ExpressJs

NodeJS web application framework

what is expressJS

- Minimalist web application framework / Express is a Web Framework used to configure Node's HTTP module.
- Runs on NodeJS

Note: Express JS is a minimalist NodeJS web application framework. It has robust set of features for web and mobile application development.

what is expressJS

- Express is a Web Framework used to configure Node's HTTP module.
- Fast
- Un-opinionated
- Minimalist
- Express is distributed via an NPM package.
- Express should be installed locally to each project, not globally

To install: npm install express –save

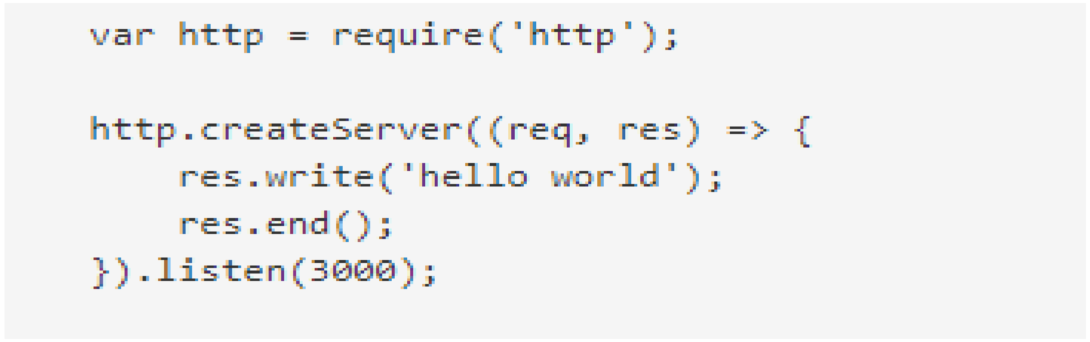
For package.json => npm init --y

NodeJS

- Javascript runtime
- Built on Chrome's V8 javascript engine.
- Event driven, non-blocking IO model
- Large package eco-system, npm!

Before we jump into ExpressJS, let's take a look at what NodeJS is quickly, this will help us understand why expressJS helps. NodeJS is a javascript runtime built on chrome's V8 javascript engine. It uses an event driven and non-blocking IO model which makes it lightweight and efficient. It also means that most of the operations are async and we'll be dealing with callbacks often. It has a large package eco-system, npm. So, most libraries you'll need for building a web app are built by someone already.

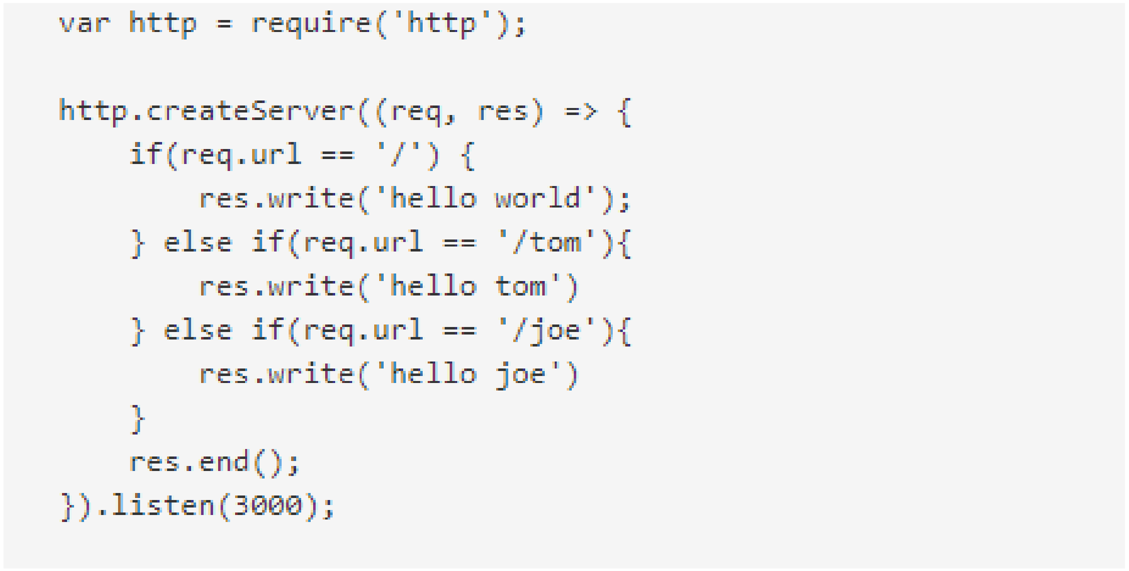
Starting Server in Node.JS



```
var http = require('http');

http.createServer((req, res) => {
    res.write('hello world');
    res.end();
}).listen(3000);
```

NodeJS API provides a built-in http module, which let's you create a server. The createServer takes in a callback, which gets 2 parameters request and response. as the name suggests, request gives you info about the request and response helps us write a response, in most web apps to the browser. And here we listen to port 3000.

