# The Goal

## - Security Best Practices in a Web Application

* <https://www.owasp.org/index.php/Authentication_Cheat_Sheet>
* <https://blog.risingstack.com/node-js-security-checklist/>
* <https://stackoverflow.com/questions/549/the-definitive-guide-to-form-based-website-authentication>
* <https://github.com/delight-im/PHP-Auth>

# Vulnerabilities and Threats

## - Cross Site Scripting (XSS)

* XSS attacks occur when an outside entity is able to execute code within your website or app. The most common attack vector here is if your website allows inputs that are not properly sanitized.

## - Cross Site Request Forgery (CSRF)

* <https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)>
* This is an attack that forces an end user to execute unwanted actions on a web application in which they're **currently authenticated**.
* In my words: CSRF happens when someone executes a script on a server where you are currently logged in, from your web client
  + So any persistent cookies, tokens or Remember me scenarios are susceptible to CSRF attacks
  + The longer the cookies/tokens/sessions last, the higher the risk
  + Both GET and POST requests are vulnerable
* What is at risk?
  + CSRF attacks are state-changing **requests**, not theft of data, since the attacker has no way to see the response to the forged **request**
* CSRF can happen just by opening a malicious website!!!
  + The user doesn’t even have to click on anything. Script can be executed in the <body onload=”maliciousScript()”> or as the src value of an embedded image.
    - **So do avoid opening suspicious emails and websites**
    - Simply not clicking on any links, images in websites is not enough any more
  + Most users are normally logged in to their Email, Bank site, Amazon or any other shopping site (with their credit card information)
  + If this user now opens another browser tab where a malicious script can be executed
  + This malicious code could randomly send requests to various known servers (Gmail, all the banks etc.)
  + If by chance, one of those requests was to a server where the user is currently logged in, in another browser tab,
  + The malicious script could request to let’s say delete emails, or change user’s password in the email server (that is why, asking for existing password to change password is important), or change the email and phone number, or make a transaction on the bank server.
  + And since the user is logged in, the request from the malicious code will be considered a request from the authenticated user, since its coming from the same browser.
* Protecting against an CSRF attack:
  + The server, upon establishing a session with a client will generate a unique token (note this is not a JWT).
  + For subsequent POST/GET requests to the server, a hidden input field in each html form will contain this token and the server will check to make sure the tokens match.

## - SQL Injection

## - Man in the Middle Attacks

## - Session Hijacking, Sniffing

# Client Side

## - Considerations

* How to log in
* How to log out
* How to remain logged in (session management)
* "Remember me" checkbox
* Managing cookies (including recommended settings)
* Require re-authentication for sensitive actions even if ‘remember me’ is on
* SSL/HTTPS end-to-end encryption
* How to store passwords
* Forgotten username/password functionality
* Use of ‘nonces’ to prevent cross-site request forgeries (CSRF)
* OpenID
* Ask for acceptance of Cookies (compliance with EU’s DGPR data protection laws)
* Browser auto-completion of usernames and passwords
* Show/Hide password
* Secret URLs (public URL protected by digest)
* Using secret questions
* Checking password strength
* E-mail validation

## - Login Process

* provide generic/local strategy as well as Social Media login
* Form Design (<https://uxdesign.cc/design-better-forms-96fadca0f49c>)
  + Use CSRF token field
  + Use terms 'Login' vs 'Sign In'
  + Ability to Show/Hide Password
* Form Input Validation

<https://github.com/monterail/vuelidate>

<https://baianat.github.io/vee-validate/guide/>

* + SQL Injection protection
  + XSS protection
  + CSRF protection (<https://www.acunetix.com/websitesecurity/csrf-attacks/>)

## - User Registration Process

* Using emails as username
* Password strength control
* Form Design
  + Use CSRF token field
  + Use terms 'Register' vs 'Sign up'
  + Ask the absolute minimum amount of information at Registration
    - Make them fill out the details later
* Form Input Validation
  + SQL Injection protection
  + XSS protection
  + Responsive Notifications when some input field is missed or wrong format, for example
    - telephone numbers
    - emails
    - dates
    - dollar amounts
* Form Input Validation using Vuelidate: <https://medium.com/js-dojo/simple-vue-js-form-validation-with-vuelidate-722331e5ab0d>

