

Web Framework 1 – Week 10 and 11

Web Security, Authentication, Authorization

Professor: Harman Mann

Summer 2024

Agenda

- Implement security features in NodeJS app
- Authentication and Authorization
- HTTP Cookie and Session

How to protect credential info in source code?

- It is a best practice to not to include API-key, database password, ... in the source code!
- Create a .env file in the root directory of your project.
- Add environment-specific variables on new lines in the form of NAME=VALUE.
 - DB_HOST=localhost
- process.env now has the keys and values you defined in your .env file
 - db.connect({
 - host: process.env.DB_HOST,
 - })

dotenv

- Dotsenv is a **zero-dependency module** that loads environment variables from a .env file into process.env.
 - Storing configuration in the environment separate from code is based on The Twelve-Factor App methodology.
- <https://12factor.net/>
- In a twelve-factor app, env vars are granular controls, each fully orthogonal to other env vars. They are never grouped together as “environments”, but instead are independently managed for each deploy. This is a model that scales up smoothly as the app naturally expands into more deploys over its lifetime.
- Not having multiple .env and not commit your .env to version control!

Activity: Add the following files and show how it works

- Create a .env file in your project folder (typically at the root) and place the key value pairs in there
 - # This is a comment
 - HOST = localhost
 - DATABASE = mydb
 - PORT = 5432
- Create a simple-yourname.js and run it. Explain how it works.
 - `require('dotenv').config()`
 - `const hostname = process.env.HOST;`
 - `const database = process.env.DATABASE;`
 - `const port = process.env.PORT;`
 - `console.log(hostname);`
 - `console.log(database);`
 - `console.log(port);`

Config

- config will read your .env file, parse the contents, assign it to process.env, and return an Object with a parsed key containing the loaded content or an error key if it failed.
 - `const result = dotenv.config()`
 - `if (result.error) {`
 - `throw result.error`
 - `}`
 - `console.log(result.parsed)`

.env Path

- Specify a custom path if your file containing environment variables is located elsewhere.
 - `require('dotenv').config({ path: '/custom/path/to/.env' })`
- Question:
- Try this command: `console.log(process.cwd())`
- What does it do?

Deploying with Vercel (again)

- Setting environment variables on your local development machine or in a VM is only half the work.
- During deployment of application in a cloud environment such as Heroku, you need to define environment variables.
 - You can find instructions on how to set them in their documentations

Part 2: Password Encryption

Password Encryption

- How to protect your site – if someone gets into your database and steals all your users' passwords that are stored in plain text?
- Encrypt password-field in the database when a user registers and when they try to login, encrypt their plain text password that is sent to the server over a HTTPS POST request, encrypt it and compare the encrypted passwords for a match.
- A great node.js library to use for the one-way encryption of passwords is `bcrypt.js`.

bcrypt.js

- Install bcrypt.js using npm – and include it in your solution using:
- **`const bcrypt = require('bcryptjs');`**
 - This hashes the password using famous hash algorithms like:
- md5, sha224, sha256, sha512, etc.

Hashing: add salt!

- Adding a salt to a hash increases its security
- A **salt is just an extra sequence of characters added to the password**
- Proper salts can render rainbow table attacks useless
- The salt needs to be stored together with the hash

Store data using hash

- If we wish to encrypt a plain text password (ie: "myPassword123"), we can use bcrypt to generate a "salt" and "hash" the text

JS bcrypt_test.js X

test > JS bcrypt_test.js > ...

```
1 const bcrypt = require("bcrypt");
2
3 const password = "password123";
4
5 async function run() {
6   const p1 = bcrypt.hash(password, 10);
7   const p2 = bcrypt.hash(password, 10);
8   const p3 = bcrypt.hash(password, 10);
9
10  const [hashed1, hashed2, hashed3] = await Promise.all([p1, p2, p3]);
11
12  console.log(hashed1);
13  console.log(hashed2);
14  console.log(hashed3);
15
16  const isMatch = await bcrypt.compare("password123", hashed1);
17  const isMatch2 = await bcrypt.compare("password1234", hashed1);
18  console.log("Password matches:", isMatch);
19  console.log("Password matches:", isMatch2);
20 }
21
22 run();
23
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
PS C:\Users\Harma\OneDrive\Documents\GitRepos\WF1-TEMP-W9-Mongoose\test> node .\bcrypt_test.js
$2b$10$xCPmIKANBJy5e4iNFSVhju1aNj5ieFnpG3J7GwY0cEDrFebHc2AIi
$2b$10$Lrjqlyt2fbe.JvLCgMw5S.DCoHGIQBhc41p.dCh4oNsBVgoejNnmK
$2b$10$.KgmxlVs3FTAdjMPvYaP0tdf9us4fN1UPSeZF8wWi8G/UM3VRiYG
Password matches: true
Password matches: false
PS C:\Users\Harma\OneDrive\Documents\GitRepos\WF1-TEMP-W9-Mongoose\test>
```

compare the
“hashed”
text

JS bcrypt_test.js X

test > JS bcrypt_test.js > ...

```
1 const bcrypt = require("bcrypt");
2
3 const password = "password123";
4
5 async function run() {
6   const p1 = bcrypt.hash(password, 10);
7   const p2 = bcrypt.hash(password, 10);
8   const p3 = bcrypt.hash(password, 10);
9
10  const [hashed1, hashed2, hashed3] = await Promise.all([p1, p2, p3]);
11
12  console.log(hashed1);
13  console.log(hashed2);
14  console.log(hashed3);
15
16  const isMatch = await bcrypt.compare("password123", hashed1);
17  const isMatch2 = await bcrypt.compare("password1234", hashed1);
18  console.log("Password matches:", isMatch);
19  console.log("Password matches:", isMatch2);
20 }
21
22 run();
23
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
● PS C:\Users\Harma\OneDrive\Documents\GitRepos\WF1-TEMP-W9-Mongoose\test> node .\bcrypt_test
$2b$10$xCPmIKAnBJy5e4iNFSVhju1aNj5ieFnpG3J7GwY0cEDrFebHc2AIi
$2b$10$Lrjqlyt2fbe.JvLCgMw5S.DCoHGIQBhc41p.dCh4oNsBVgoejNnmK
$2b$10$.KgmxlVs3FTAdjMPvYaPOtdf9us4fN1UPSeZF8wWi8G/UM3VRiYG
Password matches: true
Password matches: false
○ PS C:\Users\Harma\OneDrive\Documents\GitRepos\WF1-TEMP-W9-Mongoose\test>
```

HTTP is stateless

- HTTP is called as a stateless protocol because each request is executed independently, without any knowledge of the requests that were executed before it
- The question is How to make HTTP “act” stateful?
 - And before that : Do we really need to make HTTP behave stateful?