Starting Server in Node.JS



```
var http = require('http');

http.createServer((req, res) => {
    if(req.url == '/') {
        res.write('hello world');
    } else if(req.url == '/tom'){
        res.write('hello tom')
    } else if(req.url == '/joe'){
        res.write('hello joe')
    }
    res.end();
}).listen(3000);
```

In this example, web app has multiple end points or pages. So,when we need more URLs, we tend to add a logic inside the callback and that makes the code a bit ugly and makes it un-maintainable. We need an abstracted layer which let's us build web apps in a more manageable way.

Why do we need?

- Express provides an abstraction here. instead of passing your method, you can pass the express instance to the http createServer method, and express will handle your request from there.
- Express uses concept of middlewares to control the flow of the request and app.use will add middlewares to the app's request flow

```
var http = require('http'),
    express = require('express');
var app = express();

app.use('/tom', (req, res, next) => {
    res.write('hello tom');
    res.end();
});

app.use((req, res, next) => {
    res.write('hello world');
    res.end();
});

http.createServer(app).listen(3000);
```

Middlewares

- we need support for cookies
- we need to be able to parse data sent in POST
- we need user authentication
- we need access to databases

and more...

to build a typical web app, we need more features. like we need to parse cookies, parse POST data handle authentication and have access to databases. this is where the middleware magic kicks in.

ExpressJS – Everything is a Middleware

- Provides support for routing and serving static files.
- Almost all other functionality is provided through third-party middleware modules

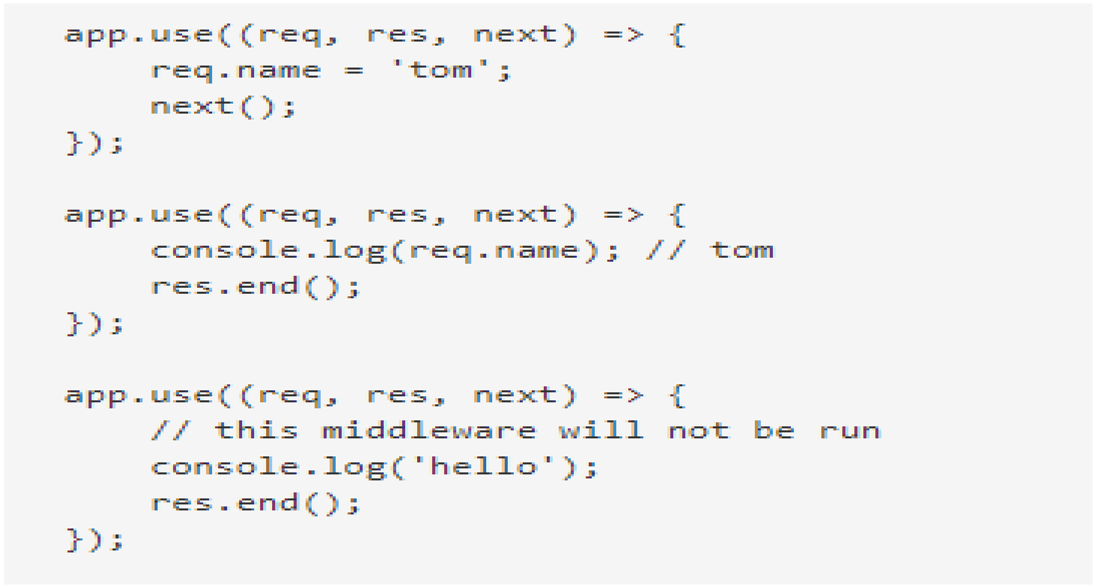
✓ Processing request body – Body Parser
✓ Working with cookies – Cookie Parser
✓ Using session – Express Sessions
✓ Handling Authentication - Passport

Middlewares

- Middleware is a function, that alters the flow of the request by modifying the request / response variables.
- it gets 3 parameters, the request, the response and next.
- request will have all the request info like the url, headers etc.
- response parameter will let you write / output your response.
- the third parameter next, calling next will execute the next middleware in the stack.
- calling app.use will let you add middleware to the request stack. so whenever your app get a request, all the middlewares that match the request will get executed.