## - Remember Me Process

* <https://paragonie.com/blog/2015/04/secure-authentication-php-with-long-term-persistence>
* <https://stackoverflow.com/questions/1354999/keep-me-logged-in-the-best-approach>
* <https://fishbowl.pastiche.org/2004/01/19/persistent_login_cookie_best_practice>

#### Security issues with Remember-Me

* It creates a large window in which an attacker can "ride" on the session.
* This gives an attacker a very long time (Months?) in which he can deliver a CSRF attack.
* Even if you have CSRF protection an attacker can still ride on a session with XSS and XmlHttpRequest (HttpOnly Cookies will prevent a full hijack).
* "Remember Me" makes other threats like XSS, CSRF, sniffing more serious.
* As long as these vulnerabilities have been addressed, then you shouldn't have a problem with real world hackers.

#### Generic Process:

**Improved Persistent Login Cookie Best Practice**

1. When the user successfully logs in with Remember Me checked, a **login cookie is issued** in addition to the standard session management cookie.
2. The login cookie contains **a series identifier and a token (**UserId and a RememberMeToken). The series and token are **unguessable random numbers** from a suitably large space. Both are stored together in a database table, **the token is hashed** (sha256 is fine).

* The login token is Password Equivalent, so if an attacker got their hands on your database, they could use the tokens to log in to any account, just as if they were clear text login-password combinations.

1. When a non-logged-in user visits the site and presents a login cookie, the series identifier is **looked up in the database**.
   1. If the **series identifier** is present and the hash of the **token** matches the hash for that series identifier, the user is considered **authenticated**. A **new token** is generated, a new hash for the token is stored over the old record, and a new login cookie is issued to the user (it's okay to re-use the **series identifier**).
   2. If the series is present but the token does not match, a **theft** is assumed. The user receives a strongly worded warning and all of the user's remembered sessions are deleted.
   3. If the username and series are not present, the login cookie is **ignored**.
2. When a logged-in user returns, look them up by the RememberMe token and make sure the UserId matches.

This approach provides defense-in-depth. If someone manages to leak the database table, it does not give an attacker an open door for impersonating users.

#### Implementation details

* store the userId and rememberme token in a separate table in the database
* How to have two cookies in one session?

### - Password Recovery Process

### - Client Side Session Management

* **cookies**
* HTML5 Web Storage
  + **session storage**:
    - window.sessionStorage
    - stores data for one session
    - data is lost when the browser is closed
  + **local storage**:
    - window.localStorage
    - stores data with no expiration date
  + Before HTML5, application data had to be stored in cookies, included in every server request. Web storage is more secure, and large amounts of data can be stored locally, without affecting website performance.
  + Unlike cookies, the storage limit is far larger (at least 5MB) and information is never transferred to the server.
  + Web storage is per origin (per domain and protocol). All pages, from one origin, can store and access the same data.

### - Client Side Security

#### Considerations

* Do we salty hash the password??
  + <https://stackoverflow.com/questions/3715920/is-it-worth-hashing-passwords-on-the-client-side>
  + <https://security.stackexchange.com/questions/53594/why-is-client-side-hashing-of-a-password-so-uncommon>
* So from the above links, **basically use TLS/SSL**
  + HTTPS is HTTP over SSL/TSL
  + TLS is the SSL v3.1
* So using https in production is enough then?! seems like it
* We as programmers DO NOT have to code anything special to achieve HTTPS.
  + It’s just a server side step, to buy and install an SSL certificate

