Universidad Complutense de Madrid
Facultad de Informática
Máster en Ingeniería Informática - Desarrollo de Aplicaciones y Sistemas Inteligentes

D4 - Memoria del proyecto

30 de junio de 2019

Aitor Cayón Ruano,
Jose Javier Cortés Tejada

Índice

1. Diseño del sistema **3**
1.1. Agente base de datos 4
1.2. Agente cluster 6
1.3. Agente reglas 7
1.4. Agente conversacional 8

2. Software de terceros **9**
2.1. Dialogflow 9
2.2. Sklearn 9
2.3. PyKnow 9
2.4. PADE 9
2.5. MongoDB 9

3. Manual de construcción **10**
3.1. Instalación 10
3.2. Arranque 11
3.3. Parada 11

4. Manual de usuario **12**

5. Reparto de tareas **14**

1. Diseño del sistema

El sistema implementado cuenta con una estructura como la mostrada en la figura 1, de forma que tenemos 4 agentes:

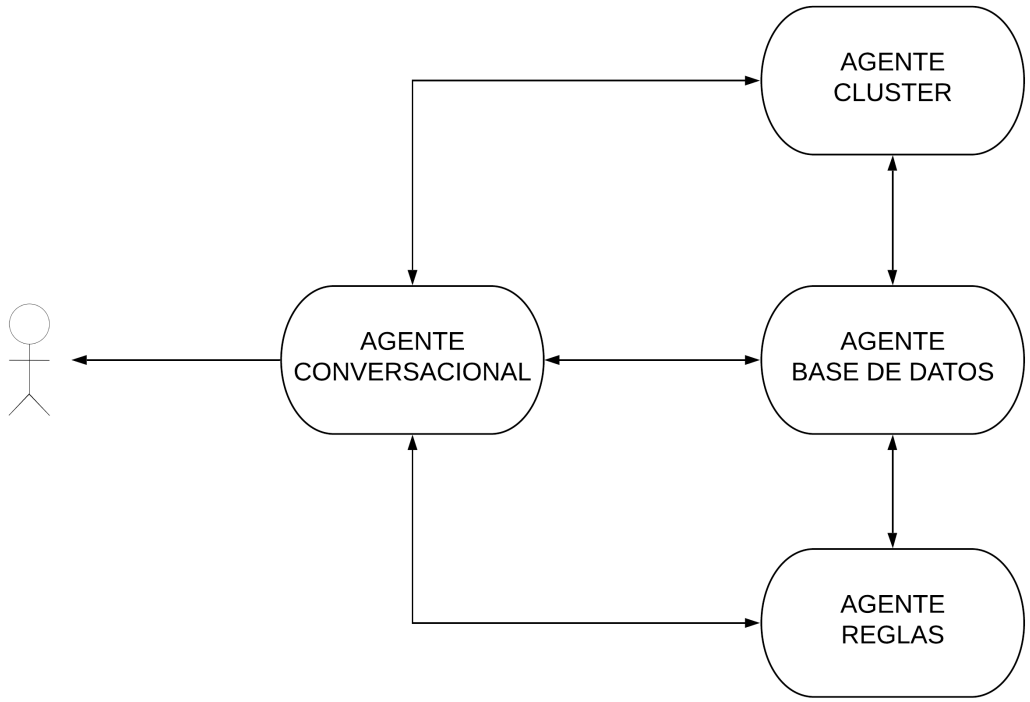

Figura 1: Diagrama con el diseño general del sistema.

- Agente base de datos: contiene toda la información del sistema y se la proporciona a los demás agentes.
- Agente cluster: es el encargado de crear clústers con los perfiles de usuarios del sistema para así poder completar aquellos datos del usuario de los que no se disponga.
- Agente reglas: realiza recomendaciones en base a las funcionalidades del sistema usando los datos del usuario así como la información sobre los platos disponible.
- Agente conversacional: interactúa con el resto de agentes y con el usuario.

