kampal



Hackaton

MANUAL

Índice

Índice **1**
Guía para el Hackathon - Desarrollo de Bots en Kampal Collective Learning **2**
1. Introducción **2**
2. Descripción de Kampal Collective Learning **2**
2.1. Fases Principales 3
3. Archivo agent.py: Estructura y Eventos **3**
3.1. Requerimientos de la Aplicación 3
3.2. Flujo de Eventos 3
4. Handlers y Formatos de Datos **4**
4.1. handle_fase(message) 4
4.2. handle_questions(message) 6
4.3. handle_description(message) 7
4.4. handle_vecinos(message) 8
4.5. handle_un_vecino(message, vecino_id) 9
4.6. handle_my_solution(message) 11
4.7. handle_top(message, question_order) 13
4.8. handle_swap(message) 14
4.9. handle_message(message) 15
5. Funciones clave **15**
5.1. query_ia(payload) 16
5.2. save_response(responses) 16
5.3. Buenas Prácticas para el Desarrollo de Bots 17
5.4. Consideraciones adicionales 18
6. Hackathon Project: Configuración y Uso **19**
6.1. Instalación 19
6.2. Estructura del Proyecto 19
6.3. Uso 20
7. Próximos Pasos **21**
8. Conclusión **21**



Guía para el Hackathon - Desarrollo de Bots en Kampal Collective Learning

1. Introducción

En este hackathon, los participantes desarrollarán bots que colaboren en la resolución de un problema dentro de Kampal Collective Learning, una herramienta diseñada para potenciar la inteligencia colectiva. En Kampal Collective Learning, muchos usuarios (o bots) trabajan a la vez, comparten, editan y copian soluciones con el objetivo de producir la mejor respuesta posible de manera conjunta.

Esta guía describe:

- El flujo de trabajo y las Fases de Kampal Collective Learning.
- Cómo está estructurado el agente (bot) que responde automáticamente a los eventos, en el fichero **agent.py**.
- Cómo se implementan y documentan las principales funciones (handlers) que manejan cada evento.
- Las funciones clave **save_response** y **query_ia** que facilitan la interacción con el servidor de Kampal Collective Learning y un modelo de IA conversacional.
- Recomendaciones para interactuar con la aplicación y participar de forma autónoma.
- Detalles de instalación y ejecución del cliente de hackathon, que se conecta vía WebSocket y realiza consultas a una API de IA.

2. Descripción de Kampal Collective Learning

La idea central es que cada participante (humano o bot) mantiene sus propias respuestas a diferentes preguntas planteadas en un proyecto. Estos pueden **editar** sus respuestas o **copiar** las de sus vecinos, aumentando así la popularidad (y, a menudo, la "salud") de las propuestas más acertadas.



2.1. Fases Principales

- **Fase 0**: Solo lectura del enunciado.
- **Fase 1**: Se puede escribir y guardar respuestas.
- **Fase 2**: Aparecen las respuestas de hasta 4 vecinos, pero no se actualizan en tiempo real. Se puede copiar.
- **Fase 3**: Ahora se ve a los vecinos en tiempo real y pueden cambiar los vecinos asignados. Continúa la posibilidad de editar y copiar.
- **Fase 4**: El sistema puede copiar automáticamente respuestas entre vecinos.
- **Fase 5**: Las respuestas con menor salud pueden desaparecer. Sigue siendo posible editar y copiar.
- **Fase 6**: Se muestran las 10 Mejores respuestas (Top 10) en lugar de vecinos. Se sigue pudiendo copiar.
- **Fase 7**: Se muestran las 10 Mejores respuestas (Top 10) en lugar de vecinos. Solo se puede copiar, no se puede editar. No hay cambios ni extinciones adicionales.
- Cuando termina la Fase 7, se obtiene la solución final.

3. Archivo agent.py: Estructura y Eventos

El fichero **agent.py** es el punto de entrada para tu bot. En él se definen una serie de funciones handler, cada una destinada a reaccionar ante un evento distinto que notifica el sistema.

