

# Session-9 & 10

Disfo Frontend - [https://gitlab.crio.do/public\\_content/node-sprint-sessions/disfo-frontend](https://gitlab.crio.do/public_content/node-sprint-sessions/disfo-frontend)

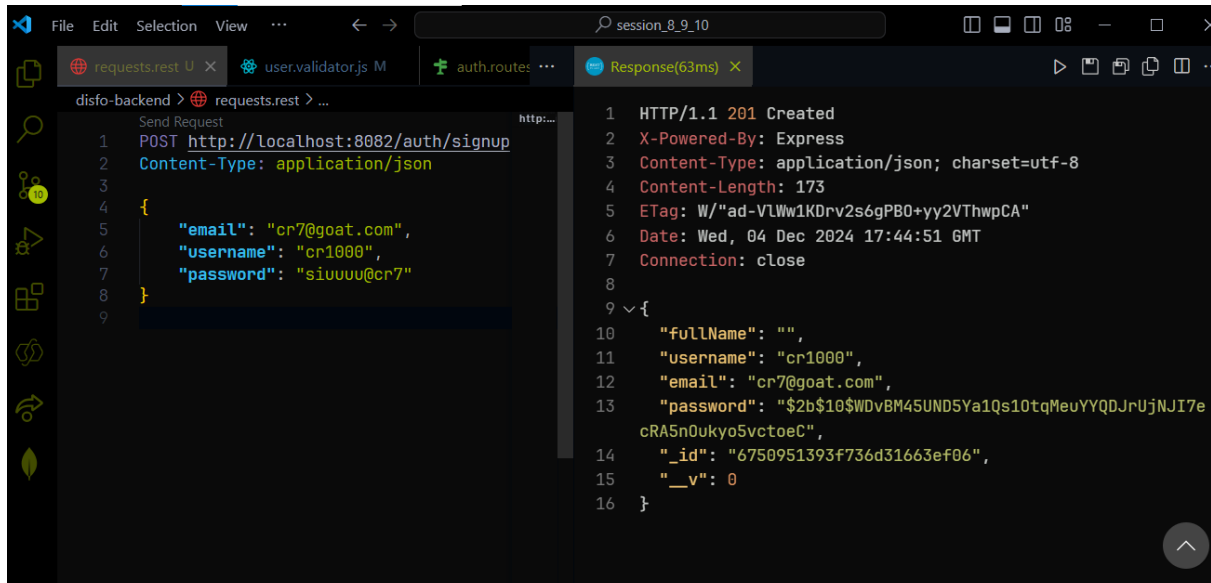
Disfo Backend - [https://gitlab.crio.do/public\\_content/node-sprint-sessions/disfo](https://gitlab.crio.do/public_content/node-sprint-sessions/disfo)

**04-12-2024**

1. Created the Skeleton in Session-8. `Index.js` , `auth.routes.js` , `auth.controller.js` .
2. Made a POST request to `/auth/signup` with email, username and password in the header. Tested out whether the POST call is getting hit or not by logging.
3. Created a `auth.service.js` with the signup service (AuthService). This service makes use of the Model Schema of the `user.model.js`, receives the payload and then creates a new User using this payload.
4. Because the auth service returns a promise here in controller we need to handle it in `try-catch block` always. Also `async-await` is required.
5. `postSignup` function is completed now. So now when we make a POST request the user is created. But the password is open and visible. It is not ideal to store password open like this.
6. This is where `bcrypt` comes into picture. `Bcrypt` hashes the password.
7. We also added `ValidateUser` validation to the `/signup` route.
8. Hashing, Encryption/Decryption, Encode/Decode
9. We store the Salt-hashed password in the database. Next time the user logs in using his password we again add salt to it, hash it and then compare the hash with the password in our DB (Which was also a Hash).
10. Install `bcrypt`. Create a hashing function in the `auth.service.js` because its a business logic to hash.
11. Once we create a hashing function in services we need to use that password to be stored into the DB with the user details. Therefore we keep everything users sends as it is and then for password we pass it through

the bcrypt hashing function and sending payload.password as the argument.