1.1. Agente base de datos

Este agente es el encargado de almacenar, actualizar y proveer a los demás agente información de los usuarios o de los platos. Para ello hace uso de una base de datos MongoDB con dos colecciones:

- users: usuarios del sistema. Cada usuario dispone de los siguientes datos:
  1. _id: identificador único.
  2. age: edad del usuario.
  3. height: altura del usuario.
  4. weight: peso del usuario.
  5. allergies: lista de alergias del usuario.

```
[
        { "_id": "1", "age":27.0, "height":1.58, "weight":90.0, "allergies":["maiz"]},
        { "_id": "2", "age":56.0, "height":1.88, "weight":85.0, "allergies":["avena"]},
        { "_id": "3", "age":27.0, "height":1.77, "weight":69.1, "allergies":["huevos"]},
        { "_id": "4", "age":29.0, "height":1.86, "weight":87.4, "allergies":[]}
]
```

Figura 2: Extracto de la colección de usuarios del sistema.

- plates: comidas del sistema. Se dividen en 3 categorías y cada una tiene atributos específicos:
  1. desayunos:
     a) tipo: tipo de menú, en este caso desayuno.
     b) dieta: tipo de dieta asociada a un objetivo (baja en grasas, alta en proteínas).
     c) fruta, cereal, lácteo y embutido: componentes del desayuno. Pueden estar vacíos en algunas instancias.
     d) dificultad: complejidad de elaboración del plato.
  2. comidas:
     a) tipo: tipo de menú, en este caso comida.
     b) primero, segundo y postre: platos del menú.
     c) tipo_dieta: tipo de menú en función de la dieta a la que esté enfocado. Puede ser vegetariana, mediterránea o alta en proteínas.
     d) dificultad: complejidad de elaboración del menú.

3. cenas:
   a) tipo: tipo de menú, en este caso cena.
   b) primero y postre: platos del menú.
   c) dificultad: complejidad de elaboración del menú.

```
[
        { "_id": "1", "tipo": "comida", "primero": "ensalada", "segundo": "sopa",
        "postre": "fruta", "dieta": "baja_grasas", "tipo_dieta":"vegetariana",
        "dificultad": "dificil" },
        { "_id": "5", "tipo": "cena", "primero": "queso fresco", "postre": "fruta",
        "dieta": "baja_grasas", "dificultad": "sencillo" },
        { "_id": "12", "tipo": "desayuno", "dieta": "alta_proteinas", "fruta": "platano",
        "lacteo": "leche", "dificultad": "sencillo" }
]
```

Figura 3: Extracto de la colección de menús del sistema.

El agente base de datos implementa un el protocolo FIPA-REQUEST para comunicarse con el resto de agentes. En este contexto este agente solo responde a peticiones externas, de forma que nunca envía ninguna petición a los demás agentes.

Además presenta una estructura de clases como la mostrada en la figura 5.

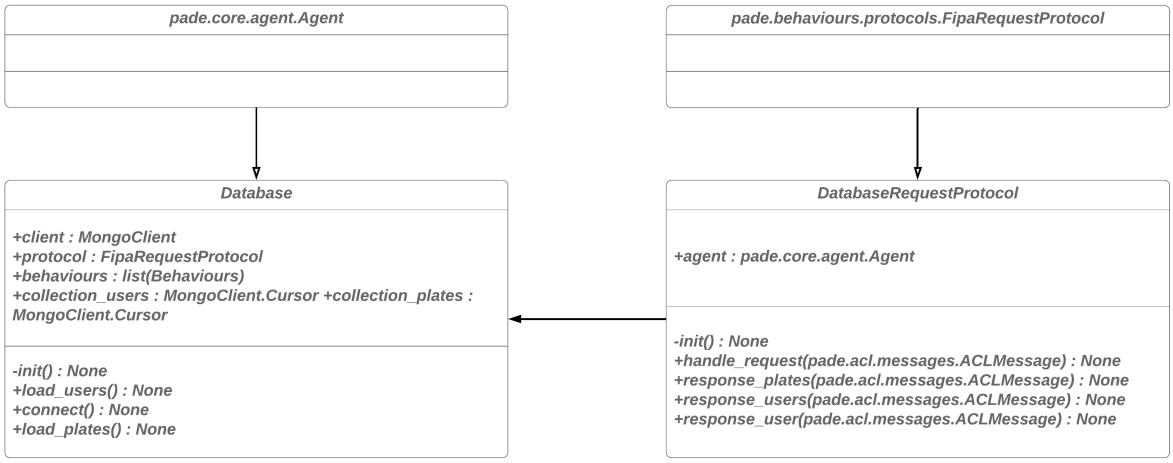

Figura 4: Diagrama de clases de agente base de datos.