Middlewares

If you look at the example, we have 3 middlewares added to the flow, the first one adds a key called 'name' and set's it a value 'tom'. since it calls next(). the next middleware in the stack will get executed. in this case, the second middleware prints the name set by the first middleware and it ends the request. since the request end in the second middleware and the it doesn't call next, the third middleware does not get called.



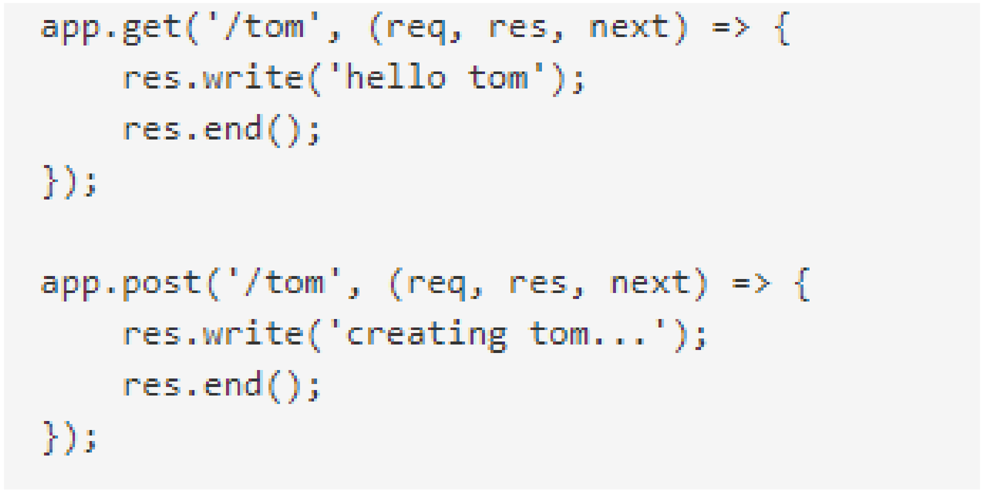
```
app.use((req, res, next) => {
    req.name = 'tom';
    next();
});

app.use((req, res, next) => {
    console.log(req.name); // tom
    res.end();
});

app.use((req, res, next) => {
    // this middleware will not be run
    console.log('hello');
    res.end();
});
```

Routing Requests

- Requests can be routed based upon the following factors
- HTTP Method – GET, POST, PUT, DELETE and many more
- URL Path – Path of the request URL can be matched against patterns.
- Routes are match based upon the order they are registered
- Each match is handled by a function which is passed a request and a response object allowing the handler to access information from the request and contribute to the generation of the response.
- Each handler can pass the request along via the next function

Routing Middleware

- In most web apps, you would want to differentiate between the type of Request, for a typical website you'd want to handle a GET and a POST differently. and for a RESTful service, you'd want to support PUT and DELETE and sometimes pre-flight HEADs too. Express routing supports this. So instead of using app.use, we could use app.get and app.post

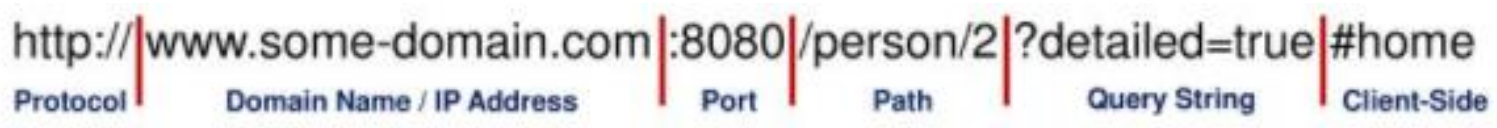


```
app.get('/tom', (req, res, next) => {
    res.write('hello tom');
    res.end();
});

app.post('/tom', (req, res, next) => {
    res.write('creating tom...');
    res.end();
});
```

