# MEAN Stack

## What is MEAN Stack?

MEAN is a user-friendly full-stack JavaScript framework ideal for building dynamic websites and applications. The origin of this word is the abbreviation series of the following words:

**M**ongoDB, **E**xpress, **A**ngular and **N**ode.js

**Technologies Covered:**

A screenshot of a cell phone

Description automatically generated

One of the main benefits of the MEAN stack is that a single language, JavaScript, runs on every level of the application, making it an efficient and modern approach to web development.

MEAN variants can be mentioned as MERN, MEVN, ... (using React, VueJS instead of Angular).

## MEAN workflow

A picture containing device

Description automatically generated

* In the MEAN stack, **MongoDB** stores the application’s data. Because both the application and the database use JavaScript, there’s no need to translate the object as it journeys from the application to the database and back.
* **Express** is a web application framework for Node.js. It balances ease of use and a full feature set.
* **AngularJS**—Google’s JavaScript frontend framework—The MEAN stack includes AngularJS to help developers build the user-facing side of the application. Because the backend, frontend and database are all built on JavaScript, there’s a smooth flow of information between all parts of your application.
* **Node.js** is the backbone of the MEAN stack. Express is purpose-built to work on top of Node.js, and AngularJS connects seamlessly to Node.js for fast data serving. Node.js comes complete with an integrated web server, making it easy to deploy your MongoDB database and application to the cloud.

# Angular

A picture containing drawing, clock, sign

Description automatically generated

## What is Angular?

* AngularJS is a JavaScript-based open-source front-end web framework mainly maintained by Google and by a community of individuals and corporations to address many of the challenges encountered in developing single-page applications. It aims to simplify both the development and the testing of such applications by providing a framework for client-side model–view–controller (MVC) and model–view–viewmodel (MVVM) architectures, along with components commonly used in rich Internet applications.
* Currently, AngularJS has been killed by Google. Instead, Google announced a framework called Angular with the same architecture but using the Typescrypt language.

## Typescript

* TypeScript is an open-source programming language developed and maintained by Microsoft. It is a strict syntactical superset of JavaScript and adds optional static typing to the language.
* It can be considered an enhanced version of Javascript by adding static type options and object-oriented classes that are not available in Javascript. TypeScript can be used to develop client-side applications (Angular2) and server-side (NodeJS).
* Actually, TypeScript was not first, there were some libraries like CoffeScript, and Dart developed by Google, but the weakness is that these two libraries use completely new syntax, which is completely different. with TypeScript, so though it was born later, TypeScript is still well received by developers.

## Advantages of TypeScript

* Easy to develop large projects: With the use of the latest techniques and object-oriented programming, TypeScript helps us develop large projects easily.
* Support for the latest Javascript features: TypeScript ensures full use of the latest Javascript techniques, such as the current version is ECMAScript 2020 (ES2020).
* Open source: TypeScript is open source so you can use it for free, and it is supported by the community.
* TypeScript is Javscript: The nature of TypeScript is that the compiler generates javascript code, so you can run anywhere as long as it supports JavaScript compilation. Alternatively, you can mix JavaScript syntax into TypeScript, which makes it easier for developers to access TypeScript.

# Node.js – Express

A drawing of a face

Description automatically generated

## What is Node.js and Express?

* Nodejs is an independent development platform built on Chrome's JavaScript Runtime that we can build network applications quickly and easily.
* Nodejs was built and developed in 2009, sponsored by Joyent, based in California, USA.
* The lower part of the Nodejs Core is written mostly in C ++, so the processing speed and performance are quite high.
* Nodejs creates applications with fast processing speed, realtime, and real time.
* Nodejs applies to products that have a lot of traffic, need to expand quickly, need to innovate in technology, or create Startup projects as quickly as possible.
* Express.js, or simply Express, is a web application framework for Node.js, released as free and open-source software under the MIT License. It is designed for building web applications and APIs. It has been called the de facto standard server framework for Node.js. It provides powerful features for web or mobile development.
* Expressjs supports HTTP and midleware methods, making the API extremely powerful and easy to use.

