Nom i Cognoms:	Victor Asensio Bermúdez
URL Repositori Github:	victor06-git/components-dades-nodejs

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 respositori

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?

   És importants organitzar el codi en una estructura com controllers/, routes/, schemes/, models/ perquè fan el codi més estructurat, llegible i optimizat. A part de ser entendible per futurs programadors ajuda a l'organització del codi. Els avantatges que ofereix són: poguer trobar més ràpidament els errors, aconseguir millors resultats, etc.

2. Analitzant el fitxer server.js, quina és la seqüència correcta per inicialitzar una aplicació Express? Per què és important l'ordre dels middlewares?

La seqüència correcta per inicialitzar l'aplicació Express és cors(), express.json(), /api-docs, app.use(...) –> next(), expressLogger, /api/chat/, errorHandler.
Amb aquest ordre la funcionalitat és perfecte, si es cambia l'ordre no funcionará del tot bé. Per exemple, cors que sol anar al principi per validar si la petició ve d'un domini no autoritzat. I també, errorHandler que sempre ha d'anar al final perquè Express funciona en Cascada, si ocorre un error en chatRoutes, es crida a next(error).

3. Com gestiona el projecte les variables d'entorn? Quins avantatges ofereix usar dotenv respecte a hardcodejar els valors?

   Les variables d'entorn les gestiona el projecte mitjançant el fitxer .env. L'utilització d'aquest fitxer en comptes de hardcodejar els valors és principalment perquè .env conté les claus reals però mai es puja al repositori, així el codi Github és segur, i cada desenvolupador té el seu propi ".env" privat.

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?

En aquest fitxer, s'utilitzen funcions com .get(), .post() sobre l'objecte router per definit com respondre a peticions específiques. El segment :id és un paràmetre dinàmic. Express capturarà qualsevol valor que es posicioni allà a la URL i el passarà al controlador dins de req.paramas.id. I finalment, s'exporta al router. La diferència entre els dos mètodes,

get permet obtenir informació del servidor, en canvi post envia informació per crear o processar dades. Les dades en **get**() es troben a la URL, en canvi a **post**() es troba al cos de la petició. I **get**() és menys segur per a dades sensibles, en canvi **post**() és més segur, ja que les dades no queden a l'historial. En el cas de només llegir o recuperar dades sense modificar res al servidor s'utilitza més convenientment el métode GET(), en canvi si quan l'acció modifica l'estat del servidor o crea un recurs nou s'utilitza POST().

2. En el fitxer chatController.js, per què és important separar la lògica del controlador de les rutes? Quins principis de disseny s'apliquen?

   Separar la lògica del controlador de les rutes és clau per seguir el principi **SOLID** de responsabilitat única (**SRP**): les rutes només gestionen el trànsit, mentre que el controlador executa la lògica de negoci. Això facilita el **manteniment** i la **reutilització** de codi (principi **DRY**), permetent que funcions com generateResponse s'utilitzin en diferents llocs. A més, millora radicalment la **testabilitat.**

3. Com gestiona el projecte els errors HTTP? Analitza el middleware errorHandler.js i explica com centralitza la gestió d'errors.

   Aquest middleware actua com una **xarxa de seguretat final** que captura qualsevol excepció que es produeixi en el cicle de vida d'una petició HTTP.

Punt d'entrada únic: En lloc de posar blocs try/catch amb respostes personalitzades a cada ruta.

Filtratge per tipus (Categorització): El middleware analitza la propietat `err.name` per identificar errors específics de la base de dades (**Sequelize**).

Abstracció de Seguretat: Per a errors genèrics (500), el middleware filtra la informació sensible. Utilitza la variable d'entorn NODE_ENV per decidir si mostra el detall de l'error (útil en desenvolupament) o un missatge genèric (seguretat en producció).

Observabilitat: Centralitza el registre a través del `logger`. Això permet que tot error, a més de respondre al client, quedi registrat amb el seu *stack trace*, mètode i ruta per a una anàlisi posterior.

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?

Mecanisme d'Integració

L'integració es basa en un sistema de parsing d'anotacions:

- **L'escaneig**: Quan l'aplicació arrenca, swagger-jsdoc llegeix el fitxer swagger.js, que té una "ruta de recerca" (apis: ['./src/routes/*.js']).
- **L'extracció**: La llibreria busca els blocs de comentaris que comencen amb /** @swagger.
- **El mapatge**: Swagger associa cada definició YAML (com /api/chat/prompt) amb la ruta real d'Express que té just a sota (router.post('/prompt', ...)).