Routing Data

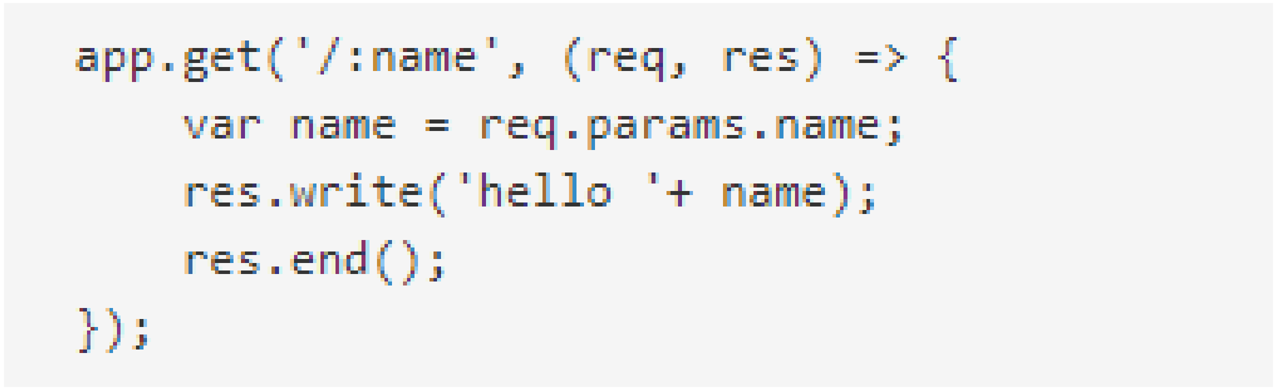
Two kinds of data are available in the URL

- URL Parameters – exposed as **params** property of the request object
- Query String Parameters – exposed as the **query** property on the request object



Routing Data

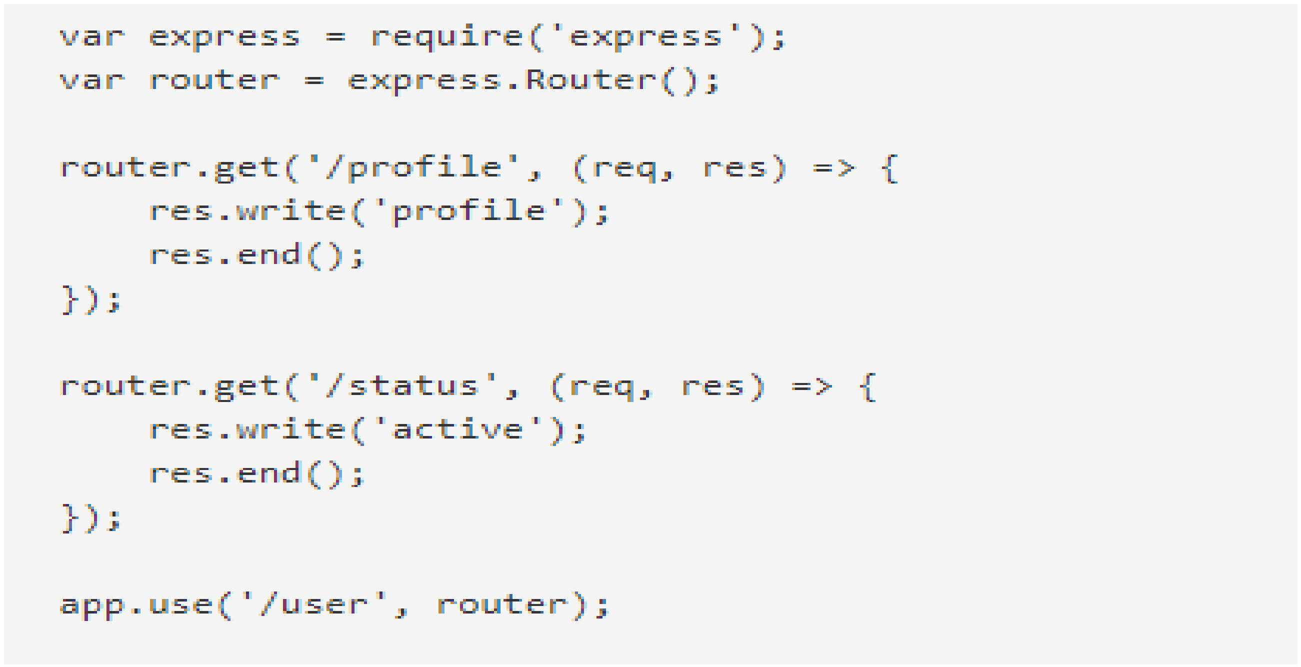
- we can also pass parameters in the URL's and can use them in functions. so, the code above will match /tom /mary /joe /everyone and prints "hello "+ name



```
app.get('/:name', (req, res) => {
    var name = req.params.name;
    res.write('hello '+ name);
    res.end();
});
```

