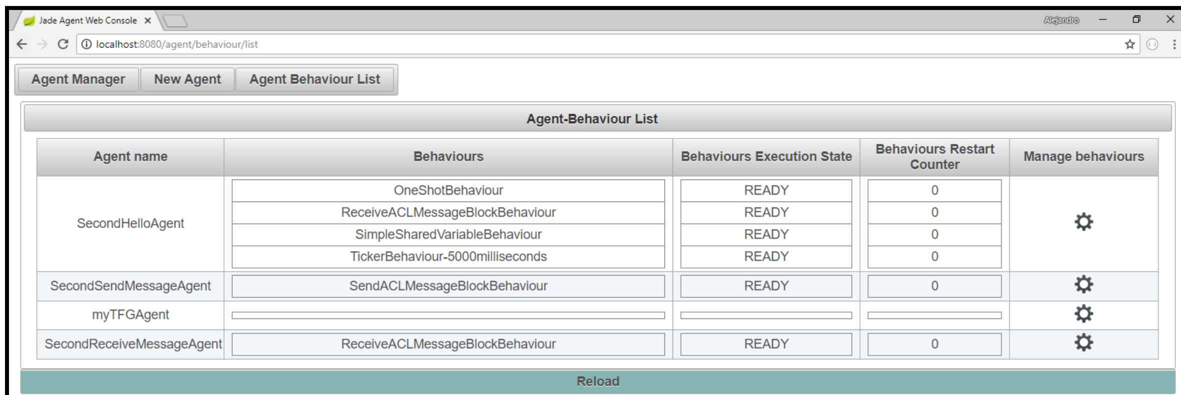

Figura 6.64 Vista agent-behaviour-list.xhtml. Agente myTFGAgent sin comportamientos

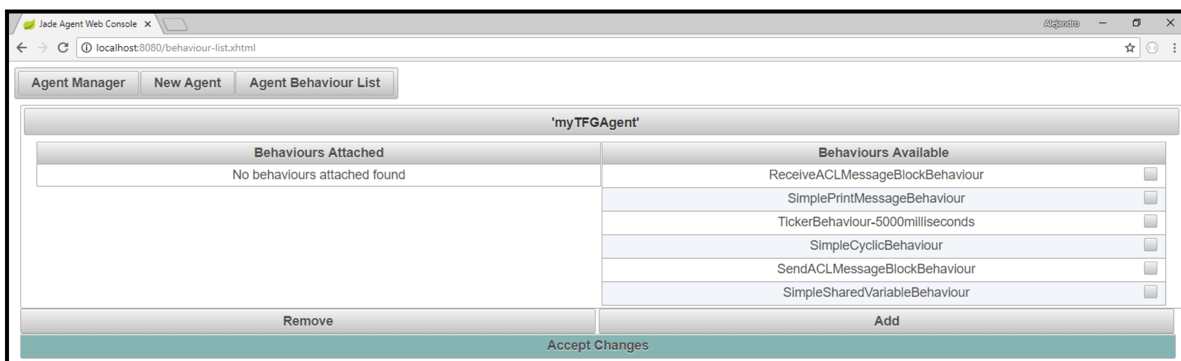

Figura 6.65 Vista behaviour-list.xhtml. Agente myTFGAgent sin comportamientos vinculados

