Express

Why Express.js?

```javascript
const http = require('http');
const fs = require('fs');

const server = http.createServer((req, res) => {
    // console.log(req.url, req.method, req.headers);
    const url = req.url;
    const method = req.method;

    if (url === '/') {
        // do something…
    }
    if (url === '/messsage' && method === 'POST') {
        // do something…
    }
    // do something…

    if (url === '/messsage' && method === 'GET') {
        // do something…
    }
    // do something…
});

server.listen(3000);
```

Server Logic is Complex!

You want to focus on your Business Logic, Not on the nitty-gritty Details

Use a Framework for the Heavy Lifting!

Framework: Helper functions, tools & rules that help you build your application!

Express

- Express.js is a web framework based on the core Node.js `http` module. Those components are called middleware.

- What Does Express.js Help You With?

| Parsing Requests & Sending Responses	Routing	Managing Data
Extract Data	Execute different Code for different Requests	Work with Files
Return Data	Filter/Validate incoming Requests	Work with Databases

Express Application Structure

- The typical structure of an Express.js app (which is usually `app.js` file) roughly consists of these parts, in the order shown:

1. Dependencies
2. Instantiations
3. Configurations
4. Middleware
5. Routes
6. Error Handling
7. Bootup

workflow
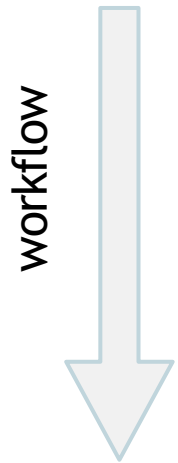
Your First Express App

- **Create a new `package.json` file**
  - `npm init`

1. **Dependencies**: Install Express
   - `npm install express`

2. **Instantiations:** Instantiate Express. Create a file named `app.js`, then add the content below:

```javascript
const express = require('express');

const app = express();

app.listen(3000, () => {
    console.log('Your Server is running on 3000');
});
```

Configurations

- There are two ways to configure our application:

1. **set**
   - `app.set('port', process.env.PORT || 3000);`
   - `const port = app.get('port');`

2. **enable/disable – for Boolean value only**
   - `app.enable('case sensitive routing')`
   - `app.set('case sensitive routing', true)`

   - `app.disable('case sensitive routing')`
   - `app.set('case sensitive routing', false)`
     - when it's enabled, then **/users** and **/Users** won't be the same. It's best to leave this option disabled by default for the sake of avoiding confusion.

Configurations – **'env'**

- During development, the app error messaging needs to be as verbose as possible, while in production it needs to be user friendly not to compromise any system or user's Personally Identifiable Information (PII) data to hackers.

```
app.set('env', 'development');
console.log(app.get('env'));
```

- The better way is to start an app with package.json

```
"scripts": {
        "start": "nodemon app.js"
}
```

