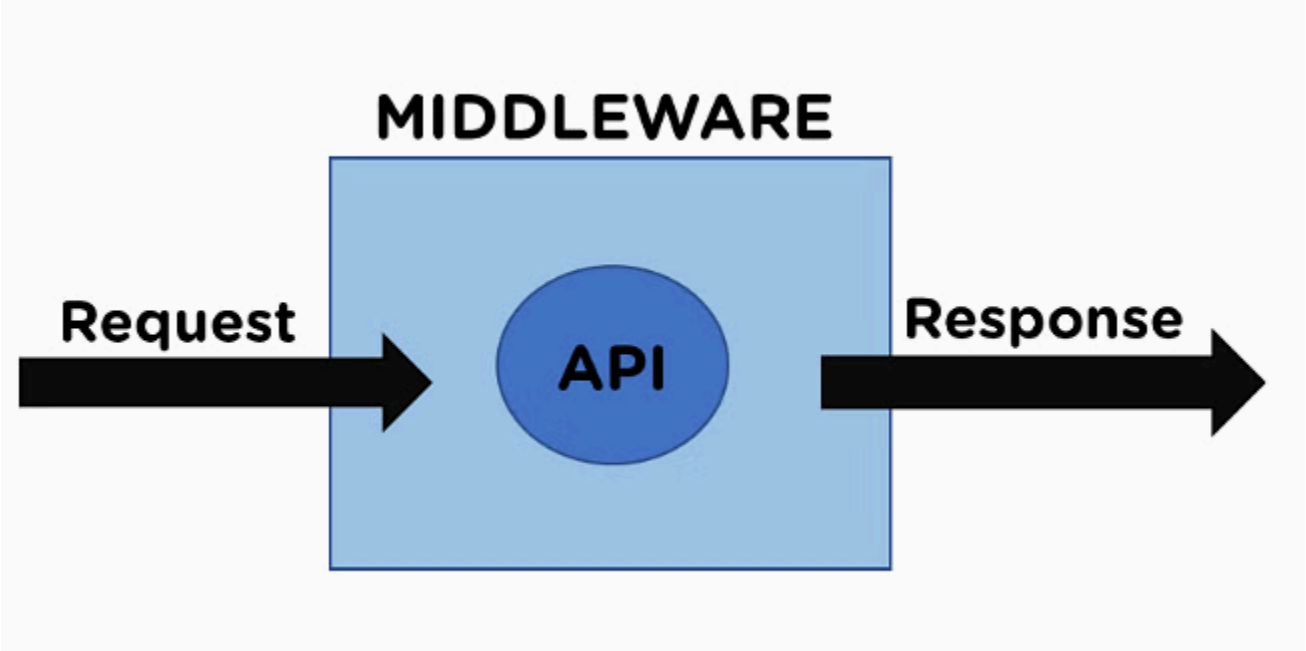
wwDAY 8

- *Middleware*

Imagine you're at a restaurant, and you've placed an order for your favorite dish. Now, before that dish reaches your table, it goes through several stages in the kitchen. Each stage involves different tasks, like chopping vegetables, cooking, and adding spices. Middleware is a bit like these stages in the kitchen—it's something that happens in between your request and the final response in a web application.

Now, let's apply this idea to a web application, like the "Node Hotel" system:

1. **Request Phase:**
    - You (the client) make a request to the Node Hotel system. It could be asking for the menu, submitting a reservation, or anything else.
2. **Middleware Phase:**
    - Middleware is like the behind-the-scenes process in the kitchen. It's a series of functions that your request goes through before it reaches the final destination.
3. **Final Response Phase:**
    - After passing through the middleware, your request gets processed, and the system sends back a response. It could be the menu you requested or confirmation of your reservation.



Example in Node.js:

