ESTRUCTURA DE LA BASE DE DATOS

Colección users:

```
{
  _id: ObjectId,
  username: "juanito123",
  email: "juan@example.com",
  password: "hashedpassword",
  favoriteMovies: [
    {
      movieId: ObjectId,
      likedAt: ISODate
    }
  ]
}
```

Colección movies:

```
{
  _id: ObjectId,
  title: "Inception",
  genre: "Sci-Fi",
  year: 2010,
  likes: Number // <- opcional, para llevar un contador total
}
```

CÓMO MANEJAR EL "LIKE" O "FAVORITO"

1. El usuario da like (favorito) a una película

- Desde frontend: el usuario pulsa el botón de "❤️".
- En backend:
    - Verificas si la película ya está en su lista.
    - Si no está: la agregas al array favoriteMovies con la fecha.
    - Opcional: incrementas el contador likes de la película.

```
// Ejemplo con Mongoose
const user = await User.findById(userId);
const alreadyLiked = user.favoriteMovies.some(fav =>
fav.movieId.equals(movieId));

if (!alreadyLiked) {
  user.favoriteMovies.push({ movieId, likedAt: new Date() });
```

```
  await user.save();


  await Movie.findByIdAndUpdate(movieId, { $inc: { likes: 1 } });
// opcional
}
```

---

2. El usuario quita el like

```
user.favoriteMovies = user.favoriteMovies.filter(fav =>
!fav.movieId.equals(movieId));
await user.save();


await Movie.findByIdAndUpdate(movieId, { $inc: { likes: -1 } }); //
opcional
```

---

3. Mostrar películas favoritas del usuario

```
const user = await
User.findById(userId).populate('favoriteMovies.movieId');
const favoriteMovies = user.favoriteMovies.map(f => f.movieId);
```

Para que populate() funcione, asegúrate de definir movieId como tipo mongoose.Schema.Types.ObjectId con referencia a Movie.

EXPLICACIÓN DETALLADA:

1. ESTRUCTURA INICIAL (ASUMIDA):

Imagina que tienes un modelo **User** definido más o menos así:

```
const UserSchema = new mongoose.Schema({
  username: String,
  favoriteMovies: [
    {
      movieId: { type: mongoose.Schema.Types.ObjectId, ref: "Movie"
},
      likedAt: Date
    }
  ]
});

const User = mongoose.model("User", UserSchema);
```

Esto significa que cada usuario tiene un campo favoriteMovies que es un **array**, y cada elemento de ese array es un objeto con dos campos:

- movieId: la referencia a la película favorita.
- likedAt: la fecha exacta en que se marcó como favorita.

2. CÓDIGO EXPLICADO LÍNEA POR LÍNEA:

```
user.favoriteMovies.push({ movieId, likedAt: new Date() });
```

¿Qué hace esto?

- **user**: es el documento (usuario) recuperado desde la base de datos con Mongoose (ej: const user = await User.findById(userId)).
- f**avoriteMovies**: es el campo que almacena las películas favoritas del usuario (un array).
- **.push()**: método de JavaScript que agrega un nuevo elemento al final del array.
- **{ movieId, likedAt: new Date() }:** es un nuevo objeto que tiene dos propiedades:
    - movieId: es el identificador de la película (probablemente un ObjectId de MongoDB). Normalmente ya lo tienes guardado en la variable movieId.
    - likedAt: guarda la fecha y hora actual en el momento del like (new Date() genera una fecha/hora actual).

Por ejemplo, luego de ejecutar esta línea, podrías tener algo así en memoria:

```
favoriteMovies: [
  { movieId: "663a54134f1c2bc123456789", likedAt: "2024-04-
25T10:00:00Z" }
]
```

---

3. GUARDANDO EN LA BASE DE DATOS:

```
await user.save();
```

- await: porque la operación es asíncrona (guarda en la base de datos y lleva tiempo).
- user.save(): toma el documento modificado (en memoria) y lo guarda en MongoDB, actualizando el documento correspondiente.

Después de esto, la base de datos MongoDB tendrá almacenado este cambio, guardando permanentemente la película favorita dentro del documento del usuario.

¿QUÉ OCURRE EXACTAMENTE EN MONGODB LUEGO DE .SAVE()?