To make HTTP act stateful

- Session
 - Session is an object which is stored on a web server
 - web session ID is stored in a visitor's browser to track sessions.
 - This session ID is passed along with any **HTTP requests** that the visitor makes while on the site
- Cookie
 - is a small piece of data in “{name}={value}” pair which is stored on the client side

Differences between Authentication vs Authorization

Authentication

- Answer the question **who you are.**
- Verify the **identity** of a user

Authorization

- Answer the question **what do you have access to?**
- Verifies the **privileges** of a user

Status codes

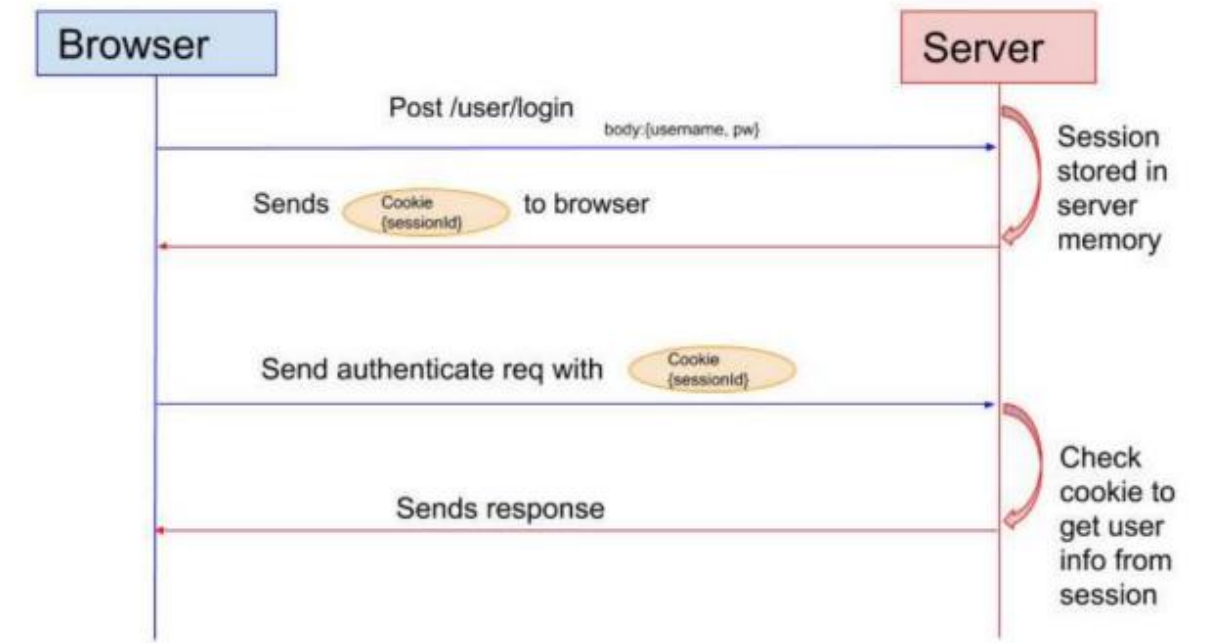
- HTTP response status codes – can be used by your application to inform the browser of whether a request was rejected because of an authentication problem or an authorization problem.
- **401 status code (Unauthorized)** Authentication error. The resource exists but it requires the user to be authenticated first to view it. It may also require permissions and be checked again after authenticating for proper authorization.
- **403 status code (Forbidden)** Authorization error. The resource exists but the user does not have permission to view it.
- **404 status code Not Found.** The resource that was requested was not found on the server. This is commonly used when a url is requested that just doesn't exist.

Web App Security

- What is Web App Security?
- → AAA – **Accounting** DB table: users, Register()/ Sign Up
 - **Authentication**: Who are you? Login()
 - **Authorization**
- What can you do? What resource can you access?
 - e.g.: Securing routes in data-service.js of an app/api
- To ensure AAA
 - http -> https -> certification
 - Db users: password encryption
 - bcrypt.js
 - Session management

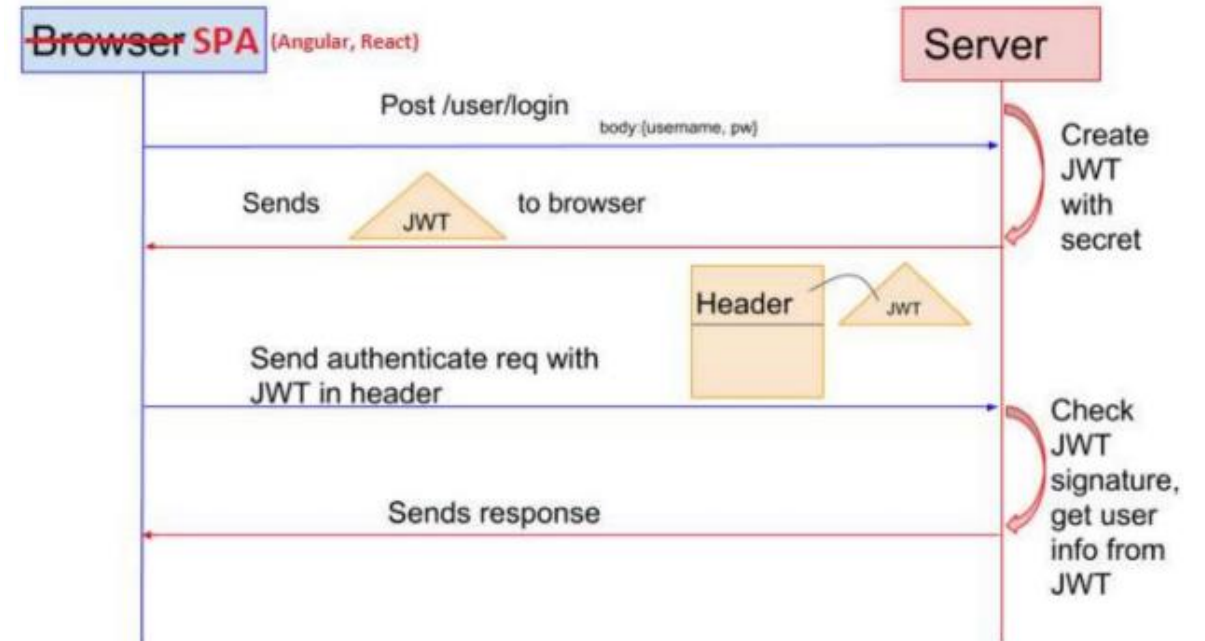
Web App Security

- Session based authentication:
 - State management – First http request: login > session created on server > encryption > cookie > save in users' browser by using client-session.js
 - In all following http requests/responses, the session is attached

