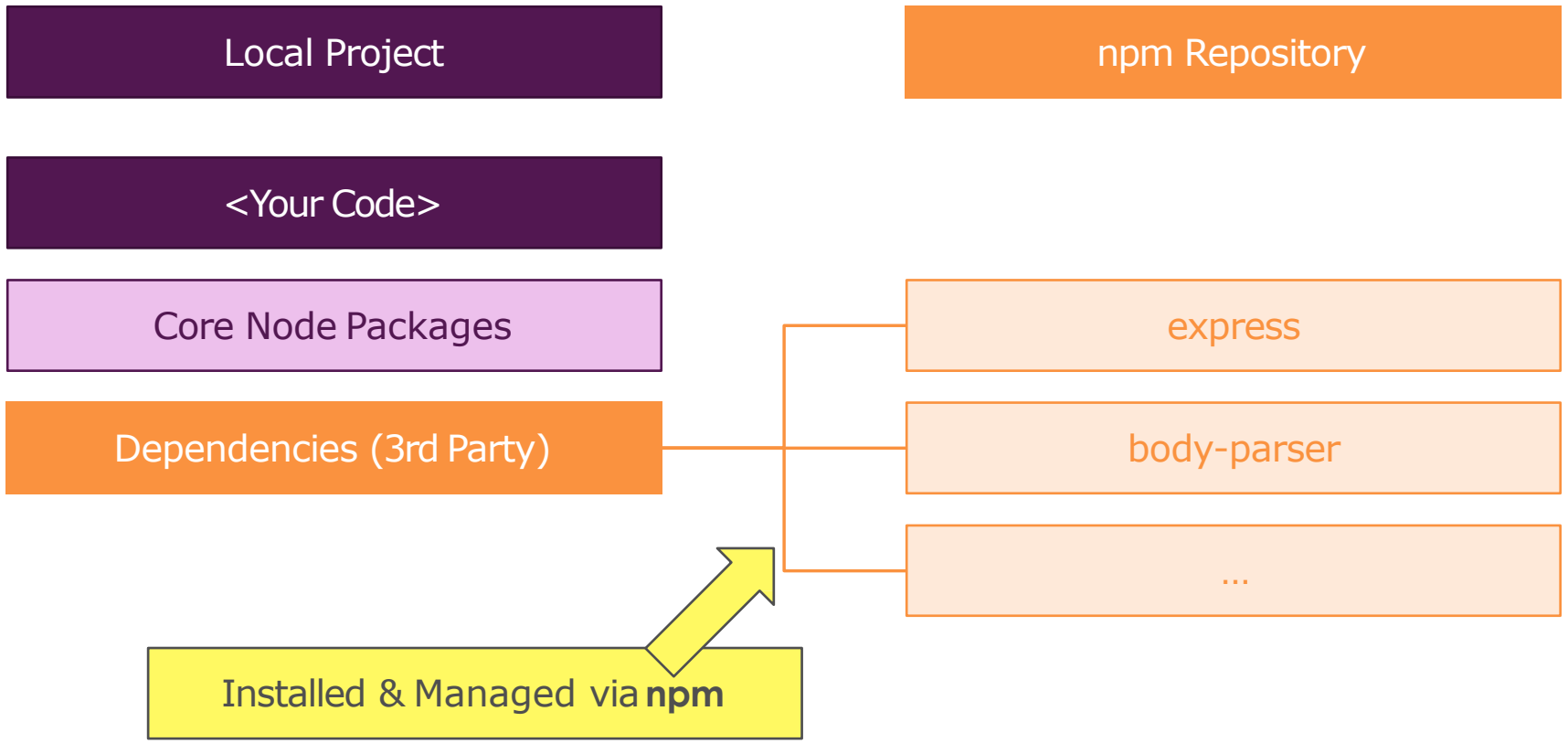
Express Framework

npm & packages Intro



What is npm?

- **npm** is a node package manager for Node.js packages, or modules if you like.
- www.npmjs.com hosts thousands of free packages to download and use.
- The NPM program is installed on your computer when you install Node.js.

- When we install a package:
  - notice folder: `node_modules`
  - This structure separate our app code to the dependencies. Later when we share/deploy our application, there's no need to copy `node_modules`, run: `npm install` will read all dependencies and install them locally.

Main Point Preview

- Express is a web application framework; it provides us with a lot of functionality to more easily make web applications. It also has a variety of useful (but optional) configuration options.

- *Science of Consciousness*: Do less and Accomplish more.