Routing Module

- When the app grows, you'd want to manage the routes in a modular way. And express provides a separate smaller router component to achieve this. In this example, we're adding a sub-router for /user so, any url's that has /user as the first part will be handed over to the router.



```
var express = require('express');
var router = express.Router();

router.get('/profile', (req, res) => {
    res.write('profile');
    res.end();
});

router.get('/status', (req, res) => {
    res.write('active');
    res.end();
});

app.use('/user', router);
```

Routing Module

- In a bigger application, you can separate the routing logic and create controllers and bootstrap the controllers like below. In this case, book would export a book router and will handle all the urls starting with /book. while the user router will handle all the urls starting with user.

```
var express = require('express'),
    user = require('./user'),
    book = require('./book');

var app = express();

app.use('/user', user);
app.use('/book', book);
```

Views

- Express doesn't bundle any template engines
- But it supports 14+ NodeJS template engines

How often do we see web pages in plain text? most apps we build usually should return html or json. To keep thing simple, ExpressJS team decided not to support any specific template engines, so how do they support 14+ template engines.

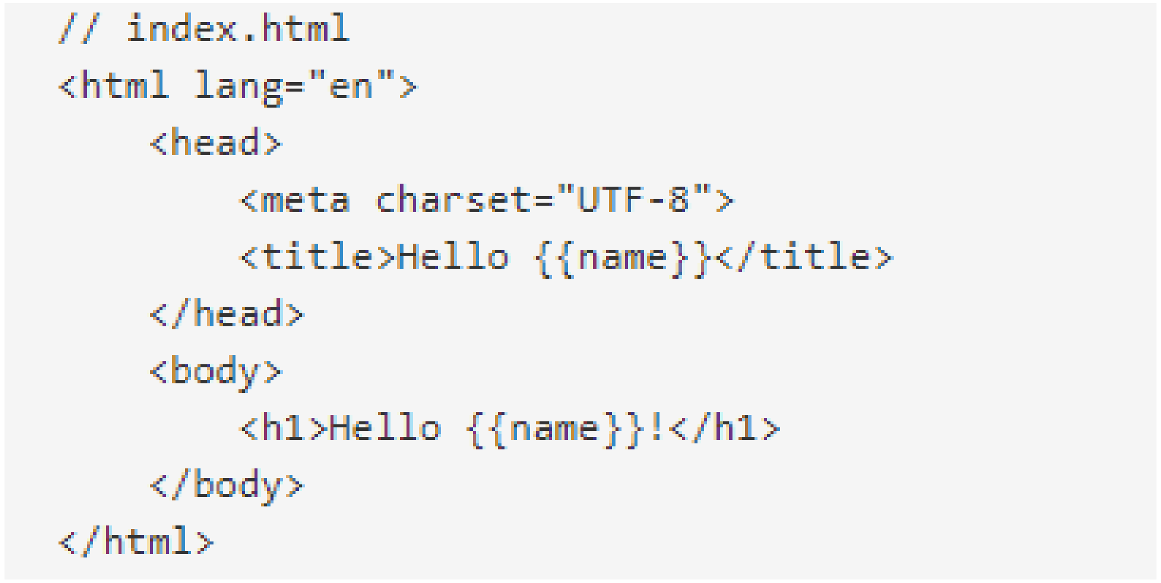
```
var express = require('express'),
    cons = require('consolidate'),
    app = express();

app.engine('html', cons.handlebars);
app.set('view engine', 'html');
app.set('views', __dirname + '/views');

app.get('/', (req, res) => {
    res.render('index', {
        name: 'world'
    });
});
```

Views

- ExpressJS uses consolidate.js library to support template engines.
- And since consolidate.js suppports about 14+ NodeJS template engines, they can be utilized in the express app.
- let's go through the snippet, then we call app.engine method, which tells express use handlebars template engine to parse all the html files.
- we call app.set, which sets let's you manage express config. then we set 'view engine', 'html' which is the engine we just defined and 'views' to the absolute path of directory which contains all the view files.
- Then inside our middleware, we call res.render, which takes in 2 parameters, the view and the data that needs to be passed to the view. the template engine, in our case handlebars will render the view file with the data passed.

Views

- And that'll render process the handlebars template and renders the output to the client. Consolidate provides support for most of the famous template engines like jade, handlebars, nunjucks. so you choose whatever flavour you like. Also, all the layout options and the way you manage layouts will be handled by the template engine.