1.2. Agente cluster

Es el encargado de agrupar los documentos con la información de los usuarios. Para ello es necesario que solicite esta información al agente base de datos, aunque el resto de su comportamiento se centra en responder a peticiones de otros agente. Al igual que el caso anterior este agente también implementa el protocolo FIPA-REQUEST para comunicarse con el resto de agentes.

Para generar los clusters usamos kmeans como algoritmo de agrupamiento. Por defecto el sistema cuenta un 100 perfiles de usuarios variados y crea 10 clusters con dichos perfiles. Para hacer los agrupamientos solo se tienen en cuanta parte de los datos de los usuarios vistos anteriormente, en concreto se usan *age*, *weight* y *height*.

El diagrama de clases para este agente es igual al del anterior, aunque cambian ciertos métodos. Se añaden un nuevo manejador para el protocolo de comunicación y los métodos necesarios para el funcionamiento del clúster como se observa en la figura 6.

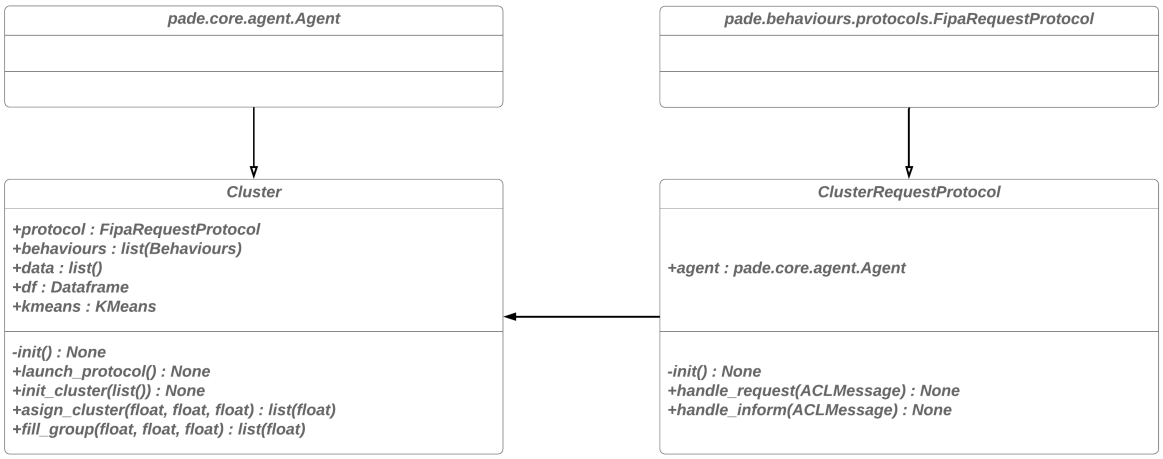

Figura 5: Diagrama de clases de agente cluster.

1.3. Agente reglas

Este agente se encarga de realizar las recomendaciones del sistema. Para ello dispone de un motor de reglas implementado con *PyKnow* y un conjunto de datos como el explicado en la sección 1.1. En este caso el diagrama de clases difiere de los anteriores (figura 6).