12. Password Encryption successful. `/auth/signup` is completed.



The screenshot shows the VS Code interface with a REST client tab open. The request is a POST to `http://localhost:8082/auth/signup` with a JSON body containing email, username, and password. The response is a 201 Created status with a JSON body containing user details and a token.

```
1 POST http://localhost:8082/auth/signup
2 Content-Type: application/json
3
4 {
5   "email": "cr7@goat.com",
6   "username": "cr1000",
7   "password": "siuuuu@cr7"
8 }
9
10 HTTP/1.1 201 Created
11 X-Powered-By: Express
12 Content-Type: application/json; charset=utf-8
13 Content-Length: 173
14 ETag: W/"ad-VLWw1KDrv2s6gPB0+yy2VThwpCA"
15 Date: Wed, 04 Dec 2024 17:44:51 GMT
16 Connection: close
17
18 {
19   "fullName": "",
20   "username": "cr1000",
21   "email": "cr7@goat.com",
22   "password": "$2b$10$WDvBM45UND5Ya1Qs10tqMeuYYQDJrUjNJI7e
23     cRA5n0ukyo5vctoec",
24   "_id": "6750951393f736d31663ef06",
25   "_v": 0
26 }
```

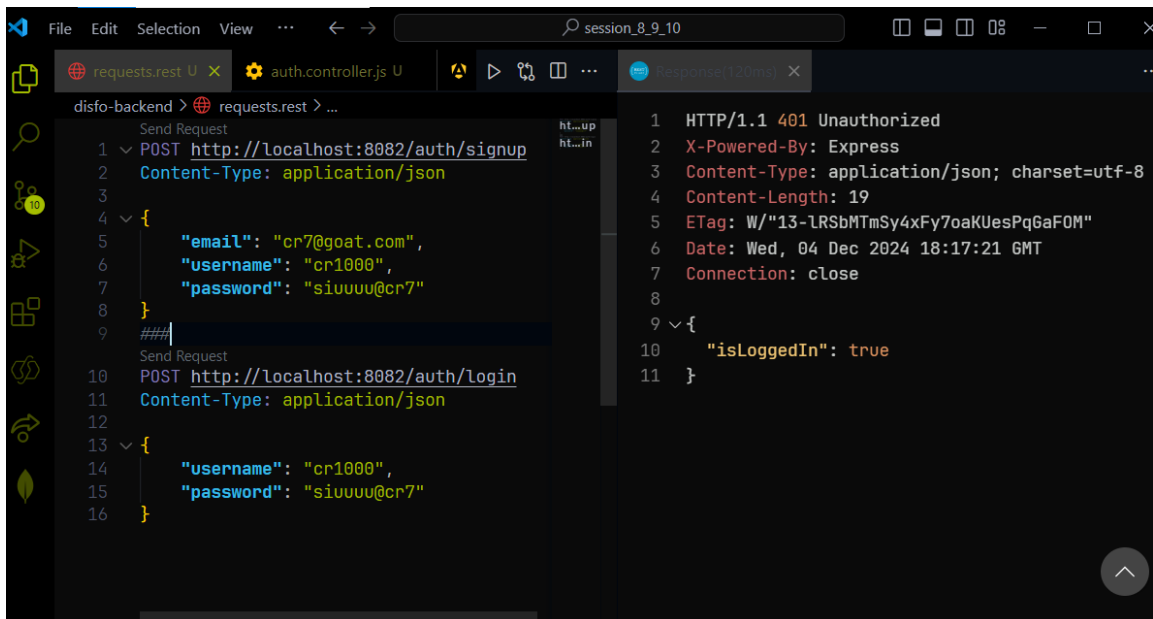
13. We will implement `/auth/login` route function.

14. First thing we need to do when we receive the username and password from the user is to check whether the user exists or not in the DB. We made use of `findByUsername` method from `user.service.js`.

15. Now once if the reqUser is found in the DB now we check for the password.

16. We created a new service called `comparePassword` which takes `plainTextPassword` and `hashedPasswords` are two parameters. Compares the values and returns the boolean values true or false.