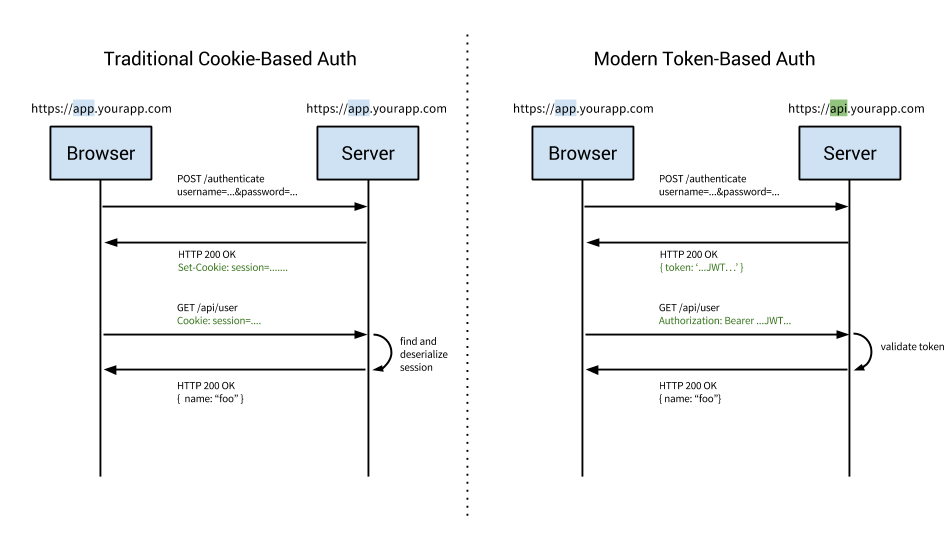
# Authentication vs Authorization

* <https://medium.com/tech-tajawal/microservice-authentication-and-authorization-solutions-e0e5e74b248a>
* <https://blog.bitsrc.io/understanding-json-web-token-authentication-a1febf0e15>
* <https://blog.sqreen.io/authentication-best-practices-vue/>
* <https://blog.jscrambler.com/vue-js-authentication-system-with-node-js-backend/?utm_medium=referral&utm_source=reddit.com>
* <https://dzone.com/articles/cookies-vs-tokens-the-definitive-guide>

There are two concepts to remember:

* **User Authentication**
  + A user needs to login and authenticated.
    - This can be done in a vanilla fashion (find username in database, then compare passwords).
    - Passport.js can be used to accomplish this
    - We need bcrypt to create/compare password hashes.
  + User login/authentication could be:
    - token-based (in which case we need the JWT package too) (Can it? I think tokens should be used for authorization and not authentication. Am I right?)
    - session based
* **Resource Authorization**
  + API endpoints need to be secured.
    - Essentially, each time an endpoint is requested, the user is checked to be authenticated.
  + This can be done using passport too!

## - Authentication Types



### **Session/Cookie-based Authentication**

* The Server does all the heavy lifting server-side.
* **It is state-full.**
  + Uses are tracked and this information is stored:
    - On the serverside in the session (in memory, or in some type of store, or a database)
    - On the clientside in a cookie
* Broadly speaking a client authenticates with its credentials and receives a session id (which can be stored in a cookie) and attaches this to every subsequent outgoing request.
* So this could be considered a "token" as it is the equivalent of a set of credentials.
* There is however nothing fancy about this session id-String.
* It is just an identifier and the server does everything else.
* It associates the identifier with a user account (e.g. in memory or in a database).
* It can restrict or limit this session to certain operations or a certain time period and can invalidate it if there are security concern and more importantly it can do and change all of this on the fly.
* Furthermore it can log the users every move on the website(s).
* Possible disadvantages are bad scale-ability (especially over more than one server farm) and extensive memory usage.

### **Token-based Authentication**

* No session is persisted server-side (**stateless**).
  + The server does not keep a record of which users are logged in or which JWTs have been issued.
  + How does the server know the token can be trusted?
    - Since the token is signed by the server, in the subsequent requests, the server can verify that the token indeed was granted by it
  + How does the server know if the client has not stolen the token?
    - Don’t know ….
* The initial steps are the same.
* Credentials are exchanged against a token which is then attached to every subsequent request (It can also be stored in a cookie).
* However for the purpose of decreasing memory usage, easy scale-ability and total flexibility (tokens can be exchanged with another client) a string with all the necessary information is issued (the token) which is checked after each request made by the client to the server.

#### Generic Token Based Authentication process:

* User Requests Access with Username / Password
* Application validates credentials
* Application provides a signed token to the client
* Client stores that token and sends it along with every request
  + The token is generally sent as an addition Authorization header in the form of Bearer {JWT},
  + but can additionally be sent in the body of a POST request
  + or even as a query parameter
* Server verifies token and responds with data
* Once a user logs out, the token is destroyed client-side, no interaction with the server is necessary.