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. It has optional additional components which 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

Your First Express App

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');
})
```

Express Middleware

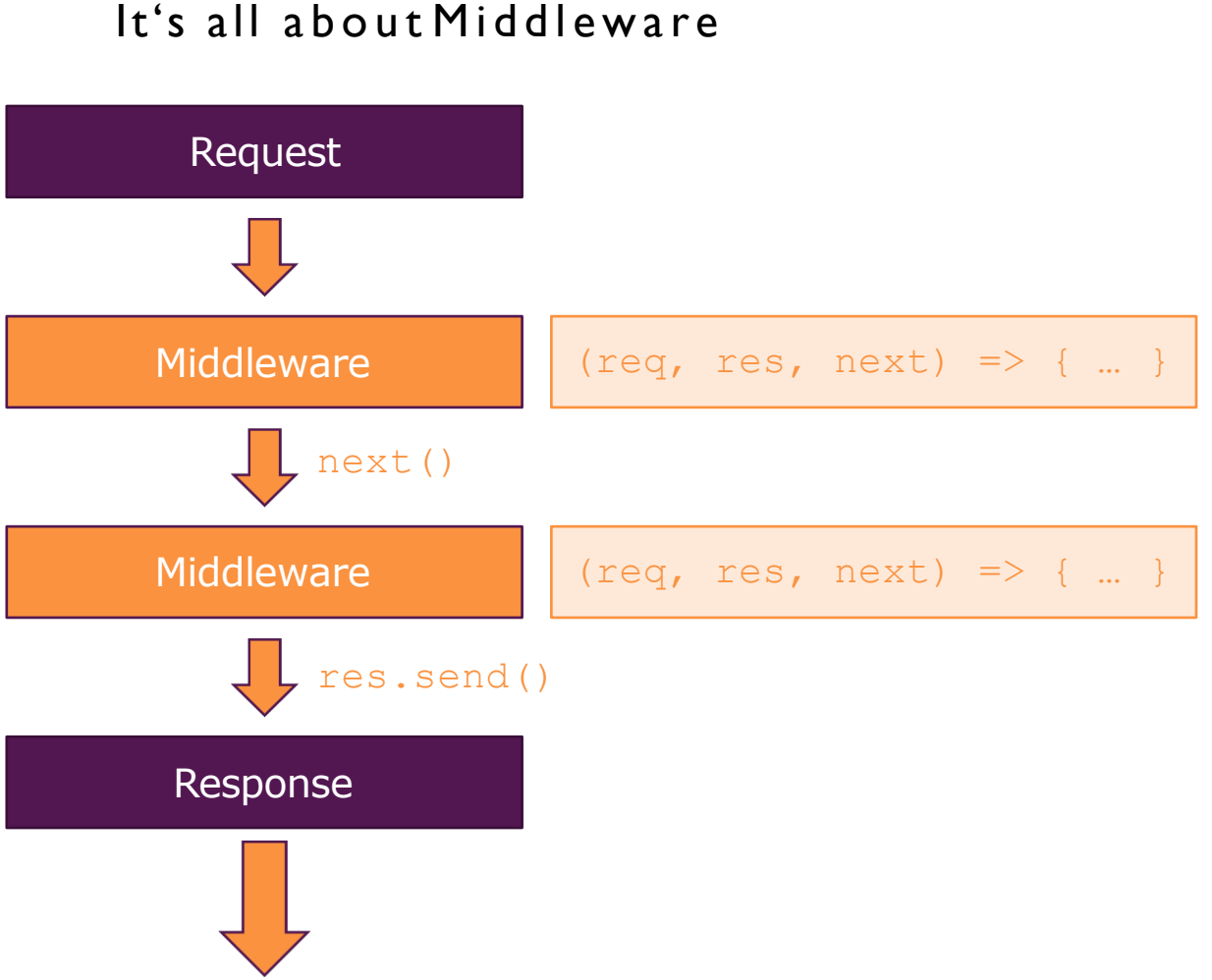
Main Point Preview

- Middleware are optional components (functions) for Express that can help process the request / response. Optionally you can specify for which route / URL(s) the middleware should be used.

- *Science of Consciousness*: Do less and Accomplish more.

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 is 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

Deprecated see upcoming slide on built-in

- 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('/');
});
```

Deprecated version shown as an example of how to use external middleware

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

Using body-parser Only for certain route

```
const express = require('express')
const bodyParser = require('body-parser')
const app = express()

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

app.post('/login', urlencodedParser, function (req, res) {
      res.send('welcome, ' + req.body.username)
})
app.post('/api/users', jsonParser, function (req, res) {
      // create user in req.body
})
```

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

Deprecated version shown as an example of how to use external middleware

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 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

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'));
});

// code below is never executed (everything receives 404)
app.get('/add-product', (req, res, next) => {
    res.sendFile(path.join(__dirname, 'views', 'add-product.html'));
});
```

Main Point

- Middleware are optional components (functions) for Express that can help process the request / response (for example: parse form input). Optionally you can specify for which route / URL(s) the middleware should be used.

- *Science of Consciousness*: Do less and Accomplish more.

Routing

Main Point Preview

- Using app.get() and app.post() we can specify how a HTTP GET or POST to a certain URL should be handled. The Router class allows us to group related routes together in separate files.

- *Science of Consciousness*: Life is found in layers.

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('*', userAuth);
app.all('/api/*', apiAuth);

var userAuth = function (req, res, next) {
      return next();
});

var apiAuth = function (req, res, next) {
      return next();
});
```

Routing Example

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

app.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>');
});

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

Configure Routes

```
app.get('/', function (req, res) {
    res.send('<html><body><h1>Hello World</h1></body></html>');
});
app.post('/submit-data', function (req, res) {
    res.send('POST Request');
});
app.put('/update-data', function (req, res) {
    res.send('PUT Request');
});
app.delete('/delete-data', function (req, res) {
    res.send('DELETE Request');
});
var server = app.listen(5000, function () {
    console.log('Node server is running..');
});
```

Main Point

- Using app.get() and app.post() we can specify how a HTTP GET or POST to a certain URL should be handled.