17. In Controller we are using this `comparePassword( )` method to check with the DB password with user inputted password. Once we are done with this testing this. Made a POST call again with username and password as the headers being passed.



18. If we mess up with username it shows the error with message. If we mess up with the user password then the `isLoggedIn` will be false.

**06-12-2024**

19. Created one more new user and tested with the Authentication.

20. Some Interview Questions in the Slides to go through.

- What is Hashing ? Explain the relevance of it
  - <https://www.crio.do/learn/crio-community/topic/what-is-hashing-explain-the-relevance-of-it/210506/>
- Why do we need bcrypt ?
  - Bcrypt can expand its Key Factor to compensate for increasingly more-powerful computers and effectively slow down its hashing speed
- Why passwords are hashed ?
  - Password hashing is used to verify the integrity of your password, sent during login, against the stored hash so that your actual password never has to be stored.
- What is encryption and Decryption?
  - Encryption is the process by which a readable message is converted to an unreadable form to prevent unauthorized parties

from reading it. Decryption is vice-versa of Encryption

21. Create a login Schema in Validators folder `login.validator.js` . Used it for the `/login` route before postLogin call.
22. So now we need to send both username and password in the header of the POST request otherwise we will get the error.
23. Do you know how websites know that we are logged in and we are the one who are making GET, POST, PUT and all requests ? - This is happening with the help of **Cookies**.
24. HttpOnly Cookie means he cannot be accessed from the Client side. Only accessible from the Server side.
25. How can we access these Cookies ? → How did we access the localStorage from the frontend ?
  - `localStorage.getItem( )`
  - `document.cookie` → for getting cookies at the client side (HttpOnly cookies are not seen)
26. XSS attacks - <https://owasp.org/www-community/attacks/xss/>
27. A cookie with the Secure attribute is only sent to the server with an encrypted request over the HTTPS protocol.
28. If the Cookie has the Secure attribute checked in then it will only be sent when it has a secure connection (HTTPS)
29. Here a problem arises with the Id and token being stored in the server for the particular user. What if we have multiple servers ? What if the server goes down ? Who will handle the requests for those users then ?
  - a. We can keep this table which stores the Id and Tokens in a Central Datastore. Now the servers can interact with the Datastore.
  - b. But still we have the problem. What if the Datastore goes down ?
30. This is where JWT Tokens come into play. Using these we can identify the user without storing the token in the servers or Datastore. This is stateless storage Mechanism.

## The idea is to generate a Token when the user logs in.

31. Install the JWT package (npm i jsonwebtoken)

32. Once the user is logged we generate a Token:

```
res.send({ isLoggedIn, token: AuthServiceInstance.generate
```

33. The generateJwt is the function from the auth.service.js which is using JWT package and generating a token for the user. It uses a Secret key as the second parameter which we store in the .env file and then access it.

- Secret Key generator by Vercel - <https://auth-secret-gen.vercel.app/>
- For checking whether the token is valid or not - <https://jwt.io/>

34. Token looks something like this: It has 3 parts separated by dots as seen below.

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiI2NzUyYmU1MDU1ZGQ2ZGRlODNlNjQ3OGUiLCJpYXQiOiJE3MzM0OTgwMjYsImV4cCI6MTczMzQ5MDA1Nn0.L1HEURZgXwqcRezsfJVU0IzBRiq36X-yBuMJqd-2Wr8
```

35. 3 Parts:

- Red part → **Header** (Base-64 format) (Tells which Algo was used and what typ it is (JWT here))
- Purple part → **Payload** (Base-64 format)

- Blue part → **Signature**

36. Now we will send this token in the header as a cookie with a token.