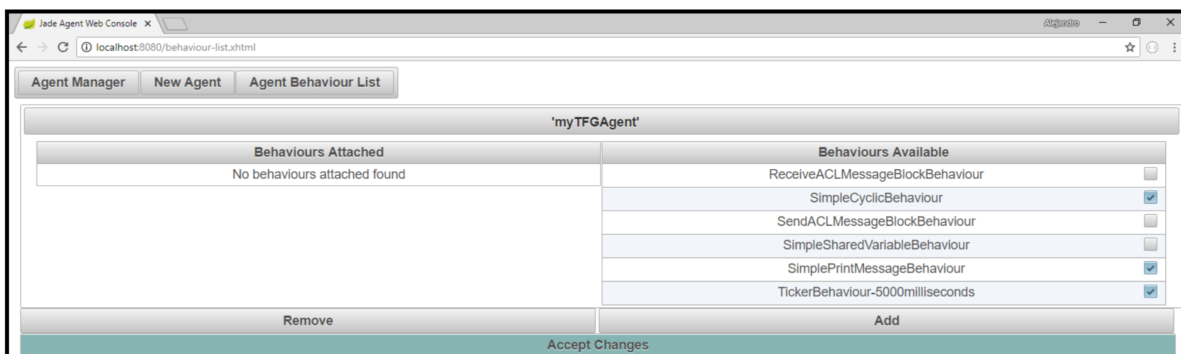

Figura 6.66 Vista behaviour-list.xhtml. Selecciono tres agentes disponibles

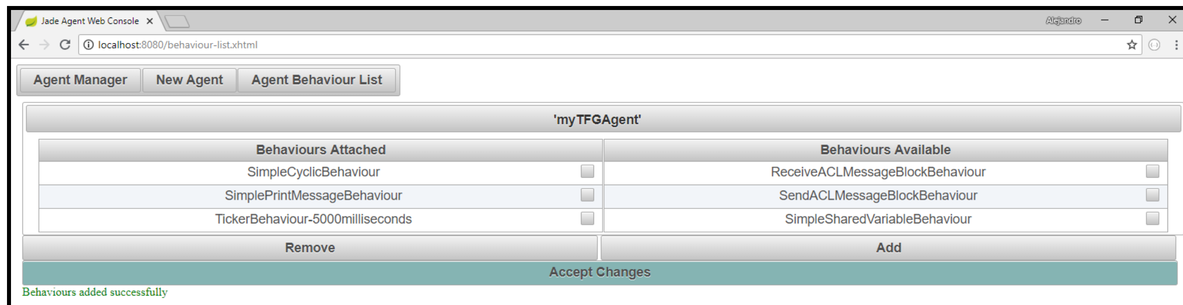

Figura 6.67 Vista behaviour-list.xhtml. Agente myTFGAgent sin comportamientos vinculados

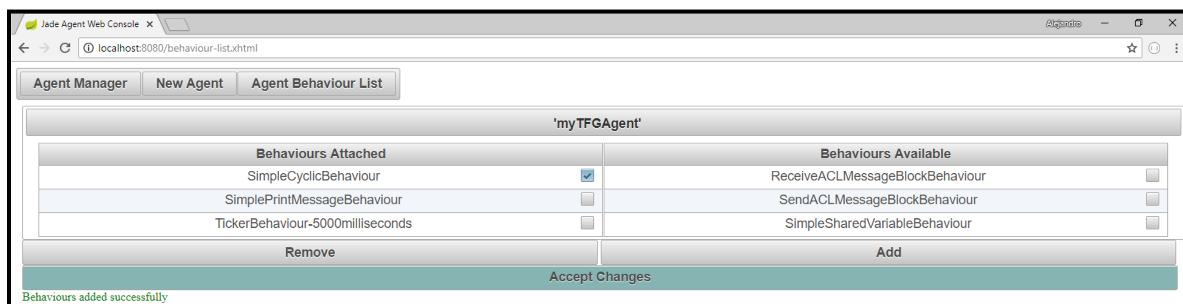

Figura 6.68 Vista behaviour-list.xhtml. Agente myTFGAgent con tres comportamientos después de añadirlos desde la tabla de la derecha

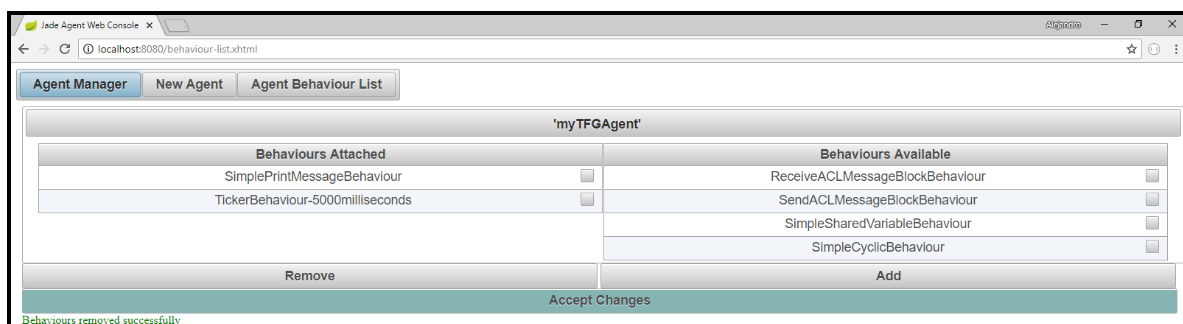

Figura 6.67 Vista behaviour-list.xhtml. Agente myTFGAgent con dos comportamientos después de haber eliminado uno