#### Advantages of Token-Based Authentication

* Stateless: Conforms to REST API standards
* Scalability:
  + Doesn’t take up storage on the server, doesn’t need session/database lookup on every request
  + The server's only job is to sign tokens on a successful login request and verify that incoming tokens are valid.
  + In fact, the server does not even need to sign tokens. Third party services such as Auth0 can handle the issuing of tokens and then the server only needs to verify the validity of the token.
* Store Data in the JWT
  + Depending on your use case you may choose to make the minimal amount of claims such as the user id and expiration of the token,
  + Or you may decide to include additional claims such as the user's email address, who issued the token, scopes or permissions for the user, and more.
    - How is this secure?
* Performance:
  + Since additional data can be stored inside the JWT, such as the user's permission level, it saves additional lookup calls to get and process the requested data.
  + For example, say you had an API resource /api/orders that retrieves the latest orders placed via your app, but only users with the role of **admin** have access to view this data.
    - In a cookie based approach, once the request is made, you'd have one call to the database to verify that the session is valid, another to get the user data and verify that the user has the role of **admin**, and finally a third call to get the data.
    - On the other hand, with a JWT approach, you can store the user role in the JWT, so once the request is made and the JWT verified, you can make a single call to the database to retrieve the orders.
* Decoupled: Can be used with browsers, Mobile Apps, Microservices, or Rest APIs
* Cross Domain and CORS
  + Cookies work well with singular domains and sub-domains, but when it comes to managing cookies across different domains, it can get hairy.
  + In contrast, a token-based approach with CORS enabled makes it trivial to expose APIs to different services and domains.
  + Since the JWT is required and checked with each and every call to the back-end, as long as there is a valid token, requests can be processed.

#### Concerns with Token-Based Authentication

* Size of Token:
  + Cookies are tiny. There is a limit of 4 KB
  + Tokens can store any amount of data and, hence, could become very big.
* Where to Store Token on Client Side:
  + In Cookie: but cookie has a size limit
  + In Local Storage: private to the domain. Secure, but same token cannot be shared even by subdomains.
  + In session Storage: expires when the browser is closed
* Security of Token:
  + Tokens Are Signed, Not Encrypted
  + Can be easily decoded
  + Sensitive data, such as passwords, should never be stored in the payload
  + Should use JSON Web Encryption (JOSE is a nodejs library for this)
* XSS vulnerability:
  + Cross Site Scripting attacks happen when a malicious user can execute script on your site. Main cause is un-sanitized input fields (such as comments, reviews etc)
  + If script can be executed, the Tokens (and non HTTPOnly cookies) can be accessed
* CSRF vulnerability:
  + Cross Site Request Forgery
  + CSRF attacks are not an issue if you are using JWT with local storage.
  + If the Token is stored in a cookie, it is vulnerable to CSRF attack

#### Recommendations

* Keep the life of tokens short
* In the event of a severe breach: change the signing algorithm, which would invalidate all active tokens and require all of your users to log in again.

## - Client Side Issues

* To use JWT tokens in the browser you have to store it in either cookie, LocalStorage or SessionStorage, which can lead to XSS attacks.

#### Links

* <https://blog.sqreen.io/authentication-best-practices-vue/>
* <https://blog.jscrambler.com/vue-js-authentication-system-with-node-js-backend/?utm_medium=referral&utm_source=reddit.com>

## - Server Side Issues

* For supporting web application only, either cookies or tokens are fine
  + In case of cookies: Think about CSRF attacks
  + In case of JWT tokens: Take care of XSS attacks
* For supporting both a web application and a mobile client,
  + Go with an API that supports token-based authentication.
* For APIs that communicate with each other, go with request signing.