MongoDB actualiza el documento del usuario en la colección users agregando este objeto en el array:

```
{
  "_id": ObjectId("663a540c4f1c2bc987654321"),
  "username": "juanito123",
  "favoriteMovies": [
    {
      "movieId": ObjectId("663a54134f1c2bc123456789"),
      "likedAt": ISODate("2024-04-25T10:00:00Z")
    }
  ]
}
```

¿Y SI NECESITAS EVITAR DUPLICADOS?

Antes de ejecutar .push(), normalmente verificas si la película ya está en la lista para evitar duplicados:

```
const exists = user.favoriteMovies.some(fav =>
fav.movieId.equals(movieId));
if (!exists) {
  user.favoriteMovies.push({ movieId, likedAt: new Date() });
  await user.save();
} else {
  console.log("La película ya está agregada como favorita");
}
```

CONCLUSIÓN

En resumen, lo que hace tu código es:

- Agregar una nueva película favorita al array del usuario.
- Registrar la fecha/hora exacta del "like".
- Guardar esos cambios en la base de datos MongoDB.

DFERENCIA ENTRE USAR **SAVE()** DE MONGOOSE Y UN MÉTODO **HTTP PUT** EN EXPRESS O UNA **API REST**:

¿POR QUÉ SE USA .SAVE() EN MONGOOSE?

Cuando trabajas directamente con **Mongoose**, utilizas métodos como .save() porque estás trabajando con documentos cargados en memoria. Por ejemplo:

```
const user = await User.findById(userId);
user.favoriteMovies.push({ movieId, likedAt: new Date() });

await user.save();
```

Aquí no importa si usas POST, PUT, o cualquier otro método HTTP, porque eso depende de cómo estructuras tu API REST, no de MongoDB directamente.

Mongoose no conoce HTTP. Es una librería ORM que interactúa directamente con la base de datos.

ENTONCES, ¿QUÉ SIGNIFICA USAR PUT O POST EN UNA API REST?

Cuando hablamos de métodos HTTP (GET, POST, PUT, DELETE) estamos refiriéndonos a operaciones en una API REST. No tienen relación directa con MongoDB o Mongoose, sino con cómo estructuras la comunicación desde tu cliente (frontend) al servidor.

Si, por ejemplo, tuvieras una API REST con Express, podrías definir una ruta así:

```
// Usando PUT (actualizar datos del usuario)
app.put('/api/users/:id/favorite-movies', async (req, res) => {
  const userId = req.params.id;
  const { movieId } = req.body;

  const user = await User.findById(userId);
  user.favoriteMovies.push({ movieId, likedAt: new Date() });

  await user.save();

  res.status(200).json({ message: "Película favorita agregada" });
});
```

En este ejemplo:

- PUT es solo el método HTTP que escoges.
- Dentro de tu backend (Express), llamas a Mongoose con .save() para guardar en MongoDB.

¿CUÁL MÉTODO HTTP DEBERÍA USAR? ¿POST O PUT?

Por convención REST:

- **POST** se usa para crear nuevos recursos o agregar nuevos elementos (es muy frecuente para agregar un like a una película).
- **PUT** normalmente se usa para actualizar un recurso completo o reemplazarlo. En tu caso, si ves "añadir un favorito" como una actualización del recurso usuario, puedes usarlo.

Personalmente, recomiendo usar:

- POST para agregar (like, favoritos).
- DELETE para eliminar (quitar un like).

EJEMPLO RECOMENDADO:

```
// Agregar favorito
```

```
POST /api/users/:id/favorite-movies


// Quitar favorito
DELETE /api/users/:id/favorite-movies/:movieId
```

CONCLUSIÓN

- **Métodos HTTP** (PUT, POST, DELETE) → son parte del protocolo REST de tu API.
- **Mongoose** (.save(), .updateOne(), .findByIdAndUpdate()) → interactúa directamente con MongoDB.

Ambos funcionan juntos, pero cumplen roles diferentes.

ENTONCES:

- En la capa REST (frontend ↔ servidor): eliges POST o PUT según tu lógica.
- En la capa backend (servidor ↔ MongoDB): utilizas métodos de Mongoose (.save(), .updateOne()).