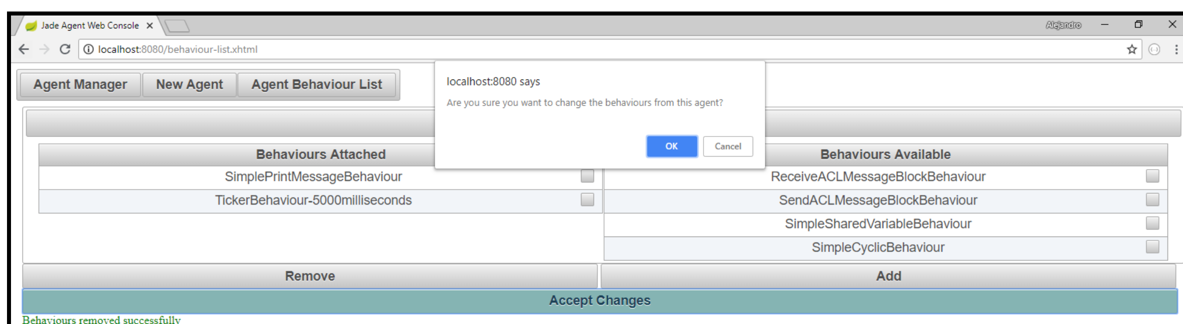

Figura 6.68 Vista behaviour-list.xhtml. Dialogo para confirmación de los cambios realizados

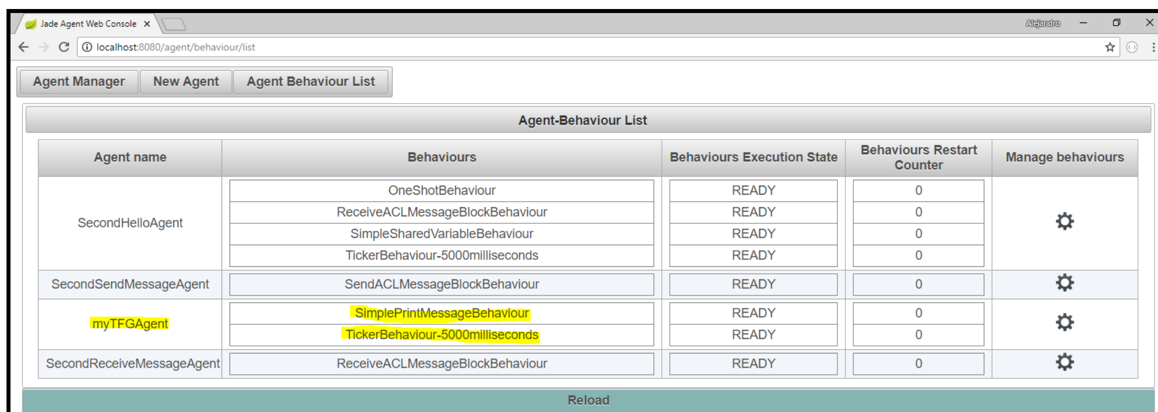

Figura 6.69 Vista agent-behaviour-list.xhtml. Agente myTFGAgent con dos comportamientos añadidos

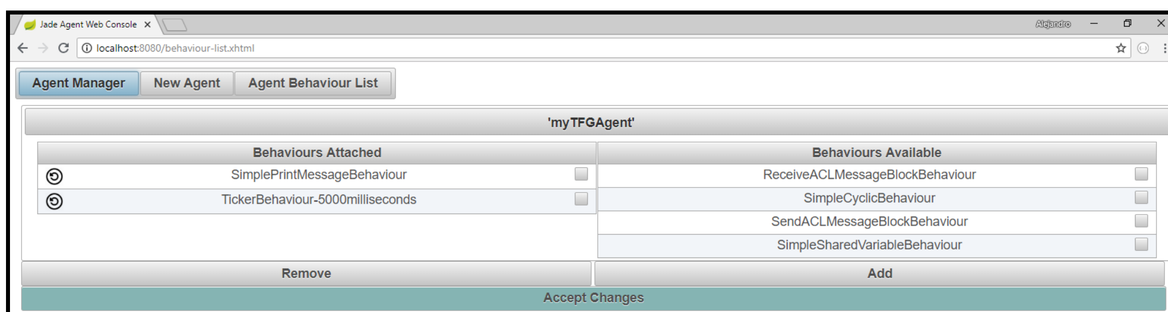

Figura 6.70 Vista behaviour-list.xhtml. Los comportamientos vinculados al agente tienen el botón de reinicio disponible