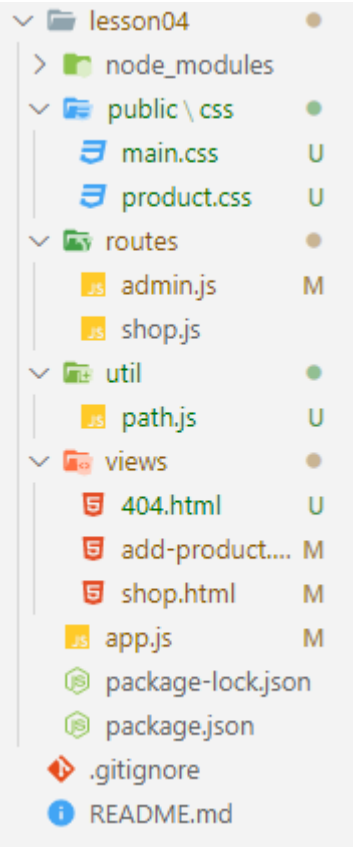
- *Science of Consciousness*: Life is found in layers, it's good to separate different areas of a large app into separate files.

Serving Static Resources

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

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

app.use('/mycss', express.static(__dirname, '/public/css'));
app.use('/img', express.static(__dirname, '/public/images'));
app.use('/js', express.static(__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.

Redirects

Main Point Preview

- HTTP POST requests should never return HTML, instead they should return an HTTP redirect. Express allows us to easily send redirects using the res.redirect() function.

- *Science of Consciousness*: Purification leads to progress. It's best to only use the HTTP methods for their intended purpose, violating their purpose makes things muddy.

GET and POST

- The purpose of GET is to retrieve data
  - It is possible to send data with it (in the url), but that's not its goal
- The purpose of POST is to send data
  - It is possible to retrieve data with it (as a response), but you shouldn't
- It's important when making web applications to use these as intended!
  - Although HTTP has more methods, the browser only uses GET and POST

Receiving data from POST

- It is very easy to return HTML on a POST request
  - But since POST was not intended for this purpose it creates problems
- POST responses cannot be bookmarked
  - This is because the input data needed to create the page is not saved, only the URL!
- Refreshing a page returned from a POST creates a "do you want to resubmit"
  - Not a 'problem', it just looks very amateurish
  - Professionals know never to return HTML from a POST

POST should always Redirect

- The correct reponse after a POST is a redirect
  - HTTP 303 code & url
  - Example: 303 /see/result/here
- Officially this is the
  - POST / Redirect / GET Pattern
  - Because the browser will make a GET request when it receives the redirect to show the user the new page

JSFiddle

```
1  const express = require("express");
2  const session = require("express-session");
3  const app = express();
4
5  app.use(express.urlencoded( { "extended": false}));
6  app.use(session({
7    "secret": "salt for cookie",
8    "resave": false,
9    "saveUninitialized": false,
10 }));
11 app.get("/", (req, res) => {
12   res.send(`<a href='add'>Add to session</a> <br>
13                <a href='view'>View session</a>`);
14 });
15 app.get("/add", (req, res) => {
16   res.send(`<form method='POST'>
17                <input name='key'>
18                <input name='value'>
19                <input type='submit'>
20              </form>`);
21 });
22 app.get("/view", (req, res) => {
23   let view = "<ul>";
24   for (const key in req.session) {
25     if (key === 'cookie') continue;
26     view += `<li>${key}: ${req.session[key]}</li>`;
27   }
28   view += "</ul><a href='/'>back</a>";
29   res.send(view);
30 });
31 app.post("/add", (req, res) => {
32   req.session[req.body.key] = req.body.value;
33   res.redirect(303, "/view");
34 });
35 app.listen(3000);
```

If you just give a URL it uses 302
Which works but is inappropriate for POST/Redirect/GET

Redirect 'back'

- If you just want to send the browser back to the previous page
  - You can simply use res.redirect('back')
  - Much easier than using a URL
  - Internally this uses the referrer HTTP header to find what the previous page was

Main Point

- HTTP POST requests should never return HTML, instead they should return an HTTP redirect. Express allows us to easily send redirects using the res.redirect() function.

- *Science of Consciousness*: Purification leads to progress. It's best to only use the HTTP methods for their intended purpose, violating their purpose makes things muddy.

Error Handling

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.

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(__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

Resources

- HTTP Resources
  - HTTP
  - Post / Redirect / Get
- Express Resources
  - ExpressJS
  - Connect
  - Express Wiki
  - morgan
  - body-parser
- Other Resources
  - Understanding Express.js
  - A short guide to Connect Middleware