In the context of Node.js, imagine you want to log every request made to your "Node Hotel" application. You could use middleware for this.

```
// Middleware Function
const logRequest = (req, res, next) => {
    console.log(`[${new Date().toLocaleString()}] Request made to: ${req.originalUrl}`);
    next(); // Move on to the next phase
};

// Using Middleware in Express
const express = require('express');
const app = express();

// Apply Middleware to all Routes
app.use(logRequest);

// Define Routes
app.get('/', (req, res) => {
    res.send('Welcome to Node Hotel!');
});

app.get('/menu', (req, res) => {
    res.send('Our delicious menu is coming right up!');
});

// Start the Server
const PORT = 3000;
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

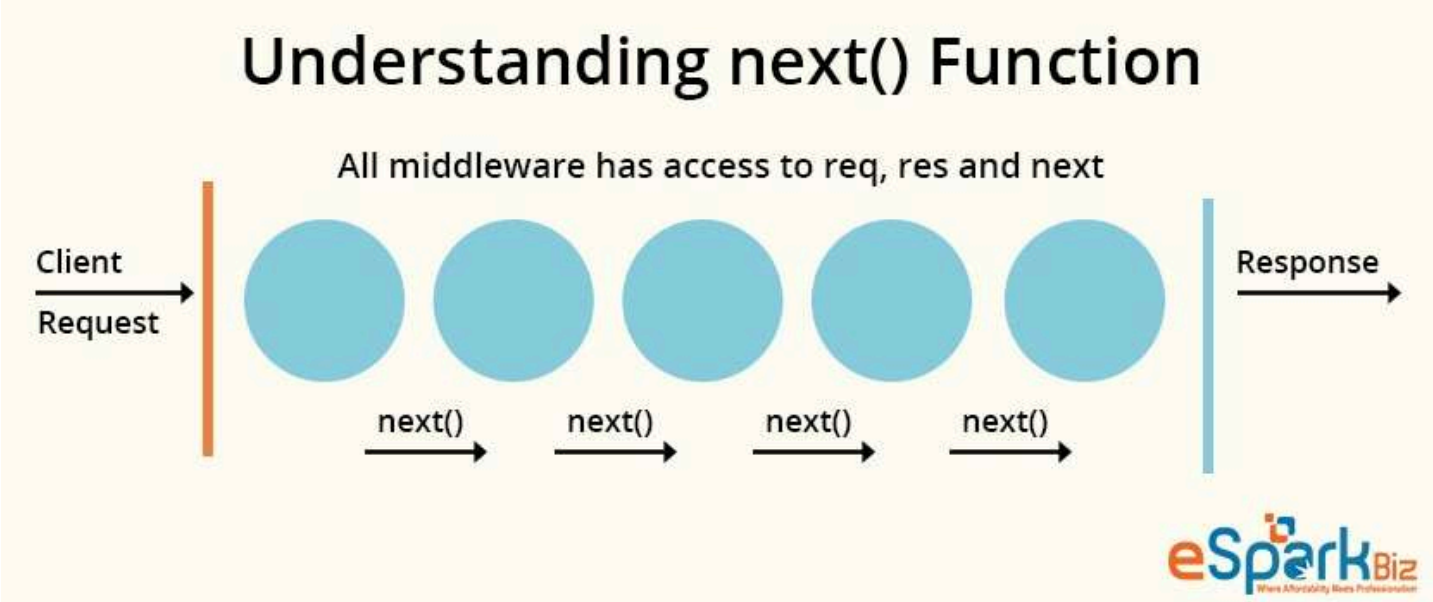
In this example, `logRequest` is our middleware. It logs the time and the requested URL for every incoming request. The `app.use(logRequest)` line tells Express to use this middleware for all routes.

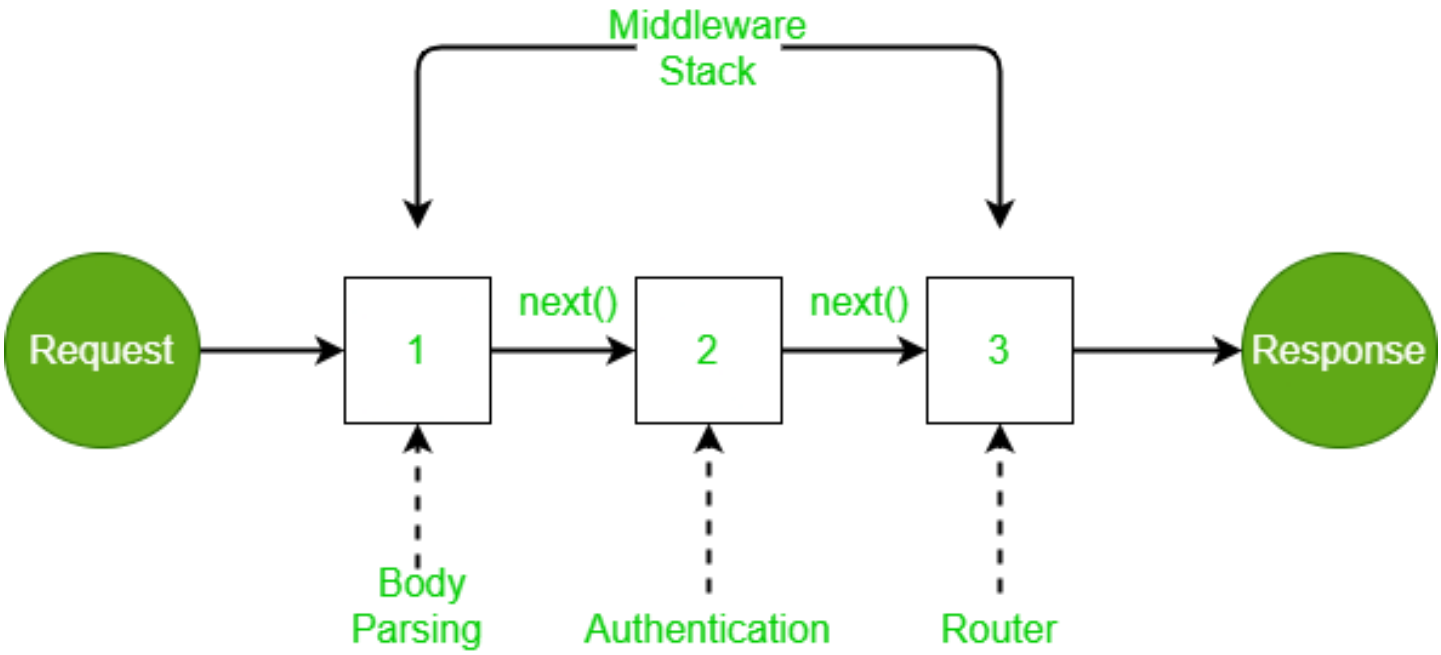
So, when you access any route (like `/` or `/menu`), the middleware runs first, logs the request, and then the route-specific code executes.

In summary, middleware is like a series of tasks that happen behind the scenes in a web application. **It's a way to add extra functionality to your application's request-response cycle**, such as **logging**, **authentication checks**, or **modifying request data**, before it reaches its final destination.

Why do we use the next() function in the middleware function

In Express.js, the `next()` function is a callback that *signals to Express* that the current middleware function has completed its processing and that it's time to move on to the next middleware function or route handler in the chain.




Middleware
Stack
Request
1
next()
2
next()
3
Response
Body
Parsing
Authentication
Router

- *Authentication & Authorization*

Like Aim: 700
Comment Aim: 400
Please Like this Video Before watching

Tag me on Linkedin for Notes – show Demo How you can post

Imagine you're the manager of the "Node Hotel" application, and you want to ensure that only authorized staff members can access certain features. This is where authentication comes in.

1. Verifying Identity (Authentication):

- **Scenario:** When a staff member, let's say a chef, wants to log in to the Node Hotel system, they need to prove that they are indeed the chef they claim to be.
- **In Practice:** In Node.js, authentication involves checking the chef's credentials, like a username and password, to make sure they match what's on record. It's like asking the chef to enter a secret code (password) and confirming that it's correct.

2. Access Control (Authorization):

Now, let's add a layer of authorization based on the roles of the staff members.

- **Scenario:** Once the chef has proven their identity, you, as the manager, want to control what they can and cannot do. For instance, chefs should be able to update the menu items, but maybe not manage staff salaries.
- **In Practice:** In Node.js, after authenticating the chef, you'll use authorization to decide what parts of the system they have access to. It's like giving the chef a key card (authorization) that lets them into the kitchen but not into the manager's office.

Implementation in Node.js:

1. **Authentication Middleware:**
    - In your Node.js application, you might use middleware like Passport to handle the authentication process.
    - Passport helps verify the identity of the chef based on their provided credentials.
2. **User Roles and Permissions:**
    - You'll define roles for staff members (e.g., chef, waiter, manager).
    - Authorization middleware will check the role of the authenticated user and grant access accordingly.

3. **Secure Endpoints:**
    - You'll protect certain routes (like updating menu items) with authentication checks.
    - Only authenticated and authorized users (like chefs) will be allowed to access these routes.

In the Hotel Context:

- **Authentication:** When Chef John logs in, the system checks if the provided username and password match what's on record for Chef John.