## Why I choose using Node.js and Express?

* Nodejs applications are written in javascript, which is a fairly common language. According to the author of the Javascript language, Ryan Dahl: “Javascript has features that make it very different from the rest of the dynamic programming languages, namely that it has no concept of multithreading, all of which are single threaded, and event driven.”
* Nodejs runs cross-server-side, using Event-driven event-driven architecture, non-blocking I / O mechanism to make it lightweight and efficient.
* Can run the Nodejs application anywhere on Mac - Window - Linux, moreover, the Nodejs community is very large and completely free. You can see how big the Nodejs community is here, the packages are completely free: <https://www.npmjs.com/>
* NodeJS applications respond well in real time and run cross-platform, multi-device.

## Express midleware

* Let’s say we had the following code:

const express = require("express");

var app = express();

// Custom middleware

function myMiddleware1(req, res, next) {

req.newProperty = "my custom property";

next();

}

// Another custom middleware

function myMiddleware2(req, res, next) {

req.newProperty = "updated value";

next();

}

app.get("/", (req, res, next) => {

res.send(`<h1>Custom Property Value: ${req.newProperty}`);

});

// Server listens on http://localhost:3000

app.listen(3000);

* As you can see, this is an extremely simple Express application that defines two middlewares and has a single route that you can visit in your browser at http://localhost:3000. If you started this application and visited that route, it would say “Custom Property Value: undefined” because defining middleware functions alone is not enough.
* We need to tell the Express application to actually use these middlewares. We can do this in a few ways. First, we can do it within a route.

app.get("/", myMiddleware1, (req, res, next) => {

res.send(`<h1>Custom Property Value: ${req.newProperty}`);

});

* If you add the first middleware function as an argument to the route, you will now see “Custom Property Value: my custom property” show up in the browser. What really happened here:
* The application was initialized
* A user visited http://localhost:3000/ in the browser, which triggered the app.get() function.
* The Express application first checked to see if there was any “global” middleware installed on the router, but it didn’t find any.
* The Express application looked at the app.get() function and noticed that there was a middleware function installed before the callback. The application ran the middleware and passed the middleware the req object, res object, and the next() callback.
* The myMiddleware1 middleware first set req.newProperty, and then called next(), which tells the Express application “Go to the next middleware”. If the middleware did not call next(), the browser would get “stuck” and not return anything.
* The Express app did not see any more middleware, so it continued with the request and sent the result.

# MongoDB

A picture containing drawing, food

Description automatically generated

## NoSQL

* NoSQL is an open source database and is abbreviated by: None-Relational SQL or there is often called Not-Only SQL.
* NoSQL is developed on Javascript Framework with JSON data type and key and value data type.
* NoSQL was born as a patch for the shortcomings and shortcomings and limitations of RDBMS (Relational Database Management System) in terms of speed, features, Ability of extension, ...
* With NoSQL you can extend data without worrying about things like creating foreign keys, primary keys, checking constraints, etc.
* NoSQL ignores the integrity of data and transactions in exchange for fast performance and scalability.
* NoSQL is used in many companies and large corporations, for example, FaceBook uses Cassandra developed by FaceBook, Google developed and used BigTable, ...

## MongoDB

* MongoDB is an open source database management system, is a NoSql database and is used by millions of people.
* MongoDB is a document-oriented database, the data is stored in JSON-style document instead of tables like relational database so the query will be very fast.
* With relational databases we have the concept of tables, relational databases (such as MySQL or SQL Server ...) use tables to store data, with MongoDB we will use the concept of collection instead of tables.
* Compared to RDBMS, in MongoDB collection corresponds to table, while document will correspond to row, MongoDB will use documents instead of row in RDBMS.
* Collections in MongoDB are structured very flexibly, allowing archived data not to follow a certain structure.
* Related information is stored together for quick query access through the MongoDB query language

# Socket.io

A picture containing drawing

Description automatically generated

## What is socket.io?

* Building a realtime application requires the use of socketio. Socketio will help parties in different locations connect with each other, transfer data instantly via intermediary server. Socketio can be used in many applications such as chatting, online games, updating the results of an ongoing match, ...
* Socketio is not a language, but just a tool to help make realtime applications. Therefore, socketio cannot be used to completely replace a language, but must be used in combination with another language. That language can be php, asp.net, nodejs, ...

