Express

Why Express.js?

```
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…

  });

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!

Alternatives to Express.js

- Vanilla Node.js
- Adonis.js
- Koa
- Sails.js
- …

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	Manage Data across Requests (Sessions)
Render HTML Pages	Filter /Validate incoming Requests	Work with Files
Return Data /HTML Responses		Work with Databases

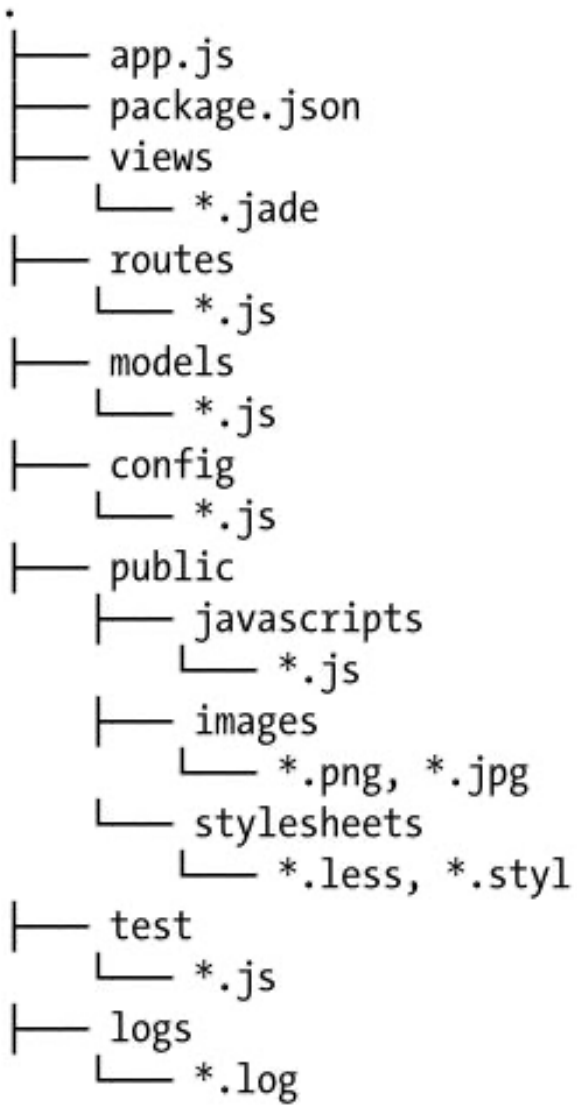
Express application generator

- Use the application generator tool, express-generator, to quickly create an application skeleton.