- **Authorization:** Once authenticated, Chef John is authorized to modify menu items but may not have permission to change other critical settings.

In simple terms, authentication in Node.js for your hotel application ensures that each staff member is who they say they are, and authorization determines what they're allowed to do once their identity is confirmed.

It's like having a secure system where only the right people get access to the right areas of your hotel management application.

In general, authentication is applied before authorization in the security process. Here's the typical sequence:

1. **Authentication:**
    - The first step is to verify the identity of the user or system entity attempting to access a resource or perform an action. This involves checking credentials such as usernames and passwords or using other authentication methods like tokens, API keys, or certificates.
2. **Authorization:**
    - Once the identity is verified through authentication, the system moves on to authorization. Authorization determines what actions or resources the authenticated user or entity is allowed to access based on their permissions, roles, or other access control mechanisms.

The reason for this order is straightforward: before you can determine what someone is allowed to do (authorization), you need to know who they are (authentication). Authentication establishes the identity, and authorization defines the permissions associated with that identity.

In the context of web applications, middleware for authentication is typically applied first in the request-response cycle to verify the user's identity. If authentication is successful, the request proceeds to authorization middleware to determine what the authenticated user is allowed to do.

It's important to note that while authentication and authorization are often discussed as distinct steps, they work together as essential components of a security strategy to control access to resources and protect against unauthorized actions.

- Now we will implement Authentication as a middleware Function. So that, Routes will be authenticated before reaching out to the server.
- Implementing authentication as a middleware function is a common and effective approach.

- *Passport.js*

Passport.js is a popular **authentication middleware for Node.js**. Authentication is the process of verifying the identity of a user, typically through a username and password, before granting access to certain resources or features on a website or application.

Think of Passport.js as a helpful tool that makes it easier for developers to handle user authentication in their Node.js applications. It simplifies the process of authenticating users by **providing** a set of pre-built strategies for different authentication methods, such as username and password, social media logins (like Facebook or Google), and more.

Here's a breakdown of some key concepts in Passport.js:

1. **Middleware:** In the context of web development, *middleware is software that sits between the application and the server.* Passport.js acts as middleware, intercepting requests and adding authentication-related functionality to them.
2. **Strategy:** Passport.js uses the concept of strategies for handling different authentication methods. A strategy is a way of authenticating users. Passport.js comes with various built-in strategies, and you can also create custom strategies to support specific authentication providers.
3. **Serialize and Deserialize:** Passport.js provides methods for serializing and deserializing user data. Serialization is the process of converting user data into a format that can be stored, usually as a unique identifier. Deserialization is the reverse process of converting that unique identifier back into user data. These processes are essential for managing user sessions.

- Install Passport

To use Passport.js in a Node.js application, you need to **install the passport package** along with the **authentication strategies** you intend to use.

For this course, we are using Local strategies authentication (username and password).

you would typically install `passport-local`