3.1. Requerimientos de la Aplicación

Python (versión 3.10).

Librerías necesarias según requirements.txt o la configuración que tengáis.

Configuración de acceso al servidor Kampal Collective Learning (tokens, URLs, etc.), usualmente en config.py.

3.2. Flujo de Eventos

Cada vez que Kampal Collective Learning emite un evento (por ejemplo, llegada de nuevas preguntas, actualización de la fase, cambio de vecinos, etc.), se invoca la función



correspondiente, pasando la información relevante a tu bot para que este la procese y decida las acciones necesarias (p. ej. editar una respuesta, copiarla, guardar).

Las funciones handler principales son:

- handle_fase
- handle_questions
- handle_description
- handle_vecinos
- handle_un_vecino
- handle_my_solution
- handle_top
- handle_swap
- handle_message

A continuación, se explican cada una y el formato de datos que manejan.

4. Handlers y Formatos de Datos

A modo de referencia, aquí tienes ejemplos de docstring y descripciones de cada función. En tu implementación, añade tu propia lógica dentro de ellas para que tu bot actúe de forma autónoma.

4.1. handle_fase(message)

Maneja el evento 'fase'. Este evento notifica el estado de la fase del hackaton e incluye diversa información relacionada con el tiempo, configuraciones y/o propiedades específicas de dicha fase. El parámetro `message` puede variar en formato y contenido.

```
def handle_fase(message):
    """
    Maneja el evento 'fase'.

    Este evento notifica el estado de la fase del hackaton e incluye diversa
    información relacionada con el tiempo, configuraciones y/o propiedades
    específicas de dicha fase. El parámetro `message` puede variar en formato
```




y contenido, por ejemplo:

1. Un diccionario con segundos restantes para la fase:
```
{
  "seconds_left": 15515
}
```

2. Una cadena con una fecha/hora límite en formato ISO8601:
```
"2025-02-27T03:02:37.518Z"
```

3. Un diccionario con duraciones o métricas intermedias:
```
{
  "0": 9,
  "1": 134.16387499363333,
  "2": 19.6874072046,
  ...
}
```

4. Un diccionario más completo con múltiples campos:
```
{
  "deadline": "2025-02-27T03:02:37.518Z",
  "durations": {
     "0": 9,
     "1": 134.16387499363333,
     "2": 19.6874072046,
     ...
  },
  "obj": {
    "blob": "",
    "default_tiempo": 10,
    "description": "",
    "flags": {
      "copia": True,
      "numsolutions": "all"
    },
    "name": "Fase 7"
  },
  "phase_lock": False,
  "status": 7
}
```

5. Otra variante con tiempo restante y número de usuarios:
```
{
  "seconds_left": 62339,
  "num_users": 20
}
```




```
    :param message: Diccionario (o, en algunos casos, cadena) con información
                    referente al estado y/o configuración de la fase actual
                    (p.ej. tiempo restante, fecha límite, flags, nombre, etc.).
    ""
```

4.2. handle_questions(message)

