19 Apr 2019

TO THE NEW™

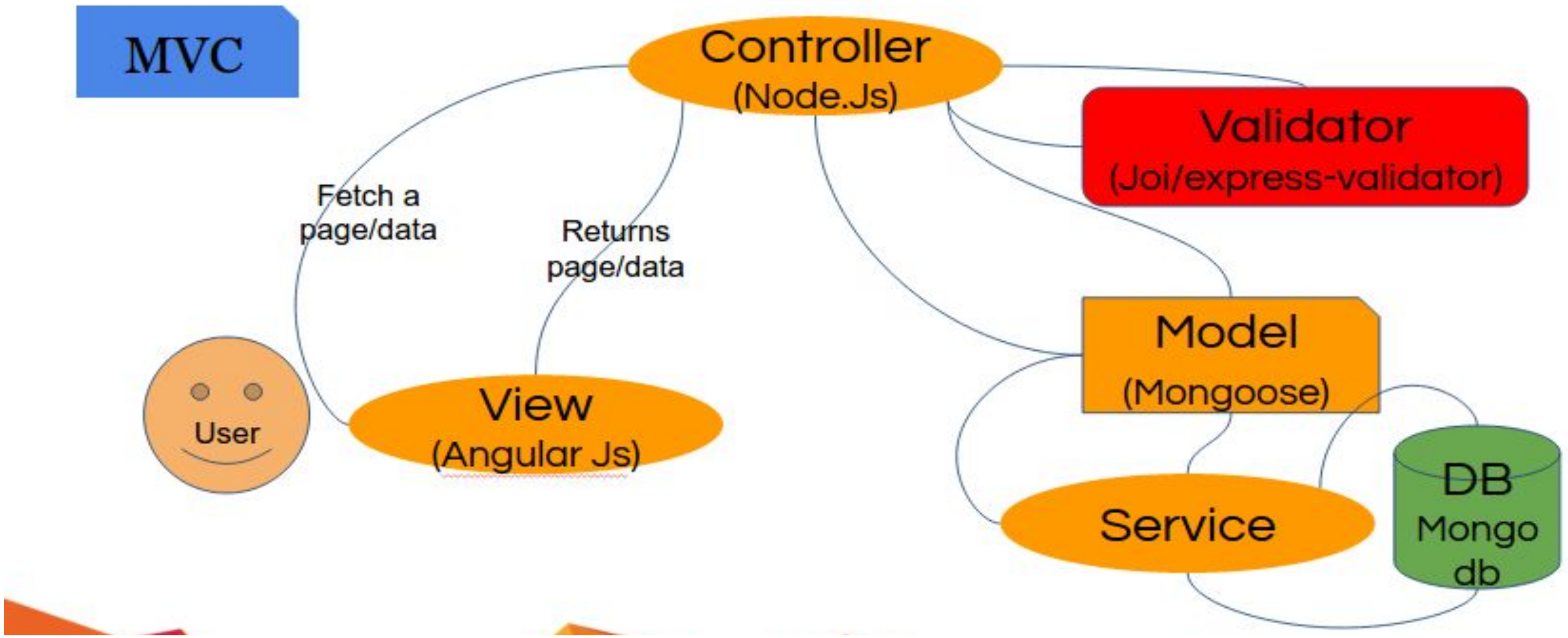
Express + Middleware

By : Khan M. Tabish & Rajesh

Agenda

- Understanding of Project Structure
- Creating Express Server
- Basic Routing
- Your own Tomcat
- Routing in depth
- Middleware
- Error Handling
- Starting with express-generator
- Exercises on ExpressJs

Understanding of Project Structure



Creating Express Server

Create Project & Install Express-Js

```
$ mkdir express-app && cd express-app
$ mkdir -p router views
$ touch router/index.js views/index.html views/about.html server.js
$ npm init
$ npm i express --save
```

Add dependencies in package.json

```
"dependencies":
 {
   "express": "~4.15.2",
   "ejs":"~2.5.6"
}
```

```
$ npm install
```

Creating Express Server

Creating Express Server(Hello.World!)

```
"use strict";
//Step1:start app at port 3000
//server.js
const Express = require('express');
const PORT = 3000;
const app = Express();
app.listen(PORT);

//Step2: Add callback to listen function
//server.js
app.listen(PORT, () => {
  console.info("Server is running @:http://localhost:%d", PORT);
});
```

Basic Routing(Exposure to REST)

```
//server.js
//...
const app = Express();
//...
app.get('/', (req, res) => {
    res.send('Hello World!');
});
//...
app.listen(PORT, () => {
    console.info("Server is running @:http://localhost:%d", PORT);
});
//...
```

Basic Routing(Exposure to REST)

