Nom i Cognoms:	Carlos Medina Gálvez
URL Repositori Github:	https://github.com/CarlosMG06/DAM2_M0486_Practiques/tree/main/RA2_3_4/PR4.2

ACTIVITAT
Objectius: ● Familiaritzar-se amb el desenvolupament d'APIs REST utilitzant Express.js ● Aprendre a integrar serveis de processament de llenguatge natural i visió artificial ● Practicar la implementació de patrons d'accés a dades i gestió de bases de dades ● Desenvolupar habilitats en documentació d'APIs i logging ● Treballar amb formats JSON i processament de dades estructurades
Criteris d'avaluació: - Cada pregunta indica la puntuació corresponent
Entrega: - Repositori git que contingui el codi que resol els exercicis i, en el directori "doc", aquesta memòria resposta amb nom "memoria.pdf"

Punt de partida
https://github.com/jpala4-ieti/DAM-M0486-Tema3-RA6-PR4.2-Punt-Partida-25-26

Preparació de l'activitat

- Clonar el repositori de punt de partida
- Llegir els fitxers README*.md que trobaràs en els diferents directoris
- Assegurar-te de tenir una instància de MySQL/MariaDB funcionant
- Tenir accés a una instància d'Ollama funcionant (al centre te'n facilitem una)

Entrega

- URL de resopitori

Exercicis

Exercici 1 (2.5 punts)

L'objectiu de l'exercici és familiaritzar-te amb **xat-api**. Respon la les preguntes dins el requadre que trobaràs al final de l'exercici.

Configuració i Estructura Bàsica:

1. Per què és important organitzar el codi en una estructura de directoris com controllers/, routes/, models/, etc.? Quins avantatges ofereix aquesta organització?
2. Analitzant el fitxer server.js, quina és la seqüència correcta per inicialitzar una aplicació Express? Per què és important l'ordre dels middlewares?
3. Com gestiona el projecte les variables d'entorn? Quins avantatges ofereix usar dotenv respecte a hardcodejar els valors?

API REST i Express:

1. Observant chatRoutes.js, com s'implementa el routing en Express? Quina és la diferència entre els mètodes HTTP GET i POST i quan s'hauria d'usar cadascun?
2. En el fitxer chatController.js, per què és important separar la lògica del controlador de les rutes? Quins principis de disseny s'apliquen?
3. Com gestiona el projecte els errors HTTP? Analitza el middleware errorHandler.js i explica com centralitza la gestió d'errors.

Documentació amb Swagger:

1. Observant la configuració de Swagger a swagger.js i els comentaris a chatRoutes.js, com s'integra la documentació amb el codi? Quins beneficis aporta aquesta aproximació?
2. Com es documenten els diferents endpoints amb els decoradors de Swagger? Per què és important documentar els paràmetres d'entrada i sortida?
3. Com podem provar els endpoints directament des de la interfície de Swagger? Quins avantatges ofereix això durant el desenvolupament?

Base de Dades i Models:

1. Analitzant els models Conversation.js i Prompt.js, com s'implementen les relacions entre models utilitzant Sequelize? Per què s'utilitza UUID com a clau primària?
2. Com gestiona el projecte les migracions i sincronització de la base de dades? Quins riscos té usar `sync()` en producció?
3. Quins avantatges ofereix usar un ORM com Sequelize respecte a fer consultes SQL directes?

Logging i Monitorització:

1. Observant logger.js, com s'implementa el logging estructurat? Quins nivells de logging existeixen i quan s'hauria d'usar cadascun?
2. Per què és important tenir diferents transports de logging (consola, fitxer)? Com es configuren en el projecte?
3. Com ajuda el logging a debugar problemes en producció? Quina informació crítica s'hauria de loguejar?

1. Aquesta organització és important per:
    - permetre la escalabilitat si es volen afegir més models i endpoints o substituir-ne algun
    - assegurar que el codi sigui llegible i mantenible a llarg termini, ja que deixa clar la seva arquitectura per a revisions futures
    - facilitar la creació de tests efectius de la lògica
2. La seqüència correcta és:
   Carregar variables d'entorn i importacions → Crear instància d'Express → Configuració de middlewares → Configuració de documentació → Configuració de middlewares personalitzats → Rutes → Gestió d'errors
   L'ordre de middlewares és important perquè cadascun s'executa en el mateix ordre que es defineix:
    - les rutes requereixen peticions i JSON parsejat
    - logging abans de les rutes per poder registrar-les
    - la gestió d'errors ha d'estar al final del tot per poder capturar els errors possibles
3. Emmagatzema les variables d'entorn en fitxers dotenv. Això serveix per seguretat de les dades sensibles, per centralitzar la seva configuració, i per poder utilitzar diferents dades segons l'entorn (desenvolupament, producció o testing), com es veu en ".env.test".