Maneja el evento 'questions'. Este evento notifica que se han recibido nuevas preguntas. En general, sólo se recibe este evento al comienzo de cada conexión o si el bot tiene que reconectarse por pérdida de conectividad en algún momento. Permite conocer las preguntas que se plantean en el proyecto y que el bot debe tratar de responder.

```
def handle_questions(message):
    """
    Maneja el evento 'questions'.

    Este evento notifica que se han recibido nuevas preguntas. El parámetro
    `message` suele ser una lista de diccionarios, donde cada uno contiene:

        {
          "question_id": "string",  # Identificador único de la pregunta
          "text": "string"          # Enunciado o contenido de la pregunta
        }

    Ejemplo:
    [
      {"question_id": "4c273e5a-d726-4ab9-8c8e-5afd8c996bd0", "text": "¿Cuál es la
fruta más sana?"},
      {"question_id": "d3d1c733-5334-4748-808b-1b547942d0e0", "text": "Dime un cuadro
que te guste"},
      {"question_id": "d95a16a6-cec1-45e5-b728-108c7091eb19", "text": "¿Quién ganó el
mundial de fútbol en 1994?"}
    ]

     ***Uso de la información de 'description'***
    Es importante (aunque no obligatorio) aprovechar la información que se
    recibe a través de 'handle_description', ya que aporta contexto adicional
    sobre el proyecto en el que se está trabajando. Esto puede ser
    muy útil para enriquecer las respuestas o para filtrar qué datos son
    relevantes.

    El código de ejemplo en esta función muestra cómo podrías:
```




```
    1. Recorrer las preguntas para generar respuestas (aquí, de forma
       predeterminada se asigna "Hola mundo").
    2. Consultar a la IA (con `ask_something`) para obtener una respuesta
       (totalmente opcional y personalizable).
    3. Llamar a `save_response` para guardar la(s) respuesta(s).

  ***** Formato que espera 'save_response' *****
  'save_response' requiere un diccionario donde:

      - Las claves sean los 'question_id' de cada pregunta.
      - Los valores sean listas de cadenas (puede ser una o varias)
        con la respuesta o respuestas generadas.

  Por ejemplo:
  {
    "4c273e5a-d726-4ab9-8c8e-5afd8c996bd0": ["Hola mundo"],
    "d3d1c733-5334-4748-808b-1b547942d0e0": ["Mi pintura favorita es..."],
    "d95a16a6-cec1-45e5-b728-108c7091eb19": ["Brasil fue el campeón..."]
  }

  Cada hacker puede implementar la lógica de resolución de preguntas
  como prefiera (IA, bases de datos, etc.) siempre
  que, al final, se genere un diccionario con esta estructura y se llame
  a 'save_response'.

  :param message: Lista de preguntas, con "question_id" y "text".
  """
```

4.3. handle_description(message)

Maneja el evento de “description”. Este evento devuelve la descripción del proyecto. Es información contextual. Aquí podría venir el planteamiento de un situación sobre la cuál se realizarán preguntas. Entonces, las preguntas serán recibidas en el evento **handle_questions** y la explicación en este evento. Se suele recibir un único evento al principio de cada conexión

```
def handle_description(message):
    """
Maneja el evento 'description'. En este caso, se espera que     'message' sea un texto
descriptivo explicando el contexto del proyecto. Se almacena en la variable global
'description' y se registra en el log.

    :param message: Texto descriptivo.
```




```
    """
```

Esta información puede ser muy útil para comprender el contexto del problema y decidir cómo responder a las preguntas.

4.4. handle_vecinos(message)

Una vez que el bot ha escrito su primera respuesta, se coloca en una red de usuarios que estarán conectados entre sí. No todos los usuarios están interconectados sino que tienen un número limitado de vecinos o usuarios con los que puede interactuar. Además durante el transcurso del proyecto, el sistema realizará permutaciones de usuarios, cambiando su posición en la red y, por tanto, sus vecinos. Este evento devuelve los identificadores de los vecinos actuales que posee el bot. Este listado puede ser de longitud variable pudiendo tener de 0 a 4 vecinos. Cada vez que el listado de vecinos cambia, se dispara este evento.

```
def handle_vecinos(message):
    """
    Maneja el evento 'vecinos'.

    Este evento notifica qué identificadores (UUIDs) de vecinos tiene el usuario
    en un momento dado. El contenido de `message` suele ser una lista de cadenas,
    representando esos UUIDs. Puede haber entre 0 y 4 vecinos simultáneos.
    Por ejemplo:

    [
      "129c6a92-17f3-4eca-b6bc-42e86d19ad42",
      "1f1dadfd-34d7-4179-bd45-d79b9be029b3",
      "3bd40ea9-6c07-4faf-adba-b5f5cbaffdee"
    ]

    :param message: Lista de identificadores (strings) de los vecinos.
    """
```




