Express

Rujuan Xing

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

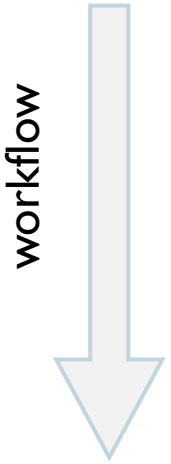
- Express.js is a web framework based on the core Node.js `http` module. Those components are called middleware.

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

```
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**
   - `app.enable('etag') === app.set('etag', true)`
   - `app.disable('etag') === app.set('etag', false)`

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": "set PORT=9999 && set NODE_ENV=development && 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

Configurations - `'case sensitive routing'`

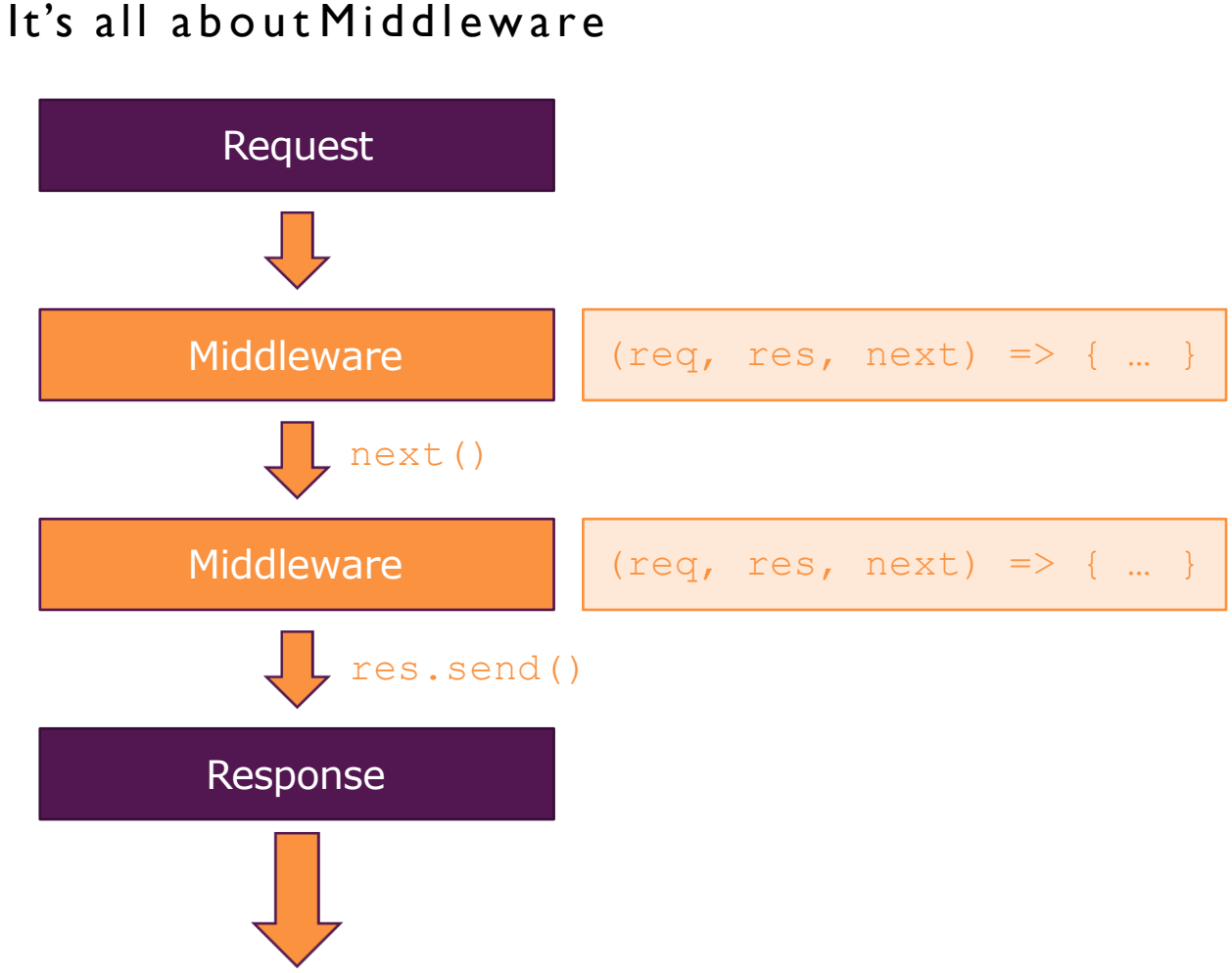
- By default, Express routes are not case-sensitive which the configuration below is `false`.