1. S'implementa amb express.Router(). Utilitza POST per crear prompts, i GET per obtenir converses i llistar els models disponibles.
La principal diferència entre HTTP GET i POST és que GET envia informació en la URL i POST la envia en el cos de la petició. En general, GET s'ha d'utilitzar per fer consultes i POST s'ha d'utilitzar per fer operacions o enviar dades sensibles com contrasenyes o tokens.
2. És important per evitar codi que compleix múltiples funcions la vegada (ex: definició dels endpoints HTTP, validació de peticions, lògica de resposta i lògica de processos de l'aplicació). També pot evitar duplicació de codi en la definició de cada ruta. S'aplica el principi de responsabilitat única (SRP) i el principi de "Don't Repeat Yourself" (DRY).
3. Els gestiona amb una estructura uniforme: nom, missatge, stack, path i mètode. El fitxer errorHandler,js captura tots els errors de l'aplicació i dona una resposta adequada segons el seu nom. També els registra en el logger.

1.
En swagger.js es defineix la configuració central. Defineix metadades de l'API, els models de dades de la BBDD, les respostes i etiquetes per classificar els endpoints.

Els comentaris @swagger en chatRoutes.js a cada endpoint són parsejats per Swagger JSDoc i, combinats amb la configiuració, apareixen a la UI de Swagger.

Com el codi i la documentació estan conjunts, podem evitar documentació desactualitzada i no tenim que anar a un wiki extern per consultar-la. A més a més, com la documentació es puja a Git junt amb el codi, sabem quina documentació correspon a cada versió.

2.
Es documenten amb un resum de què fan (*summary*), l'esquema que fan servir (*schema*), exemples (*examples*) i respostes possibles (*responses*).
És important documentar els paràmetres d'entrada i sortida per evitar pèrdua de temps corregint errors simples com noms incorrectes de paràmetres, perquè Swagger pugui validar les peticions abans d'enviar-les i perquè nous ulls puguin entendre ràpidament el funcionament de les peticions.

3.
Des de la Swagger UI, podem provar els endpoints així:
Fem clic al botó "Try it Out" que apareix en el menú desplegable de cadascun, modifiquem els paràmetres (GET) o el body de la petició (POST) i fem clic a "Execute".

Pel que fa a avantatges, això agilitza les proves de funcionament, Swagger valida l'input abans d'enviar i els exemples definits a la documentació s'incorporen automàticament.

1.
La relació Conversation-Prompt s'implementa fent servir els mètodes hasMany i belongTo definits en models de Sequelize.

S'utilitza UUID com a clau primària perquè és un identificador únic *universal*, de manera que no farà conflicte amb altres bases de dades. A més a més, la seva impredictibilitat aporta seguretat dificultant atacs que endevinen IDs existents.

2.
Fa servir sync() per gestionar la sincronització. En la producció, això comporta riscos de pèrdua o inconsistència de dades, especialment en el cas extrem de "force: true".

3.
Usar l'ORM Sequelize ofereix principalment aquests avantatges:

- Seguretat: Atac d'injecció SQL no és possible
- Compatibilitat: Mateix codi per MySQL, PostgreSQL, SQLite...
- Validació: Centralitzada a la definició de cada model i automàtica

1.
S'implementa amb un format comú del namespace Winston, incloent marca de temps, captura d'errors, nivell de log, missatge i metadades.

Aquests són els nivells de log i els seus casos d'ús:
- Error: fallades crítiques que requereixen un adob immediat
- Warn: imprevistos que no impedeixen el funcionament, però l'alenteixen
- Info: informació general del funcionament
- Verbose: informació complexa i detallada (ex: flux d'execució llarg)
- Debug: informació per depurar (info tècnica, valors de variables, configuracions...)
- Silly: Informació ultra-detallada rarament necessària

2. És important per no tenir un punt únic de falla i poder recuperar les dades en cas de pèrdues. (caiguda de la consola, corrupció del fitxer...)
En el projecte, sempre es configura el transport per consola i especificant la ruta es configura el transport per fitxer.