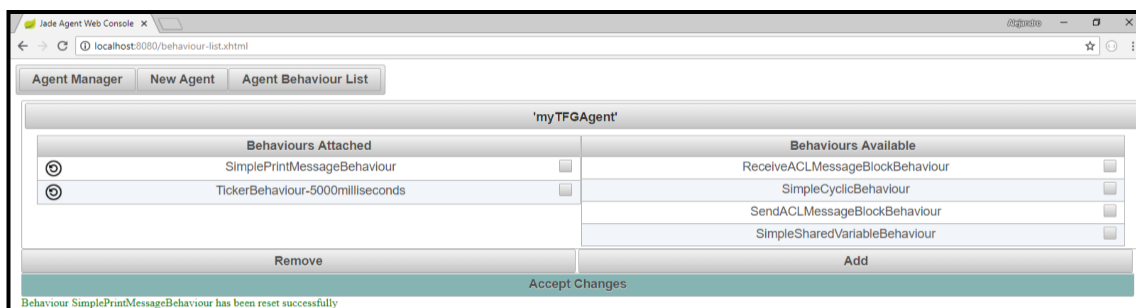

Figura 6.71 Vista behaviour-list.xhtml. El comportamiento SimplePrintMessageBehaviour ha sido reiniciado con éxito

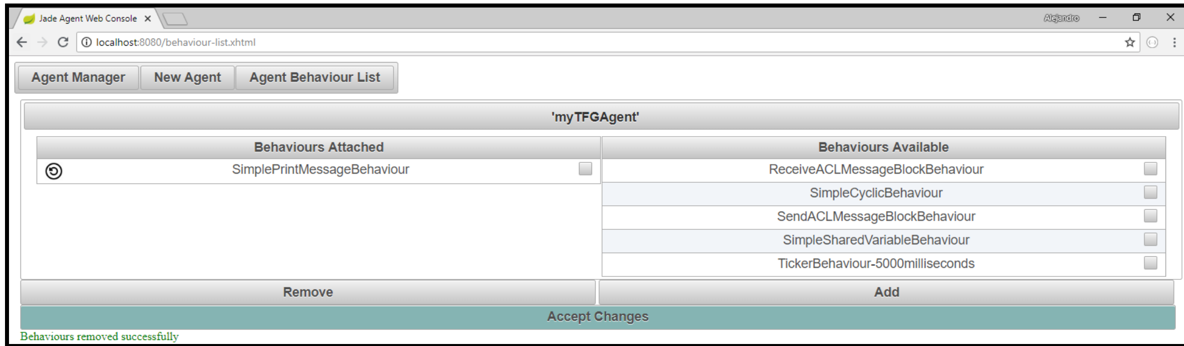

Figura 6.72 Vista behaviour-list.xhtml. El comportamiento tickerBehaviour ha sido eliminado

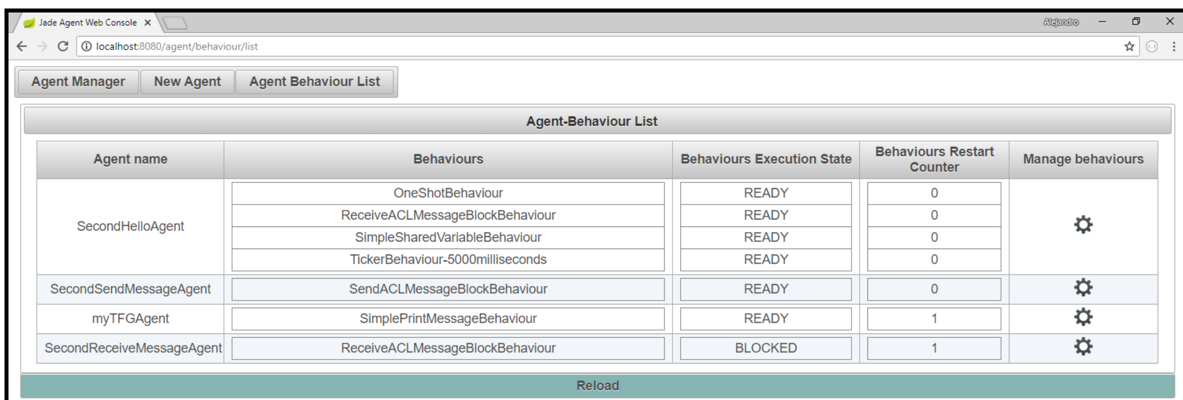

Figura 6.73 Vista agent-behaviour-list.xhtml. El valor del contador de reinicios del comportamiento SimplePrintMessageBehaviour es uno y el comportamiento TickerBehaviour ya no aparece