```
app.enable('case sensitive routing');
```

- For example, 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.

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) {}`



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

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

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('/add-product', (req, res, next) => {
        const html = `
    <!DOCTYPE html>
    <html>
    <body>
    <form action="/product" method="post">
      <input type="text" name="title"><br>
      <input type="submit" value="Submit">
    </form>
    </body>
    </html>`;
    res.send(html);
});

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

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

```
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.route` Return currently-matched route
- `request.cookies` Cookies, requires cookie-parser
- `request.signedCookies` Signed cookies, requires cookie-parser
- `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.redirect(status, url)` Redirect request
- `response.redirect(url)` Redirect to new path with status 302
- `response.send(status,data)` Send response
- `response.json(data)` Send JSON and force proper headers
- `response.jsonp(data)` JSON data will be wrapped in JS function call
- `response.sendfile(path, options, callback)` Send a file
- `response.render(templateName, locals, callback)` Render a template
- `response.locals` Pass data to template()
- `response.status(status)` Send status code

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

Response Object Examples

```
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).

`next()`

- `next()` : Go to next request handler function(middleware, route), could be in the same URL route.
- `next('route')` : bypass the remaining route callback(s) and go to next one. next('route') will work only in middleware functions that were loaded by using the app.METHOD() or router.METHOD() functions.
- `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.

Routing `app.all()`

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

```
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

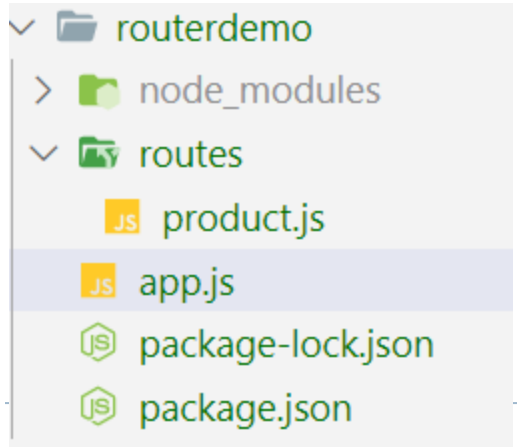
- The Router class is a mini Express.js application that has only middleware and routes. This is useful for **abstracting modules** based on the business logic that they perform.
- You can think of it as a "mini-application," capable only of performing middleware and routing functions. Every Express application has a built-in app router.

routes/product.js

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

router.get('/add-product', (req, res, next) => {
    const html = `
       <form action="/product" method="post">
      <input type="text" name="title"><br>
      <input type="submit" value="Submit">
    </form>`;
    res.send(html);
});

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



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

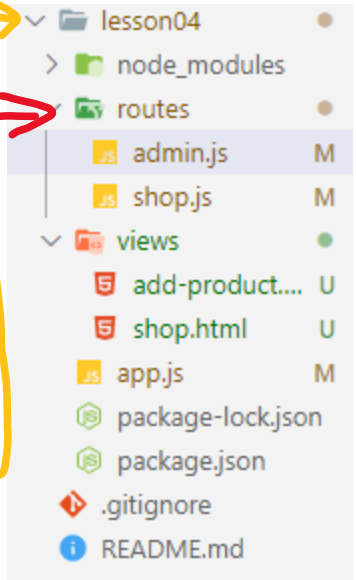
```
app.use('/admin', adminRoutes);
```

Serving HTML Pages

- `path.join([...paths])`:The `path.join()` method joins all given path segments together using the platform-specific separator as a delimiter, then normalizes the resulting path.
- `__dirname` tells you the absolute path of the directory containing the currently executing file.