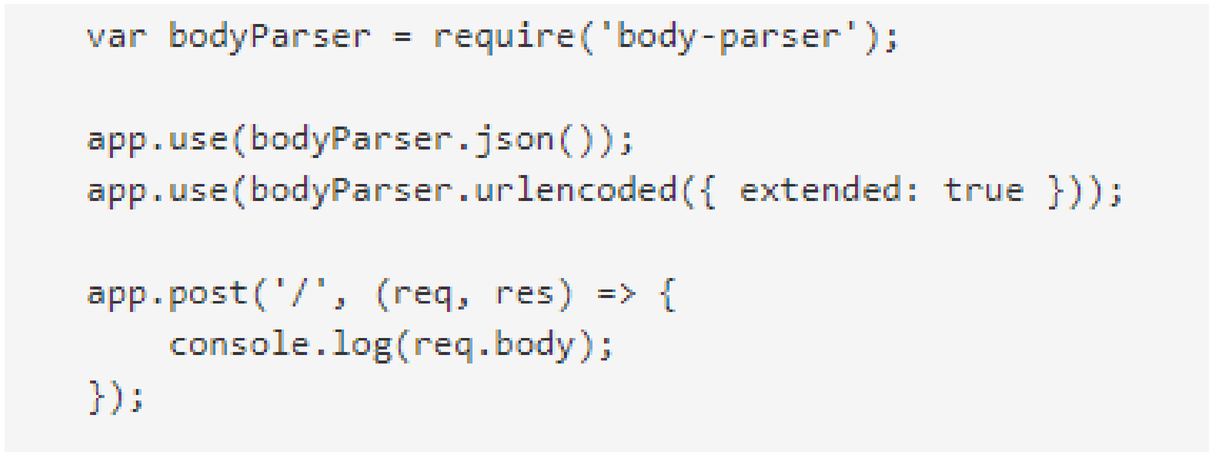
```
// index.html
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Hello {{name}}</title>
    </head>
    <body>
        <h1>Hello {{name}}!</h1>
    </body>
</html>
```

Body Parser Middleware

We don't have to write our own middlewares, there's a large number of middlewares out there which can let you accomplish most common tasks. The express team themselves support a few essential middlewares. Since expressJS on its own cannot parse the content of a POST or a cookie, it requires a middle-ware. For example, the body parser.

Body Parser Middleware

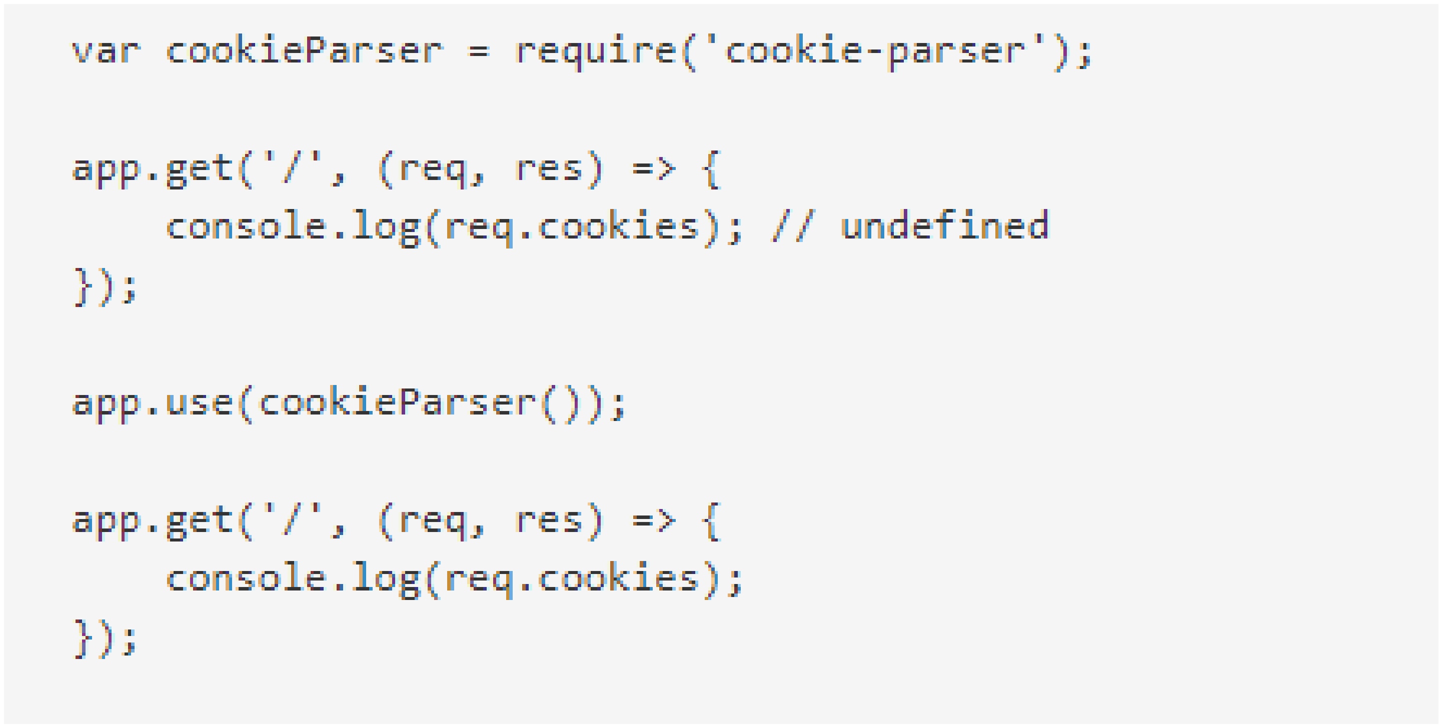
- Body Parser determines whether to process the request body as URL encoded of JSON data based upon the request Content-Type
- Body Parser makes the data available through the body property on the request object
- Body Parser should be loaded before any routes which need access to the request body data



```
var bodyParser = require('body-parser');

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.post('/', (req, res) => {
    console.log(req.body);
});
```