```
npm install passport passport-local
```

Once you've installed these packages, you can set up and configure Passport.js in your application.

```
const express = require('express');
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;
const app = express();
// Initialize Passport
app.use(passport.initialize());
```

- Passport Local Strategy

- The Passport Local Strategy is a part of the Passport.js authentication middleware for Node.js. It's specifically designed for handling username and password-based authentication.
- The Passport Local Strategy, by default, expects to extract the username and password from the request body. It is a common practice for username and password-based authentication systems to send the credentials as part of the request body, especially in login forms.

- Add username & password

- Now we have to add username and password in the person schema

```
33      username: {
34          type: String,
35          required: true,
36          unique: true,
37      },
38      password: {
39          type: String,
40          required: true,
41      }
42  });
```

- Configure the Local Strategy

- Define and configure the Local Strategy using `passport-local`.
- You need to provide a verification function that checks the provided username and password.

```
passport.use(new LocalStrategy(
  async (username, password, done) => {
    // Your authentication logic here
  }
));
```

- In the Local Strategy's verification function, you typically query your database to find the user with the provided username. You then compare the provided password with the stored password.
- In the context of `LocalStrategy`, Passport.js expects the verification function to have the following signature:

```
function(username, password, done)
```

- The `done` callback should always be the last parameter, and it's essential to maintain this order for Passport.js to work correctly. If you change the order of parameters, you risk breaking the expected behavior of Passport.js.

