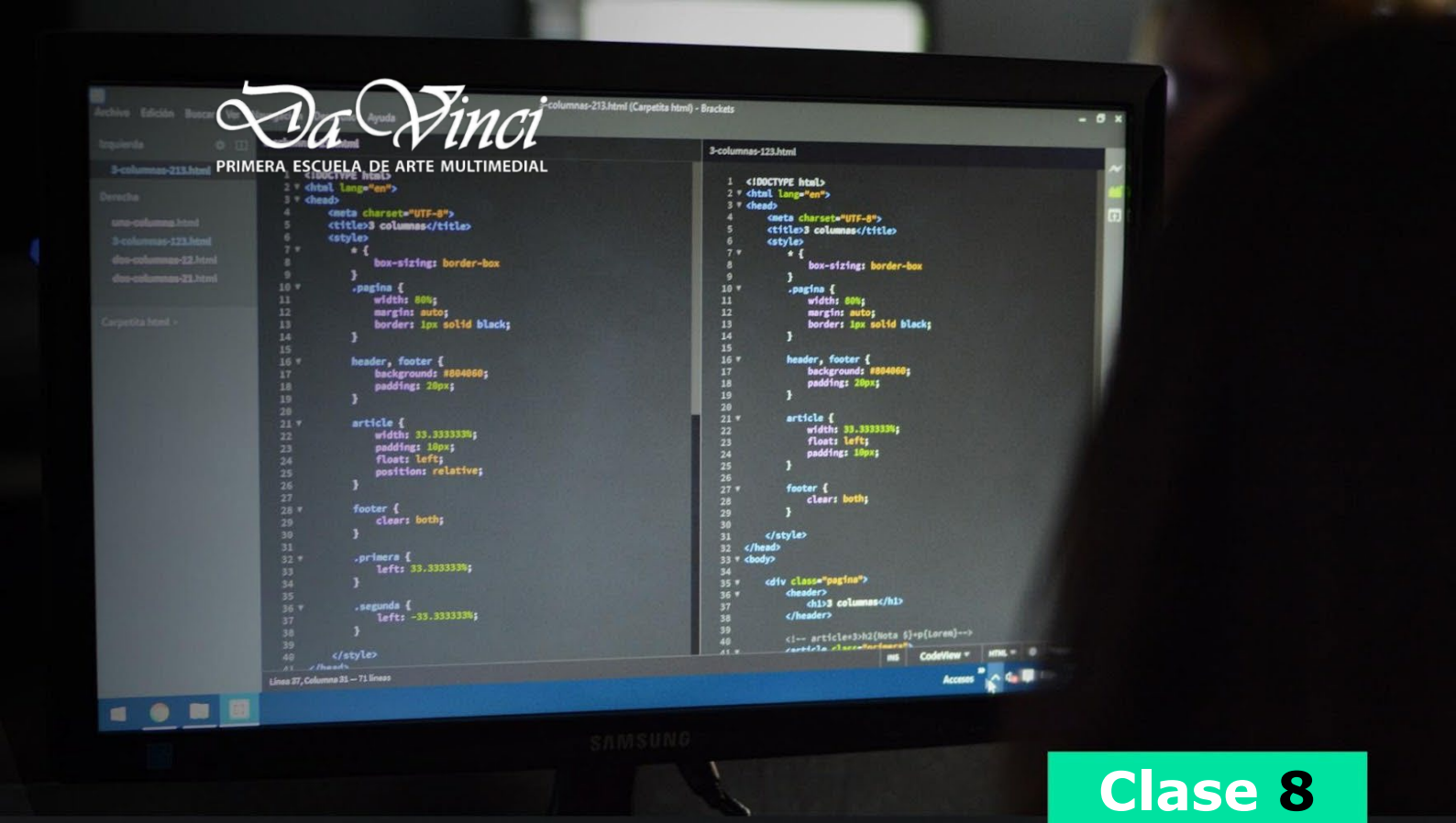

Clase 8

Diseño y Programación Web

Materia:
Aplicaciones Híbridas

Docente contenidista: MARCOS GALBÁN, Camila Belén

Mongoose

Mongoose es una biblioteca de Node.js para interactuar con bases de datos MongoDB. Facilita la conexión y el trabajo con MongoDB proporcionando una capa de abstracción que maneja gran parte de la complejidad de las operaciones de la base de datos.