#### Links

* <https://stackoverflow.com/questions/549/the-definitive-guide-to-form-based-website-authentication>
* <https://blog.bitsrc.io/understanding-json-web-token-authentication-a1febf0e15>
* <https://hackernoon.com/your-node-js-authentication-tutorial-is-wrong-f1a3bf831a46>
* <https://blog.risingstack.com/node-hero-node-js-authentication-passport-js/>
* <https://medium.com/of-all-things-tech-progress/starting-with-authentication-a-tutorial-with-node-js-and-mongodb-25d524ca0359>
* <https://www.codementor.io/kgasta/nodejs-authentication-methods-e0c0i6k40>
* <https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication>

# Server Side

## - Considerations

* Authentication
  + Passwords
    - Hashing
    - Salting
  + Brute Force Protection
* Server side Session Management
  + <https://www.owasp.org/index.php/Top_10_2013-A2-Broken_Authentication_and_Session_Management>
  + Guest Sessions (not logged in)
  + User Sessions (logged in)
* Tokens
* Access permissions/User Roles
* endpoint Authorization/Protection
* re-authenticating users for sensitive actions such as changing passwords etc
  + To avoid CSRF attacks which would attempt to change a user’s password on a server where the user is logged in.

## - Express Middleware

Express is an un-opinionated, lightweight framework. Therefore, we need additional middleware to achieve certain functionalities. Sails.js is more opinionated and bundles lots of things.

* <https://expressjs.com/en/resources/middleware.html>
* <https://blog.jscrambler.com/setting-up-5-useful-middlewares-for-an-express-api/>

Important: the order of Middleware in the app.js/server.js is important!!

#### Body Parser

* <https://expressjs.com/en/resources/middleware/body-parser.html>
* <https://medium.com/@adamzerner/how-bodyparser-works-247897a93b90>

TLDR: It returns a function that acts as middleware. The function listens for req.on(‘data’) and constructs req.body from the chunks of data it gets.

* This module is needed to read HTTP POST requests in Express.js v4 and above.
* body-parser is an express middleware that reads a form's input and stores it as a JavaScript object accessible through 'req.body'.
* That is, You need to use bodyParser() if you want the form data to be available in req.body.
  + body-parser doesn’t handle multipart bodies.
  + We would need busboy, Multer or multiparty libraries for this
* Use of bodyParser.urlencoded({extended: true})
  + The "extended" syntax allows for rich/nested objects and arrays to be encoded into the URL-encoded format, allowing for a JSON-like experience with URL-encoded

#### Express Validator

* <https://express-validator.github.io/docs/>

Used for validation of form input such as email, dates, address fields.

#### Express Session

Used to persist user's session

- A user session can be stored in two main ways with cookies: on the server or on the client. (see express-session vs cookie-session comparison above)

#### Helmet

* <https://github.com/helmetjs/helmet>

Helps you secure your Express apps by setting various HTTP headers Configures the Content Security Policy;

* Removes the header X-Powered-By that informs the name and the version of a server;
* Configures rules for HTTP Public Key Pinning;
* Configures rules for HTTP Strict Transport Security;
* Treats the header X-Download-Options for Internet Explorer 8+;
* Disables the client-side caching;
* Prevents sniffing attacks on the client Mime Type;
* Prevents ClickJacking attacks;
* Protects against XSS (Cross-Site Scripting) attacks.

#### CORS

* <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>
* <https://expressjs.com/en/resources/middleware/cors.html>
* <https://www.html5rocks.com/en/tutorials/cors/>
* <https://www.moesif.com/blog/technical/cors/Authoritative-Guide-to-CORS-Cross-Origin-Resource-Sharing-for-REST-APIs/>
* Cross-Origin Resource Sharing (CORS) is a mechanism that uses additional HTTP headers
  + to tell a browser to let a web application running at one origin (domain) have permission
  + to access selected resources from a server at a different origin.
* For security reasons, browsers restrict cross-origin HTTP requests initiated from within scripts.
* CORS allow AJAX requests to skip the Same-origin policy and access resources from remote hosts
  + either by using the default options
  + or allowing specific origins only
  + The default configuration is the equivalent of:

{

"origin": "\*",

"methods": "GET,HEAD,PUT,PATCH,POST,DELETE",

"preflightContinue": false,

"optionsSuccessStatus": 204

}

* Since our vue/react clients are running on a different URL than the API server,
* If we do not use CORS, the ajax request from the client will meet an access control error,
* If we use CORS without any options, the default configurations as shown above will be applied.
  + This is not good for security, since anyone can request resources and could crash the server
  + Or a XSS attack could happen
* If we setup CORS correctly, only certain origins will be white listed.
  + Cross Site Scripting (XSS) attacks can be prevented that way
  + since the malicious code won’t have an origin which is allowed by the server
* CORS must enable cross-origin requests while still protecting the browser's same-origin policy
* CORS requires a preflight/pre-request handshake before the actual HTTP request
  + In the preflight request, the client asks for a type of connection and the server responds that it'll allow it
  + this pre-flight request is permitted via the Access-Control-Expose-Headers
  + Once the client knows it is allowed a certain header,
    - then it uses that header in the actual request
    - this is permitted via the Access-Control-Allow-Headers
* Access-Control-Expose-Headers
  + This header lets a server whitelist headers that browsers are allowed to access.
  + By default, only the following headers are exposed
    - Cache-Control
    - Content-Language
    - Content-Type
    - Expires
    - Last-Modified
    - Pragma
* Access-Control-Allow-Headers
  + Used in response to a preflight request to indicate which HTTP headers can be used when making the actual request.
  + to check the headers, open the Network tab in the Chrome DevTools, and review the 'XHR' details
    - Do I need to set both the 'Authorization' and 'Credentials' headers for the Session management to work?
      * Authorization header is needed to send the token in the request
      * Credentials header is needed to send a cookie with the request
      * the Credentials header has to be set on both server (in the CORS options) and client (in the Ajax/Axios/XHR requests) for it to work
        + But the question is: do I need it?

## - Middleware Comparison

### express-session vs cookie-session

* express-session is more abstract, it supports different session stores (like files, DB, cache and whatnot).
  + it should be used for larger and sensitive payloads
* cookie-session is a simple / lightweight cookie-based session implementation, where cookie is the only Storage engine supported.
  + That is, all the session info is stored on the client, in a cookie.
  + It should only be used for small amount of data, and when session data isn't sensitive.
  + In case of cookie-session, the server is entirely stateless
  + It is helpful in applications where no database is used in the back-end.
  + Cases where database is used, it can still be useful, for example,
    - To act like a cache to stop frequent database lookups which is expensive.
* Both middleware make use of client-side cookies to maintain a user's context i.e. Session.
* The difference lies in:
  + What gets stored in the cookies, and
  + Whether server-side store is needed

The table below compares cookie-session middleware and express- session middleware with respect to Sessions:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Client-side Store (cookie)** | | **Server-side Store (in memory, db etc)** | |
| **Middleware** | ***Used?*** | ***What is stored?*** | ***Used?*** | ***What is stored?*** |
| **cookie-session** | Yes | All session data | No | N/A |
| **Express-session** | Yes | Only Session Id | Yes | All Session data |

### express-session vs passport

* <https://www.airpair.com/express/posts/expressjs-and-passportjs-sessions-deep-dive>
* <https://www.npmjs.com/package/express-session>
* express-session creates a session to store user data in
  + if ‘cookie’ is specified in the options,
    - then the cookie is stored in express session
  + if ’store’ is specified in the options
    - then we need a persistent store such as a database
    - there is a list of supported stores on the npm link above
    - if connect-mongodb-session is used for store,
      * the session is stored in the mongodb in the ‘sessions’ document
  + option saveUninitialized = false
    - Is useful for implementing login sessions, reducing server storage usage, or complying with laws that require permission before setting a cookie.
    - Choosing false will also help with race conditions where a client makes multiple parallel requests without a session.
* If a client has never visited the site before
  + That means there is no session and cookie on the server
  + The client **request** has no cookie header
    - But the response has some problem (no data) in my case !!
  + The server **response** has a ‘set-cookie’ header
* If page is refreshed while a cookie is active, then there is a cookie in **request** only
  + If the cookie is not modified on the server, the response will not have a cookie (VERIFY)
* the session from mongodb is deleted after the time specified in the cookie’s maxAge property
  + if a client then access the site again after their existing cookie has expired
    - their **request** will have the header ‘cookie’ with the value of the expired cookie
    - a new cookie is created on the server for them
    - the **response** of that request has header ‘set-cookie’ with a fresh cookie
  + while a cookie is active, the **response** has the header ‘cookie’
* express session works even when no authentication/user login is in place
* express-session makes the following objects available to the ‘req’
  + req.session.cookie
  + req.session.user
  + req.session.passport.user (if passport is used)
* passport is needed to implement some type of authentication
  + req.user is made available from the passport session
  + passport's serialize and deserialize functions receives the req.session.passport.user
  + and the deserialize function creates the req.user from it
* what is req.cookie? is it the cookie sent from the client?
* do we need to store the user in the express session?
* how much of user should we store in the express session?
* do we save the whole user in the session or just some token?