```
passport.use(new LocalStrategy(async (username, password, done) => {
    try {
        console.log('Received credentials:', username, password);
        const user = await Person.findOne({ username });
        if (!user)
            return done(null, false, { message: 'Incorrect username.' });

        const isPasswordMatch = (user.password === password ? true : false);
        if (isPasswordMatch)
            return done(null, user);
        else
            return done(null, false, { message: 'Incorrect password.' })
    } catch (error) {
        return done(error);
    }
}));
```

- In the context of Passport.js, `done` is a callback function that is provided by Passport to signal the completion of an authentication attempt. It is used to indicate whether the authentication was successful, and if so, to provide information about the authenticated user.
- The `done` function takes three parameters: `done(error, user, info)`.
- If the authentication is successful, you call `done(null, user)` where `user` is an object representing the authenticated user.
- If the authentication fails, you call `done(null, false, { message: 'some message' })`. The second parameter (`false`) indicates that authentication failed, and the third parameter is an optional info object that can be used to provide additional details about the failure.

- Passport Authenticate

- Once you have configured Passport Local Strategy, the next steps typically involve integrating it into your application.
- In route, we should use `passport.authenticate()` to initiate the authentication process.
- To authenticate any routes, we need to pass this as an middleware

Let's suppose we want to Authenticate this Route

```
app.get('/', function (req, res) {
    res.send('Welcome to our Hotel');
})
```

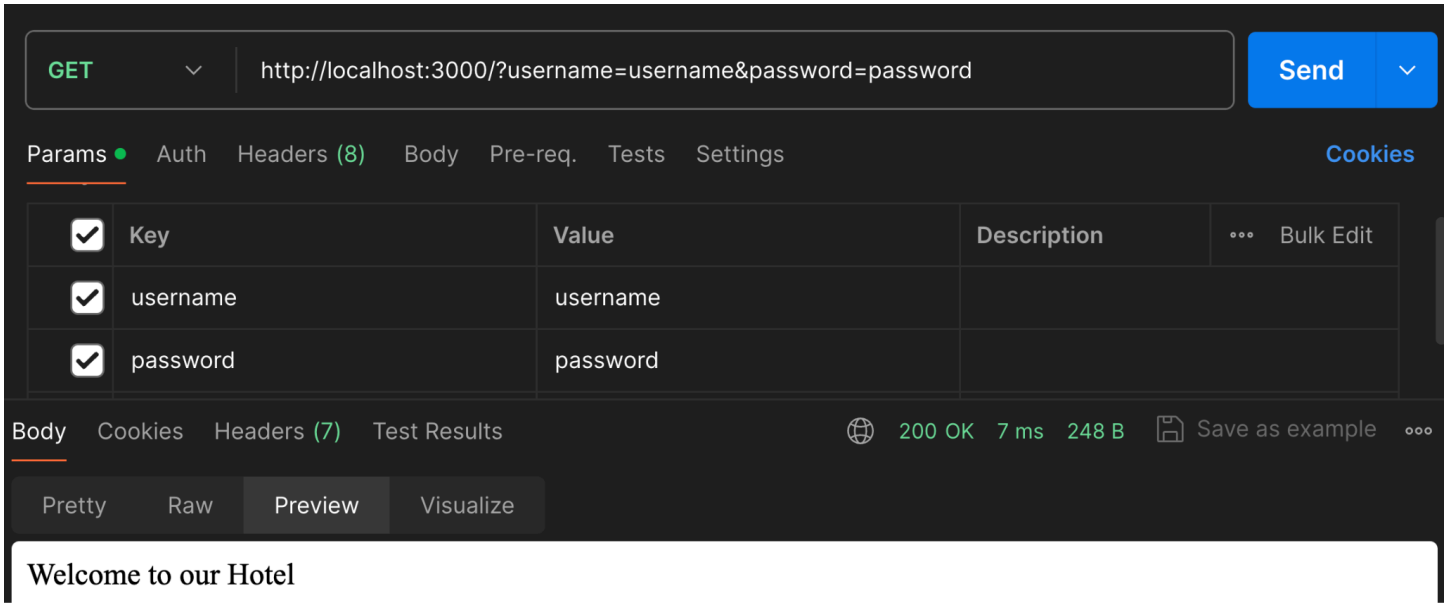
- We have to initialize the passport

```
app.use(passport.initialize());
app.get('/', passport.authenticate('local', { session: false }), function (req, res) {
    res.send('Welcome to our Hotel');
})
```

Or we can also write like this