Características

Modelos y Esquemas

Mongoose utiliza esquemas para definir la estructura de los documentos dentro de una colección, permitiendo definir campos, tipos de datos, validaciones y valores predeterminados.

Validación

Mongoose proporciona validación de datos al nivel del esquema, garantizando que los datos almacenados en la base de datos cumplan con ciertos requisitos.

Consultas y Operaciones CRUD

Facilita la creación de consultas y la ejecución de operaciones CRUD (Crear, Leer, Actualizar, Eliminar).

Middleware (Hooks)

Permite definir middleware (pre y post hooks) que se ejecutan durante ciertas etapas del ciclo de vida de un documento, como antes de guardar o después de eliminar.

Population

Ofrece una funcionalidad para referenciar documentos en otras colecciones, similar a una relación entre tablas en bases de datos relacionales.

Plugins

Permite la extensión de esquemas con funcionalidad reutilizable mediante plugins.

Configuración de la conexión a MongoDB

Utiliza el método connect de Mongoose para establecer una conexión con tu base de datos MongoDB. Puedes especificar la URL de conexión y las opciones de configuración necesarias.

Ejemplo

```
const mongoose = require('mongoose');

// Conexión a la base de datos
mongoose.connect('mongodb://localhost:27017/mydatabase',
{ useNewUrlParser: true, useUnifiedTopology: true });

// Definición del esquema
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  age: Number
});

// Creación del modelo basado en el esquema
const User = mongoose.model('User', userSchema);

// Creación de un nuevo usuario
const newUser = new User({
  name: 'John Doe',
  email: 'john.doe@example.com',
  age: 30
});

// Guardar el usuario en la base de datos
newUser.save((err) => {
  if (err) return console.error(err);
  console.log('Usuario guardado con éxito');
});

// Búsqueda de usuarios
User.find({ name: 'John Doe' }, (err, users) => {
  if (err) return console.error(err);
  console.log(users);
});
```

Explicación:

- **Importar Mongoose**: Importar la biblioteca Mongoose.
- **URL de Conexión**: Definir la URL de conexión a la base de datos MongoDB. Si se está usando un servicio de MongoDB en la nube, la URL será diferente.
- **Opciones de Configuración**: Se especifican opciones para manejar la forma en que Mongoose interactúa con MongoDB.
- **Conectar a la Base de Datos**: Usar mongoose.connect para establecer la conexión. then y catch se utilizan para manejar la promesa de la conexión.
- **Manejo de Eventos de Conexión**: Utilizar db.on y db.once para manejar eventos como errores de conexión o una conexión exitosa.
- **Definición del Esquema**: Definir un esquema para los documentos.
- **Creación del Modelo**: Crear un modelo basado en el esquema.
- **Guardar un Documento**: Crear y guardar un nuevo documento en la base de datos.

Las opciones de configuración en Mongoose al conectarse a MongoDB ayudan a controlar varios aspectos del comportamiento de la conexión y la interacción con la base de datos. A continuación, se describen las opciones que se mencionaron en el ejemplo:

useNewUrlParser: true

- Esta opción permite que Mongoose use el nuevo analizador de URL de MongoDB. El nuevo analizador es más robusto y corrige varias inconsistencias y problemas con el antiguo analizador.
- Se recomienda siempre establecer esto en true para futuras versiones y mejoras.