Beneficis de l'aproximació "Code-first"

1. **Documentació com a "Contracte Viu"**: Al fitxer chatRoutes.js veiem paràmetres com stream o conversationId. Si un desenvolupador canvia el nom d'un camp al codi però no a Swagger, la discrepància es detecta ràpidament perquè estan al mateix fitxer.
2. **Facilitat per al Front-end**: La documentació generada inclou exemples (com el prompt per defecte sobre cançons poc conegudes). Això permet que l'equip de Front-end sàpiga exactament què enviar sense llegir la lògica del controlador.
3. **Proves Sandbox (Interactivitat)**: Swagger UI utilitza aquestes rutes per generar botons "Try it out". Això permet testejar el comportament de l'API (per exemple, veure com respon l'anàlisi de sentiment) des del navegador, estalviant temps de desenvolupament.
4. **Autodocumentació d'Errors**: Es defineixen clarament els codis 201, 400 i 404. Això força el desenvolupador a pensar en els casos d'error mentre escriu la ruta.

2. Com es documenten els diferents endpoints amb els decoradors de Swagger? Per què és important documentar els paràmetres d'entrada i sortida?

   Els endpoints es documenten mitjançant blocs de comentaris JSDoc amb l'etiqueta @swagger, que utilitzen format **YAML** per definir rutes, mètodes i etiquetes. S'utilitzen decoradors com parameters per a dades a l'URL, requestBody per a objectes JSON i responses per enumerar els possibles codis d'estat (200, 400, 500)

3. Com podem provar els endpoints directament des de la interfície de Swagger? Quins avantatges ofereix això durant el desenvolupament?

Per provar els endpoints directament des de la interfície de Swagger (habitualment accessible a /api-docs), s'utilitza la funcionalitat interactiva que genera automàticament swagger-ui-express.

Com fer-ho pas a pas

1. **Selecció de l'endpoint:** Fas clic sobre la ruta que vols provar (per exemple, POST /api/chat/prompt).
2. **Activar el mode d'edició:** Prem el botó **"Try it out"**. Això desbloqueja els camps d'entrada.
3. **Configuració de dades:** Omples el requestBody amb el JSON corresponent o els paràmetres de la URL. Swagger ja et proporciona un "Schema" d'exemple per defecte.
4. **Execució:** Prem el botó blau **"Execute"**. La interfície farà una petició real al teu servidor.
5. **Inspecció:** Just a sota apareixerà la resposta real del servidor (codi d'estat, cos de la resposta i capçaleres), a més de la comanda curl equivalent.

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?

   Les relacions es defineixen mitjançant associacions de Sequelize: **Conversation.hasMany**(Prompt) i **Prompt.belongsTo**(Conversation). Això crea una clau forana ConversationId a la taula de prompts, establint un vincle **1:N**.

   L'ús d'**UUID** com a clau primària és fonamental per la **seguretat**, ja que impedeix que algú endevini IDs de converses alienes incrementant números a la **URL**.

   A més, facilita l'**escalabilitat** en sistemes distribuïts, permetent generar identificadors únics sense consultar la base de dades central, i evita col·lisions de dades durant migracions o fusions de taules.

2. Com gestiona el projecte les migracions i sincronització de la base de dades? Quins riscos té usar `sync()` en producció?

El projecte gestiona la base de dades principalment mitjançant **Sequelize**, que ofereix dues vies: el mètode automàtic sync() i el sistema de migracions manual.

Gestió de la Base de Dades

Sincronització (sync()): Sequelize compara el codi dels teus models (Conversation.js, Prompt.js) amb les taules de la base de dades i les crea o modifica automàticament per reflectir els canvis.

Migracions: Són fitxers de control de versions per a la base de dades. Permeten descriure canvis (com afegir una columna) de forma incremental i reversible, facilitant el treball en equip.