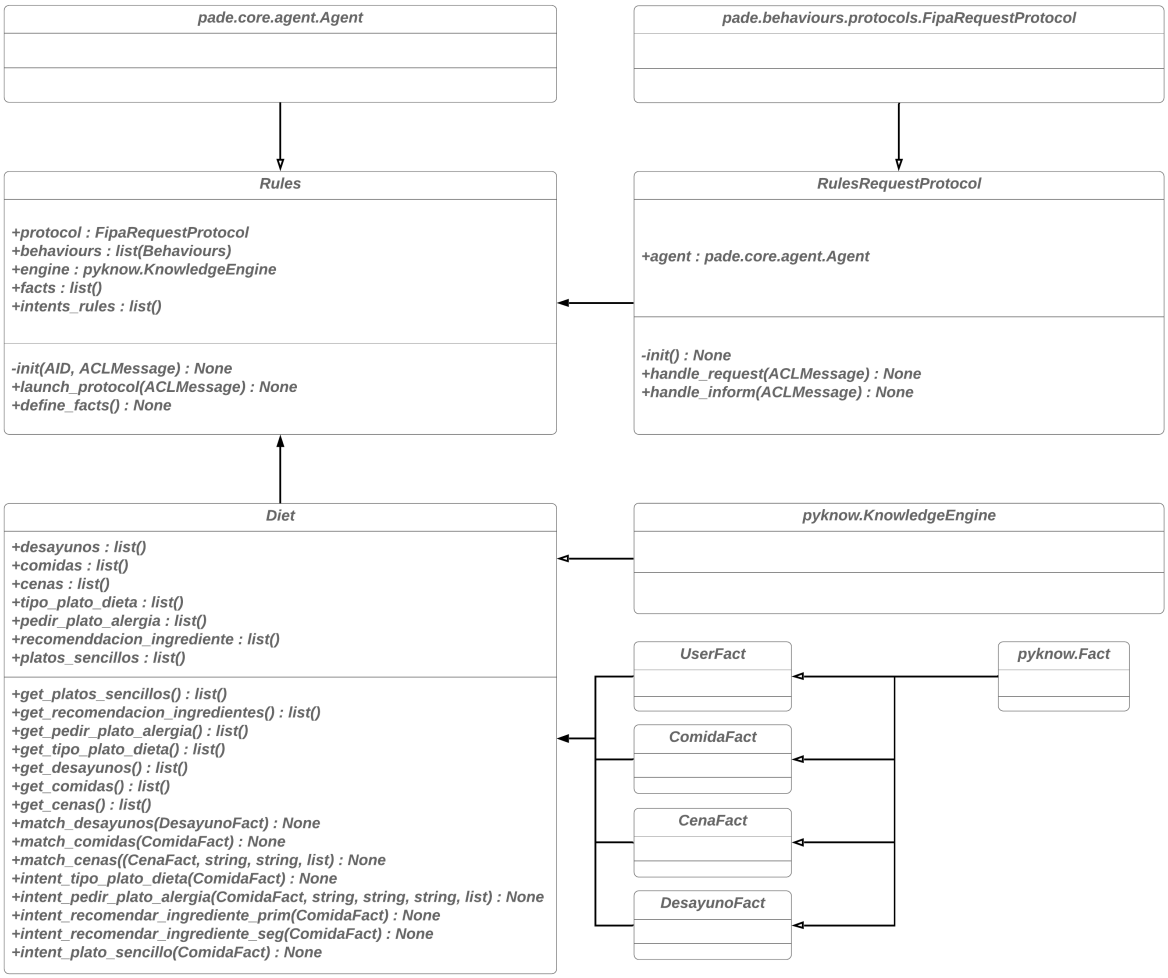

Figura 6: Diagrama de clases de agente reglas.

Ahora contamos con una clase *Diet* que hereda de *pyknow.KnowledgeEngine* e implementa el motor de reglas. Esta clase además dispone de 4 clases internas (*UserFact, DesayunoFact, ComidaFact* y *CenaFact*) que son usadas por las reglas a la hora de hacer las recomendaciones.

La clase *Rules* posee una instancia de *Diet* con la que hacer las recomendaciones e implementa el protocolo FIPA-REQUEST para comunicarse con otros agentes.

1.4. Agente conversacional

Este agente es el encargado de mantener la sesión con el usuario y además hace peticiones a otros agentes. Este agente implementa de nuevo el protocolo FIPA-REQUEST para comunicarse con otros agentes y además posee un comportamiento *TimedBehaviour* para el envío de mensajes. A diferencia de JADE, donde disponemos de los *CyclicBehaviour* para enviar un mensaje y esperar su respuesta, aquí no podemos replicar este comportamiento de forma sencilla. El comportamiento *TimedBehaviour* funciona como un temporizador, de manera que cada un determinado número de segundos (especificado por nosotros) hace una acción. Esto implica que no podemos esperar a mandar un mensaje y que otro agente nos responda como si podemos hacer en JADE.

Otra cosa a tener en cuenta es que en PADE el hilo principal de ejecución del programa crea procesos hijos para cada agente lo cual implica que se cierre la salida estandart, es decir, no se puede escribir por consola en los procesos que ejecutan los agente.

Debido a estas dos restricciones la conversación del usuario con este agente se ha hecho usando ficheros externos, de forma que cada 10-15 segundos se envía un mensaje con cada *input* con tiempo suficiente para que se reciba la respuesta. Esto es así debido a limitaciones de la plataforma.