- Run with command: `npm start`

- The most common values for `env` setting are:
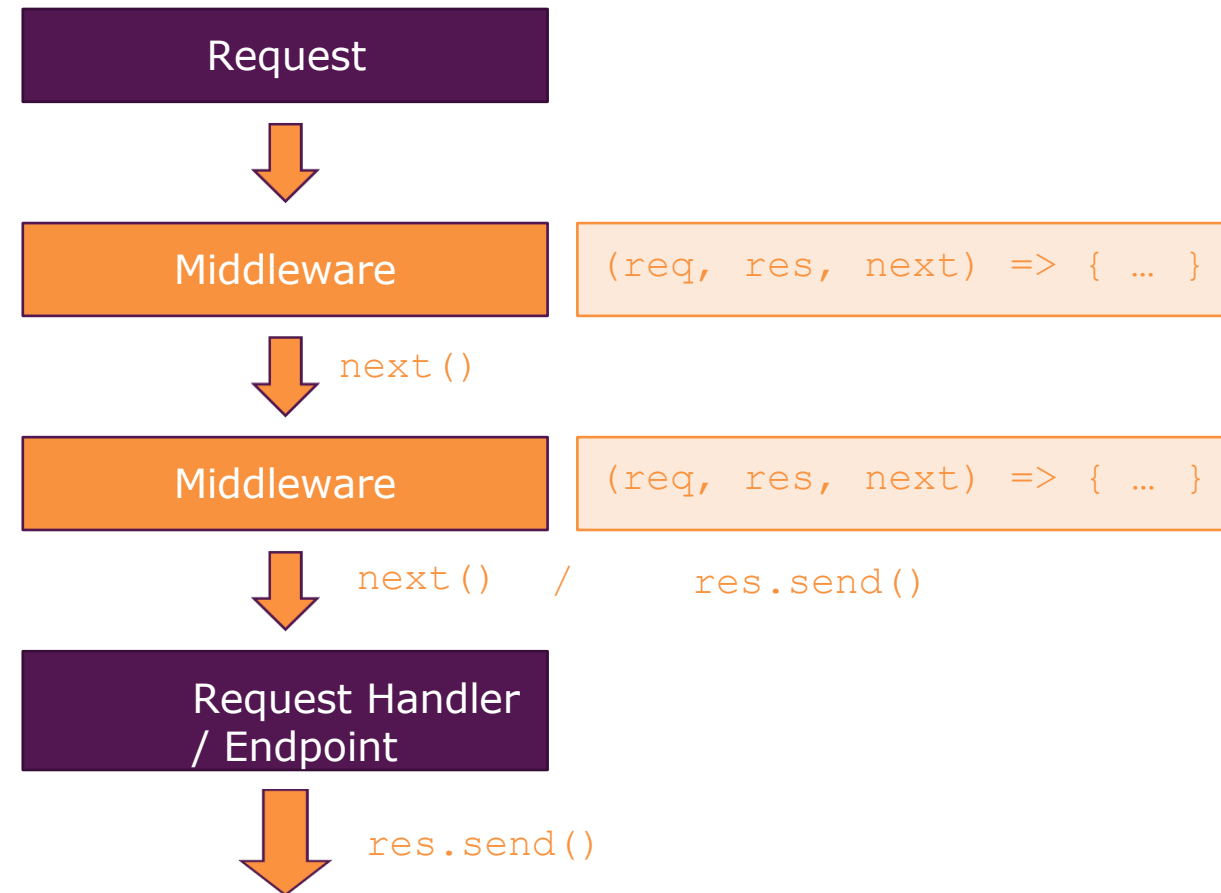 - development
  - test
  - stage
  - preview
  - production

https://en.wikipedia.org/wiki/Personally_identifiable_information

Middleware

- Middleware is a useful pattern that allows developers to reuse code within their applications and even share it with others in the form of NPM modules.

- The definition of middleware is a **function** with **three** arguments:

  - `function (req, res, next) {}`

- Error-handling middleware always takes **four** arguments.

  - `function (err, req, res, next) {}`

It's all about `Middleware`

```
Request
```

⬇

```
Middleware          (req, res, next) => { … }
```

`next()`

⬇

```
Middleware          (req, res, next) => { … }
```

`next()    /        res.send()`

⬇

```
Request Handler
/ Endpoint
```

`res.send()`

⬇

Using Middleware

- To use a middleware, we call the `app.use()` method which accepts:
  - **app.use([path,] callback[, callback...])**
  - One optional string path
  - One mandatory callback function

```javascript
app.use((req, res, next) => {
    console.log('This middleware always run!');
    next();
});

app.use('/products', (req, res, next) => {
    console.log('In the middleware!');
    res.send('The "Products" Page');
});

app.use('/', (req, res, next) => {
    console.log('In another middleware!');
    res.send('Hello from Express');
});
```

Built-in MiddleWare express parser

- Node.js body parsing middleware to handle HTTP POST request.
- Parse incoming request bodies in a middleware before your handlers, available under the `req.body` property.

