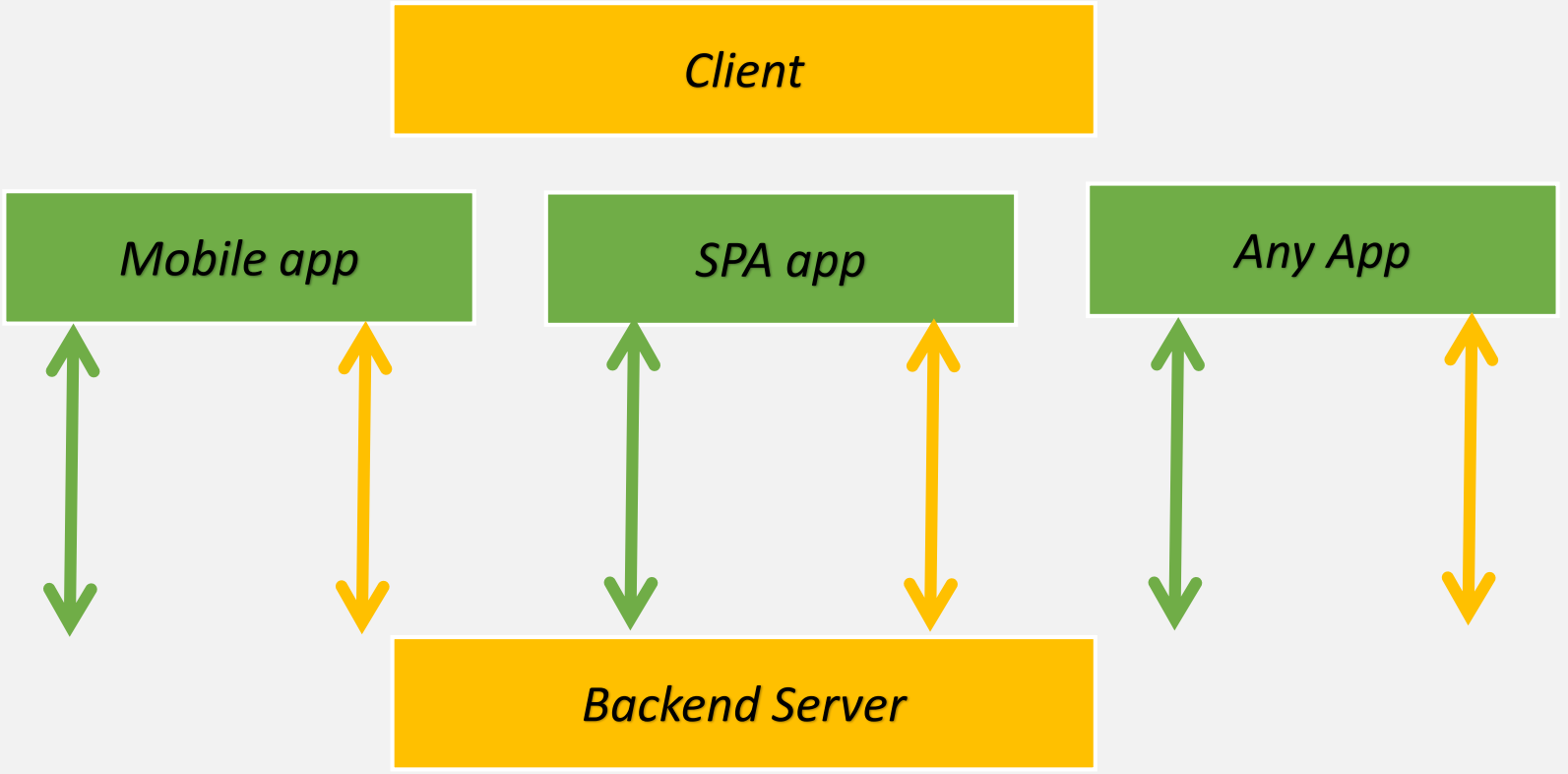
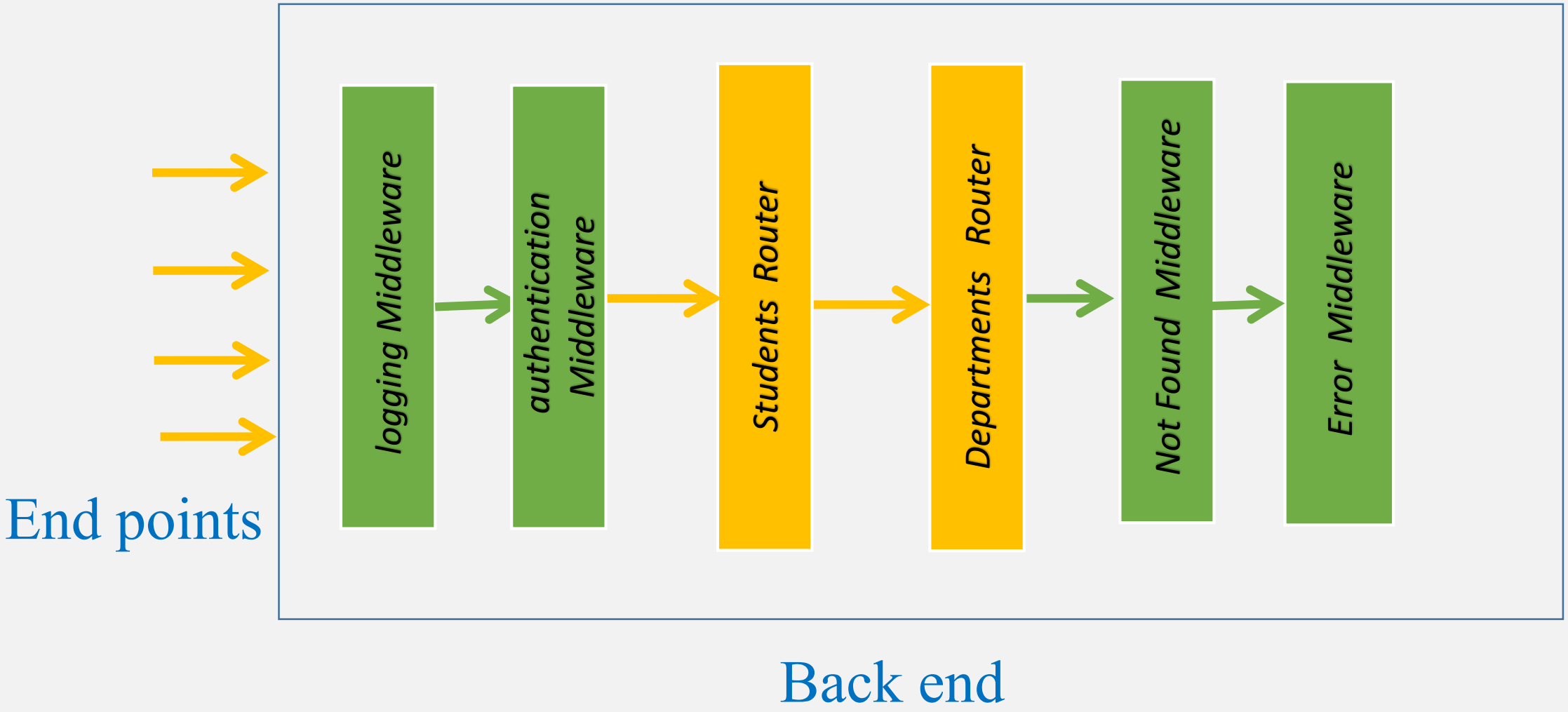