7. CONCLUSIONES Y LÍNEAS FUTURAS

Nos encontramos con un TFG que tiene una fuerte componente de investigación. JADE es un middleware y framework de código abierto que es utilizado a diario en multitud de proyectos. El denominador común es el beneficio que encuentran en el uso de sistemas multi-agentes para cubrir necesidades que surgen de su lógica de negocio. Los sistemas multi-agentes permiten desarrollar sistemas desacoplados, autónomos y flexibles, pero sin JADE emplearíamos multitud de recursos en desarrollar un sistema multi-agente para Java que implementase los estándares de la industria, como FIPA, el estándar de SMA establecido por la IEEE *Computer Society.* Y una vez implementado necesitaríamos que pudiesen cooperar con agentes desarrollados por terceros. Esta enorme cantidad de recursos hace de JADE una herramienta muy potente y valorada, pero en la era web seguimos anclado en las aplicaciones de escritorio, cuando la vertiente del software colaborativo se ha visto potenciada en gran magnitud debido a la conexión a lo ancho del globo. ¿Cómo puedo lograr que mis agentes JADE funcionen en mi aplicación web empresarial? ¿Cómo los puedo gestionar si la consola solo se encuentra disponible en entorno de escritorio? ¿Tengo que crear dos agentes totalmente distintos con la lógica de negocio duplicada solo porque son entornos incompatibles? Son solo algunas de la multitud de preguntas e incógnitas que el desarrollador de agentes debe enfrentar al enfocar un proyecto sobre una plataforma web empresarial. Este proyecto nace para suplir este hueco entre ambas plataformas y proporcionar un envoltorio que, por un lado, permita migrar y desarrollar nuevos agentes de una forma metódica muy cercana al desarrollo en entornos de escritorios y, por otro, ofrecer una plataforma web con la que gestionar todos tus agentes de forma remota.

A lo largo de toda la memoria se ha explicado con minucioso detalle las numerosas decisiones de diseño e implementación, el objetivo de cada componente, como interactúan entre ellos, la arquitectura del framework y como tras distintas etapas de refinación hemos alcanzado un código legible que ha intentado adaptarse a los principios SOLID. Hemos elaborado una capa sobre la librería JADE, una extensión que sin necesidad de modificar ningún aspecto de la estructura e implementación de

la librería ha conseguido suministrar esta compatibilidad. Tanto para aplicaciones empresariales web con JavaEE como aplicaciones desarrolladas con Spring MVC, con leves diferencias en el uso de anotaciones para la inyección de los agentes, utilizando los recursos y librerías que cada entorno te proporciona.

Nuestros agentes no heredan directamente de la clase *Agent* sino de una extensión llamada *SpringAgent* (o *EJBAgent* para JavaEE, clases con la misma implementación) que hereda de *Agent* y aporta esta abstracción. La única diferencia final para el desarrollador es la especificación de un método extra *getInstance* con una implementación trivial. La lógica de negocio particular se implementa exactamente del mismo modo que en Java SE.

Además de esta extensión hemos conseguido desacoplar los comportamientos de los distintos agentes: permitimos manejar las instancias de manera aislada y que sean vinculadas y desvinculadas de sus agentes en cualquier momento. Ofrecemos un control total sobre cualquier comportamiento de un agente concreto. Y esto solo por nombrar alguno de los principales beneficios del framework desarrollado.

El desarrollo de la consola web también abre un gran espectro de posibilidades. La estructura dividida de cliente web y API REST evoluciona el sistema con una orientación hacia los micro-servicios. La estructura de la API permite que pueda incluirse en cualquier aplicación de terceros que utilice Spring MVC con Spring Boot añadiendo los componentes *jade* y *api* de la capa *Service, agentModel* y *responseMessageModel* de la capa *Model* y el controlador *JadeRestController.* A partir de aquí solo se necesitan inyectar los agentes y comportamientos que quieren gestionarse de forma remota como servicios en el controlador.