```
$ npm install -g express-generator

$ express MyApp

$ cd MyApp
$ npm install
```

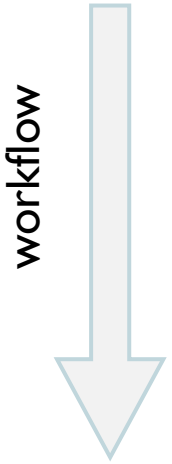


Watching for File Changes

- The following file-watching tools can leverage the watch() method from the core Node.js fs module and restart our servers when we save changes from an editor.
  - **forever** https://npmjs.org/package/forever
  - **node-dev** https://npmjs.org/package/node-dev
  - **nodemon** https://npmjs.org/package/nodemon
  - **supervisor** https://npmjs.org/package *Written by the creators of NPM*
  - **up** https://npmjs.org/package/up *Written by the Express.js team*

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



Your First Express App

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

1. Dependencies: Install Express
   - `npm install express –save`
2. Instantiations: Instantiate Express

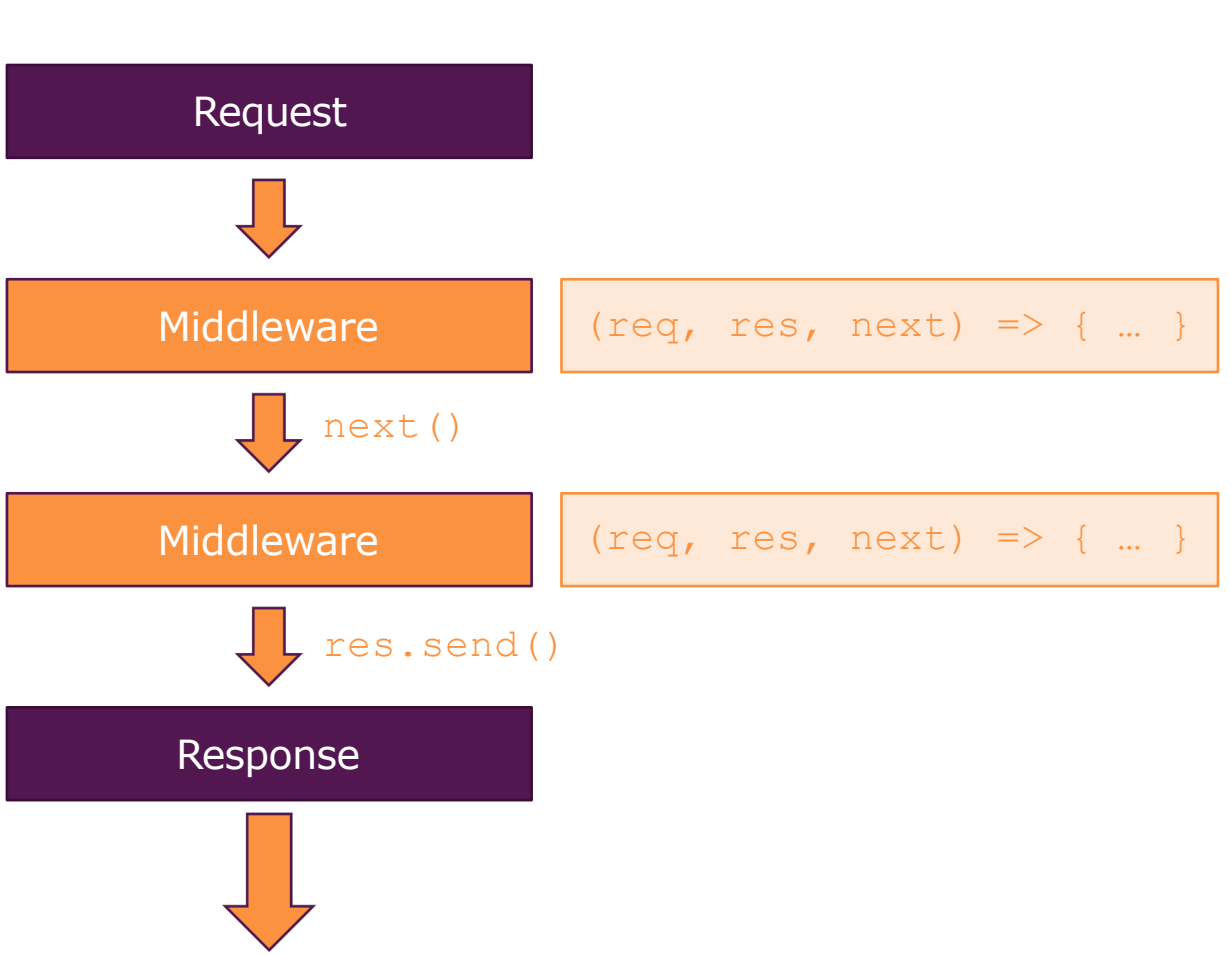
```
const express = require('express');

const app = express();

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

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:
  - request
  - response
  - next



Using Middleware

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

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

app.use('/add-product', (req, res, next) => {
    console.log('In the middleware!');
    res.send('<h1>The "Add Product" Page</h1>');
});

app.use('/', (req, res, next) => {
    console.log('In another middleware!');
    res.send('<h1>Hello from Express</h1>');
});
```

Middleware `body-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.

- The `body-parser` module has 4 distinct middlewares:
  - **`json()`** Processes JSON data
  - **`urlencoded()`** Processes URL-encoded data: name=value&name2=value2
  - **`raw()`** Returns body as a buffer type
  - **`text()`** Returns body as string type

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

Middleware `body-parser`