```

6 const postSignup = async (req, res) => {
7   try {
8     const newUser = await AuthServiceInstance.signup(req.body);
9     res.status(201).send(newUser);
10  } catch (error) {
11    if (error.code === 11000) {
12      return res.status(409).send({ message: 'User with this email already exists!' });
13    }
14    req.status(500).send({ message: 'Failed to sign user up!' });
15  }
16 };
17
18 const postLogin = async (req, res) => {
19   try {
20     const { username, password } = req.body;
21     const reqUser = await userServiceInstance.findByUsername(username);
22     if (!reqUser) {
23       return res.status(404).send({ message: 'User with username: ${username} could not found' });
24     }
25     const isLoggedin = await AuthServiceInstance.comparePassword(password, reqUser.password);
26     if (!isLoggedin) {
27       res.status(401).send({ isLoggedin });
28     }
29     res.cookie('remember-user-token', AuthServiceInstance.generateJwt({ userId: reqUser._id }));
30     res.status(200).send({ message: 'Failed to login: Try again!' });
31   } catch (error) {
32     // ...
33   }
34 };

```

```

1 HTTP/1.1 200 OK
2 X-Powered-By: Express
3 Access-Control-Allow-Origin: http://localhost:8081
4 Vary: Origin
5 Access-Control-Allow-Credentials: true
6 Set-Cookie: remember-user-token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmQiaWoiI2NzuzMjluODk2Zjg0ODY0YVZuI2Z0c2VtZGILCjpwYXkiOiJlMzZlM1M0V5OTcsImV4cCI6MTczMzUwMzAyM399.gAqKPhyZHRUC6G3CET0gF1xVC
7 Content-Type: application/json; charset=utf-8
8 Content-Length: 19
9 Etag: W/"13-LRSbMTmSy4xFy7oakUesPgSaFOM"
10 Date: Fri, 06 Dec 2024 17:43:17 GMT
11 Connection: close
12
13 {
14   "isLoggedin": true
15 }

```

37. Now we have make request from the Frontend using the user credentials. But we get one error called as **CORS** because we are requesting it from the **:8081** frontend domain to **:8082** backend domain.

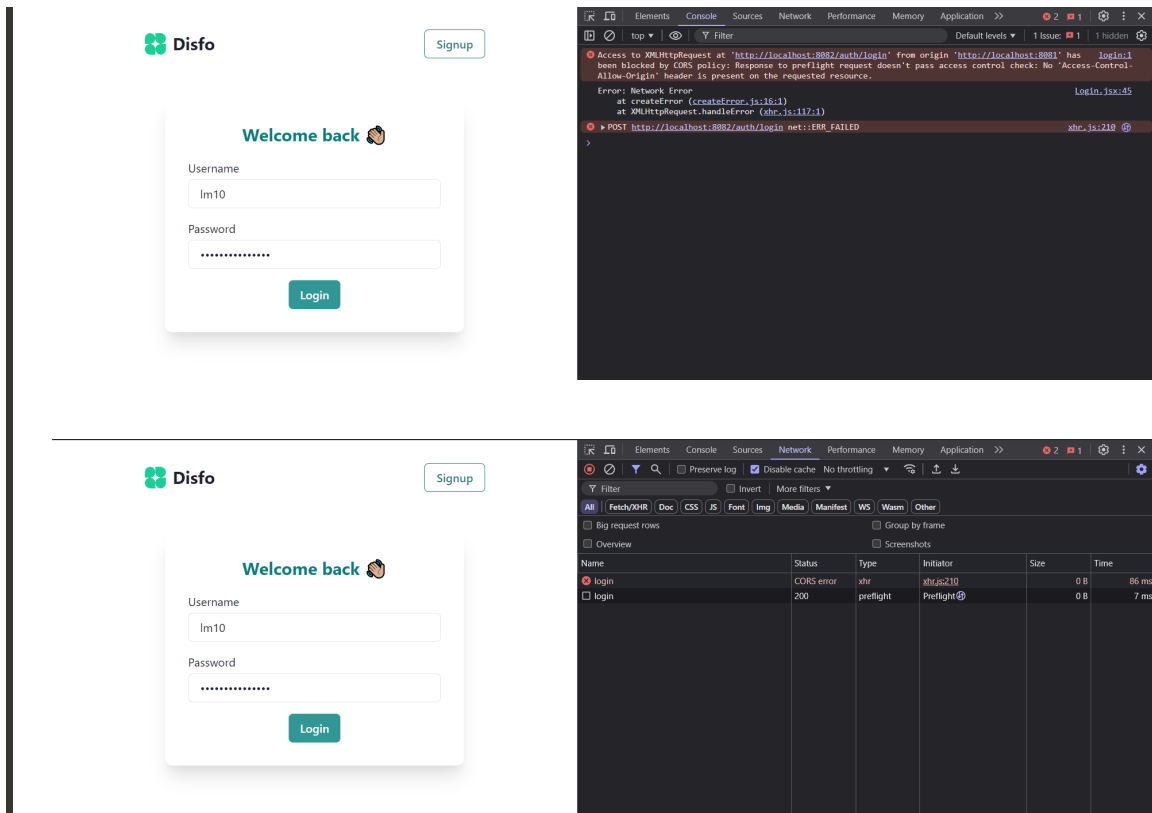
38. Before the actual request is made the browser checks for the CORS access by making a pre-flight request.

Preflight request

## Preflight request

**It is an HTTP request of the OPTIONS method, sent before the request itself, in order to determine if it is safe to send it**

- It is only after the server has sent a positive response that the actual HTTP request is sent.



## Session-10 (CORS Package usage)

1. In order to handle this error we will use CORS Package - <https://www.npmjs.com/package/cors>
2. Import and use the cors package in the index.js file.
3. {withCredentials: true} → Option in the frontend POST call code what it does it it says whatever cookies are associated with the client domain should be sent when a request is made to that server domain.
4. We also need to specify at the server side explicitly like below:

```
app.use(cors({
  origin: "http://localhost:8081", // If we don't mention
  // methods: ["GET", "POST", "PUT", "DELETE", "PATCH"],
  optionsSuccessStatus: 200,
  credentials: true, // Explicitly tells that Cookies and
}));
```

## Why Do You Get a CORS Error Without It?

When the client makes a `fetch` or `axios` request with `credentials: "include"` or `withCredentials: true` and the server doesn't allow credentials by setting `credentials: true` in the `cors` middleware, the browser blocks the request due to a mismatch in expectations. This triggers a CORS error because:

- The browser checks if the server explicitly allows sending cookies or credentials.
- If `credentials: true` is not set, the server's response does not include the required `Access-Control-Allow-Credentials: true` header.
- Without this header, the browser refuses to include sensitive data like cookies in the request.

- In Docs it's present to bypass the Access-Control-Allow-Credentials which will be empty without `credentials: true` option.

- `credentials`: Configures the `Access-Control-Allow-Credentials` CORS header. Set to `true` to pass the header, otherwise it is omitted.

5. We can add some options in the headers like below.

```
if (!isLoggedIn) {
  res.status(401).send({ isLoggedIn });
}
res
  .cookie(
    'remember-user-token',
    AuthServiceInstance.generateJwt({ userId: reqUser._id }),
    { httpOnly: true, maxAge: 1 * 60 * 60 * 1000 } // we can also write: "expires: new Date()" in place of maxAge.
  )
  .send({ isLoggedIn });
```

6. Some Interview Questions in the Slides to go through

## Session 9 Slides Completed

## Session 10 Slides Started

1. Agenda - Authorizing users using passport.js package.
2. The reason why we built using these tokens is that the server to know that we are one that are logged in and making requests.
3. We created one more function in the `auth.service.js` called `verifyJwt`.
  - `verifyJwt` (token, SecretKey)



- Now we added a GET request from the already existing user with the Authorization token (User received this token when first logged in)

```
GET http://localhost:8082/user/lm10
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXV
```

4. We will see both Authorizations here first one using JWT second one using Passport.
4. Created a file called `authorize-jwt.middleware.js` for adding at the `/:username` route for checking before it goes to the controller.
5. Result of verification is the payload that was used to create the token.
6. Below is the code written so far and used this middleware in the

`user.routes.js`.