## Why socket.io?

* WebSocket doesn't work ideally in the presence of Firewall and Antivirus.
* WebSocket doesn't work ideally in the presence of Proxy and Load Balancer either.
* It handles graceful degradation for you to numerous technical alternatives to get bi-directional near-time communication flowing (web sockets, ajax long polling, flash, etc)
  + As of March 2013, that site lists web sockets at 61% support. This is not "practically full".
* It handles browser inconsistencies and varying support levels for you
  + (these first 2 things are basically the same value created by jQuery, to put it in perspective)
* It includes additional features beyond bare bones web sockets such as room support for basic publish/subscribe infrastructure and things like automatic reconnect
* AFAIK it is more popular and easier to get help with than vanilla web sockets, at least at the moment.

## Why I need socket.io for this project?

* Because I need a realtime update from Back-end to Front-end and data exchange between Admin side and Client side.

# Passport in Node.js

## What is Passport?

* Passport.js one of the most popular Nodejs modules that supports your authentication. It is designed to be a very flexible middleware that gives you a high degree of customization with a variety of authentication scenarios: you can use Twitter, Facebook, Google and even a username-password in the database.
* You can also customize exactly which routes require authentication.

## Authentication Choices

A screenshot of a cell phone

Description automatically generated

Above is a high-level overview of the main authentication choices available to developers today. Here is a quick overview of each:

* Session-Based Authentication — Utilizes browser Cookies along with backend “Sessions” to manage logged-in and logged-out users.
* JWT Authentication — A stateless authentication method where a JSON Web token (JWT) is stored in the browser (usually localStorage). This JWT has assertions about a user and can only be decoded using a secret that is stored on the server.
* OAuth and OpenID Connect Authentication — A modern authentication method where an application uses “claims” generated from other applications to authenticate its own users. In other words, this is federated authentication where an existing service (like Google) handles the authentication and storage of users while your application leverages this flow to authenticate users.

Because I need to create a local authentication, so I choose Session-Based Authentication and use `passport-local` Passport JS Strategy.

## Passport passport-local strategy.

To understand the basic tenets of session-based-authentication, we need a few concepts:

* Basic HTTP Header Protocol
* What a cookie is
* What a session is
* How the session (server) and cookie (browser) interact to authenticate a user

## How Passport JS Local Strategy works

* There is one last thing that we need to learn in order to fully understand Session-Based Authentication–Passport JS.
* Passport JS has over 500 authentication “Strategies” that can be used within a Node/Express app. Many of these strategies are highly specific (i.e. passport-amazon allows you to authenticate into your app via Amazon credentials), but they all work similar within your Express app.
* In my opinion, the Passport module could use some work in the department of documentation. Not only does Passport consist of two modules (Passport base + Specific Strategy), but it is also a middleware, which as we saw is a bit confusing in its own right. To add to the confusion, the strategy that we are going to walk through (passport-local) is a middleware that modifies an object created by another middleware (express-session). Since the Passport documentation has little to say around how this all works, I will attempt to explain it to the best of my ability in this post.
* Let’s first walk through the setup of the module.
* We will need to install Passport and a Strategy to your project.

npm install --save passport passport-local

* Once we have done that, we will need to implement Passport within your application. In the passportExample.js file, I have added all the pieces we need for the passport-local strategy. I have removed comments to simplify.
* In the code above, I have two helper functions that will assist in creating and validating a password.

function validPassword(password, hash, salt)