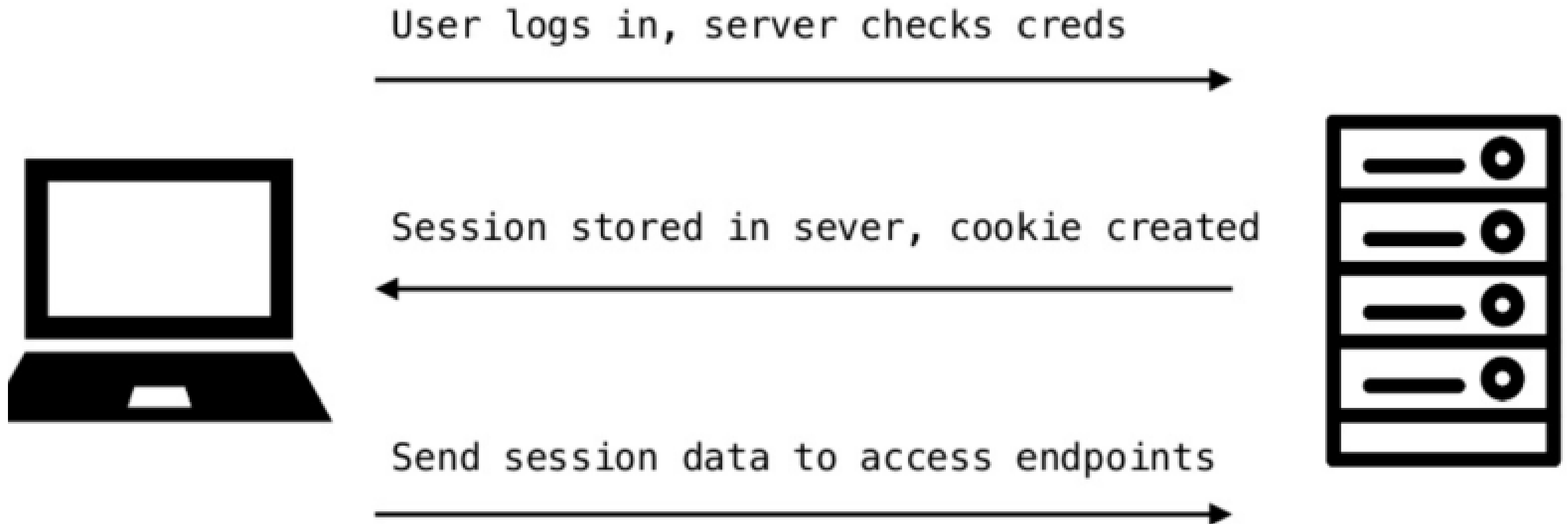


Securing a WEB API

- WEB API:
 - No UI, no browser, no session
- JWT(JSON Web Token)
 - Token based authentication
 - Similar to session based, but it's for Web API



Traditional Authentication System



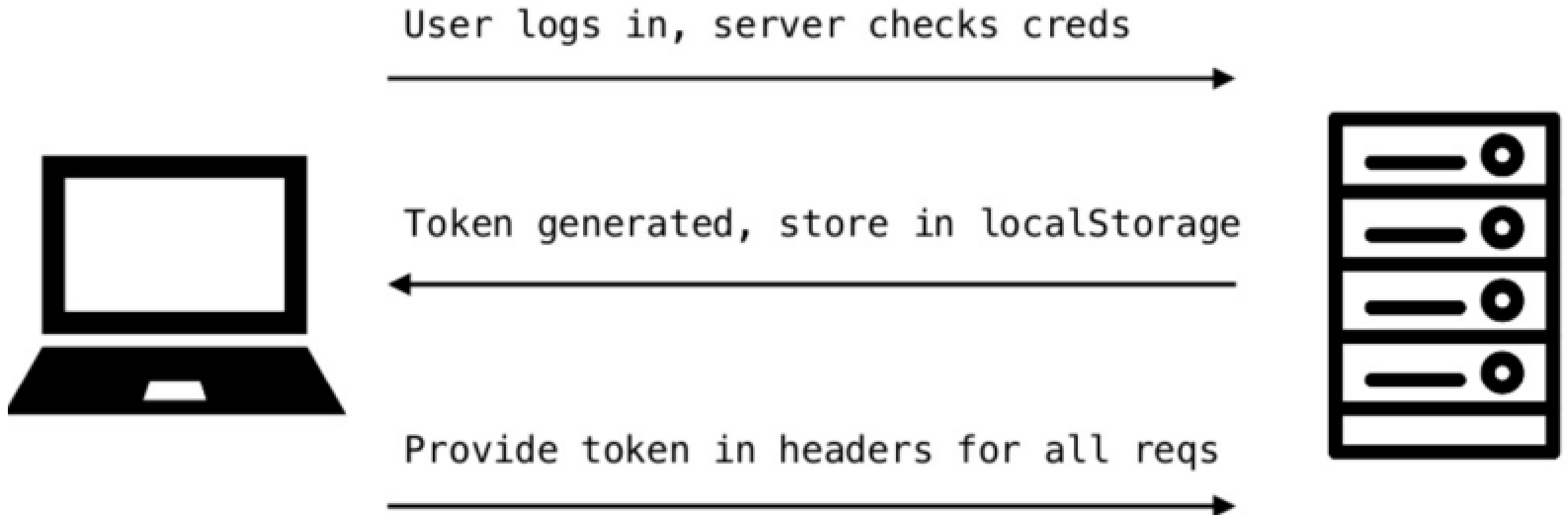
Issues with traditional systems

- Sessions: **Record needs to be stored on server (Space problem)**
- Scalability: With session in memory, load increases drastically in distributed system
- CORS: When using multiple devices grabbing data via AJAX requests, we may run into forbidden requests.
- CSRF Attacks: Riding session data to send commands to server from a browser that is trusted via session.

JSON web tokens (JWT)

- A JSON web token, or JWT (“jot”) for short, is a standardized, optionally validated and/or encrypted container format that is used to securely transfer information between two parties.