- Express built-in middleware has 4 distinct methods:
  - `express.json([options])`: It parses incoming requests with JSON payloads. >= v4.16.0
  - `express.urlencoded([options])`: Processes URL-encoded data: name=value&name2=value2. >= v4.16.0
  - `express.raw([options])`: It parses incoming request payloads into a Buffer. >= v4.17.0
  - `express.text([options])`: It parses incoming request payloads into a string

- The result will be put in the `request` object with `req.body` property and passed to the next middleware and routes.

- NOTE: All built-in middleware are based on `body-parser` module. It does not support multipart( ). instead, use busboy, formidable, or multiparty

Example: Built-in MiddleWare express parser

```
app.use(express.urlencoded({
    extended: true
}));

app.use('/products', (req, res, next) =>{
        res.send('Added');
});

app.use('/products', (req, res, next) => {
    console.log(req.body); // { title: 'book' }
    res.send('Returning products')
});
```

The `extended` option allows to choose between parsing the URL-encoded data with the `querystring` library (when `false`) or the `qs` library (when `true`).

Using body parsing Only for certain route

```javascript
const express = require('express');
const app = express();

const jsonParser = express.json();
const urlencodedParser = express.urlencoded({ extended: false });

app.use('/login', urlencodedParser, function (req, res) {
    res.send('welcome, ' + req.body.username);
});

app.use('/api/users', jsonParser, function (req, res) {
    // create user in req.body
});
```

Request Object

- `request.params` Parameters middleware
- `request.query` Extract query string parameter
- `request.body` Payload, requires body-parser

http://expressjs.com/en/api.html#req

Request Object Examples

- `request.query`

  Optional

  ```
  http://localhost:3000/search?q=nodejs&lang=eng
  {"q": "nodejs", "lang": "eng"}
  ```

- `request.params`

  Mandatory

  ```
  app.get('/api/:id/:name/:city',
          function(req, res) {
                  res.end(req.params);
          }); // //
  http://localhost:3000/api/1/John/Fairfield
  { id: 1, name: 'John', city: 'Fairfield' }
  ```

Other Request Header Properties

`request.get(headerKey)` Value for the header key

`request.accepts(type)` Checks if the type is accepted

`request.acceptsLanguage(language)` Checks language

`request.acceptsCharset(charset)` Checks charset

`request.is(type)` Checks the type

`request.ip` IP address

`request.ips` IP addresses (with trust-proxy on)

`request.path` URL path

`request.host` Host without port number

`request.fresh` Checks freshness

`request.stale` Checks staleness

`request.xhr` True for AJAX-y requests

`request.protocol` Returns HTTP protocol

`request.secure` Checks if protocol is https

`request.subdomains` Array of subdomains

`request.originalUrl` Original URL

Response Object

- `response.send(status,data)` Send response
- `response.json(data)` Send JSON and force proper headers
- `response.sendfile(path, options, callback)` Send a file
- `response.status(status)` Send status code
- `res.redirect([status,] path)` Redirects to the URL derived from the specified path

Response Object Examples

```javascript
app.use('/posts', (req, res) => {
    let data = [{
            "userId": 1,
            "id": 1,
            "title": "sunt aut"
        },
        {
            "userId": 1,
            "id": 2,
            "title": "qui est esse",
            "body": "est rerum tempore"
        },
        {
            "userId": 1,
            "id": 3,
            "title": "ea molestias quasi"
        }
    ];
    res.json(200, data);
});

// a common way to send status number
response.status(200).send('Welcome')
```

> The `response.send()` method conveniently outputs any data application thrown at it (such as strings, JavaScript objects, and even Buffers) with automatically generated proper HTTP headers (Content-Length, ETag, or Cache-Control).

Routing app.VERB()

- Routes an HTTP request, where METHOD is the HTTP method of the request, such as GET, PUT, POST, and so on, in lowercase.