```
app.use(passport.initialize());
const localAuthMiddleware = passport.authenticate('local', { session: false });

app.get('/', localAuthMiddleware, function (req, res) {
    res.send('Welcome to our Hotel');
})
```

- Now, we can test the ‘ / ‘routes it needs parameter



- Same way, we can also pass authenticate /person routes.

- Passport Separate File

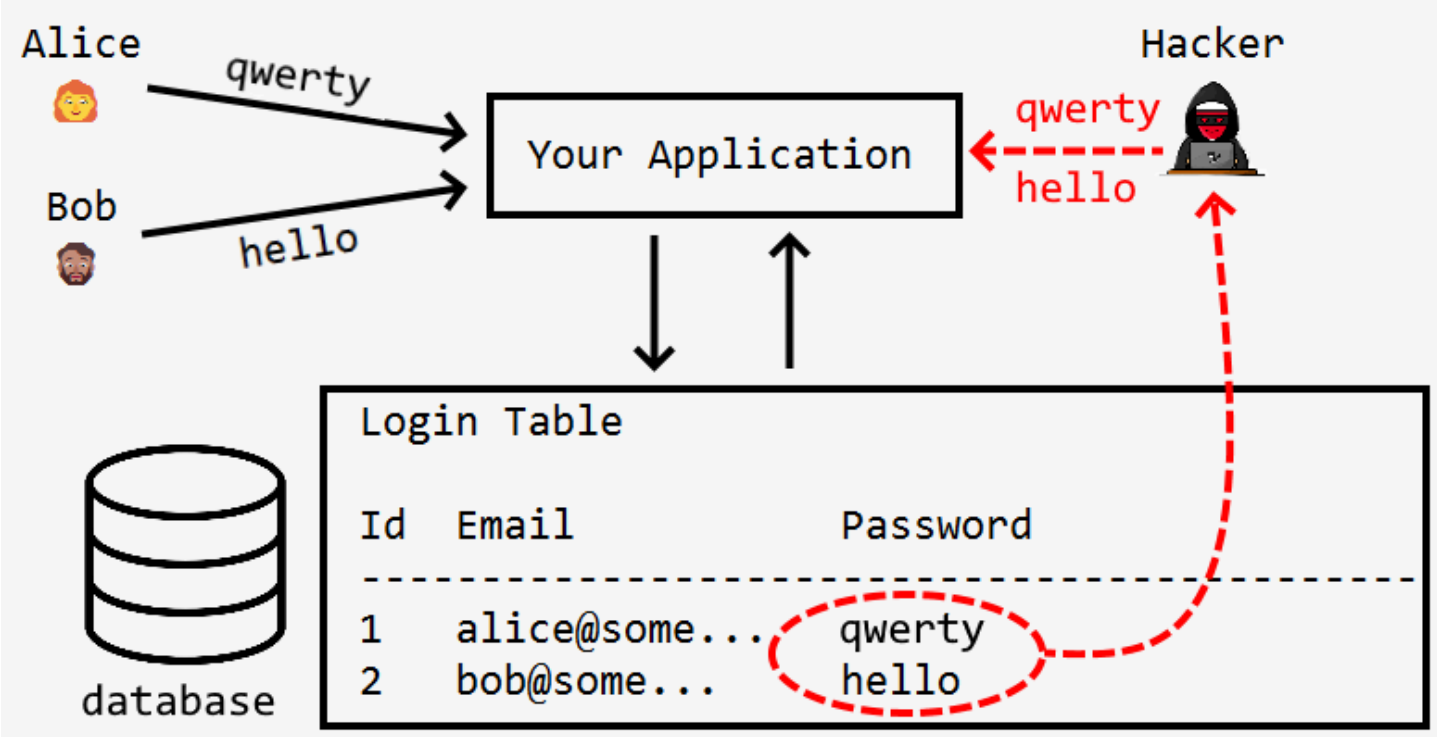
- Now, rather than all the passport codes in a server file. We can separate it into another file name auth.js
- And in the server.js file, we will import the passport

```
const passport = require('./auth'); // Adjust the path as needed

// Initialize Passport
app.use(passport.initialize());
const localAuthMiddleware = passport.authenticate('local', { session: false });

app.get('/', localAuthMiddleware, function (req, res) {
    res.send('Welcome to our Hotel');
})
```