```
const AuthService = require("../services/auth.service");
const AuthServiceInstance = new AuthService();

const authorize = (req, res, next) => {
  try {
    const token = req.headers.authorization.split(" ");
    const result = AuthServiceInstance.verifyJwt(token)
    // console.log(result);

    next();
  } catch (error) {
    if (error.name === "TokenExpiredError") {
      return res.sendStatus(403).send({ message: "JWT"
    }
    if (error.name === "JsonWebTokenError") {
      return res.sendStatus(403).send({ message: error
    }
    res.sendStatus(401);
  }
}
```

```
module.exports = authorize;
```







8. Now we know that the result in the above code holds the payload. The payload contained `userId`, `iat` and `exp` as we saw earlier. So we will extract the `userId` from payload.
9. Now we will use this `userId` and find out who the user is and then attach it to the request body. But why are we attaching the user to the request body?

```
try {  
    const token = req.headers.authorization.split(" ")  
    // const result = AuthServiceInstance.verifyJwt(token)  
    // console.log(result);  
    const { userId } = AuthServiceInstance.verifyJwt(token)  
    const user = await UserServiceInstance.findById(userId)  
    req.user = user;  
    next();  
}
```

10. Understand this - Before the flow reaches the Controller the middleware `authorize-jwt` is reached and then `authorize` function is called. Token is extracted from the headers. Token is verified. The result of verification is if it's valid we get the `userId` of the user. Once we get the `userId` we can find the user in the DB. Then attach the user to the request object.
11. Now in `user.controller.js` we have `getUserByUsername` function. Inside that we added a new line now. This is to stop the Current user from accessing the user details of another user. Say if Person A is logged in and made a request for Person B profile then he should not be able to see or modify the data of Person B.

```
if (req.user.username !== username) {  
    return res.sendStatus(403);  
}
```

12. Basically it performs a security check to ensure that the authenticated user is authorized to access or modify the resource specified in the request.
  13. Now created a new user called Reacher. Logged in as Reacher. Using Reacher's Token I try to get the details but as lm10 in the request. I get forbidden as expected. When i make the it reacher in `/user/reacher` during request it gives me the details.
- 
14. Now this was one way to do this Authorization. But here we will learn one more way to do this using `passport.js` package.

<p><b>Passport.js</b> Simple, unobtrusive authentication for Node.js</p> <p> <a href="https://www.passportjs.org/">https://www.passportjs.org/</a></p>	 Simple, unobtrusive authentication for Node.js 
<p><b>passport-jwt</b> Passport authentication strategy using JSON Web Tokens</p> <p> <a href="https://www.passportjs.org/packages/passport-jwt/">https://www.passportjs.org/packages/passport-jwt/</a></p>	 Simple, unobtrusive authentication for Node.js 

15. We will need two packages. Passport and Passport-jwt here for our usage in disfo. → `npm i passport passport-jwt`.
16. Created a config folder and then `passport.js` file. Skeleton is below.

```
const JwtStrategy = require('passport-jwt').Strategy;
const ExtractJwt = require('passport-jwt').ExtractJwt;

const options = {
  jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken,
  secretOrKey: process.env.JWT_SECRET_KEY,
};

// JwtStrategy constructor takes two parameters, options and a callback function
const strategy = new JwtStrategy(options, (payload, done) {
```

```
});  
  
module.exports = strategy;
```

17. Once the strategy is completed with its logic we now create a middleware called as `authorize-passport.middleware.js`. Just these 4 lines.

```
const passport = require("passport");  
const JwtStrategy = require("../config/passport");  
  
passport.use(JwtStrategy);  
  
module.exports = passport.authenticate();
```

18. Now we will make use of this middleware in the `user.route.js` instead of previous one which one was using the jwt-authorization thing.
19. Now we create a token for reacher and then request using that token we get the details now.
20. So Basically we had two steps:
- a. Create a Strategy ( In Config → `passport.js` )
  - b. Create a middleware ( `authorize-passport.middleware.js` )

| [Read more about Passport in Docs.](#)

21. Some Interview questions in the slides.

---

**Session-10 Completed**