- routes/admin.js

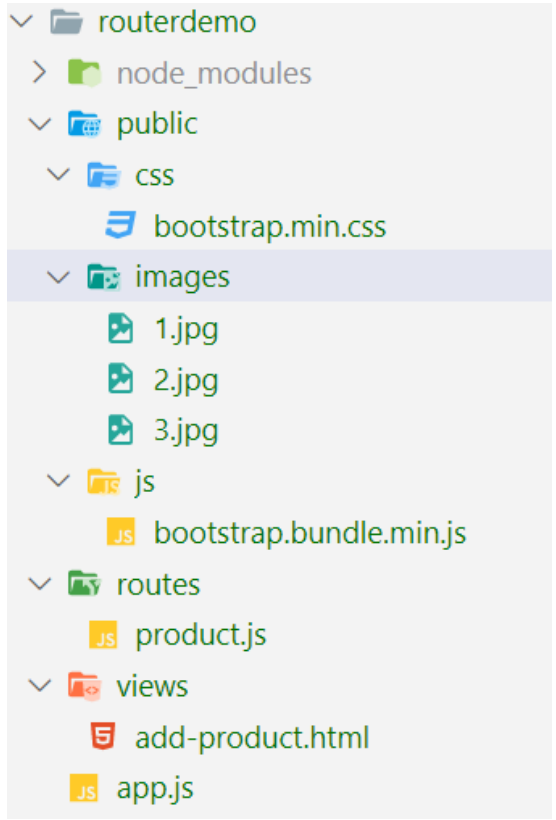
```
router.get('/add-product', (req, res, next) => {
    res.sendFile(path.join(__dirname, '..', 'views', 'add-product.html'));
});
```



Serving Static Resources

- **static** is the **only** middleware that comes with Express.js before version 4.15.x. It enables pass-through requests for static assets.

```
app.use(express.static(path.join(__dirname, 'public')));
<link rel="stylesheet" href="/css/main.css">

app.use('/mycss', express.static(path.join(__dirname, 'public', 'css')));
app.use('/img', express.static(path.join(__dirname, 'public', 'images')));
app.use('/js', express.static(path.join(__dirname, 'public', 'js')));
<link rel="stylesheet" href="/mycss/main.css">
```



Once Express sees a request to the following paths `/mycss` or `/img` or `/js` it will stream those resources immediately without looking at the rest of the Routes or other Middleware.

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.

Returning a 404 page

- The **HTTP 404**, **404 Not Found**, **404**, **Page Not Found**, or **Server Not Found** error message is a Hypertext Transfer Protocol (HTTP) standard response code, in computer network communications, to indicate that the browser was able to communicate with a given server, but the server could not find what was requested.

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

- **IMPORTANT:** You define 404 page not found middleware last, after other app.use() and routes calls.

https://en.wikipedia.org/wiki/HTTP_404

Middleware Order Matters

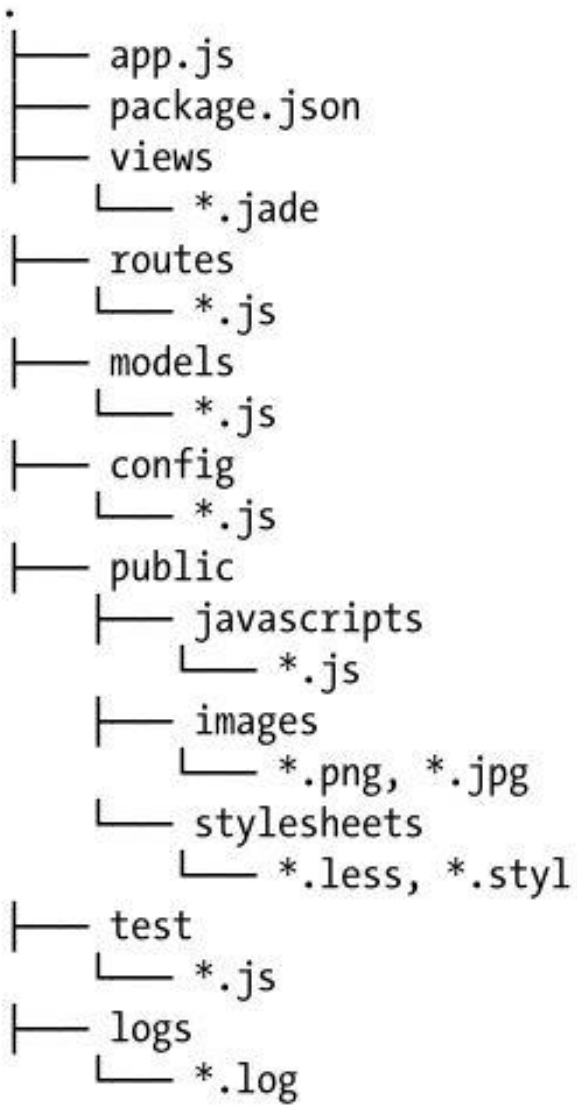
- The order of middleware loading is important: middleware functions that are loaded first are also executed first.

```
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'));
});
```

Express application generator

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

```
$ npm install -g express-generator

$ express -e -c less -f MyApp
// -e ejs
// -c set the stylesheet engine to less
// -f force on non-empty directory
// -h for help

$ 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*

Resources

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