### jsonwebtoken vs passport-jwt

jsonwebtoken:

* An implementation of JSON Web Tokens.
* This was developed against draft-ietf-oauth-json-web-token-08. It makes use of node-jws
* This needs 'jws' (jws or jwt?)

passport-jwt:

* A Passport strategy for authenticating with a JSON Web Token.
* This module lets you authenticate endpoints using a JSON web token.
* It is intended to be used to secure RESTful endpoints without sessions.
* the passport-jwt uses 'jsonwebtoken' as a dependency

### bcrypt vs bcryptjs

JWT

<http://cryto.net/~joepie91/blog/2016/06/13/stop-using-jwt-for-sessions/>

## **- Routes**

Every route file must have lines 1, 2 and 3

1. const express = require('express')

2. const router = express.Router()

// bunch of routes

3. module.exports = router

In case of fetch route in a client-side code. If you do not specify credentials: 'include' in fetch options, cookies are neither sent to server nor saved by a browser, although the server response sets cookies.

## - Security

### Links

* <https://expressjs.com/en/advanced/best-practice-security.html>
* <https://blog.risingstack.com/node-js-security-checklist/>

### In production

* The first step in securing any web application is using SSL.
  + That keeps your cookie confidential, prevents replay attacks, ensures the user is talking to the right server, prevents MitM, and prevents attackers from changing the data on the network
* Set the secure flag on the cookie, so it's only sent over SSL (HTTPS).
* Set the http-only flag, to prevent JavaScript from reading it.

To test production setup on localhost, generate a self-signed SSL certificate <https://blog.mgechev.com/2014/02/19/create-https-tls-ssl-application-with-express-nodejs/>

### Considerations

* Do we need CORS (whitelist origins) when using Tokens for authentication?
* Is token enough for authentication or
  + Is it just an identified for user information stored on the server in a session?
* tokens can be used to allow a third-party access to one's own data
  + e.g when we allow an app to access our Facebook
  + this will further need user roles so only selective data is allowed to third parties

### TODOs

* I should do different APIs with different scenarios for token expiration
  + e.g a bank's API should allow only short lived sessions/tokens
  + and email or social media is a long term session/token
* I should learn to force a session/token to expire pre-maturely
  + e.g in response to a suspicion of un-authorized activity

## - Server Side Session Management (using express-session)

* If you want two separate session objects, one for regular usage and one for admin usage with no overlap between them, then you have to do two separate app.use('/path1', session(...)) and app.use('/path2', session(...)) statements so you have two separate session managers for different paths and make sure each has a different cookie name (using the name parameter to the session() options). And, then you have to design your URLs to be sub-paths of those so they get the right path.
* Usually, people only use one session and then just keep a flag in the session whether it's admin login or not and you can check that flag when needed.
* It’s common practice to associate a session with an anonymous user, one who hasn’t authenticated with your application. However, when a user does successfully authenticate with your application, it is absolutely paramount that the authenticated user doesn’t continue to use the same session ID.

<https://dzone.com/articles/securing-nodejs-managing-sessions-in-expressjs>

### Cookies

Cookies are simply key/value pairs that let us get around HTTP being a stateless protocol. When an application wants some data to last for more than one connection they can use cookies to store that data on the client side.

Cookies can be used to store non-sensitive data such as preferences for themes, and other personalization.

Cookies are used to store user session data too. This information is sensitive and must be secured. But cookies are inherently unsecure and hence must never be used to store passwords or other sensitive information.

