Mongoose **middleware hooks** (also called **pre** and **post** middleware) are functions that are executed during specific stages of the Mongoose document lifecycle. These hooks allow you to run custom logic before or after certain operations such as saving, updating, or removing documents.

Mongoose middleware is particularly useful for tasks like validation, logging, encryption, and other operations that need to run automatically during database operations.

**Types of Mongoose Middleware**

1. **Pre Middleware**: Executes before an operation (e.g., save, remove, update).
2. **Post Middleware**: Executes after an operation.

**Common Operations with Middleware**

* **save**: Runs before or after saving a document (insert or update).
* **validate**: Runs before document validation.
* **remove**: Runs before or after removing a document.
* **update**: Runs before or after updating a document.
* **find / findOne**: Runs before or after querying documents.

**How to Use Pre and Post Middleware**

**1. Pre Middleware**

import mongoose from 'mongoose';

import bcrypt from 'bcrypt';

const userSchema = new mongoose.Schema({

    username: { type: String, required: true, unique: true },

    password: { type: String, required: true }

});

// Pre-save hook to hash the password before saving the user document

userSchema.pre('save', async function (next) {

    if (!this.isModified('password')) return next(); // Skip hashing if password is not modified

    this.password = await bcrypt.hash(this.password, 8); // Hash the password

    next(); // Proceed with the save operation

});

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

export default User;

* Pre middleware runs before a specified operation is executed.
* You can manipulate the document before it is saved to the database.

**Example: Hashing a password before saving**

In this example:

* The **pre** middleware runs before the save operation.
* It hashes the password before saving it to the database, ensuring that the password is never stored in plain text.

**2. Post Middleware**

* Post middleware runs after a specified operation is executed.
* It cannot modify the document or the result of the operation, but it can be useful for things like logging, sending notifications, or performing additional actions.

**Example: Logging after a document is saved**

userSchema.post('save', function (doc) {

    console.log(`User with username ${doc.username} has been saved.`);

});

In this example:

* The **post** middleware runs after a save operation.
* It logs a message to the console after the user document has been successfully saved.

**Common Middleware Operations**

**Pre Middleware Examples**

1. **Pre-save hook** to set a default value or manipulate data before saving:
2. userSchema.pre('save', function (next) {
3. if (!this.username) {
4. this.username = 'defaultUsername';
5. }
6. next();
7. });
8. **Pre-update hook** to check conditions before updating:

userSchema.pre('update', function (next) {

    if (this.getUpdate().age < 18) {

        throw new Error('Age cannot be less than 18');

    }

    next();

});

1. **Pre-remove hook** to perform some action before deleting a document:

userSchema.pre('remove', function (next) {

    console.log(`User with ID ${this.\_id} is about to be removed.`);

    next();

});

**Post Middleware Examples**

1. **Post-save hook** to log something after a document is saved:

userSchema.post('save', function (doc) {

    console.log('Document saved:', doc);

});

1. **Post-find hook** to modify the result of a query after it's executed:

userSchema.post('find', function (docs) {

    docs.forEach(doc => {

        doc.username = doc.username.toUpperCase();

    });

});

1. **Post-remove hook** to clean up related data after deleting a document:

userSchema.post('remove', async function (doc) {

    await mongoose.model('Post').deleteMany({ user: doc.\_id });

    console.log(`Deleted posts related to user ${doc.\_id}`);

});

**When to Use Pre and Post Middleware**

1. **Pre Middleware**:
   * Use **pre** middleware when you need to validate, transform, or modify data before it’s saved or modified.
   * Examples: Hashing passwords, modifying properties, adding timestamps, setting default values, and validating data.
2. **Post Middleware**:
   * Use **post** middleware when you need to perform actions after the operation has completed (e.g., logging, sending notifications, cleanup tasks).
   * Examples: Logging after saving a document, sending emails, cleaning up related documents after deletion.

**Asynchronous Middleware**

Mongoose middleware can be asynchronous, meaning you can use async/await to perform asynchronous operations (like hashing or making external API calls).

userSchema.pre('save', async function (next) {

    if (!this.isModified('password')) return next();

    this.password = await bcrypt.hash(this.password, 8);

    next(); // Don't forget to call next() to proceed

});

In the above code:

* The middleware is asynchronous because we are using await to hash the password before continuing.

**Example: Complete User Schema with Pre and Post Middleware**

import mongoose from 'mongoose';

import bcrypt from 'bcrypt';

const userSchema = new mongoose.Schema({

    username: { type: String, required: true, unique: true },

    password: { type: String, required: true },

});

// Pre-save middleware to hash password before saving

userSchema.pre('save', async function (next) {

    if (!this.isModified('password')) return next();

    this.password = await bcrypt.hash(this.password, 8);

    next();

});

// Post-save middleware to log a message

userSchema.post('save', function (doc) {

    console.log(`User ${doc.username} has been saved successfully.`);

});

// Method to compare passwords during login

userSchema.methods.comparePassword = async function (candidatePassword) {

    return await bcrypt.compare(candidatePassword, this.password);

};

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

export default User;

**Summary**

* **Pre Middleware** runs before an operation (e.g., save, update), useful for modifying or validating data.
* **Post Middleware** runs after an operation, useful for tasks like logging or side effects.
* Middleware hooks allow you to centralize business logic related to database operations and ensure consistent behavior across your application.

**custom methods**

In Mongoose, **custom methods** are functions that you can define on a **schema** to perform specific operations on documents or collections. These methods can be used to extend the functionality of the Mongoose model beyond the built-in CRUD operations, allowing you to add reusable logic or queries that are specific to your application's needs.

You can define custom instance methods (which operate on individual document instances) or custom static methods (which operate on the model itself, affecting queries and collections).

### ****Types of Custom Methods in Mongoose****

1. **Instance Methods**:
   * These methods are defined on a single document instance and can access and manipulate the data of that specific document.
2. **Static Methods**:
   * These methods are defined on the model itself and are used for operations involving the entire collection or executing complex queries.

### ****1. Instance Methods****

Instance methods are used to perform operations on individual document instances (i.e., when you retrieve a document from the database).

#### ****Defining Instance Methods****

Instance methods are defined using the schema.methods object. These methods are available on individual documents, so you can access and modify the properties of the document.

##### Example: Custom Instance Method

Suppose you have a User model and you want to add a method to check if a user’s password matches a provided password.

userSchema.methods.isPasswordCorrect = async function(password){

    return await bcrypt.compare(password,this.password)

}