```
const bodyParser = require('body-parser’);
app.use(bodyParser.urlencoded());

app.use('/add-product', (req, res, next) => {
    console.log('In the middleware!');
    res.send('<form action="/product" method="post"><input name="title"><button type="submit">Sub
mit</button></form>');
});


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

    res.redirect('/');
});
```

- **Note:** `body-parser` does not support multipart( ). instead, use busboy, formidable, or multiparty.

Built-in MiddleWare express parser

- The express.json() and express.urlencoded() middleware have been added to provide request body parsing support out-of-the-box. This uses the expressjs/body-parser module module underneath.

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

- This option allows to choose between parsing the URL-encoded data with the querystringlibrary (when false) or the qs library (when true).
- This middleware is available in Express v4.16.0 onwards.

`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

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('/product', (req, res, next) => {
    console.log(req.body);
    res.redirect('/');
});
```

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

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.

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.

```
const express = require('express');
const options = {
    "caseSensitive": false,
    "strict": false
};
const router = express.Router(options);

router.get('/add-product', (req, res, next) => {
    console.log('In the middleware!');
    res.send('<form action="/product" method="post"><input name="title"><button type="submit">Submit</button></form>');
});

router.post('/product', (req, res, next) => {
    console.log(req.body);
    res.redirect('/');
});

module.exports = router;
```

Where options is an object that can have following properties:

- **caseSensitive**: Boolean
- **strict**: Boolean

Filtering Paths

- routes/admin.js

```
router.get('/admin/add-
product', (req, res, next) => {
    res.send('<form action="/admin/product" me
thod="post">…</form>');
});

router.post('/admin/product', (req, res, next)
 => {
    res.redirect('/');
});
```

- app.js

```
app.use(adminRoutes);
```

- routes/admin.js

```
router.get('/add-
product', (req, res, next) => {
        res.send('<form action="/admin/product
" method="post>…</form>');
});

router.post('/product', (req, res, next) => {
   res.redirect('/');
});
```

- app.js

```
app.use('/admin', adminRoutes);
```

Error Handling - Synchronous

Error Handling refers to how Express catches and processes errors that occur both synchronously and asynchronously. Express comes with a default error handler so you don't need to write your own to get started.

- **Catching Errors**
  - Errors that occur in synchronous code inside route handlers and middleware require no extra work. If synchronous code throws an error, then Express will catch and process it. For example:

```
app.get('/', function (req, res) {
        throw new Error('BROKEN') // Express will catch this on its own.
})
```

- How about asynchronous?

https://expressjs.com/en/guide/error-handling.html

Error Handling - Asynchronous

- For errors returned from asynchronous functions invoked by route handlers and middleware, you must pass them to the next() function, where Express will catch and process them. For example:

```
app.get('/', function (req, res, next) {
    fs.readFile('/file-does-not-exist', function (err, data) {
        if (err) {
            next(err) // Pass errors to Express.
        } else {
            res.send(data)
        }
    })
})
```

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.

```
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.render('error', { error: err })
}
```

```
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.

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('/add-product', (req, res, next) => {
    res.sendFile(path.join(__dirname, 'views', 'add-product.html'));
});
```

Resources

- Express Resources
  - ExpressJS
  - Connect
  - Express Wiki
  - morgan
  - body-parser
- Other Resources
  - Understanding Express.js
  - A short guide to Connect Middleware

Homework - Exercise

1. Create a npm project and install Express.js (Nodemon if you want)
2. Change your Express.js app which serves HTML files (of your choice with your content) for "/", "/users" and "/products".
3. For "/users" and "/products", provides GET and POST requests handling (of your choice with your content) in different routers.
4. Provide your own error handling