El diagrama de clases de este agente corresponde con el de la figura 7

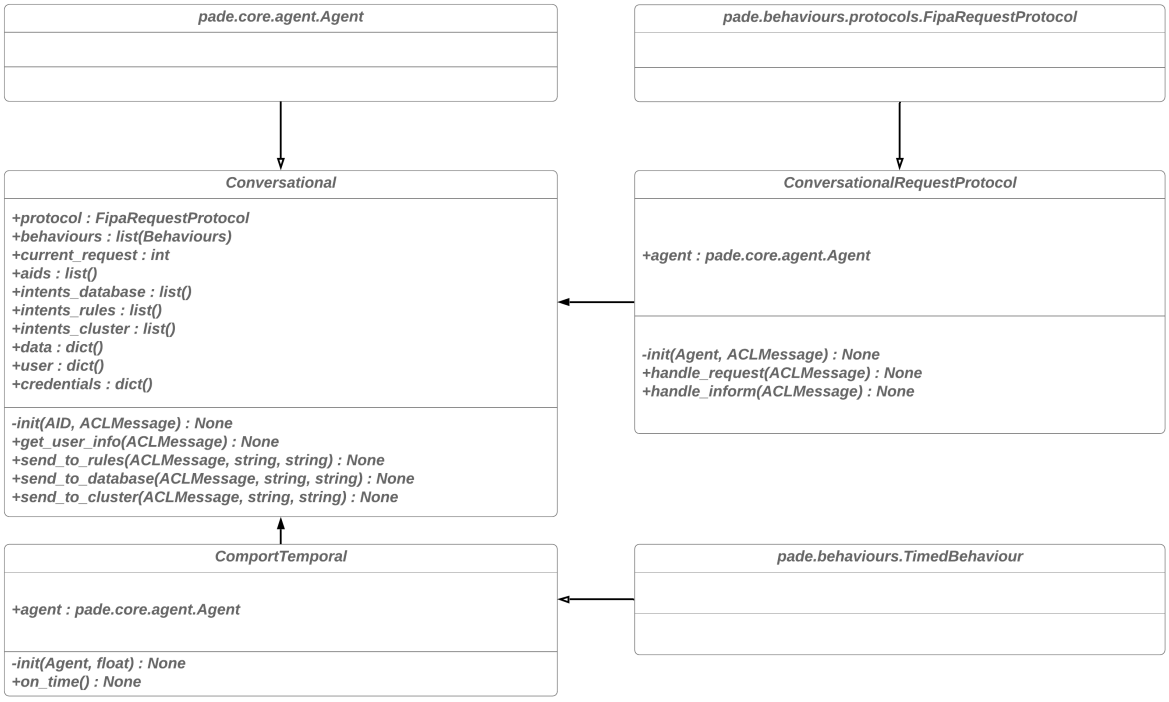

Figura 7: Diagrama de clases de agente conversacional.

2. Software de terceros

2.1. Dialogflow

DialogFlow es una API para la construcción de interfaces conversacionales basadas en texto. Permite a nuestro sistema analizar las respuestas del usuario para identificar la intencionalidad de las mismas y actuar en consecuencia. Constituye la base para la interacción con el usuario y para dirigir el comportamiento del sistema, es decir, nos permite distinguir entre las distintas posibles solicitudes de menús y las actualizaciones a realizar sobre su perfil.

2.2. Sklearn

Sklearn es una de las principales de python para machine learning. EN nuestro proyecto la hemos usado para implementar el algoritmo kmeans para agrupar los perfiles de usuario en función de la edad, altura y peso. Esto nos permite garantizar salidas más correctas para nuestro sistema a pesar de que la información proporcionada por el usuario esté incompleta.

2.3. PyKnow

Sistema experto basado en CLIPS. En nuestro proyecto lo hemos usado para implementar un motor de reglas con el que hacer las recomendaciones a los usuarios en base a los hecho asertados y la información de su perfil.

2.4. PADE

Framework para sistemas multiagente en python. Proporciona las mismas funcionalidades que JADE al implementar los protocolos FIPA. Lo hemos usado en nuestro proyecto para la comunicación entre agentes.