Por otro lado, nuestro cliente web puede tomarse como un ejemplo de las posibilidades que ofrece la API REST, ya que no hemos usado siquiera todas las funciones desarrolladas. Nuestro cliente puede gestionar completamente la creación, eliminación y ciclo de vida de los distintos agentes, al igual que manejar y asignar comportamientos de forma individualizada. Pero cada organización puede desarrollar el cliente que más beneficios le aporte, con las funcionalidades y extensiones que considere oportunas, ya sea un cliente web o un cliente móvil. Cualquier dispositivo capaz de realizar y consumir peticiones HTTP es candidato a

poder usarse como cliente; y sin interfaz ninguna, en sistemas con menos recursos gráficos pueden utilizarse clientes como Postman y gestionar los agentes a bajo nivel de forma rápida y eficiente.

Por todo esto consideramos que este proyecto de código abierto puede ser bastante útil y soluciona de manera general los problemas planteados. Ningún software es perfecto, esa algo que la experiencia a lo largo de los años remarca a todos los desarrolladores, por lo que a pesar de todas las pruebas realizadas con distintos tipos de agentes y situaciones aún tenemos que enfrentar el sistema ante entornos reales, ante sistemas más complejos que alcancen los casos aislados que pudiesen producir un comportamiento erróneo. Este es otro motivo por el que se ha descrito de forma tan minuciosa el proyecto: la extensión y mejora por parte de terceros. Los proyectos de código abierto crecen, se mantienen y mejoran gracias a la comunidad, y es algo de lo que nos beneficiamos todos. Con este fin, mantenemos la licencia LGPL (*Lesser General Public License*) versión 2 que especifica JADE en http://jade.tilab.com.

Llegados a este punto hay varias líneas futuras a considerar:

- Implementar un historial de acciones en el cliente web, tanto para la creación, eliminación y gestión del ciclo de vida de los distintos agentes como para los comportamientos, para visualizar qué comportamientos hemos añadido a qué agente, qué comportamiento hemos eliminado o cuáles hemos reiniciado.
- Desarrollo de un cliente móvil con Android, IOs o una aplicación híbrida, como puede ser Ionic o ReactNative.
- Ampliar el framework para el uso de varios contenedores distintos, permitiendo elegir dónde será ejecutado tu agente. Actualmente utilizamos un único contenedor, pero dada la arquitectura del framework esta modificación se podría desarrollar a corto-medio plazo. En primera instancia sería suficiente con gestionar una lista de contenedores y que cada agente tuviese un parámetro para identificar el contenedor al que ha sido vinculado.
- Añadir perfiles de usuario a nuestra consola web. Cada usuario solo podría visualizar y gestionar los agentes que ha creado. Se podría establecer una

serie de permisos para permitir la existencia de administradores en el sistema. Esto daría lugar a una nueva serie de requisitos de registro, inicio y cierre de sesión. Necesitaríamos una base de datos donde almacenar el nombre y la contraseña (debidamente cifrada) de los usuarios, y podríamos implementarlo de forma aislada para nuestra consola web o de forma genérica en nuestra API REST. Cada opción tiene sus beneficios e inconvenientes que habría que valorar. La API REST, al no tener estado, tendría que hacer uso de *bearer tokens*, y para mantener el sistema lo más desacoplado posible podríamos utilizar una base de datos en la nube, como *MongoDB Atlas.* Esto sería un cambio de mayor complejidad que debería estimarse a largo plazo.

REFERENCIAS BIBLIOGRÁFICAS

http://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf

https://www.javacodegeeks.com/2014/11/java-annotations-tutorial.html

https://dzone.com/articles/how-annotations-work-java

http://programacionjade.wikispaces.com/

http://jade.tilab.com/doc/tutorials/jsp/JADE4JSP.html

https://javaee.github.io/javaserverfaces-spec/

http://www.academia.edu/9332879/Building_a_Web-bridge_for_JADE_agents

https://dzone.com/articles/how-annotations-work-java

https://dzone.com/articles/java-annotations-are-a-big-mistake?fromrel=true

https://sanaulla.info/2013/03/21/introduction-to-functional-interfaces-a-concept-recreated-in-java-8/

https://howtodoinjava.com/core-java/annotations/complete-java-annotations-tutorial/