Token based Authentication Systems



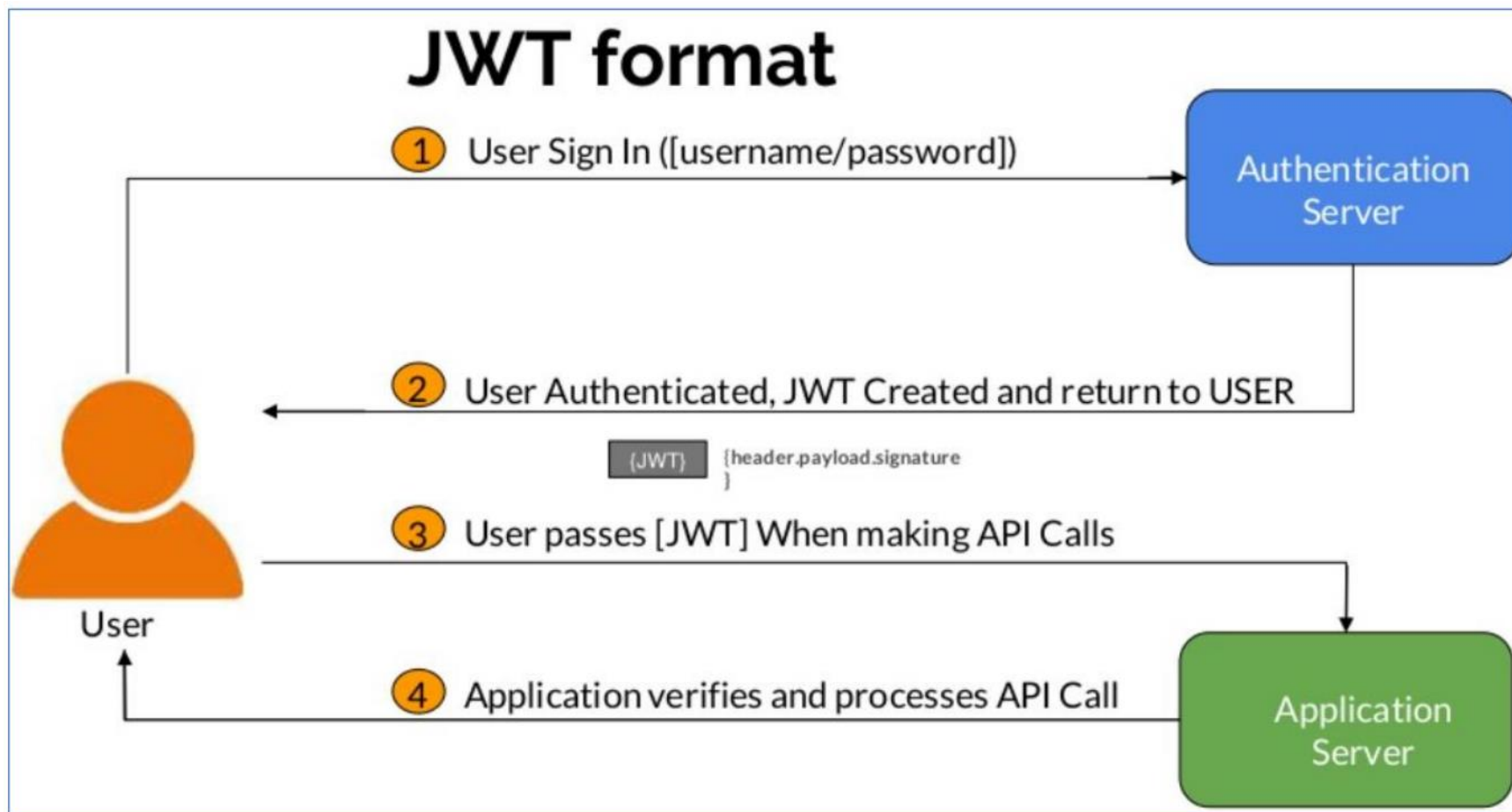
What is JWT

- Official website: – <https://jwt.io/>
- RFC: – <https://tools.ietf.org/html/rfc7519>

JWT Benefits

- They are contained and help maintain stateless architecture
- They work well across multiple programming languages

JWT workflow



JWT Format

- JSON Web Token (JWT) is a compact URL-safe means of representing claims to be transferred between two parties
- **Header.payload.signature**
 - Every JWT carries a header with claims about itself.
- These claims establish the algorithms used, whether the JWT is signed or encrypted, and in general, how to parse the rest of the JWT.
 - The payload is the element where all the interesting user data is usually added.
 - The purpose of a signature is to allow one or more parties to establish the authenticity of the JWT
 - JWT Handbook:
 - <https://auth0.com/resources/ebooks/jwt-handbook/thankyou>

JWT Format

- Header.payload.signature
- Header: consists of two parts:
 - The type of the token, which is JWT
 - The hashing algorithm being used, such as SHA256 or RSA
- For example: { – “alg”:HS256”, – “typ”:”JWT”}

HMAC SHA256 vs RSA SHA256

- HMAC SHA256: Symmetric key cryptography, single shared private key.
 - **Faster**
 - **good between trusted parties.**
- RAS SHA256: Asymmetric key cryptography, use public/private keys
 - **Slower**
 - **Good between untrusted parties**

Reserved Claims

- iss (issuer) The person that issued the token
- sub (subject): The subject of the token
- aud (audience): Audience the token is intended for
- nbf (not before): Starting time token is available
- iat (issued at): When the token was issued.
- jti (JWT ID) : Unique identifier for the token
- exp (expiration) : Time the token expires

Payload: Example

- var payload = {
 - sub : '456598',
 - exp : '12345678',
 - jti: '841234',
 - role: 'admin' }

Part 3: Coding

- What we will do for the next 2 weeks:
 - 1 – Creating a User System
 - Signup and Login (if you do not want to make the HTML pages, we can use POSTMAN)
 - Create associated routes to handle them
 - Make a Model file to handle submissions to the Database
 - 2 – Using Sessions to Store our Login attempt
 - 3 – Using JWT to store our Login attempt

Links to the repository

- Follow everything in this order:

1. Refactored - [HarmanSMann/WF1-TEMP-W9-Mongoose at refactor-book_router \(github.com\)](#)
2. User System - [HarmanSMann/WF1-TEMP-W9-Mongoose at w10-UserSystem \(github.com\)](#)
3. Adding Bcrypt - [HarmanSMann/WF1-TEMP-W9-Mongoose at w10-adding-bcrypt \(github.com\)](#)
4. (OPTION 1) - Adding Session - [HarmanSMann/WF1-TEMP-W9-Mongoose at w10-adding-sessions \(github.com\)](#)
5. (OPTION 2) - Adding JWT - [HarmanSMann/WF1-TEMP-W9-Mongoose at w10-adding-jwt \(github.com\)](#)