3. Quins avantatges ofereix usar un ORM com Sequelize respecte a fer consultes SQL directes?

L'ús de Sequelize ofereix **seguretat nativa** contra la injecció SQL i una gran **abstracció**, permetent canviar de motor de base de dades sense refer el codi. Millora la **productivitat** en utilitzar sintaxi de JavaScript en lloc de cadenes de text SQL, fent el projecte més llegible i fàcil de mantenir. També automatitza la gestió de relacions complexes i validacions de dades directament als models. Tot i que el SQL pur pot ser més ràpid en consultes extremadament complexes, l'ORM redueix errors humans i accelera el desenvolupament. A més, facilita la gestió del cicle de vida de les dades amb **hooks** i migracions integrades.

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?

L'implementació utilitza Winston per crear un sistema de logs flexible que combina **timestamp, metadades i stack traces** d'errors. El logging estructurat permet passar objectes JSON com a segon argument, facilitant la depuració sense embrutar el missatge principal. Mitjançant DailyRotateFile, el projecte gestiona el cicle de vida dels fitxers per evitar que ocupin massa espai. La configuració a través de process.env.LOG_LEVEL permet filtrar el soroll: en producció se solen registrar nivells info o superiors, mentre que en desenvolupament s'usa debug per seguir tot el flux. Finalment, el middleware expressLogger automatitza el registre de cada petició HTTP, aportant traçabilitat total a l'aplicació.

2. Per què és important tenir diferents transports de logging (consola, fitxer)? Com es configuren en el projecte?

   És molt important perquè et permet tenir un millor control dels errors que ocorren per consola com poder visualitzar també el que ha passat durant X dias, actuant com una memòria històrica que evita que la informació es perdi quan es tanca la terminal. Aquest sistema de **logging estructurat** amb Winston permet que, si un error va ocórrer fa tres matinades, puguis anar al fitxer corresponent i analitzar el *stack trace* i les metadades (com la IP o el model d'Ollama usat) per recrear l'escenari exacte del problema.

3. Com ajuda el logging a debugar problemes en producció? Quina informació crítica s'hauria de loguejar?

Reconstrucció d'escenaris: Permet reproduir errors intermitents que només ocorren sota condicions de càrrega o amb dades d'entrada molt específiques que no es repliquen en local.
Visibilitat sense interrupció: Pots inspeccionar l'estat de les variables i el flux d'execució sense aturar el servei ni afectar l'experiència de l'usuari.
Anàlisi de latències: Mitjançant el registre de temps (com fas amb `expressLogger`), pots identificar colls d'ampolla en serveis externs (Ollama) o consultes lentes a la base de dades.

Perquè un log sigui útil, ha de ser **estructurat** i contenir context suficient per ser accionable:

1. **Context d'Identificació:** IDs únics de conversa (`conversationId`), de prompt i de petició (`request-id`). Això permet filtrar totes les línies de log relacionades amb una sola transacció.
2. **Stack Traces d'Errors:** El missatge d'error per si sol és insuficient; cal el rastre complet de la pila per localitzar el fitxer i la línia exacta de la fallada.
3. **Metadades de l'Entorn:** El model d'IA utilitzat, la versió de l'API, i si s'ha utilitzat mode `stream`.
4. **Dades del Client:** Adreça IP, mètode HTTP, URL de l'endpoint i codi d'estat de la resposta.
5. **Indicadors de Temps:** Timestamp precís i la durada total del processament de la petició

En producció és molt important el logging perquè proporciona traçabilitat.

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.

Definiré el json d'entrada d'aquesta manera: '{ "text" : "Aquest joc té diferents modes que m'agraden però començo la batalla final i em sembla una idea molt mala en implementar diferents atacs que no funcionen seqüencialment." }'
El json de sortida serà d'aquesta manera: '{ "id": "f47ac10b-58cc-4372-a567-0e02b2c3d479", "text": "Aquest joc té diferents modes que m'agraden però començo la batalla final i em sembla una idea molt mala en implementar diferents atacs que no funcionen seqüencialment.", "sentiment": "positivo", "score": 0.98, "createdAt": "2024-05-20T10:00:00.000Z", "updatedAt": "2024-05-20T10:00:00.000Z" }'
I alhora d'emmagatzemar la informació ho faré d'aquesta manera: **'id**: generat automàticament pel model **text**: el comentari, ve del json d'entrada **sentiment**: calculat segons la lógica del controlador (positive | negative) **score**: calculat/generat segons la lógica del controller **createdAt**: generat automàticament per sequelize **updatedAt**: generat automàticament per sequelize'