\* @param {\*} password - The plain text password

\* @param {\*} hash - The hash stored in the database

\* @param {\*} salt - The salt stored in the database

\* This function uses the crypto library to decrypt the hash using the salt and then compares

\* the decrypted hash/salt with the password that the user provided at login

function genPassword(password)

\* @param {\*} password - The password string that the user inputs to the password field in the register form

\* This function takes a plain text password and creates a salt and hash out of it. Instead of storing the plaintext

\* password in the database, the salt and hash are stored for security

* In addition to the comments, I’ll note that these functions require the NodeJS built-in crypto library. Some would argue a better crypto library, but unless your application requires a high degree of security, this library is plenty sufficient!
* Next up, let’s take a look at the passport.use() method.

passport.use(

new LocalStrategy(function (username, password, cb) {

User.findOne({ username: username })

.then((user) => {

if (!user) {

return cb(null, false);

}

// Function defined at bottom of app.js

const isValid = validPassword(password, user.hash, user.salt);

if (isValid) {

return cb(null, user);

} else {

return cb(null, false);

}

})

.catch((err) => {

cb(err);

});

})

);

* First, I’ll mention that with all Passport JS authentication strategies (not just the local strategy we are using), we will need to supply it with a callback that will be executed when we call the passport.authenticate() method. For example, we might have a login route in your app:

app.post('/login', passport.authenticate('local', { failureRedirect: '/login' }), (err, req, res, next) => {

if (err) next(err);

console.log('You are logged in!');

});

* Our user will type in their username and password via a login form, which will create an HTTP POST request to the /login route. Let’s say our post request contained the following data:

{

"email": "sample@email.com",

"pw": "sample password"

}

* This WILL NOT WORK. The reason? Because the passport.use() method expects your POST request to have the following fields:

{

"username": "sample@email.com",

"password": "sample password"

}

* It looks for username and password field. If we wanted the first json request body to work, we need to supply the passport.use() function with field definitions:

passport.use(

{

usernameField: 'email',

passwordField: 'pw'

},

function (email, password, callback) {

// Implement our callback function here

}

);

* By defining the usernameField and passwordField, we can specify a custom POST request body object.
* ...

## Conceptual Overview of Session Based Authentication

Here is a basic flow of our app:

1. Express app starts and listens on http://www.expressapp.com (just assume this is true for the sake of the example).
2. A user visits http://www.expressapp.com/login in the browser
3. The express-session middleware realizes that there is a user connecting to the Express server. It checks the Cookie HTTP header on the req object. Since this user is visiting for the first time, there is no value in the Cookie header. Because there is no Cookie value, the Express server returns the /login HTML and calls the Set-Cookie HTTP header. The Set-Cookie value is the cookie string generated by express-session middleware according to the options set by the developer (assume in this case the maxAge value is 10 days).
4. The user realizes that he doesn’t want to login right now, but instead, wants to go for a walk. He closes his browser.
5. The user returns from his walk, opens the browser, and returns to http://www.expressapp.com/login again.
6. Again, the express-session middleware runs on the GET request, checks the Cookie HTTP header, but this time, finds a value! This is because the user had previously created a session earlier that day. Since the maxAge option was set to 10 days on the express-session middleware, closing the browser does not destroy the cookie.
7. The express-session middleware now takes the connect.sid value from the Cookie HTTP header, looks it up in the MongoStore (fancy way to say that it looks up the id in the database in the sessions collection), and finds it. Since the session exists, the express-session middleware does not do anything, and both the Cookie HTTP header value and the MongoStore database entry in the sessions collection stays the same.
8. Now, the user types in his username and password and presses the “Login” button.
9. By pressing the “Login” button, the user sends a POST request to the /login route, which uses the passport.authenticate() middleware.
10. On every request so far, the passport.initialize() and passport.session() middlewares have been running. On each request, these middlewares are checking the req.session object (created by the express-session middleware) for a property called passport.user (i.e. req.session.passport.user). Since the passport.authenticate() method had not been called yet, the req.session object did not have a passport property. Now that the passport.authenticate() method has been called via the POST request to /login, Passport will execute our user-defined authentication callback using the username and password our user typed in and submitted.
11. We will assume that the user was already registered in the database and typed in the correct credentials. The Passport callback validates the user successfully.
12. The passport.authenticate() method now returns the user object that was validated. In addition, it attaches the req.session.passport property to the req.session object, serializes the user via passport.serializeUser(), and attaches the serialized user (i.e. the ID of the user) to the req.session.passport.user property. Finally, it attaches the full user object to req.user.
13. The user turns off his computer and goes for another walk because our application is boring.
14. The user turns on his computer the next day and visits a protected route on our application.
15. The express-session middleware checks the Cookie HTTP header on req, finds the session from yesterday (still valid since our maxAge was set to 10 days), looks it up in MongoStore, finds it, and does nothing to the Cookie since the session is still valid. The middleware re-initializes the req.session object and sets to the value returned from MongoStore.
16. The passport.initialize() middleware checks the req.session.passport property and sees that there is still a user value there. The passport.session() middleware uses the user property found on req.session.passport.user to re-initialize the req.user object to equal the user attached to the session via the passport.deserializeUser() function.
17. The protected route looks to see if req.session.passport.user exists. Since the Passport middleware just re-initialized it, it does, and the protected route allows the user access.
18. The user leaves his computer for 2 months.
19. The user comes back and visits the same protected route (hint: the session has expired!)
20. The express-session middleware runs, realizes that the value of the Cookie HTTP header has an expired cookie value, and replaces the Cookie value with a new Session via the Set-Cookie HTTP header attached to the res object.
21. The passport.initialize() and passport.session() middlewares run, but this time, since express-session middleware had to create a new session, there is no longer a req.session.passport object!
22. Since the user did not log in and is trying to access a protected route, the route will check if req.session.passport.user exists. Since it doesn’t, access is denied!
23. Once the user logs in again and triggers the passport.authenticate() middleware, the req.session.passport object will be re-established, and the user will again be able to visit protected routes.

# Openlayers

A picture containing drawing

Description automatically generated

## What is Openlayers?

* OpenLayers is a pure JavaScript library for displaying map data in most modern web browsers, with no server-side dependencies. OpenLayers implements a JavaScript API for building rich web-based geographic applications, similar to the Google Maps and MSN Virtual Earth APIs, with one important difference – OpenLayers is Free Software, developed for and by the Open Source software community.
* It can display map tiles, vector data and markers loaded from any source. OpenLayers has been developed to further the use of geographic information of all kinds.
* OpenLayers is capable of dealing with most projections. If we do not explicitly set one, our map is going to use their default which is the Web Mercator projection (EPSG:3857). The same projection is used e.g. for the maps of the OpenStreetMap-project and commercial products such as Bing Maps or Google Maps.

## Why Openlayers?

* It’s free to use and distributed under the BSD 2-Clause license.
* There are about 240 contributors. Consequently, releases every 3-4 months.
* The documentation contains QuickStart’s, tutorials and a lot of examples.
* The API documentation is well-structured.
* The power and flexibility are the two most strong characteristics of OpenLayers. The library has all the required features in the core functionality.
* OpenLayers supports GeoJSON, GeoRSS, KML, GML, and map data from any source using OGC-standards as WMS or WFS.
* Using projections, different Geo data sources, developing integration with other GIS applications? Then OpenLayers will be the right choice!

# Another technology

## Bcrypt

* Bcrypt is a password-hashing function designed by Niels Provos and David Mazières, based on the Blowfish cipher and presented at USENIX in 1999.

## Mongoose

* Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box.

## Cors

* Cross-origin resource sharing is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served. A web page may freely embed cross-origin images, stylesheets, scripts, iframes, and videos.

## Bootstrap

* Bootstrap is a free and open-source CSS framework directed at responsive, mobile-first front-end web development. It contains CSS- and JavaScript-based design templates for typography, forms, buttons, navigation, and other interface components.