Refactor -> User System

1. Make a new model for users: user.js
2. Make a user router to handle routing: /signup (or call it register), /login
 1. Signup:
 1. Get – send an html page with a form for fname, lname, username, email, password submission
 2. Post – receive request data
 1. Create a user object and save each request item to the object
 2. Save the object to our database
 2. Login:
 1. Get – send an html page with a form for email, password submission
 2. Post – receive request data
 1. Extract data
 2. Find user from DB
 3. Compare password
 4. Send out a Hello Message
3. Add the user router into the index.js

Part 1.1: User model

```
models > user.js > ...
1  const mongoose = require("mongoose");
2
3  const userSchema = new mongoose.Schema(
4    {
5      first_name: {
6        type: String,
7        required: true,
8      },
9      last_name: {
10       type: String,
11       required: true,
12     },
13     username: {
14       type: String,
15       required: true,
16       unique: true,
17       trim: true,
18       minlength: 3,
19     },
20     email: {
21       type: String,
22       required: true,
23       unique: true,
24       trim: true,
25       lowercase: true, // Convert email to lowercase
26       match: [/.+@.+.+/, "Please fill a valid email address"],
27     },
28     password: {
29       type: String,
30       required: true,
31       minlength: 6,
32     },
33   },
34   { timestamps: true }
35 );
36
37 const User = mongoose.model("User", userSchema);
38
39 module.exports = User;
```

Part 1.2: User router

```
router >  user_router.js > ...
```

```
1  const express = require("express");  
2  const router = express.Router();  
3  const path = require("path");  
4  const User = require("../models/user");  
5  const { validateUser } = require("../middleware/validator");  
6
```

Part 1.2.1: User router – Sign up route

```
15 router
16   .route("/sign-up")
17   .get((req, res) => {
18     res.sendFile(path.join(__dirname, "../views", "sign-up.html"));
19   })
20   .post(validateUser, (req, res) => {
21     const { first_name, last_name, username, email, password } = req.body;
22
23     const newUser = new User({
24       first_name,
25       last_name,
26       username,
27       email,
28       password,
29     });
30
31     newUser
32       .save()
33       .then(() => {
34         res.status(201).json({ message: "Successfully Added" });
35       })
36       .catch((err) => {
37         console.error(err);
38         res.status(500).json({ error: "Internal Server Error" });
39       });
40   });
```




Part 1.2.2: User router - Login

```
42 router
43 .route("/login")
44 .get((req, res) => {
45   res.sendFile(path.join(__dirname, "../views", "login.html"));
46 })
47 .post((req, res) => {
48   const { email, password } = req.body;
49
50   User.findOne({ email })
51     .then(user => {
52       if (!user) {
53         console.log(`User with email ${email} not found`);
54         return res.status(400).json({ error: `User with email ${email} not found` });
55       }
56
57       // Compare passwords
58       if (user.password !== password) {
59         console.log(`Incorrect password for user ${email}`);
60         return res.status(400).json({ error: `Incorrect password for user ${email}` });
61       }
62
63       // Passwords match, login successful
64       console.log(`Hello ${user.first_name}, with username ${user.username} (${email}) logged in successfully`);
65       res.json({ message: `Hello ${user.first_name}, with username ${user.username} (${email}) logged in successfully` });
66     })
67     .catch(error => {
68       console.error("Error during login:", error);
69       res.status(500).json({ error: "Internal Server Error" });
70     });
71 });
```


Part 1.3: Adding it to index.js

```
JS index.js > ...
1  require("dotenv").config();
2  const path = require("path");
3  const express = require("express");
4  const app = express();
5
6  const mongoose = require("mongoose");
7  const PORT = process.env.PORT || 8000;
8  const book_router = require("./router/book_router");
9  const user_router = require("./router/user_router");
10
11 // application level middleware
12 app.use(express.urlencoded({ extended: true }));
13 app.use(express.json());
14 app.use("/bookstore", book_router);
15 app.use("/user", user_router);
16
17 mongoose.connect(process.env.DB_HOST);
18 let db = mongoose.connection;
19
20 db.once("open", () => {
21   console.log("Connected to MongoDB");
22 });
23 db.on("error", (err) => {
24   console.log("DB Error:" + err);
25 });
26
27 app.get("/", (req, res) => {
28   res.redirect("/bookstore");
29 });
30
31 app.listen(PORT, () => {
32   console.log(`Server started on http://127.0.0.1:${PORT}`);
33 });
34
```

Part 1.4: Updating the Validator Middleware

middleware >  validator.js >  validateBook >  <function>

```
1  const { body, validationResult } = require("express-validator");
2
3  // Validator for validating book data
4  const validateBook = [
5    body("title", "Title is required").notEmpty(),
6    body("author", "Author is required").notEmpty(),
7    body("pages", "Pages is required").notEmpty(),
8    body("rating", "Rating is required").notEmpty(),
9    body("genres", "Genre is required").notEmpty(),
10   (req, res, next) => {
11     const errors = validationResult(req);
12     if (!errors.isEmpty()) {
13       return res.status(400).json({ errors: errors.array() });
14     }
15     next();
16   },
17 ];
```

```
19  // Validator for validating user data
20  const validateUser = [
21    body("first_name", "First Name Field is required").notEmpty(),
22    body("last_name", "Last Name Field is required").notEmpty(),
23    body("username", "Username Name Field is required").notEmpty(),
24    body("email", "Valid email is required")
25      .isEmail()
26      .withMessage("Email is not valid")
27      .notEmpty()
28      .withMessage("Email Field is required"),
29    body("password", "Password is required").notEmpty(),
30    (req, res, next) => {
31      const errors = validationResult(req);
32      if (!errors.isEmpty()) {
33        return res.status(400).json({ errors: errors.array() });
34      }
35      next();
36    },
37  ];
38
39  module.exports = {
40    validateBook,
41    validateUser,
42  };
43
```

That is everything for our User System right now

- Next: User System -> Bcrypt
 - When storing the user information, we save their password as plain text
 - We should fix that; we will be using bcrypt to hash the password.
- This will involve changing:
 1. Bcrypt integration to our user route
 2. Updating the validator for both signup and login

Install: **npm install bcrypt**

1 – Updating the Validator options

```
19 // Validator for validating user data
20 const validateUserSignup = [
21   body("first_name", "First Name Field is required").notEmpty(),
22   body("last_name", "Last Name Field is required").notEmpty(),
23   body("username", "Username Name Field is required").notEmpty(),
24   body("email", "Valid email is required")
25     .isEmail()
26     .withMessage("Email is not valid")
27     .notEmpty()
28     .withMessage("Email Field is required"),
29   body("password", "Password is required").notEmpty(),
30   (req, res, next) => {
31     const errors = validationResult(req);
32     if (!errors.isEmpty()) {
33       return res.status(400).json({ errors: errors.array() });
34     }
35     next();
36   },
37 ];
```

```
39 const validateUserLogin = [
40   body("email", "Valid email is required")
41     .isEmail()
42     .withMessage("Email is not valid")
43     .notEmpty()
44     .withMessage("Email Field is required"),
45   body("password", "Password is required").notEmpty(),
46   (req, res, next) => {
47     const errors = validationResult(req);
48     if (!errors.isEmpty()) {
49       return res.status(400).json({ errors: errors.array() });
50     }
51     next();
52   },
53 ];
54
55 module.exports = {
56   validateBook,
57   validateUserSignup,
58   validateUserLogin,
59 };
60
```