- Each route is defined by a method call on an application object with a URL pattern as the first parameter (regex are supported)

- `app.METHOD(path, [callback...], callback);`

```
app.use('/products', (req, res, next) =>
{   console.log(req.body);
    next(); //res.send('….')
});


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

The callbacks that we pass to `get()` or `post()` methods are called **request handlers** because they take `requests` (`req`), process them, and write to the `response` (`res`) objects.

Routing `app.all()`

- This method is like the standard app.METHOD() methods, except it matches all HTTP verbs.
- app.all(path, callback [, callback ...])

```javascript
app.all('/secret', function(req, res, next) {
    console.log('Accessing the secret section ...')
    next() // pass control to the next handler
})

app.all('*', requireAuthentication, loadUser)

app.all('/api/*', requireAuthentication)
```

The Router Class

- The Router class is a mini Express.js application that has only middleware and routes. This is useful for **abstracting modules** based on the business logic that they perform.

- You can think of it as a "mini-application," capable only of performing middleware and routing functions. Every Express application has a built-in app router.

```
routes/product.js
const express = require('express');
const options = {
    "caseSensitive": false
};
const router = express.Router(options);


router.get('/products', (req, res, next) => {
    res.send('All Products');
});


router.post('/products', (req, res, next) =>{
    console.log(req.body);
    res.send('Product Saved')
});
module.exports = router;
```
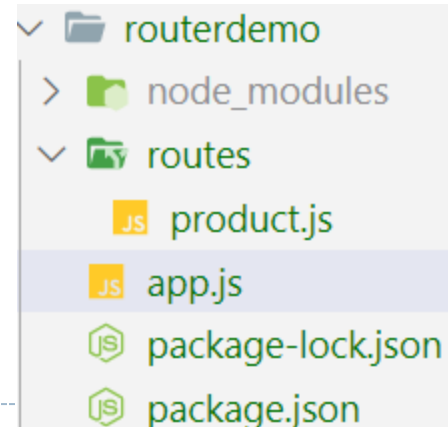
```
app.js
const express = require('express');
const productRouter = require('./routes/product');
const app = express();

app.use(express.urlencoded({ extended: true }));
app.use(productRouter);

app.listen(3000, () => console.log('listening on 3000...'));
```

```
∨ 📁 routerdemo
  > 📁 node_modules
  ∨ 📁 routes
       JS product.js
    JS app.js
    🟢 package-lock.json
    🟢 package.json
```

Error Handling in Express

- Define error-handling middleware functions in the same way as other middleware functions, except error-handling functions have **four** arguments instead of three: `(err, req, res, next)`

```
app.use(function (err, req, res, next) {
    res.status(500).send('Something broke!');
});
```

> Responses from within a middleware function can be in any format that you prefer, such as an HTML error page, a simple message, or a JSON string.

- **IMPORTANT:** You define error-handling middleware last, after other app.use() and routes calls.

Error Handling in Express

- For organizational (and higher-level framework) purposes, you can define several error-handling middleware functions, much as you would with regular middleware functions.

```javascript
function logErrors (err, req, res, next) { console.error(err.stack); next(err);
}

function clientErrorHandler (err, req, res, next) {
        if (req.xhr) { res.status(500).send({ error: 'Something failed!' })
} else { next(err) } }

function errorHandler (err, req, res, next) {
        res.status(500) res.send('error')
}

app.use(logErrors)
app.use(clientErrorHandler)
app.use(errorHandler)
```

Notice that when *not* calling "next" in an error-handling function, you are responsible for writing (and ending) the response. Otherwise those requests will "hang" and will not be eligible for garbage collection.

`next()`

- `next()`: Go to next request handler function(middleware, route), could be in the same URL route.

- `next('route')`: Skip current route and go to next one.

- `next('somethingElse')`: Go to Error Handler

Middleware Order Matters

- The order of middleware loading is important: middleware functions that are loaded first are also executed first.

```
app.use((req, res, next) => {
    res.status(404).sendFile(path.join(__dirname, 'views', '404.html'));
});

//below is not executed
app.get('/products', (req, res, next) => { res.sendFile(path.join(
    dirname, 'views', 'add-product.html'));
});
```