useUnifiedTopology: true

- Esta opción activa el nuevo motor de administración de conexiones introducido en MongoDB 3.2.
- Mejora la estabilidad y el manejo de eventos relacionados con la conexión, como la recuperación de errores de red y cambios en la topología del clúster.
- Es muy recomendable establecer esto en true para obtener los beneficios de las mejoras en el manejo de conexiones.

useFindAndModify: false

- Esta opción configura Mongoose para que use las funciones de findOneAndUpdate(), findOneAndRemove() en lugar de las funciones findAndModify() de MongoDB.
- Las funciones findOneAndUpdate() y findOneAndRemove() son más consistentes con el comportamiento de Mongoose y proporcionan una API más clara.
- Se recomienda establecer esto en false para mantener la consistencia y evitar advertencias de deprecación.

useCreateIndex: true

- Esta opción hace que Mongoose use createIndex() en lugar de ensureIndex() para crear índices en MongoDB.
- createIndex() es el método recomendado y el futuro soporte para ensureIndex() está en desuso.
- Es recomendable establecer esto en true para evitar advertencias de deprecación y asegurar la compatibilidad futura.

```
// Opciones de configuración (recomendadas)
const options = {
    useNewUrlParser: true, // Usar el nuevo analizador de URL
    useUnifiedTopology: true, // Activar el nuevo motor de administración
de conexiones
    useFindAndModify: false, // Usar findOneAndUpdate() en lugar de
findAndModify()
    useCreateIndex: true // Usar createIndex() en lugar de ensureIndex()
  };

  // Establece la conexión
  mongoose.connect(mongoDBUrl, options)
    .then(() => {
      console.log('Conexión a la base de datos establecida con éxito');
    })
    .catch((error) => {
      console.error('Error al conectar a la base de datos:', error);
    });
```

Operaciones CRUD

Leer (Read)

Para leer documentos, se utilizan los métodos find(), findOne(), findById(), entre otros.

Encontrar todos los documentos:

```
User.find({}, (err, users) => {
if (err) console.error(err);
console.log(users);
});
```

Encontrar un documento por una condición:

```
User.findOne({ email: 'john.doe@example.com' }, (err, user) => {
if (err) console.error(err);
console.log(user);
});
```

Encontrar un documento por ID:

```
User.findById('60d5ec49e8d8e24f1c8d4e77', (err, user) => {
if (err) console.error(err);
console.log(user);
});
```

Actualizar (Update)

Para actualizar documentos, se utilizan métodos como updateOne(), updateMany(), findByIdAndUpdate(), findOneAndUpdate(), entre otros.

Actualizar un documento por una condición:

```
User.updateOne({ email: 'john.doe@example.com' }, { edad: 35 }, (err,
res) => {
    if (err) console.error(err);
    console.log(res);
  });
```

Actualizar múltiples documentos:

```
  User.updateMany({ edad: { $lt: 18 } }, { menor: false }, (err, res) =>
{
```

```
    if (err) console.error(err);
    console.log(res);
  });
```

Encontrar por ID y actualizar:

```
  User.findByIdAndUpdate('60d5ec49e8d8e24f1c8d4e77', { edad: 36 }, { new:
true }, (err, user) => {
    if (err) console.error(err);
    console.log(user);
  });
```

Encontrar uno y actualizar:

```
  User.findOneAndUpdate({ email: 'john.doe@example.com' }, { edad: 37 },
{ new: true }, (err, user) => {
    if (err) console.error(err);
    console.log(user);
  });
```

Eliminar (Delete)

Para eliminar documentos, se utilizan métodos como deleteOne(), deleteMany(), findByIdAndDelete(), findOneAndDelete(), entre otros.