4.5. handle_un_vecino(message, vecino_id)

Este evento se disparará cada vez que un vecino actualice alguna de sus propiedades. Esto puede ocurrir por varias causas:

- Un vecino modifica alguna de sus respuestas
- La salud del vecino cambia
- Los vecinos del vecino cambian

```
def handle_un_vecino(message, vecino_id):
    """
   Maneja el evento 'un_vecino'.

    Este evento notifica que un vecino (identificado por `vecino_id`) se ha
    actualizado con nueva información. El parámetro `message` es un diccionario
    que puede incluir varios campos:

    - "answer_items": Estructura de respuestas 'activas', con índices (p.ej. "0", "1",
"2"...)
      y cada índice contiene un diccionario con uno o más textos. Por ejemplo:
      {
        "0": { "0": "Texto 1" },
        "1": { "0": "Texto 2" },
        ...
      }

    - "frozen_answer_items": Estructura similar a "answer_items" pero corresponde
      a respuestas 'congeladas' (registradas o fijadas) al inicio de una fase.
      Puede o no estar presente, según el momento o la lógica del hackaton.
      El formato es análogo:
      {
        "0": { "0": "Texto congelado 1" },
        "1": { "0": "Texto congelado 2" },
        ...
      }

    - "energia": Valor (entero) que refleja la energía o potencia asociada
      al vecino. Ejemplo: 200000000

    - "salud": Valor (float) que indica el estado de salud o integridad
```




```
  del vecino. Ejemplo: 0.0

- "vecinos": Diccionario que mapea UUIDs de otros vecinos asociados,
  incluyendo metadatos como la prioridad o posición:
  {
    "1f1dadfd-34d7-4179-bd45-d79b9be029b3": {
      ".priority": 0.0,
      "pos": 0
    },
    "41e71111-2ff7-4eb5-ba3a-d1e8268469ea": {
      ".priority": 1.0,
      "pos": 1
    },
    ...
  }

En conjunto, un ejemplo de `message` podría lucir así:

{
  "answer_items": {
    "0": {
      "0": "La mayoría de los expertos coinciden..."
    },
    "1": {
      "0": "La última cena\" de Leonardo Da Vinci..."
    },
    "2": {
      "0": "En el Mundial de Fútbol de 1994..."
    }
  },
  "frozen_answer_items": {
    "0": {
      "0": "Algunas respuestas almacenadas al inicio de la fase..."
    },
    "1": {
      "0": "Otra respuesta congelada..."
    }
  },
  "energia": 200000000,
  "salud": 0.0,
  "vecinos": {
    "1f1dadfd-34d7-4179-bd45-d79b9be029b3": {
       ".priority": 0.0,
       "pos": 0
    },
    "41e71111-2ff7-4eb5-ba3a-d1e8268469ea": {
```




```
            ".priority": 1.0,
            "pos": 1
        },
        ...
    }
}

:param message: Diccionario con la información actualizada del vecino
                (energía, salud, respuestas activas y/o congeladas, etc.).
:param vecino_id: Identificador único (UUID) del vecino que se actualiza.
```

4.6. handle_my_solution(message)

Este evento notifica que mi propia solución (en lugar de la de un vecino) se ha actualizado con nueva información. Normalmente se dispara en las siguientes situaciones:

- Cada vez que guardo una respuesta (por ejemplo, tras llamar a 'save_response'), el sistema envía este evento para notificar la información actualizada de mi solución.
- El sistema actualiza la 'salud' de mi respuesta (p. ej., puede reflejar una validación interna, un penalizador de calidad o algún factor externo de evaluación).
- El sistema actualiza automáticamente una o varias de mis respuestas, pudiendo cambiar el contenido de forma completa, parcial, o incluso eliminarlas si así lo requiere la lógica de la aplicación.
- El sistema cambia mis 'vecinos' asociados, modificando así la red de conexiones o referencias que tengo en el entorno.

```
def handle my_solution(message)
"""
    Maneja el evento 'my_solution'.

    Este evento notifica que mi propia solución (en lugar de la de un vecino)
    se ha actualizado con nueva información. Normalmente se dispara en las
    siguientes situaciones:

    1. **Cada vez que guardo una respuesta** (por ejemplo, tras llamar a
       'save_response'), el sistema envía este evento para notificar la
       información actualizada de mi solución.

    2. **El sistema actualiza la 'salud' de mi respuesta** (p. ej., puede
       reflejar una validación interna, un penalizador de calidad o algún
```

```
   factor externo de evaluación).

3. **El sistema actualiza automáticamente una o varias de mis respuestas**,
   pudiendo cambiar el contenido de forma completa, parcial, o incluso
   eliminarlas si así lo requiere la lógica de la aplicación.

4. **El sistema cambia mis 'vecinos'** asociados, modificando así la red
   de conexiones o referencias que tengo en el entorno.

El parámetro `message` es un diccionario que puede incluir los mismos
campos que en 'handle_un_vecino', pero aplicados al usuario actual.
Por ejemplo:

- "answer_items": Respuestas o textos activos del usuario,
  estructurados por índices (p.ej. "0", "1", "2"...).

- "frozen_answer_items": Respuestas 'congeladas' o registradas
  al inicio de una fase, en el mismo formato que "answer_items".

- "energia": Valor (entero) que refleja la energía o potencia
  del usuario/solución.

- "salud": Valor (float) que indica el estado de salud o integridad
  de la solución.

- "vecinos": Opcionalmente, puede haber un diccionario con la configuración
  de vecinos o referencias asociadas al usuario, si así lo define la lógica
  de la aplicación.

Ejemplo de `message`:

{
  "answer_items": {
    "0": {
      "0": "Texto de mi respuesta actual"
    },
    ...
  },
  "frozen_answer_items": {
    "0": {
      "0": "Texto congelado al inicio de la fase"
    },
    ...
  },
  "energia": 250000000,
  "salud": 10.0,
```




```
      "vecinos": {
        "id-vecino-1": {
          ".priority": 0.0,
          "pos": 0
        },
        ...
      }
    }

    :param message: Diccionario con la información actualizada de la propia
                    solución (energía, salud, respuestas activas y/o
                    congeladas, etc.).
    """
```

4.7. handle_top(message, question_order)

Este evento se disparará periódicamente a partir de una fase del experimento cada minuto o dos minutos aproximadamente. Indica, para cada pregunta, las 10 soluciones más populares en ese momento. Esta popularidad viene definida por el parámetro health

```
def handle_top(message, question_order):
    """
    Maneja el evento 'top'.

    Se espera que `message` sea un diccionario con la siguiente estructura:
    {
      "1": {
        "propuesta": {
          "0": {
            "answer_id": "string",
            "distance": float,
            "health": float,
            "itempos": {
              "0": {"text": "cadena de texto"},
              "1": {"text": "cadena de texto"},
              ...
            },
            "ranking": float
          }
        }
      },
```




```
      "2": { ... },
      ...
    }

    - Cada clave (por ejemplo, "1", "2", "3"...) indica la posición en el ranking.
    - Dentro de "propuesta", la clave "0" contiene la información de respuesta:
        * answer_id: identificador único de la respuesta
        * distance, health, ranking: métricas asociadas a la respuesta
        * itempos: estructura con subíndices y textos
    - Se procesa la información para ver la 'propuesta' en cada posición y
      extraer los datos que sean relevantes (por ejemplo, mostrar el texto o
      guardar el ranking).

    :param message: Diccionario con la información del top/ranking.
    :param question_order: Orden o identificador adicional de la pregunta
                           (puede indicar la prioridad o el id de la pregunta).
"""
```

4.8. handle_swap(message)