Routing module

```
//router/index.js
module.exports = (app) => {
    app.get('/', (req, res) => {
        res.send('Hello World!');
    });
    app.get('/emp.json', (req, res) => {
        res.send(JSON.stringify([
            {name: ‘EMP-1’, id: 1},
            {name: 'EMP-2’, id: 2},
            {name: ‘EMP-1’, id: 3}
        ]));
    });
}
```

```
//server.js
//require router module
const router = require('./router');
//add routes
router(app);
```

Basic Routing(Exposure to REST)

Routing module- Send file

```
//router/index.js
const fs = require('fs');
const path = require('path');
module.exports = (app) => {
  //...
  app.get('/', (req, res) => {
      // maybe test for existence here using fs.stat
      res.writeHead(200, {"Content-Type": "text/html"});
      fs
        .createReadStream(path.resolve('views', 'index.html'))
        .pipe(res);
    });
}
```

Your own Tomcat(Public Server)

```
//server.js
const router = require('./router');
//..
//add static server
app.use('/static', Express.static('views'));
//..
//add routes
```

Routing in Depth

```
//router/users.js
//"use strict";
module.exports = (app) => {
    app.get('/user', (req, res) => {
        res.send(JSON.stringify([
            {name: ‘User-1’, id: 1},
            {name: ‘User-2’, id: 2},
            {name: ‘User-3’, id: 3} ])); });
    app.post('/user', (req, res) => {
        res.send('Got a POST request');
    });
    app.put('/user', (req, res) => {
        res.send('Got a PUT request at /user');
    });
    app.delete('/user', (req, res) => {
        res.send('Got a DELETE request at /user'); });
}
```

```
//server.js
const router = require('./router');
//...require new routes
const userRouter = require('./router/users');
router(app);
//...ad user routes
userRouter(app);
//...
app.listen(PORT, () => {
});
```

Routing in Depth

Chainable Routing- app.route()

```
//router/books.js
app.route('/book')
  .get((req, res) => {
    res.send('Get a random book');
  })
  .post((req, res) => {
    res.send('Add a book');
  })
  .put((req, res) => {
    res.send('Update the book');
  });
```