Eliminar un documento por una condición:

```
  User.deleteOne({ email: 'john.doe@example.com' }, (err) => {
    if (err) console.error(err);
    console.log('Usuario eliminado');
  });
```

Eliminar múltiples documentos:

```
  User.deleteMany({ edad: { $lt: 18 } }, (err) => {
    if (err) console.error(err);
    console.log('Usuarios eliminados');
  });
```

Encontrar por ID y eliminar:

```
 User.findByIdAndDelete('60d5ec49e8d8e24f1c8d4e77', (err) => {
    if (err) console.error(err);
    console.log('Usuario eliminado');
  });
```

Encontrar uno y eliminar:

```
User.findOneAndDelete({ email: 'john.doe@example.com' }, (err) => {
   if (err) console.error(err);
   console.log('Usuario eliminado');
 });
```

Documentos embebidos y referencias

En MongoDB y Mongoose, hay dos formas principales de modelar relaciones entre documentos: documentos embebidos y referencias. Ambas tienen sus propias ventajas y desventajas, y la elección entre ellas depende de los requisitos específicos de tu aplicación y las operaciones que planeas realizar con los datos.

Documentos Embebidos

Descripción

Los documentos embebidos almacenan datos de subdocumentos directamente dentro del documento principal. Esta estructura es anidada y se utiliza cuando los datos relacionados están estrechamente asociados y se accede a ellos juntos frecuentemente.

Ventajas

- **Desempeño**: Las operaciones de lectura y escritura son rápidas porque no se requiere realizar múltiples consultas.
- **Atomicidad**: Las operaciones en un solo documento son atómicas, lo que simplifica las transacciones.
- **Simplicidad**: Facilita el diseño del esquema cuando los datos relacionados son pequeños y no cambian con frecuencia.

Desventajas

- **Limitaciones de tamaño**: Los documentos en MongoDB tienen un límite de 16 MB, lo que puede ser un problema si los subdocumentos crecen mucho.
- **Actualizaciones complejas**: Las actualizaciones parciales pueden ser más difíciles si solo una parte del subdocumento necesita ser cambiada frecuentemente.

Referencias

Descripción

Las referencias almacenan la relación entre documentos utilizando identificadores, similares a las claves foráneas en bases de datos relacionales. Los documentos relacionados se almacenan en colecciones separadas y se pueden recuperar mediante consultas adicionales.

Ventajas

- **Flexibilidad**: No hay limitaciones de tamaño porque los datos relacionados se almacenan en documentos separados.
- **Escalabilidad**: Es más fácil escalar cuando los datos relacionados crecen significativamente.
- **Separación de preocupaciones**: Permite una mejor organización y modularidad de los datos.

Desventajas

- **Desempeño**: Requiere múltiples consultas para recuperar datos relacionados, lo que puede ser más lento.
- **Complejidad**: Las operaciones pueden ser más complejas debido a la necesidad de manejar la consistencia de las relaciones.

¿Cuándo usar cada uno?

Documentos embebidos:

- Cuando los datos relacionados son pequeños y se accede a ellos junto con el documento principal.
- Cuando se necesita operaciones atómicas.
- Ejemplo: Una publicación de blog con unos pocos comentarios.

Referencias:

- Cuando los datos relacionados son grandes o pueden crecer indefinidamente.
- Cuando los datos relacionados tienen su propia vida útil y pueden ser accedidos independientemente.
- Ejemplo: Una aplicación de comercio electrónico donde un producto tiene muchas reseñas.

La elección entre usar documentos embebidos y referencias en una aplicación Node.js con MongoDB depende en gran medida del caso de uso específico, las necesidades de rendimiento y la estructura de los datos. Aquí hay algunos ejemplos de tipos de aplicaciones donde cada enfoque podría ser más adecuado:

Documentos Embebidos

Aplicaciones de Blog:

- **Uso**: Publicaciones de blog con comentarios embebidos.
- **Razonamiento**: Los comentarios suelen ser pequeños y se accede a ellos frecuentemente junto con la publicación. Las operaciones atómicas garantizan que una publicación y sus comentarios se manejen de manera coherente.
- **Ejemplo**: Cada publicación contiene una lista de comentarios embebidos.