```
// sets up Passport with a local authentication strategy, using a Person model
for user data. - Auth.js file

const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;
const Person = require('./models/Person'); // Adjust the path as needed

passport.use(new LocalStrategy(async (username, password, done) => {
    try {
        console.log('Received credentials:', username, password);
        const user = await Person.findOne({ username });
        if (!user)
            return done(null, false, { message: 'Incorrect username.' });

        const isPasswordMatch = user.password === password ? true : false;
        if (isPasswordMatch)
            return done(null, user);
        else
            return done(null, false, { message: 'Incorrect password.' })
    } catch (error) {
        return done(error);
    }
}));

module.exports = passport; // Export configured passport
```

- *Store Plain Password*

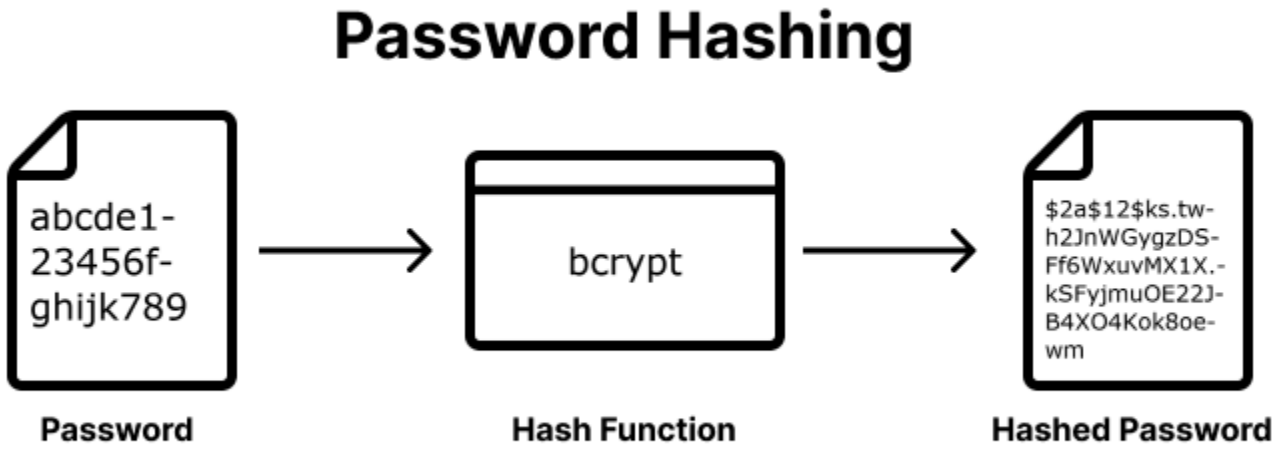
- Storing plain passwords is not a secure practice. To enhance security, it's highly recommended to hash and salt passwords before storing them.

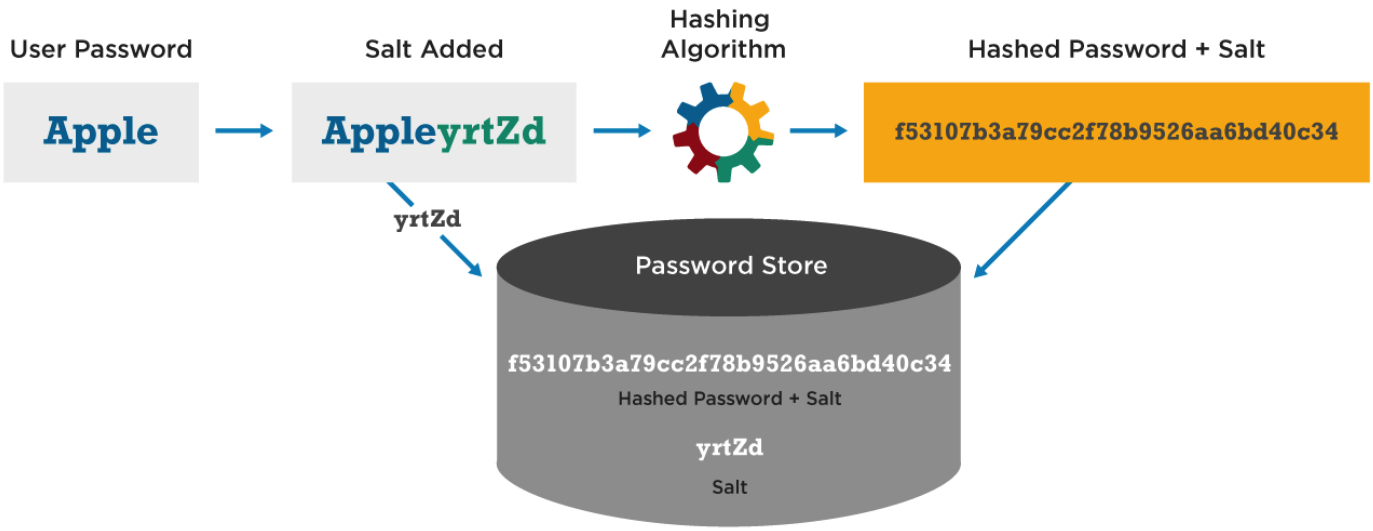


- You can use the `bcrypt` library for password hashing in your Node.js application.