2 – Updating our user route to handle bcrypt

- Signup
 - Add a hashed password
- Login
 - Fetch user information
 - Decrypt the user password
 - compare

Changes to signup

```
20 // User signup
21 router
22   .route("/sign-up")
23   .get((req, res) => {
24     res.sendFile(path.join(__dirname, "../views", "sign-up.html"));
25   })
26   .post(validateUserSignup, (req, res) => {
27     const { first_name, last_name, username, email } = req.body;
28
29     bcrypt
30     .hash(req.body.password, 10)
31     .then((hashedPassword) => {
32       const newUser = new User({
33         first_name,
34         last_name,
35         username,
36         email,
37         password: hashedPassword,
38       });
39
40       return newUser.save();
41     })
42     .then(() => {
43       res.status(201).json({ message: "Successfully Added" });
44     })
45     .catch((err) => {
46       console.error(err);
47       res.status(500).json({ error: "Internal Server Error" });
48     });
49   });
50
```

Changes to Login

```
51 // User login
52 router
53   .route("/login")
54   .get(validateUserLogin, (req, res) => {
55     res.sendFile(path.join(__dirname, "../views", "login.html"));
56   })
57   .post((req, res) => {
58     const { email, password } = req.body;
59
60     User.findOne({ email })
61       .then((user) => {
62         if (!user) {
63           console.log(`User with email ${email} not found`);
64           return res
65             .status(400)
66             .json({ error: `User with email ${email} not found` });
67         }
68
69         return bcrypt.compare(password, user.password).then((isMatched) => {
70           if (!isMatched) {
71             return res
72               .status(400)
73               .json({ error: `Incorrect password for user ${email}` });
74           }
75
76           console.log(
77             `Hello ${user.first_name}, with username ${user.username} (${email}) logged in successfully`
78           );
79           res.json({
80             message: `Hello ${user.first_name}, with username ${user.username} (${email}) logged in successfully`,
81           });
82         });
83       })
84       .catch((error) => {
85         console.error("Error during login:", error);
86         res.status(500).json({ error: "Internal Server Error" });
87       });
88     });
```

That completes the additions to the User System

- The Next steps towards the Bookstore application
 - Providing persistence through sessions authentication
 - Adding authorization for who has access to content.
- Install: express-session connect-mongo
 - Express session will handle creating the session key for us
 - Connect-mongo will be used to store the session key onto MongoDB

Implementing Sessions

- Inside index.js we need to attach the session to our app
- We need to add onto the session when the user logs in to their account. We want our session to have flag indicating that a user is logged in.

Index.js updates

```
1 require("dotenv").config();
2 const path = require("path");
3 const express = require("express");
4 const session = require("express-session"); //new
5 const MongoStore = require("connect-mongo"); //new
6 const mongoose = require("mongoose");
7
8 const app = express();
9
10 const PORT = process.env.PORT || 8000;
11 // Configure session middleware
12 app.use(
13   session({
14     secret: process.env.SESSION_SECRET,
15     resave: false,
16     saveUninitialized: true,
17     store: MongoStore.create({
18       mongoUrl: process.env.DB_HOST,
19       collectionName: "sessions",
20       ttl: 5 // this is in seconds
21     }),
22     cookie: { maxAge: 1000 * 5 }, // maxAge: 1000 * 60 * 60 * 24 1 day
23   })
24 );
```

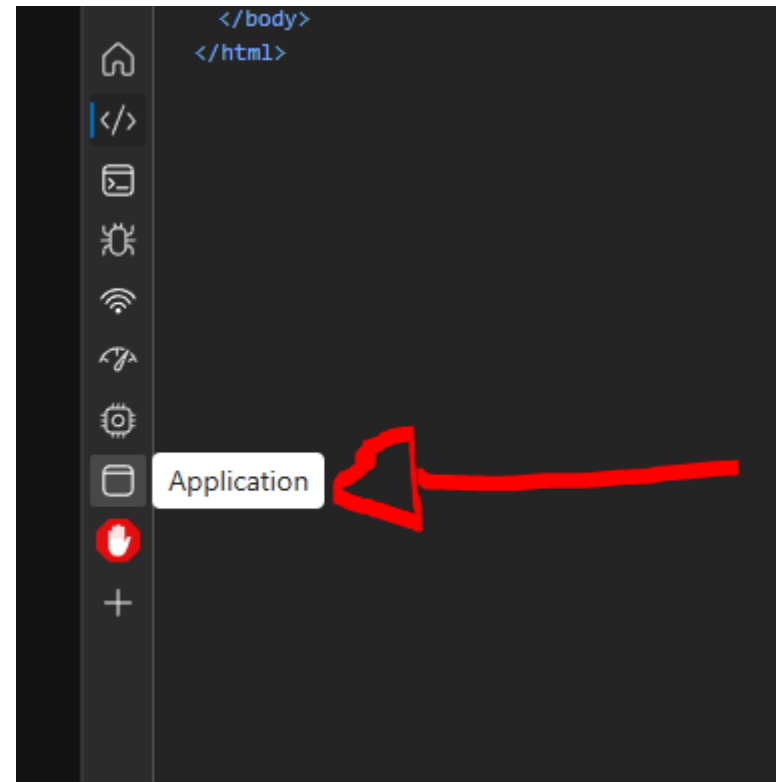
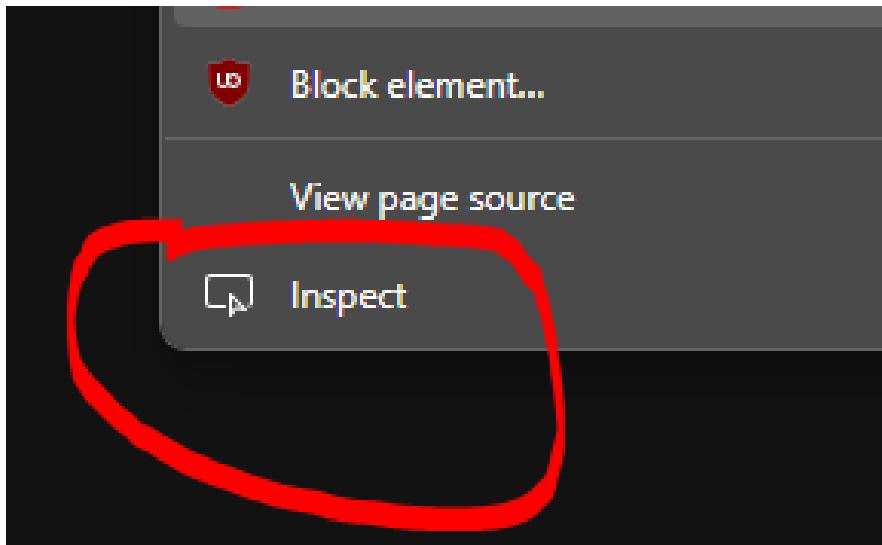
```
45 app.get("/test-session", (req, res) => {
46   if (req.session.views) {
47     console.log(req.session)
48     req.session.views++;
49     res.setHeader("Content-Type", "text/html");
50     res.write("<p>Views: " + req.session.views + "</p>");
51     res.end();
52   } else {
53     req.session.views = 1;
54     res.end("Welcome to the session demo. Refresh!");
55   }
56 });
```