Aplicaciones de Gestión de Tareas:

- **Uso**: Tareas con subtareas embebidas.
- **Razonamiento**: Las subtareas son pequeñas y se accede a ellas junto con la tarea principal. Se benefician de operaciones atómicas.
- **Ejemplo**: Cada tarea contiene una lista de subtareas embebidas.

Aplicaciones de Perfil de Usuario:

- **Uso**: Perfiles de usuario con direcciones embebidas.
- **Razonamiento**: Las direcciones están estrechamente asociadas con el perfil del usuario y se accede a ellas frecuentemente junto con el perfil.
- **Ejemplo**: Cada perfil de usuario contiene una lista de direcciones embebidas.

Referencias

Aplicaciones de Comercio Electrónico:

- **Uso**: Productos y reseñas referenciadas.
- **Razonamiento**: Las reseñas pueden ser numerosas y acceder a ellas de manera independiente del producto es beneficioso. Permite escalar mejor con colecciones grandes.
- **Ejemplo**: Cada producto referencia una lista de reseñas.'

Aplicaciones de Redes Sociales:

- **Uso**: Usuarios y publicaciones referenciadas.
- **Razonamiento**: Las publicaciones pueden ser numerosas y es beneficioso acceder a ellas de manera independiente del usuario. Además, las publicaciones pueden referenciar otros usuarios, como en menciones o etiquetados.
- **Ejemplo**: Cada usuario referencia una lista de publicaciones.

Aplicaciones de Gestión de Proyectos:

- **Uso**: Proyectos y tareas referenciadas.
- **Razonamiento**: Las tareas pueden ser numerosas y acceder a ellas de manera independiente del proyecto es beneficioso. Permite manejar grandes volúmenes de datos y mantener la flexibilidad.
- **Ejemplo**: Cada proyecto referencia una lista de tareas.

Consideraciones Adicionales

Frecuencia de Acceso:

- Si se accede frecuentemente a los datos relacionados junto con el documento principal, los documentos embebidos pueden ser más eficientes.
- Si los datos relacionados se acceden de manera independiente o rara vez junto con el documento principal, las referencias pueden ser más adecuadas.

Tamaño y Crecimiento de los Datos:

- Si los datos relacionados son pequeños y no crecen significativamente, los documentos embebidos pueden ser adecuados.

- Si los datos relacionados son grandes o pueden crecer significativamente, las referencias son preferibles para evitar problemas de tamaño de documento.

Consistencia y Atomicidad:

- Si se necesita operaciones atómicas y consistencia dentro de un documento, los documentos embebidos son la mejor opción.
- Si se puede manejar la consistencia a nivel de aplicación y no se necesita operaciones atómicas en los datos relacionados, las referencias pueden ser suficientes.

Validaciones

La validación de datos es crucial en cualquier aplicación para asegurar la integridad y consistencia de los datos que se procesan y almacenan. En aplicaciones Node.js, se pueden realizar validaciones tanto en el esquema de Mongoose como antes de insertar o actualizar datos en la base de datos utilizando una biblioteca de validación como Joi.

Validaciones en Mongoose

Mongoose proporciona mecanismos para validar datos en el esquema del modelo. Estas validaciones se ejecutan automáticamente cuando se intenta guardar un documento.

Ejemplo de validación con Mongoose:

```
const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/mydatabase', {
  useNewUrlParser: true,
  useUnifiedTopology: true
});

const userSchema = new mongoose.Schema({
  nombre: {
    type: String,
    required: [true, 'El nombre es obligatorio'],
    minlength: [3, 'El nombre debe tener al menos 3 caracteres']
  },
  email: {
    type: String,
    required: [true, 'El correo electrónico es obligatorio'],
    unique: true,
    match: [/\S+@\S+\.\S+/, 'El correo electrónico no es válido']
```

```
  },
  edad: {
    type: Number,
    min: [0, 'La edad no puede ser negativa'],
    max: [120, 'La edad no puede ser mayor a 120']
  }
});

const User = mongoose.model('User', userSchema);

const newUser = new User({
  name: 'Jane',
  email: 'jane.doe@example.com',
  age: 25
});

newUser.save()
  .then(user => console.log('Usuario guardado con éxito:', user))
  .catch(err => console.error('Error al guardar el usuario:', err));
```

Validaciones con Joi

Joi es una poderosa biblioteca de validación de datos para Node.js que permite definir esquemas de validación flexibles y robustos.