Este evento se dispara cada vez que el usuario cambia de posición en la red. Se dispararán además eventos a través de los otros manejadores con la información de los nuevos vecinos y sus soluciones. Simplemente indica que el usuario ha sido reubicado en la red.

```
def handle_swap(message):
    """
    Maneja el evento 'swap'. Notifica un evento de cambio de posición del usuario
    en el sistema. Probablemente disparará eventos de actualizaciones de
'handle_vecinos' y 'handle_un_vecino'

    Se reciben dos valores 0 ó 1 que simplemente indican un cambio en la posición del
usuario. Da igual si es 0 o 1,

    """
```




4.9. handle_message(message)

Este evento se disparará de forma poco frecuente. Es un mensaje que se puede enviar a usuarios específicos o a todos los usuarios pero generalmente enfocado a dar notificaciones a humanos. Podrían ser mensajes de este tipo:

- “Por favor, al terminar salid de la aplicación”
- “Por favor usuario23, no utilices vocabulario soez”

En general el bot no debería prestar demasiada atención a este tipo de mensajes

```
def handle_message(message):
    """
    Maneja el evento 'message'.

    Este evento suele representar notificaciones poco frecuentes
    de los organizadores, por ejemplo:
    "Mensaje enviado por los administradores con alguna notificación especial"

    :param message: Cadena de texto con el aviso o comunicado.
    """
```

5. Funciones clave

En el archivo agent.py hay dos funciones esenciales para la interacción con la plataforma y con modelos de inteligencia artificial. Estas funciones clave ofrecen los dos principales mecanismos de interacción de tu bot:

- Consultar IA para obtener ideas o contenido para las soluciones.
- Guardar respuestas para que el sistema Kampal Collective Learning refleje los cambios realizados por tu bot.



5.1. query_ia(payload)

- Objetivo: Permite realizar consultas de manera gratuita a un modelo de IA conversacional (p. ej., ChatGPT, Gemini, LLaMA, etc.) proporcionado por la organización
- Uso: Se le envía un payload con el formato típico de conversación (roles, mensajes, temperatura, número máximo de tokens, etc.), y la función retorna el objeto de respuesta de la IA.
- El número de peticiones por minuto estará restringido. Se situará en torno a las 2-3 por minuto. Si se excede el número de peticiones por minuto, se devolverá un error de tipo 429

Ejemplo:

```
payload = {
    "model": "modelo_elegido",
    "messages": [
        {"role": "system", "content": "Eres un experto en frutas"},
        {"role": "user", "content": "¿Cuál es la fruta más sana?"}
    ],
    "temperature": 0.7,
    "max_tokens": 50
}
response = query_ia(payload)
```

A continuación, se puede trabajar con la respuesta (por ejemplo, response.response o response.status_code) según lo que devuelva el modelo.

5.2. save_response(responses)

- Objetivo: Guarda o actualiza en el servidor de Kampal Collective Learning las respuestas propuestas por el bot.
- Uso: Se le envía un diccionario responses en el que las claves son los identificadores de las preguntas (question_id), y los valores son listas con una o varias cadenas, que representan las respuestas que el bot desea registrar.
- El número de peticiones por minuto estará restringido. Se situará en torno a las 2-3 por minuto. Si se excede el número de peticiones por minuto, se devolverá un error de tipo 429



Ejemplo:

```
responses = {
    "4c273e5a-d726-4ab9-8c8e-5afd8c996bd0": ["La fruta más sana es la manzana"],
    "d3d1c733-5334-4748-808b-1b547942d0e0": ["La pintura que más me gusta es..."]
}
result = save_response(responses)
```

- Una vez enviada la información, el servidor registra esas respuestas, y el evento handle_my_solution notificará cualquier actualización al bot.

5.3. Buenas Prácticas para el Desarrollo de Bots

Documentar: Usa docstrings y comentarios en tu código para que tu equipo u otros desarrolladores entiendan qué hace cada función.

Persistencia: Cuando guardes respuestas con save_response, asegúrate de formar correctamente el diccionario según la convención:

```
{
    question_id: [respuesta1, respuesta2, ...],
    ...
}
```

Contexto: Aprovecha la información de **handle_description** para dar a tu bot una mejor visión de qué se está resolviendo, y así generar respuestas coherentes.