Updating User
route to add
user into
session (only in
the login route)

```
28   User.findOne({ email })
29   .then((user) => {
30     if (!user) {
31       console.log(`User with email ${email} not found`);
32       return res
33         .status(400)
34         .json({ error: `User with email ${email} not found` });
35     }
36
37     bcrypt.compare(password, user.password).then((isMatched) => {
38       if (!isMatched) {
39         return res
40           .status(400)
41           .json({ error: `Incorrect password for user ${email}` });
42       }
43
44       // Save user info in session
45       req.session.user = {
46         id: user._id,
47         email: user.email,
48         username: user.username,
49         first_name: user.first_name,
50         // Add any other necessary user data
51       };
52
53       // Save the session explicitly
54       req.session.save((err) => {
55         if (err) {
56           console.error("Error saving to session storage: ", err);
57           return next(new Error("Error saving session"));
58         }
59       });
60     });
61   });
62 }
```

Well, how do we see it

- In your browser, we can do into the inspect too and go to application to see the connection sid





Application



Application



Manifest



Service workers



Storage



Storage



▶ Local storage



▶ Session storage



IndexedDB



▼ Cookies



http://localhost:8000



Private state tokens



Interest groups



▶ Shared storage



Cache storage



Storage buckets



Background services



Back/forward cache



↕ Background fetch



↻ Background sync



Bounce tracking mitigations



Notifications



Payment handler



🕒 Periodic background sync



▶ ↕ Speculative loads



☁ Push messaging



📄 Reporting API

Frames



▶ top



Filter



Only show cookies with an issue

Name	Value	Do...	Path	Expires / Max-...	Size	HttpOnly	Secure
connect.sid	s%3A2rB69LFuNVOkGqJBjlsfwSbj8RmWGssy.sEOgEx618ePhN0...	loc...	/	2024-07-16T01:...	91	✓	
csrftoken	S4LjOuxkZ0dVDD7e3a13fdSDk2X0i1WD	loc...	/	2025-07-10T03:...	41		

Well how do we delete the session when a user want to logout -> we delete the session

- We will have a logout route that handles the user session data and delete it (session.destroy)

```
121 // User logout
122 router.post("/logout", (req, res) => {
123   req.session.destroy((err) => {
124     if (err) {
125       console.error("Error during logout:", err);
126       return res.status(500).json({ error: "Internal Server Error" });
127     }
128
129     res.json({ message: "Logged out successfully" });
130   });
131 });
```

How does our browser know we are logged in?

- We can send over a response to the user indicating that the user session is still happening “check status”

```
105 // Check login status
106 router.get("/check-login", (req, res) => {
107   if (req.session.user) {
108     // If user session exists, return user data
109     res.json({
110       isLoggedIn: true,
111       user: req.session.user,
112     });
113   } else {
114     // If user session does not exist, return isLoggedIn false
115     res.json({
116       isLoggedIn: false,
117     });
118   }
119 });
```

And finally, Authorization

- We are able to setup a user to be logged in now
- What we also want to do is restrict what a user can access
 - Who has access to all the information on the DB – Admin
 - What about seeing class information – Students in the class
 - Who can see your grades? – Yourself, Your Teacher, anyone with higher authority

The blatant way of doing it

- Create a hierarchy system. Who has access to what and for each route, setup middleware that checks the level of privilege for the user.
- Inside your User Collection (Table) indicate the role they have on the application
- What about who has access to your information:
 - Someone who matches your ID in the DB and someone who has higher privilege

What about who has access to your information:
Someone who matches **your ID in the DB** and
someone who has higher privilege

- We can setup a function to determine a few things, and pass that along as middleware
 - Are they logged in?
 - Does their `_ID` match the route they are trying to reach
 - Do they have the authority to look into the information
- The order of operation does matter, but the final order will be up to you and the applications you make with your team.

Find all users – Making a friend example

- “I want to be able to look through the list of users on the app and find people who I can relate to.”
- From our end: what information can we provide? Filter the options to showcase to them what information they need. Make sure they are a user of our site BEFORE showcasing the list

```
133 router.get("/api/users", authMiddleware, (req, res) => {
134   User.find({}, { password: 0, createdAt: 0, updatedAt: 0, __v: 0 })
135   .then((users) => {
136     res.json(users);
137   })
138   .catch((err) => {
139     console.error("Error fetching user list:", err);
140     res.status(500).json({ error: "Internal Server Error" });
141   });
142 });
```