3. A l'hora de corregir bugs, el logging ajuda a traçar els passos que han anat a parar al bug, a fer un seguiment del rendiment, i a correlar errors amb seccions del codi.

Aquests són els tipus d'informació crítica que s'han de loguejar:
- Autenticació i autorització
- Transaccions, commits i rollbacks
- Salut del sistema
- Configuració
- Crides a dependències externes
- Auditoria de compliment de regulacions de seguretat de dades
- Mètriques de negoci (ex: noves registracions d'usuaris clients)

Exercici 2 (2.5 punts)

Dins de **practica-codi** trobaràs **src/exercici2.js**

Modifica el codi per tal que, pels dos primers jocs i les 2 primeres reviews de cada joc, crei una estadística que indiqui el nombre de reviews positives, negatives o neutres.

Modifica el prompt si cal.

Guarda la sortida en el directori data amb el nom **exercici2_resposta.json**

Exemple de sortida

```
{
  "timestamp": "2025-01-09T12:30:45.678Z",
  "games": [
    {
      "appid": "730",
      "name": "Counter-Strike 2",
      "statistics": {
        "positive": 1,
        "negative": 0,
        "neutral": 1,
        "error": 0
      }
    },
    {
      "appid": "570",
      "name": "Dota 2",
      "statistics": {
        "positive": 1,
        "negative": 1,
        "neutral": 0,
        "error": 0
      }
    }
  ]
}
```

Exercici 3 (2.5 punts)

Dins de **practica-codi** trobaràs **src/exercici3.js**

Modifica el codi per tal que retorni un anàlisi detallat sobre l'animal.
Modifica el prompt si cal.

La informació que volem obtenir és:

- Nom de l'animal.
- Classificació taxonòmica (mamífer, au, rèptil, etc.)
- Hàbitat natural
- Dieta
- Característiques físiques (mida, color, trets distintius)
- Estat de conservació

Guarda la sortida en el directori **data** amb el nom **exercici3_resposta.json**

```
{
    "analisis": [
        {
            "imatge": {
                "nom_fitxer": "nom_del_fitxer.jpg",
            },
            "analisi": {
                "nom_comu": "nom comú de l'animal",
                "nom_cientific": "nom científic si és conegut",
                "taxonomia": {
                    "classe": "mamífer/au/rèptil/amfibi/peix",
                    "ordre": "ordre taxonòmic",
                    "familia": "família taxonòmica"
                },
                "habitat": {
                    "tipus": ["tipus d'hàbitats"],
                    "regioGeografica": ["regions on viu"],
                    "clima": ["tipus de climes"]
                },
                "dieta": {
                    "tipus": "carnívor/herbívor/omnívor",
                    "aliments_principals": ["llista d'aliments"]
                },
                "caracteristiques_fisiques": {
                    "mida": {
                        "altura_mitjana_cm": "altura mitjana",
                        "pes_mitja_kg": "pes mitjà"
                    },
                    "colors_predominants": ["colors"],
                    "trets_distintius": ["característiques"]
                },
                "estat_conservacio": {
                    "classificacio_IUCN": "estat",
                    "amenaces_principals": ["amenaces"]
                }
            }
        }
    ]
}
```

Exercici 4 (2.5 punts)

Implementa un nou endpoint a xat-api per realitzar anàlisi de sentiment

Haurà de complir els següents requisits

- Estar disponible a l'endpoint POST /api/chat/sentiment-analysis
- Disposar de documentació swagger
- Emmagatzemar informació a la base de dades
- Usar el logger a fitxer

Abans d'implementar la tasca, explica en el requadre com la plantejaràs i fes una proposta de json d'entrada, de sortida i de com emmagatzemaràs la informació a la base de dades.

Primer, implementaré l'endpoint /api/chat/sentiment-analysis en chatRoutes.js, afegint documentació @swagger definint la seva estructura. La documentació i la funció seran similars a /api/chat/prompt/, però amb una prompt preestablerta:

```
`Analitza el sentiment del següent text. Conclou si és un text negatiu,
neutral o positiu:\n\n ${text.trim()}`
```

Crearé un nou schema Swagger anomenat SentimentAnalysisRequest.
En el body del schema, hi haurà les propietats "conversationId", "text" (el text a analitzar), "model" i "stream".

Emmagatzemaré la informació a la base de dades fent servir els models Prompt i Conversation ja definits.
Les respostes seran les mateixes que /api/chat/prompt.