Cookie Parser Middleware

This is a cookie parser middleware for getting the cookie information from the request header.



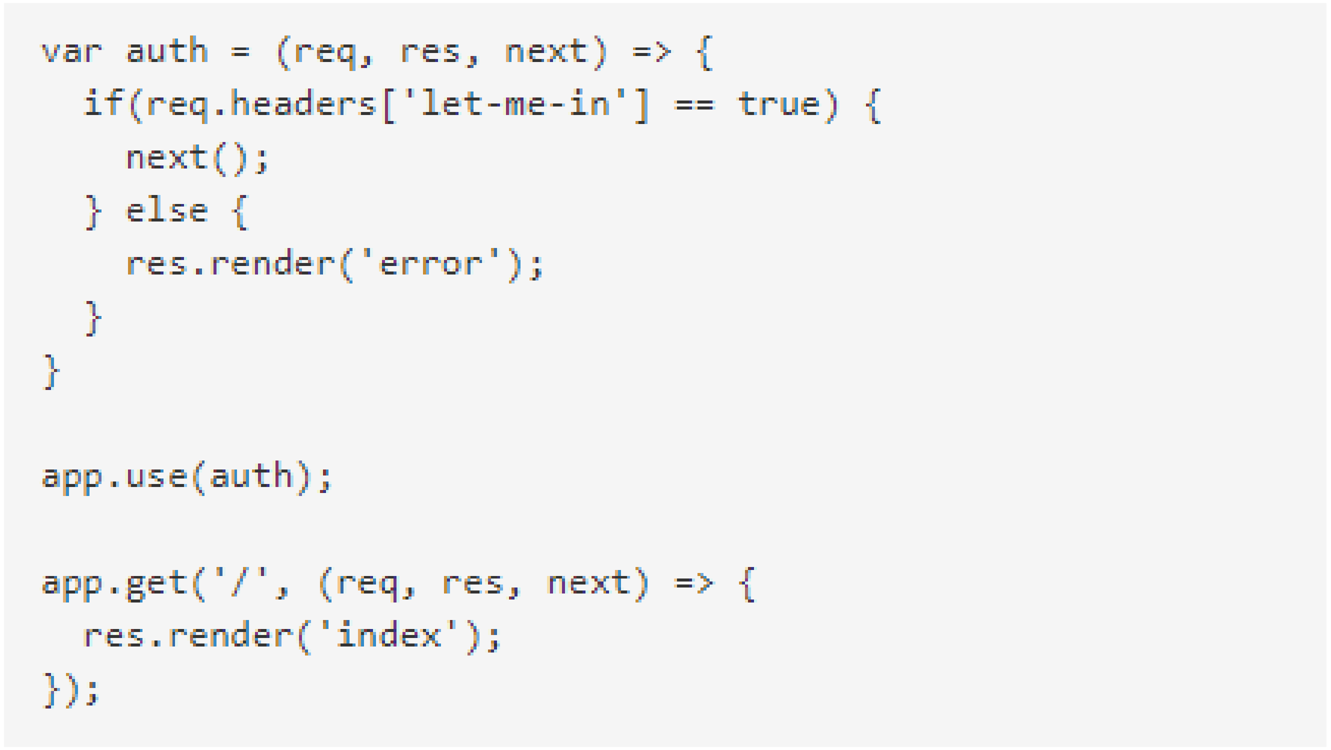
```
var cookieParser = require('cookie-parser');

app.get('/', (req, res) => {
    console.log(req.cookies); // undefined
});

app.use(cookieParser());

app.get('/', (req, res) => {
    console.log(req.cookies);
});
```