```
npm install bcrypt
```

- *bcrypt.js*





- Now we have to update our person model to store hashed passwords. Modify the registration logic to hash the password before saving it to the database.
- Because the end user didn't know about hashing, we have to internally maintain it. Like we are saving the hashed password before saving it into the database
- We are using a Mongoose middleware hook to perform an action before saving a document to the database. Specifically, it's using the `pre` middleware to execute a function before the `save` operation.

```
personSchema.pre('save', async function(next) {
    const person = this;

    // Hash the password only if it has been modified (or is new)
    if (!person.isModified('password')) return next();

    try {
        // Generate a salt
        const salt = await bcrypt.genSalt(10);

        // Hash the password with the salt
        const hashedPassword = await bcrypt.hash(person.password, salt);

        // Override the plain password with the hashed one
        person.password = hashedPassword;
        next();
    } catch (error) {
        return next(error);
    }
});
```

- The `pre('save', ...)` middleware is triggered before the `save` operation on a Mongoose model instance.
- Inside the middleware function, it checks if the password field has been modified (or if it's a new document). If not, it skips the hashing process.
- If the password has been modified, it generates a new salt using `bcrypt.genSalt` and then hashes the password using `bcrypt.hash`.
- The original plain text password in the `person` document is then replaced with the hashed password.
- The `next()` function is called to proceed with the save operation.

- The line `const salt = await bcrypt.genSalt(10);` is responsible for generating a salt, which is a random string of characters used as an additional input to the password hashing function. Salting is a crucial step in password hashing to prevent attackers from using precomputed tables (rainbow tables) to quickly look up the hash value of a password.

- `bcrypt.genSalt(rounds)`: This function generates a salt using the specified number of "rounds." The `rounds` parameter indicates the complexity of the hashing algorithm. The higher the number of rounds, the more secure the salt, but it also increases the computational cost.

```
if (!person.isModified('password')) return next();
```

This line is a conditional check that prevents unnecessary rehashing of the password when the document is being saved.

- `person.isModified('password')`: This method is provided by Mongoose and returns `true` if the specified field (`'password'` in this case) has been modified. It returns `false` if the field hasn't been modified.
- `return next();`: If the password field has not been modified, the function immediately returns, skipping the rest of the middleware. This is because there's no need to rehash the password if it hasn't changed.

- How `bcrypt` works

When you use `bcrypt` to hash a password, the library internally stores the salt as part of the resulting hashed password. This means that you don't need to separately store the salt in your database; it is included in the hashed password itself.

Here's a simplified explanation of how it works:

1. **Hashing a Password:**

When you hash a password using `bcrypt.hash`, the library generates a random salt, hashes the password along with the salt, and produces a hashed password that incorporates both the salt and the hashed value.

```
const salt = await bcrypt.genSalt(10);
const hashedPassword = await bcrypt.hash('userPassword', salt);
```

The `hashedPassword` now contains both the hashed password and the salt.

- Modify Auth code

We have to also modify the password-matching logic in the passport auth file.

- Let's create a **comparePassword** named function which compares or check the password.

```
passport.use(new LocalStrategy(async (username, password, done) => {
    try {
        console.log('Received credentials:', username, password);
        const user = await Person.findOne({ username });
        if (!user)
            return done(null, false, { message: 'Incorrect username.' });

        const isPasswordMatch = await user.comparePassword(password, user.password);
        if (isPasswordMatch)
            return done(null, user);
        else
            return done(null, false, { message: 'Incorrect password.' })
    } catch (error) {
        return done(error);
    }
}));
```

- Compare function

We also have to write the compare function

```
// Define the comparePassword method
personSchema.methods.comparePassword = async function(candidatePassword) {
    try {
        // Use bcrypt to compare the provided password with the hashed password
        const isMatch = await bcrypt.compare(candidatePassword, this.password);
        return isMatch;
    } catch (error) {
        throw error;
    }
};
```

Verifying a Password

When you later want to verify a user's entered password during login, you use `bcrypt.compare`. This function internally extracts the salt from the stored hashed password and uses it to hash the entered password for comparison.

```
const isMatch = await bcrypt.compare('enteredPassword', storedHashedPassword);
```

The `compare` function automatically extracts the salt from `storedHashedPassword` and uses it to hash the entered password. It then compares the resulting hash with the stored hash. If they match, it indicates that the entered password is correct.