https://dzone.com/articles/java-8-type-annotations

https://dzone.com/articles/cdi-di-p1

https://dzone.com/articles/cdi-di-p2

https://gerardnico.com/wiki/code/design_pattern/injection

https://gerardnico.com/wiki/lang/java/container

https://docs.jboss.org/weld/reference/latest/en-US/html/injection.html

http://www.andygibson.net/blog/tutorial/getting-started-with-jsf-2-0-and-cdi-in-jee-6-part-1/

https://netbeans.org/kb/docs/javaee/cdi-intro.html

https://martinsdeveloperworld.wordpress.com/2014/02/23/injecting-configuration-values-using-cdis-injectionpoint/

http://jade.tilab.com/pipermail/jade-develop/2008q3/012842.html

http://massapi.com/source/googlecode/27/76/277625054/traffic-jams-read-only/src/jade/core/mobility/AgentMobilityService.java.html#378

https://github.com/ubenzer/Agent-Based-Programming-Experiments-with-JADE/blob/master/lib/jade/src/jade/core/Agent.java

https://www.javabrahman.com/java-8/java-8-java-util-function-function-tutorial-with-examples/

https://stackoverflow.com/questions/29945627/java-8-lambda-void-argument

https://stackoverflow.com/questions/2186931/java-pass-method-as-parameter/19624032#19624032

http://www.java2s.com/Tutorials/Java/java.util.function/Consumer/index.htm

https://stackoverflow.com/questions/10889563/ejb-3-1-localbean-vs-no-annotation?noredirect=1&lq=1

https://stackoverflow.com/questions/34411371/is-it-possible-to-inject-ejb-implementation-and-not-its-interface-using-cdi?noredirect=1&lq=1

https://stackoverflow.com/questions/18744910/using-jsf-as-view-technology-of-spring-mvc

https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html#spring-introduction

https://stackoverflow.com/questions/19588214/spring-application-deployment-on-java-ee-servers

https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html#spring-introduction

https://spring.io/guides/gs/serving-web-content/

https://projects.spring.io/spring-framework/

https://spring.io/guides/gs/convert-jar-to-war/

https://spring.io/guides/gs/spring-boot/

https://spring.io/blog/2014/03/07/deploying-spring-boot-applications

https://spring.io/understanding/view-templates

https://github.com/acichon89/springmvcfacelets

https://dzone.com/articles/mvc-10-in-java-ee-8-getting-started-using-facelets

https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html

https://dzone.com/articles/developing-jsf-applications-with-spring-boot

https://docs.oracle.com/javaee/7/tutorial/jsf-facelets001.htm

https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#spring-web

http://www.baeldung.com/spring-mvc-tutorial

https://crunchify.com/simplest-spring-mvc-hello-world-example-tutorial-spring-model-view-controller-tips/

https://jaxenter.com/how-to-write-a-java-ee-application-using-spring-boot-and-docker-on-netbeans-ide-8-2-133200.html

https://stackoverflow.com/questions/29431579/spring-boot-app-does-not-deploy-on-glassfish-4-1

https://github.com/djdjalas/SpringBootIn50/tree/springboot-db-intergration

https://hashnode.com/post/spring-mvc-or-spring-boot-cj8egyxy901vcpgwu57m25mvx

https://hellokoding.com/spring-boot-hello-world-example-with-jsp/

https://hellokoding.com/spring-mvc-4-hello-world-example-with-xml-configuration-maven-and-jsp/

https://docs.spring.io/spring-boot/docs/1.1.x/reference/html/boot-features-developing-web-applications.html#boot-features-jsp-limitations

http://www.springboottutorial.com/creating-web-application-with-spring-boot

https://github.com/spring-projects/spring-boot/tree/v2.0.1.RELEASE/spring-boot-samples/spring-boot-sample-tomcat-jsp

https://dzone.com/articles/mvc-10-in-java-ee-8-getting-started-using-facelets

https://docs.oracle.com/javaee/7/tutorial/jsf-facelets001.htm

https://stackoverflow.com/questions/18387993/spring-jsf-integration-how-to-inject-a-spring-component-service-in-jsf-managed

https://stackoverflow.com/questions/46187725/spring-boot-jsf-integration/46190826#46190826

https://stackoverflow.com/questions/3008395/sometimes-i-see-jsf-url-is-jsf-sometimes-xhtml-and-sometimes-faces-why

https://github.com/ubenzer/Agent-Based-Programming-Experiments-with-JADE

https://spring.io/guides/gs/consuming-rest/

https://stackoverflow.com/a/31947188/4733587

https://www.developer.com/java/data/what-is-primefaces.html

https://www.primefaces.org