2.5. MongoDB

Entorno de bases de datos no relacionales. Las hemos usado en nuestro proyecto para almacenar los perfiles de usuario y la información relativa a los menús usados por el sistema.

3. Manual de construcción

3.1. Instalación

NOTA: EN LA ENTREGA SE PROPORCIONA EL ENTORNO CONFIGURADO, SE PUEDE SALTAR EL PASO DE INSTALACIÓN

Para la instalación del sistema se ha proporcionado un *script* en la entrega del proyecto con el que se configura el entorno automáticamente. No obstante es necesario hacer una instalación limpia de Anaconda y es imperativo usar Ubuntu (u otra distro de Linux, sin embargo el proyecto ha sido desarrollado y testeado en Ubuntu).

El fichero de configuración suministrado en la entrega crea un entorno virtual donde se instalarán todos los paquetes necesarios para la ejecución del proyecto. Durante la configuración del entorno es posible que se pida la contraseña del equipo para hacer varias operaciones.

Este *script* instalará las siguientes dependencias:

1. *mongo server.*
2. *mongo client.*
3. *pade.*
4. *pymongo.*
5. *pyknow.*
6. *dialogflow.*
7. *sklearn.*
8. *pandas.*
9. *kmeans.*

Se recomienda crear una carpeta donde meter todos los ficheros del proyecto y ahí ejecutar por consola el *script*. Hecho esto nos habrán aparecido varios directorios, entre ellos *pade* con la librería PADE instalada y *env* con el entorno virtual sobre el que trabajaremos.

3.2. Arranque

Antes de ejecutar el proyecto hay que hacer ciertos preparativos. Es necesario abrir dos pestañas en la consola y estar en la carpeta raíz del proyecto. Hecho esto, y tras haber seguido los pasos del apartado anterior, tenemos que activar el entorno virtual en ambas consolas mediante la orden:

conda activate ./env

A continuación vamos a una de las consolas con el entorno virtual ya activo y ejecutamos las siguientes órdenes (son dos guiones no uno):

cd mongo/
mongod –config config.yml

Esto nos lanzará la base de datos de acuerdo a la configuración del fichero *config.yml*. Ahora volvemos a la otra consola y solo tenemos que ejecutar el fichero ***main.py***. Este fichero está preparado para lanzar los 4 agentes del sistema, no obstante hay un par de cosas a tener en cuenta. Cada agente usa un puerto y además PADE por debajo utiliza varios, entonces es probable que las primeras ejecuciones del sistema lancen numerosos errores. Hemos tratado de evitarlo limpiando los puertos antes de usarlos pero aún así hay algunos que se nos escapan, por lo tanto recomendamos ejecutar el fichero ***main.py*** varias veces, cortando la ejecución con ***crtl + C*** hasta que no salgan errores (2-3 veces son suficiente).

Para ejecutar este fichero basta con hacer ***python main.py*** y nos saldrá por consola el logo de PADE. Además se nos pedirán un usuario y una contraseña, basta con pulsar intro en ambos casos.

3.3. Parada

El sistema terminará su ejecución cuando no haya más mensajes por parte del usuario, en este caso el agente conversacional mandará un mensaje a cada agente en ejecución para que termine su ejecución. En JADE disponemos de un método *on_stop* que se encarga de la finalización del agente, sin embargo en PADE no tenemos nada parecido. Por ello la parada de los agentes se hará poniéndolos en pausa hasta que expire el temporizador, lo cual finalizará su ejecución.

Esto se hará automáticamente y para desactivar el entorno virtual en ambas consolas solo será necesario hacer ***conda deactivate***.

4. Manual de usuario

Como ya se ha comentado antes, la ejecución del proyecto está totalmente automatizada debido a las limitaciones de PADE. En este apartado se detalla el contenido del fichero de prueba así como las salidas que genera.

El fichero de prueba contiene la siguiente información, de forma que cada línea pertenece a un caso de uso:

```
1  hola
2  quiero un plato con muchas proteinas!
3  me gustaria probar un plato de dieta mediterranea
4  soy alergico al kiwi, que puedo prepararme?
5  la tortilla de patatas, el gazpacho y el chorizo están genial!
6  me fascina el puerro y el zumo de naranja
7  me gusta mucho el gazpacho
8  la tortilla de patatas es genial
9  que puedo preparar con una zanahoria?
10 quiero un plato sencillo de preparar
11 quiero bajar de peso
12 mido 150 cm
```

Figura 8: Ejemplo de fichero de prueba del sistema.