Colaboración: Implementa estrategias de copiado de respuestas para sumar popularidad (y, potencialmente, mejorar la salud) y para descubrir soluciones que puedan ser útiles para todo el grupo.

Gestionar Fases: Ten en cuenta que en diferentes Fases se habilitan o restringen acciones (leer, editar, copiar, extinción de respuestas, etc.). Ajusta la lógica de tu bot según la Fase actual.



Resiliencia: Prepárate para eventos como handle_my_solution en los que tu respuesta puede desaparecer o ser sustituida por otra y para cambios en los vecinos (**handle_swap**, **handle_vecinos**).

5.4. Consideraciones adicionales

1. **Funciones y Variables Auxiliares:** Los participantes tienen la libertad de crear e invocar funciones y variables auxiliares dentro de agent.py para almacenar y gestionar la información que consideren oportuna (por ejemplo, guardar el historial de vecinos, mantener un registro de los cambios en sus respuestas, etc.). De este modo, cada bot puede implementar su propia estrategia de colaboración.
2. **Uso de Librerías Externas:** Está permitido instalar librerías auxiliares (mediante pip install o indicando dependencias en requirements.txt) siempre que sean útiles para el funcionamiento interno del bot (por ejemplo, para análisis de datos, lógica de inferencia, etc.). Sin embargo, el día del experimento, la conexión de los bots con Internet estará restringida exclusivamente a:
    a. El WebSocket de Kampal Collective Learning para recibir y enviar eventos.
    b. La API de IA para consultar y guardar respuestas.
3. Cualquier otra llamada al exterior (como actualizaciones de librerías en tiempo real, peticiones HTTP adicionales, etc.) estará bloqueada. Por ello, si tu librería realiza comprobaciones o conexiones a Internet de manera automática (por ejemplo, para descargar modelos o datos), es posible que te funcione en las pruebas de los días anteriores, pero no el día del experimento.
4. **Parámetro message:** por cuestiones de rendimiento, el parámetro **message** que suele ser un diccionario, no viene siempre completo con todas las claves y sus valores, sólo muestra las claves actualizadas, por lo que se debe manejar ese diccionario con cuidado, suponiendo que los campos que no llegan, conservan el valor previo. Por ejemplo,
    a. En un proyecto de 3 preguntas mi vecino contesta a las 3 y guarda
    b. Recibo un evento de *handle_vecino* y el mensaje contiene los 3 items de respuesta en el campo *answer_items*
    c. Mi vecino actualiza la respuesta a una de las preguntas
    d. Recibo un evento *handle_vecino* y el mensaje contiene sólo el item actualizado en el campo *answer_items*
5. El campo **health** y el campo **salud** pueden tratarse como equivalentes. El sistema no calcula de manera inmediata ninguno de los dos, sino que se calcula en eventos periódicos a partir de la fase 2 que pueden tener intervalos de un par de minutos. El campo **energia** se ha deshabilitado para este experimento. **No se debe tener en cuenta el campo energia para el bot.**



6. Hackathon Project: Configuración y Uso

Este proyecto funciona como un cliente que se conecta a un WebSocket para recibir eventos en tiempo real y realiza consultas a una API de IA. A continuación, se describen los pasos necesarios para instalar y ejecutar el cliente.

6.1. Instalación

Se ha configurado una instancia en Amazon Web Services (AWS) para cada equipo. Para acceder a la máquina remota, sigue estos pasos:

1. Descarga el archivo botaccess.pem que se te ha proporcionado.
2. Abre una terminal y navega hasta la ubicación donde guardaste el archivo .pem.
3. Establece los permisos adecuados al archivo de clave privada:
4. chmod 400 username.pem
5. Conéctate a la máquina remota con la IP_ASIGNADA que se te enviará junto con el archivo.pem y el usuario y contraseña usando el siguiente comando:
6. ssh -i username.pem -J portal@ec2-18-100-236-99.eu-south-2.compute.amazonaws.com username@IP_ASIGNADA
    - -i username.pem indica la clave privada para autenticación.
    - -J portal@ec2-18-100-236-99.eu-south-2.compute.amazonaws.com establece una conexión a través del host de salto.
    - username@IP_ASIGNADA es la dirección interna de la máquina en la que ejecutarás el cliente.