Ejemplo de validación con Joi:

```
const Joi = require('joi');
const express = require('express');
const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/mydatabase', {
  useNewUrlParser: true,
  useUnifiedTopology: true
});

const userSchema = new mongoose.Schema({
  nombre: String,
  email: String,
  edad: Number
});

const User = mongoose.model('User', userSchema);

const app = express();
app.use(express.json());
```

```
const userValidationSchema = Joi.object({
  nombre: Joi.string().min(3).required().messages({
    'string.base': 'El nombre debe ser un texto',
    'string.min': 'El nombre debe tener al menos {#limit} caracteres',
    'any.required': 'El nombre es obligatorio'
  }),
  email: Joi.string().email().required().messages({
    'string.email': 'El correo electrónico no es válido',
    'any.required': 'El correo electrónico es obligatorio'
  }),
  edad: Joi.number().integer().min(0).max(120).messages({
    'number.base': 'La edad debe ser un número',
    'number.min': 'La edad no puede ser negativa',
    'number.max': 'La edad no puede ser mayor a {#limit}'
  })
});

app.post('/users', async (req, res) => {
  const { error, value } = userValidationSchema.validate(req.body);

  if (error) {
    return res.status(400).send(error.details[0].message);
  }

  try {
    const newUser = new User(value);
    const savedUser = await newUser.save();
    res.status(201).send(savedUser);
  } catch (err) {
    res.status(500).send(err.message);
  }
});

app.listen(3000, () => {
  console.log('Servidor escuchando en el puerto 3000');
});
```

También se puede combinar las validaciones de Joi y Mongoose para asegurarse de que los datos son válidos tanto antes de llegar al modelo de Mongoose como al guardarlos en la base de datos.

Esquemas en Mongoose

En Mongoose, un Schema (Esquema) es la estructura que indica cual es la forma en la que están estructurados los documentos que se almacenan en una colección de MongoDB. Cada schema está compuesto por campos y tipos de datos permitidos, además de opciones para llevar acabo validación de dichos documentos.

Los schemas se utilizan para especificar el modelo de los datos de una aplicación, en este caso Mongoose para una aplicación construida en NodeJS. Cuando se construye un schema se genera una estructura y reglas de validación de los datos, para que estos sean analizados y validados previo a su captura.

En otras palabras, los esquemas sirven como guías de la estructura de los documentos.

Mongoose ignora todas las propiedades que no sean definidas dentro del modelo de un esquema.

Los esquemas en Mongoose soportan los siguientes tipos de datos:

- String - Cadena de caracteres
- Number - Número
- Date - Fecha y hora
- Boolean - Valor booleano
- ObjectID - Identificador único de un documento
- Mixed - Tipo genérico que puede contener cualquier tipo de dato
- Buffer - Almacenamiento de datos binarios
- Array - Colección de elementos con un tipo específico

Estos son los validadores disponibles en los schemas de mongoose:

- Required - Requerido
- Min - Valor mínimo numérico
- Max - máximo numérico

- Minlength - Longitud mínima de una cadena
- Maxlength - Longitud máxima de una cadena
- Enum - Valor permitido de un conjunto de valores
- Match - Expresión regular para validar una cadena
- Validate - Validador personalizado para un campo
- Unique - Indica que el valor de este campo debe ser único en la colección
- Sparse - Permite que el campo sea nulo o no exista para algunos documentos