Durante la ejecución del sistema veremos que se imprime por pantalla información con el siguiente formato:

1. **user: ....**
2. **response: ....**
3. **intent: ....**
4. **data: .....**

Esta información es el input que le hemos introducido al sistema (primera línea), mientras que el resto de información son los datos obtenidos de *Dialogflow*, en concreto la respuesta del API al input del usuario, el *intent* detectado en el input y los datos que se han extraído de él.

Cada línea del fichero de prueba supone un mensaje a otro agente, y esta la información de dichos mensajes se mostrará con el siguiente formato:

```
[database_agent@localhost:8095] 30/06/2019 21:16:30.678 --> Identification process done.
[rules_agent@localhost:8096] 30/06/2019 21:16:30.679 --> Identification process done.
[conversation_agent@localhost:8098] 30/06/2019 21:16:30.679 --> Identification process done.
[cluster_agent@localhost:8097] 30/06/2019 21:16:30.679 --> Identification process done.
```

Figura 9: Ejemplo del formato de los mensajes enviados en PADE.

Estos mensajes indican el agente que los muestra por pantalla, el momento en el que lo hace y la información proporcionada en ellos. A continuación se muestran algunos ejemplos de ejecución del sistema usando el contenido del fichero de prueba. La información devuelta por los agentes es mostrada en forma de *json*.

```
======================================================================
user: quiero un plato sencillo de preparar

response:  Claro, ahora mismo te lo paso!
intent:  sencillo
data:  ['plato_sencillo']
[rules_agent] 30/06/2019 21:19:03.389 --> generando un plato sencillo de preparar ...
[conversation_agent] 30/06/2019 21:19:03.397 --> plato sencillo
{"primero": "pescado", "segundo": "verdura", "postre": "fruta"}
======================================================================
```

Figura 10: Ejemplo de funcionamiento donde se solicita un plato sencillo. El agente de reglas lanza todas sus reglas y devuelve al agente conversacional uno de los platos catalogados como sencillos

```
user: la tortilla de patatas, el gazpacho y el chorizo están genial!

response:  Genial, me lo apunto
intent:  marcar_favorito
data:  ['tortilla de patatas', 'gazpacho', 'chorizo']
[database_agent] 30/06/2019 21:17:48.328 --> petición recibida de conversation_agent@localhost:8098
[database_agent] 30/06/2019 21:17:48.328 --> se ha actualizado la información del usuario
[database_agent] 30/06/2019 21:17:48.331 --> petición recibida de conversation_agent@localhost:8098
[database_agent] 30/06/2019 21:17:48.331 --> se ha actualizado la información del usuario
[database_agent] 30/06/2019 21:17:48.333 --> petición recibida de conversation_agent@localhost:8098
[database_agent] 30/06/2019 21:17:48.334 --> se ha actualizado la información del usuario
======================================================================
user: me fascina el puerro y el zumo de naranja

response:  Genial, me lo apunto
intent:  marcar_favorito
data:  ['puerro', 'zumo de naranja']
[database_agent] 30/06/2019 21:18:03.278 --> petición recibida de conversation_agent@localhost:8098
[database_agent] 30/06/2019 21:18:03.278 --> se ha actualizado la información del usuario
[database_agent] 30/06/2019 21:18:03.281 --> petición recibida de conversation_agent@localhost:8098
[database_agent] 30/06/2019 21:18:03.281 --> se ha actualizado la información del usuario
======================================================================
user: me gusta mucho el gazpacho

response:  Genial, me lo apunto
intent:  marcar_favorito
data:  ['gazpacho']
======================================================================
user: la tortilla de patatas es genial

response:  Genial, me lo apunto
intent:  marcar_favorito
data:  ['tortilla de patatas']
======================================================================
```

Figura 11: Ejemplo de funcionamiento donde se pide que se añadan como favorito varios platos. En los dos primeros casos los nuevos favoritos se mandan al agente base de datos para que se almacenen actualice la información del usuario, mientras que en los dos útlimos casos no es necesario

5. Reparto de tareas

La tareas se han repartido de forma equitativa entre los dos miembros del grupo. No tenemos una lista exacta del reparto pero ha sido igual para ambos.