**Use SSL/TLS for communication**

It is always a good practice to send your data over HTTPS rather than HTTP and it is imperative if your app transmits sensitive data.  
Encrypting data transmitted between the client and server helps mitigate several attacks like man in the middle attack, packet sniffing,  
eavesdropping etc. Let’s see how to set up TLS/SSL in Express 4.x:

Lets first generate a self-signed certificate:

$ openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 365

This generates a self-signed certificate valid for 365 days.

Next, enable HTTPS on Express. Additionally, redirect all HTTP traffic to HTTPS:

const fs = require('fs');  
const https = require('https');  
const express = require('express');const NODE\_ENV = process.env.NODE\_ENV || 'development';  
const PORT = process.env.PORT || 3443;  
   
const app = express();https.createServer({  
 key: fs.readFileSync('/path/to/key.pem'),  
 cert: fs.readFileSync('/path/to/cert.pem')  
}, app).listen(PORT);// Redirect http requests to use https in production  
if (NODE\_ENV === 'production') {  
 app.use((req, res, next) => {  
 if (req.header('x-forwarded-proto') !== 'https') {  
 res.redirect(`https://${req.header('host')}${req.url}`);  
 } else {  
 next();  
 }  
 });  
}

**Content-Security-Policy:** Prevents a range of injection attacks including Cross Site Scripting(XSS) attack.

Syntax:  
Content-Security-Policy: policy

For a detailed explanation of CSP, go through [this](https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP) link.

To set these headers in NodeJS, use the [helmet](https://helmetjs.github.io/) npm package:

const express = require('express');  
const helmet = require('helmet');  
   
const app = express();  
   
**app.use(helmet())**

**Preventing CSRF attacks**

Cross site request forgery, also known as XSRF, Sea Surf or Session Riding, is an attack vector that tricks a web browser into executing an unwanted action in an application to which a user is logged in. CSRF attacks specially targets state-changing requests and can force the victim to transfer funds, change email/password and so on.

CSRFs are typically conducted using social engineering, such as an email or link that tricks the victim into sending a request to a server on behalf of the attacker. The server has no way to distinguish a forged request from a genuine one.

In NodeJS, to prevent CSRF attack, we usually use the [csurf](https://github.com/expressjs/csurf) express middleware:

const cookieParser = require('cookie-parser');  
const csrf = require('csurf');  
const bodyParser = require('body-parser');  
const express = require('express');const csrfProtection = csrf({ cookie: true });  
const parseForm = bodyParser.urlencoded({ extended: false });  
   
// create express app   
const app = express();  
   
// we need this because "cookie" is true in csrfProtection   
app.use(cookieParser());  
   
app.get('/form', csrfProtection, (req, res) => {  
 // pass the csrfToken to the view   
 res.render('send', { csrfToken: req.csrfToken() });  
});

In the view use the CSRF token passed:

<form action="/process" method="POST">  
 <input type="hidden" name="csrf\_token" value="{{csrfToken}}">  
   
 Enter amount: <input type="number" name="amount">  
 <button type="submit">Submit</button>  
</form>

**Preventing XSS attacks**

The thumb rule to prevent this category of attack is to always validate and sanitize user data before processing or storing in database. Never trust data coming from user.  
Validation must be done on the server-side as client-side validation can be easily bypassed using tools such as Burp Suite, TamperData etc.

A common approach to validate and sanitize user data is to use a library like [validator.js](https://github.com/chriso/validator.js).

Example: To validate an email

import validator from 'validator';if(validator.isEmail('[foo@bar.com](mailto:foo@bar.com)')) {  
 // Process email or store in DB  
}