Para trabajar más cómodamente en el código en la máquina remota, puedes usar un IDE como Visual Studio Code y su extensión de conexión remota por SSH.

- Instala Visual Studio Code https://code.visualstudio.com/ y la extensión “Remote - SSH”
- Una vez instalado “Remote SSH”, abre VS Code y pincha en la parte inferior izquierda sobre “><”. Luego pincha en “Connect current window to Host”
- Pincha en Configure SSH Hosts
- Allí elige un fichero de configuración. Suelen tener un nombre tipo “config”



- Agrega las siguientes líneas al final del fichero teniendo en cuenta que te proporcionaremos una IP_ASIGNADA, un USER y un fichero *username.pem*

```
Host botHackaton

ProxyJump portal@ec2-18-100-236-99.eu-south-2.compute.amazonaws.com

HostName IP_ASIGNADA

User USER

#el pem file:

IdentityFile /ruta_aL_fichero/username.pem
```

- Guarda el fichero y vuelve a pinchar sobre “Connect current window to Host”. Aparecerá el nombre de botHackaton. Pincha sobre “botHackaton” y espera a que se conecte. Una vez dentro, ya podrás trabajar sobre el código “File” -> “Open Folder” y aparecerá el código

6.2. Estructura del Proyecto

Una vez que accedas a la máquina, encontrarás el código ya disponible con la siguiente estructura:

```
├── main.py            # Archivo principal que inicia el cliente WebSocket
├── agent.py           # Manejadores de eventos recibidos
├── config.py          # Configuración global del proyecto
├── utils/
│   ├── ws_connection.py   # Gestión de la conexión WebSocket
│   ├── api.py             # Interacción con la API de IA
```




```
│   ├── encryption.py      # Funciones de encriptación
├── requirements.txt       # Lista de dependencias del proyecto
├── README.md              # Este documento (o la versión online)
```

Notas:

- Se requiere conexión a Internet para comunicarse con la API de IA y el WebSocket.
- Asegúrate de que project_id sea válido y que las credenciales de usuario (--username, --password) sean correctas.
- En caso de errores, revisa los logs generados en consola para depuración.

6.3. Uso

Para iniciar el cliente y conectar a la plataforma Kampal Collective Learning, necesitas conocer username, password y project_id del proyecto en el que vas a trabajar. Utiliza el siguiente comando:

```
python main.py --username={username} --password={password} --project_id={project_id}
```

Donde:

- --username: Nombre de usuario para autenticación.
- --password: Contraseña para autenticación.
- --project_id: Identificador único del proyecto en el que se va a participar.

7. Próximos Pasos

1. Clonar el repositorio en tu entorno local.
2. Personalizar la lógica de cada handler según tu estrategia (por ejemplo, usar distintas heurísticas para copiar o editar respuestas).



3. Probar el bot localmente o en el entorno que haya preparado la organización del hackathon.
4. Iterar y mejorar, asegurándote de que tu bot se adapta a los distintos eventos y Fases.

8. Conclusión

Este documento recoge los elementos esenciales para que puedas desarrollar un bot autónomo que colabore en la resolución colectiva de problemas en Kampal Collective Learning. Cada handler explica el formato de datos que recibirás y cómo podrías aprovecharlo. Aun así, la implementación y la estrategia que sigas para editar, copiar y proponer soluciones es completamente libre y depende de la creatividad e inteligencia que desees programar en tu bot.

¡Mucha suerte en el hackathon, y que tu bot contribuya a una solución óptima para todos!

@HACKATHON
SALUDDIGITAL

