# 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>

## Client Side

### - Considerations

* How to log in
* How to log out
* How to remain logged in (session management)
* Managing cookies (including recommended settings)
* 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
* "Remember me" checkbox
* 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
  + Use CSRF token field
  + Use 'Login' vs 'Sign In'
  + Ability to Show/Hide Password
* Form Input Validation
  + SQL Injection protection
  + XSS protection

### - User Registration Process

* Using emails as username
* Password strength control
* Form Design
  + Use CSRF token field
  + Use '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>

### - Password Recovery Process

### - Client Side Session Management

* cookies
* session storage
* local storage

### - 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>

There are two concepts to remember:

* **User Authentication**
  + A user needs to login and authenticated.
    - This is done in a vanilla fashion (find username in database, then compare passwords).
    - We need bcrypt to create/compare password hashes.
  + User login/authentication could be:
    - token-based (in which case we need the JWT package too)
    - session based
* **Resource Authorization**
  + API endpoints need to be secured. This is done with the help of the middleware 'passport'. Essentially, each time an endpoint is requested, the user is checked to be authenticated.
  + This can be done using passport too!

### - Client Side Issues

* To use JWT tokens in the browser you have to store it in either 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>

#### 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
* Server verifies token and responds with data

REVIEW THIS

*Explain the middleware jwt, bcrypt, passport, passport-jwt and so on*

*/\*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'*

*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.*

*HURRAH!! the passport-jwt uses 'jsonwebtoken'*

*\*/*

*/\*bcrypt vs bcryptjs\*/*

JWT

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

## Server Side

### - Considerations

* Authentication
  + Hashing and salting the passwords
  + Brute Force Protection
* Server side Session Management
* Tokens
* Access permissions/User Roles/endpoint Authorization
* Passwords
  + Hashing
  + Salting

### - 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/>

#### 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 |

#### 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 headersConfigures 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 the 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?

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

### - Security

#### Links

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

#### Considerations

* Do we need CORS (whitelist origins) when using Tokens for authentication?
* Is a 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

#### Secured 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
  + **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.

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 Anser: 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*

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