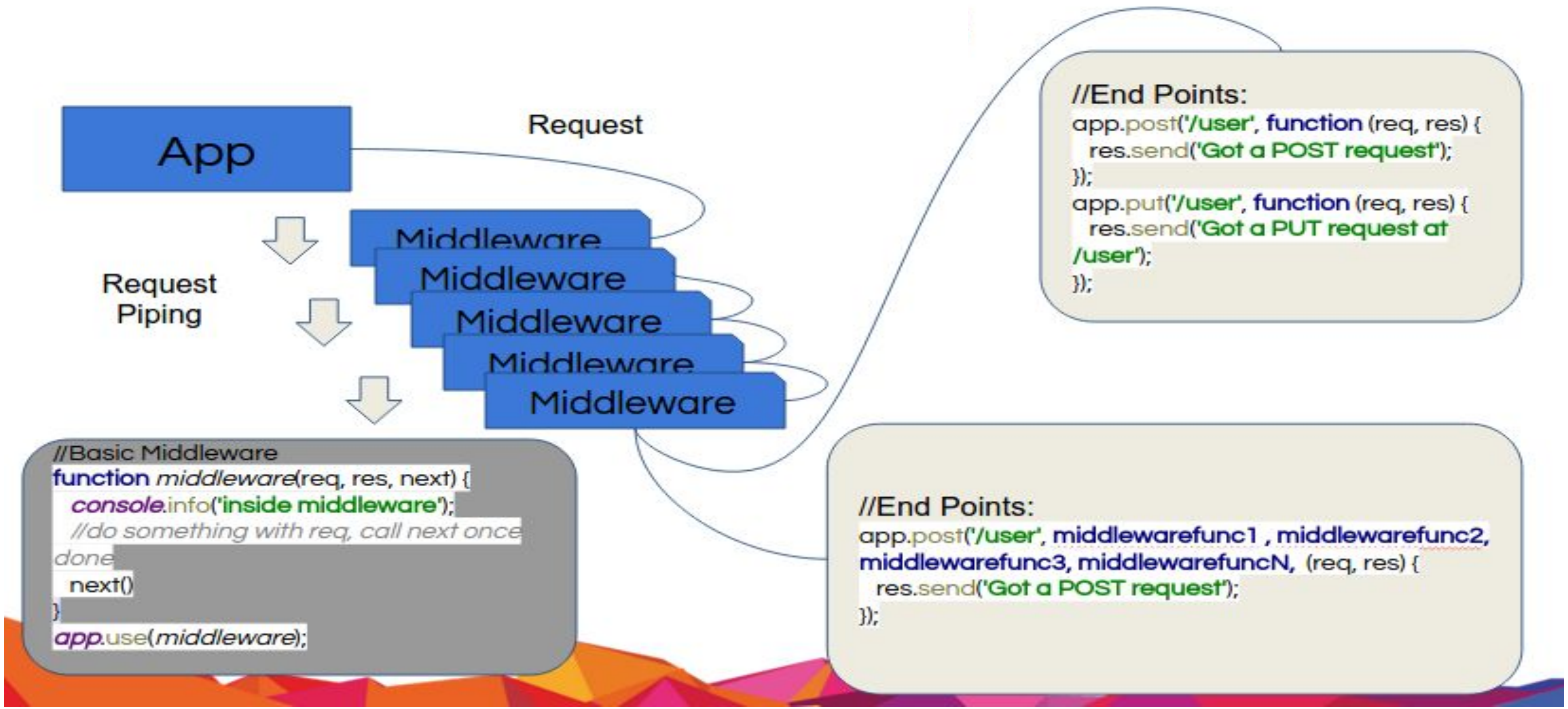
Routing in Depth

Express Router- express.Router()

```
//router/birds.js
"use strict";
const express = require('express');
const router = express.Router();
// middleware that is specific to this router
router.use((req, res, next) => {
   console.log('Time: ', Date.now());
   next();
});
```

```
// define the home page route
router.get('/', (req, res) => {
   res.send('Birds home page');
});
// define the about route
router.get('/about', (req, res) => {
   res.send('About birds');
});
module.exports = (app) => {
   app.use('/birds', router);
};
```

Middleware



```
//Basic Middleware
function middleware(req, res, next) {
  console.info('inside middleware');
  //do something with req, call next once
done
  next()
}
app.use(middleware);
```

```
//End Points:
app.post('/user', function (req, res) {
  res.send('Got a POST request');
});
app.put('/user', function (req, res) {
  res.send('Got a PUT request at
/user');
});
```

```
//End Points:
app.post('/user', middlewarefunc1 , middlewarefunc2,
middlewarefunc3, middlewarefuncN, (req, res) {
  res.send('Got a POST request');
});
```

Middleware : Overview

Middleware functions are functions that have access to the request object (req), the response object (res), and the next function in the application's request-response cycle. The next function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.

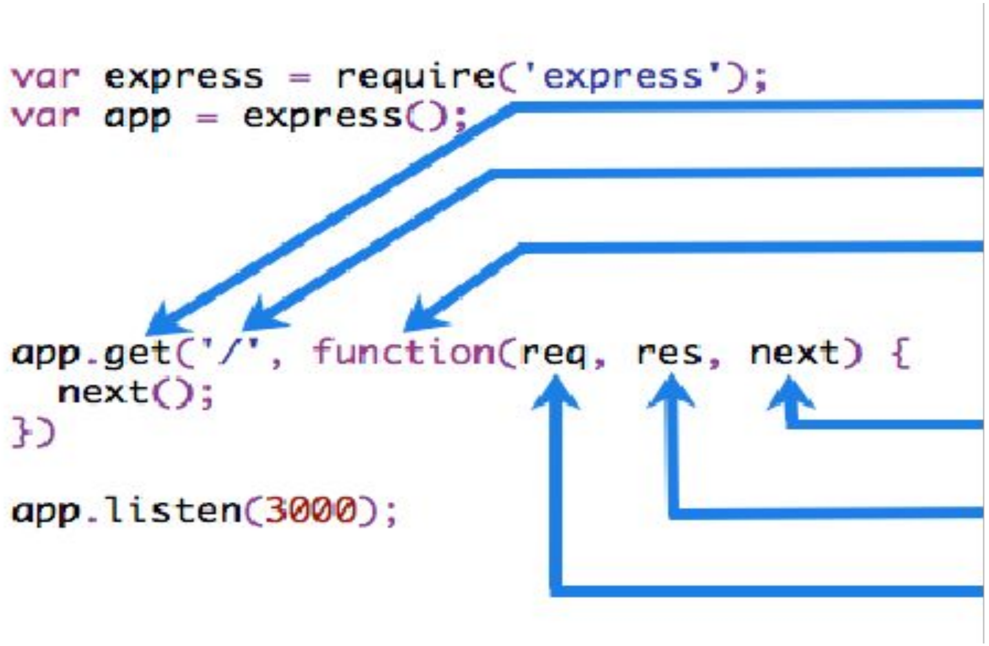
Middleware functions can perform the following tasks:

Execute any code.
Make changes to the request and the response objects.
End the request-response cycle.
Call the next middleware in the stack.

Understanding Middleware

If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.

The following figure shows the elements of a middleware function call:



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

app.get('/', function(req, res, next) {
  next();
})

app.listen(3000);
```

HTTP method for which the middleware function applies.
Path (route) for which the middleware function applies.
The middleware function.

Callback argument to the middleware function, called "next" by convention.
HTTP response argument to the middleware function, called "res" by convention.
HTTP request argument to the middleware function, called "req" by convention.

Middleware function myLogger

Here is a simple example of a middleware function called “myLogger”. This function just prints “LOGGED” when a request to the app passes through it. The middleware function is assigned to a variable named myLogger.