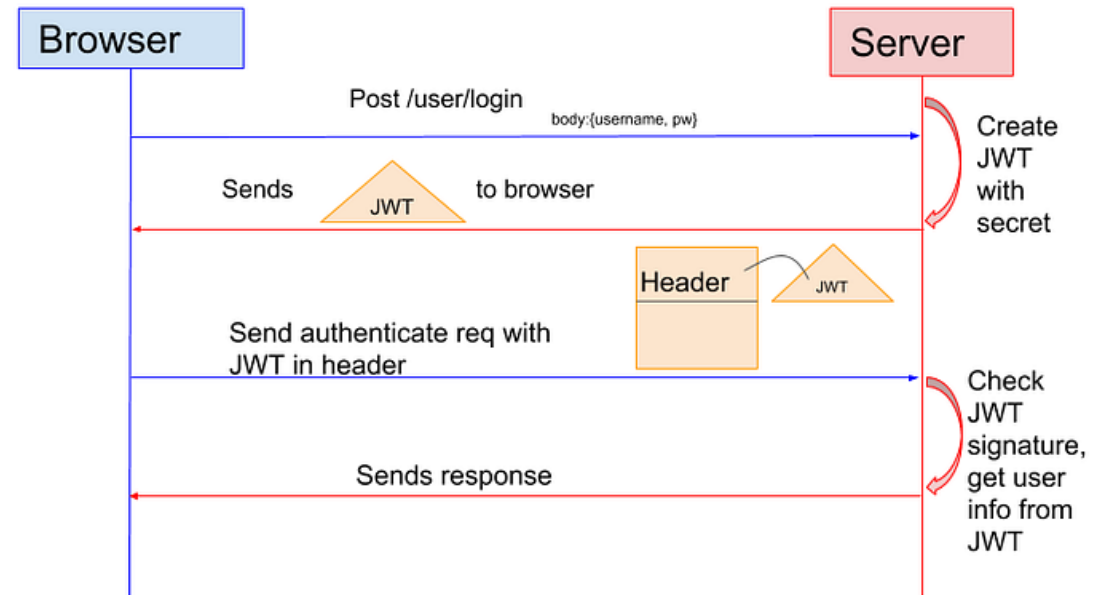
```
middleware > JS auth.js > <unknown>
1 module.exports = (req, res, next) => {
2   if (req.session && req.session.user) {
3     next(); // User is logged in, proceed to the next middleware/route handler
4   } else {
5     res.status(401).json({ message: "Unauthorized: Please log in" });
6   }
7 };
8
```

We are done with Sessions

- Implementing the session
- Adding the user into the session when they login
- Authenticate the user
- Authorize what they can access

Part 3.5 - JWT

- Sessions are stored on the server
- JWTs are created on the server and sent to the client.
- So, for us, similarly to sessions, once the user logs in, we send them a token
- The difference – the client must send their token with each request



Index file change

- Nothing, we only need to create the token when they log into the app

In the user router

- When the user logs in, we make them a token
- The normal command is `jwt.sign()` to create the token, I have moved it into its own function so we can make it more modular
- The command: `jwt.sign({Information you want to store}, "SECRET PASSWORD", {additional information})`

```
return jwt.sign({  
  userId: user.id,  
  username: user.username,  
  email: user.email,  
  firstName: user.first name,  
  lastName: user.last name  
}, process.env.ACCESS_TOKEN_SECRET, { expiresIn: '30m' });
```

User router login

- We create a token and send the token with our response
- The client side has to handle what to do with the token

```
// Generate JWT token with additional user information
const token = jwtUtils.generateAccessToken({
  _id: user._id,
  username: user.username,
  email: user.email,
  first_name: user.first_name,
  last_name: user.last_name,
});

res.json({
  token,
  message: `Hello ${user.first_name}, with username ${user.username} (${email}) logged in suc
});
```


Made a Util file for JWT

- Create token
- Validate token

```
3 function generateAccessToken(user) {
4   return jwt.sign({
5     userId: user._id,
6     username: user.username,
7     email: user.email,
8     firstName: user.first_name,
9     lastName: user.last_name
10  }, process.env.ACCESS_TOKEN_SECRET, { expiresIn: '30m' });
11 }
12
```

```
27 const verifyToken = (req, res, next) => {
28   const token = req.header("Authorization")?.replace("Bearer ", ""); // Extract token from Au
29   if (token == "null") {
30     return res.status(401).json({ error: "Not Logged in" });
31   }
32   if (!token) {
33     return res.status(401).json({ error: "Unauthorized" });
34   }
35   try {
36     const decoded = jwt.verify(token, process.env.ACCESS_TOKEN_SECRET); // Verify JWT token
37     req.user = decoded; // Attach user information to request object
38     next(); // Move to the next middleware
39   } catch (err) {
40     console.error(err);
41     res.status(401).json({ error: "Unauthorized" });
42   }
43 };
44
```

How do we store the information to the client

- Inside our login.html page, there is a small section where we call localStorage and save the token in there

```
fetch('/user/login', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({ email, password }),
})
.then(response => response.json())
.then(data => {
  if (data.token) { // Assuming the response includes a 'token' prop
    // Store token in local storage
    localStorage.setItem('token', data.token);
    showSuccessMessage(data.message);
    setTimeout(() => {
      window.location.href = '/bookstore'; // Redirect to homepage
    }, 2000); // Redirect after 2 seconds
  } else {
    showErrorMessage(data.error);
  }
})
.catch(error => {
  console.error('Error logging in:', error);
  showErrorMessage('An error occurred. Please try again later.');
```

What about sending the request

```
// Perform form submission with token included
fetch("/bookstore/add", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    Authorization: `Bearer ${token}`,
  },
  body: JSON.stringify({
    title: document.getElementById("title").value,
    author: document.getElementById("author").value,
    pages: document.getElementById("pages").value,
    rating: document.getElementById("rating").value,
    genres: document.getElementById("genres").value,
    // Add other fields as needed
  }),
})
  .then((response) => {
    if (response.status === 201) {
      return response.json().then((data) => {
        alert(data.message);
        window.location.href = "/bookstore"; // Redirect to homepage
      });
    } else {
      throw new Error("Failed to add book");
      console.error("Error adding book:", response.error);
    }
  })
  .catch((error) => {
    console.error("Error adding book:", response.error);
    // Handle error scenario
  });
});
```

How do we get the information on the server?

- We can breakdown the header call
- Req.headers

```
const verifyToken = (req, res, next) => {  
  const token = req.header("Authorization")?.replace("Bearer ", ""); // Extract to  
  if (token == "null") {  
    return res.status(401).json({ error: "Not Logged in" });  
  }  
  if (!token) {  
    return res.status(401).json({ error: "Unauthorized" });  
  }  
  try {  
    const decoded = jwt.verify(token, process.env.ACCESS_TOKEN_SECRET); // Verify  
    req.user = decoded; // Attach user information to request object  
    next(); // Move to the next middleware  
  } catch (err) {  
    console.error(err);  
    res.status(401).json({ error: "Unauthorized" });  
  }  
};
```

Lets go through an example

- On a fresh browser
 - Go to home page
 - Go to add a book page
 - What is our result, how did we get here
-
- Lets limit access to the add book section for people that are logged in

Lets go through an example

- Lets limit access to the add book section for people that are logged in
 - The page can be restricted to anyone by checking if they sent over a token
 - When they login and get into the page, when they complete their work and submit a request
 - We check the header for the token
 - Verify the content
 - Save it into our DB