#### 4 ways to steal your cookies

1. By intercepting it over an unsecure line (packet sniffing / session hijacking)
2. By an XSS hack (or similar client-side exploit)
3. By directly accessing the user's browser (via either malware or physical access to the box)
4. By reading it from the server database (probably SQL Injection, but could be anything)

#### 2 ways to use cookies securely

1. To avoid 1 from above (Man in the Middle, Sniffing, Hijacking),
   1. Use HTTPs on the server
   2. Use the ‘secure’ (only possible on https) flag when the cookie is created
   3. Remember: the cookie will not be set and sent to the client if ‘secure: true’ and the protocol is not HTTPS
2. To avoid 2 from above (XSS attacks),
   1. Properly sanitize your data/input
   2. Use only a token or session id in the cookie on the client-side and store the actual sensitive user data on the server side only. These cookies are called session cookies.
   3. Use HTTPOnly flag in the cookie
      * Since the session ID is of no use to the client, there is absolutely no reason that the front-end application should ever have access to the session ID in the cookie.
      * Now, only the agent (i.e., browser) will have access to the cookie in order to resubmit it on the next request to the same domain. This will directly help mitigate cross-site scripting threats that could have otherwise accessed the contents of our session cookie.

Some very complicated article about cookies: <https://lcamtuf.blogspot.com/2010/10/http-cookies-or-how-not-to-design.html>

#### Secured Cookies

* <https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies>
* Don’t use the default session cookie name

app.use(session({

secret: 's3Cur3', // this should be coming from the .env file

name: 'sessionId' // this should be random name, and coming from the .env file

}))

* Set the following cookie options to enhance security:
  + **secure** - Ensures the browser only sends the cookie over HTTPS.
  + **httpOnly** - Ensures the cookie is sent only over HTTP(S), not client JavaScript,
    - Helping to protect against cross-site scripting (XSS) attacks.
    - This way they won't show up in document.cookies
    - To prevent cross-site scripting ([XSS](https://developer.mozilla.org/en-US/docs/Glossary/XSS)) attacks, HttpOnly cookies are inaccessible to JavaScript's [Document.cookie](https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie" \o "Get and set the cookies associated with the current document. For a general library see this simple cookie framework.) API; they are only sent to the server. For example, cookies that persist server-side sessions don't need to be available to JavaScript, and the HttpOnly flag should be set.
  + **domain** - indicates the domain of the cookie;
    - Use it to compare against the domain of the server in which the URL is being requested. If they match, then check the path attribute next.
  + **path** - indicates the path of the cookie;
    - Use it to compare against the request path.
    - If this and domain match, then send the cookie in the request.
  + **expires** - use to set expiration date for persistent cookies.

#### Setting multiple cookies

<http://www.connecto.io/blog/nodejs-express-how-to-set-multiple-cookies-in-the-same-response-object/>

REVIEW this

*\*\*\*\*\*\*\*\*\*\* this is someone's secured cookies not working due to reverse-proxy settings on the server*

*https://github.com/expressjs/session/issues/281#issuecomment-191283280*

*const expressSession = require('cookie-session')*

*var expiryDate = new Date(Date.now() + 7 \* 24 \* 60 \* 60 \* 1000); // 7 days*

*const session = expressSession({*

*secret: sessionSecret,*

*resave: false,*

*saveUninitialized: true,*

*cookie: {*

*secureProxy: true,*

*httpOnly: true,*

*domain: 'beintoo.net',*

*expires: expiryDate*

*}*

*})*

*app.use(session)*

*I just changed require('express-session') to require('cookie-session') and added secureProxy: true, everything worked out of the box.*

*Note also that both packages are maintained by expressjs so probably in my use case, I was lucky finding out that cookie-session fits my needs.*

*\*\*\*\*\*\* Someone's Answer: Note that this was caused by a misconfigured reverse proxy in front of express*

*(X-Forwarded-Proto was missing). Secure cookies are fully supported by express-session as well*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

# UI Frameworks

## Vue

* Vuetify (<https://vuetifyjs.com/en/>)
* Vue Material (<https://vuematerial.io/>)
* Bootstrap Vue (<https://bootstrap-vue.js.org/>)
* MD Bootstrap (<https://mdbootstrap.com/vue/>)
* Bulma (<https://bulma.io/>)

<https://dev.to/rhymes/what-vuejs-framework-should-i-use-4nk1>

## React