Cookie Parser Middleware

- Express provides the ability to write cookies to the response, but middleware is needed to parse cookies in the request
- The most common middleware is named Cookie Parser
- It is installed using: npm install cookie-parser
- The cookie parser middleware must be loaded before any routes which need access to cookies
- The response cookie function writes a cookie to the response by using the Set-Cookie header

Middleware for Auth

Middlewares can also be used for authentication, for example you can have an auth middleware which checks if an auth header is present and blocks the flow.



```
var auth = (req, res, next) => {
  if(req.headers['let-me-in'] == true) {
    next();
  } else {
    res.render('error');
  }
}

app.use(auth);

app.get('/', (req, res, next) => {
  res.render('index');
});
```

Passport Authentication

Authentication for Express is commonly provided by Passport

Passport provides many different strategies for authenticating users

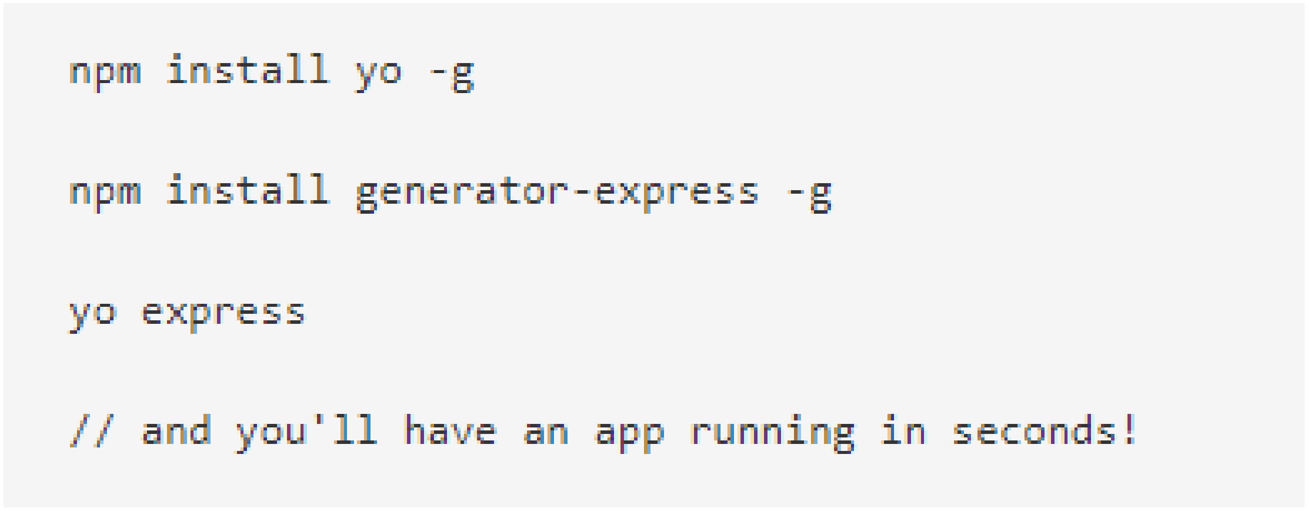
- User accounts stored in local databases
- Facebook, Twitter, Google
- OAuth and OAuth2 Providers
- WS Federation and SAML2

Passport Authentication (express-session)

- Passport relies **express-session** middleware to use session-based cookies to track logged in users
- Sessions are not required if credentials like an API-key are provided on every request
- Passport is well suited for user applications as well as REST services

Do I have to write everything myself?

So, we saw a bunch of things about expressJS. most are essential for any web app. So, do i have to write everything myself? Luckily not, people have come up with Yeoman generator, which scaffolds the app with all the essentials including templates and database. use them! but don't trust them blindly, read through the code they generate



```
npm install yo -g

npm install generator-express -g

yo express

// and you'll have an app running in seconds!
```