```
const myLogger = (req, res, next) => {
  console.log('LOGGED');
  next();
}
```

Notice the call above to next(). Calling this function invokes the next middleware function in the app. The next() function is not a part of the Node.js or Express API, but is the third argument that is passed to the middleware function. The next() function could be named anything, but by convention it is always named “next”. To avoid confusion, always use this convention.

Middleware function myLogger

Every time the app receives a request, it prints the message “LOGGED” to the terminal.
The order of middleware loading is important: middleware functions that are loaded first are also executed first.

If myLogger is loaded after the route to the root path, the request never reaches it and the app doesn’t print “LOGGED”, because the route handler of the root path terminates the request-response cycle.

The middleware function myLogger simply prints a message, then passes on the request to the next middleware function in the stack by calling the next() function.

Loading Middleware

To load the middleware function, call app.use(), specifying the middleware function. For example, the following code loads the myLoggermiddleware function before the route to the root path (/).

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

const myLogger = (req, res, next) => {
        console.log('LOGGED');
        next();
}

app.use(myLogger)
app.get('/', (req, res) => {
        res.send('Hello World!');
})
app.listen(3000);
```

Middleware functions request time.

Next, we’ll create a middleware function called “requestTime” and add it as a property called requestTime to the request object.

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

const requestTime = (req, res, next) => {
  req.requestTime = Date.now();
  next();
}

app.use(requestTime);
app.get('/', (req, res) => {
  var responseText = 'Hello World!<br>'
  responseText += '<small>Requested at: ' + req.requestTime + '</small>'
  res.send(responseText);
});

app.listen(3000);
```

Middleware

```
//server.js
//logging middleware
function middleware(req, res, next) {
    console.info('inside middleware');
    //do something with req, call next once done
    next();
}
//how to use it
app.use(middleware);
//add static server
app.use('/static', Express.static('views'));
```

Exercise- Create a request logging middleware

Error-handling Middleware

Error-handling middleware always takes four arguments. You must provide four arguments to identify it as an error-handling middleware function. Even if you don’t need to use the next object, you must specify it to maintain the signature. Otherwise, the next object will be interpreted as regular middleware and will fail to handle errors.

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

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

Default Error Handler

Express comes with a built-in error handler, which takes care of any errors that might be encountered in the app.
This default error-handling middleware function is added at the end of the middleware function stack.
If you pass an error to next() and you do not handle it in an error handler, it will be handled by the built-in error handler; the error will be written to the client with the stack trace. The stack trace is not included in the production environment.
If you call next() with an error after you have started writing the response (for example, if you encounter an error while streaming the response to the client) the Express default error handler closes the connection and fails the request.
So when you add a custom error handler, you will want to delegate to the default error handling mechanisms in Express, when the headers have already been sent to the client:

```
function errorHandler (err, req, res, next) {
  if (res.headersSent) {
    return next(err);
  }
  res.status(500);
  res.render('error', { error: err });
}
```

Note that the default error handler can get triggered if you call next() with an error in your code more than once, even if custom error handling middleware is in place

Error Handling

```
//server.js
userRouter(app);
//error handling middleware
//This should be after last app.use
function errorHandler(err, req, res, next) {
  console.info(err);
  res.status(500).send(err.message);
}
app.use(errorHandler);
//route/index.js
app.get('/error', (req, res, next) => {
    return next(new Error("Throwing error from /error"));
    res.send('This wil never reached');
});
```

Third party Middleware

The following example illustrates installing and loading the cookie-parsing middleware function cookie-parser.

```
$ npm install cookie-parser

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

const cookieParser = require('cookie-parser');

// load the cookie-parsing middleware
app.use(cookieParser());
```

Parse Cookie header and populate req.cookies with an object keyed by the cookie names. Optionally you may enable signed cookie support by passing a secret string, which assigns req.secret so it may be used by other middleware.

Third party Middleware

Parse incoming request bodies in a middleware before your handlers, availabe under the req.body property.
This does not handle multipart bodies, due to their complex and typically large nature.

```
$ npm install body-parser
```

Starting with express-generator

##Express Generator ###Install

```
$ npm i express-generator -g
```

###Create gapp

```
$ express <app-name>
```

Exercise

1. Create a user application which will have a home page, add user page and about us page

2. On Home, It will list all user in a table. There will be one more column Action in a table which will have a delete option to delete the user.

3. On add New user page, there will be form to create a new user. There will be a middleware that will add the created_on date in user when new user will be created.

4. On About-us page, there will some description of app and app creator.

Thank You

Any queries? Open to Q&A