express
TM
Eman Fathi
https://expressjs.com

Working with Node JS as RESTFULL API

Transfer Data instead of user Interface



Backend Structure



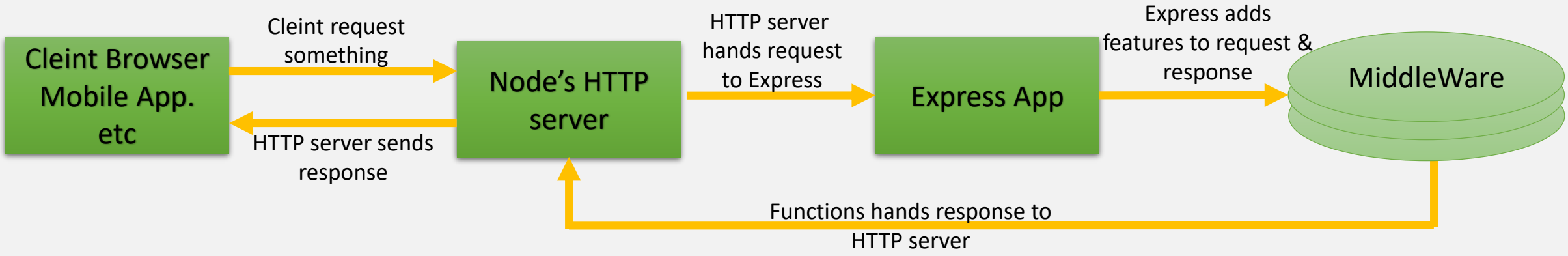
Getting started with Express

Express is perhaps the most popular Node.js web app framework. It's so popular that it's part of the MEAN Stack acronym. **MEAN** refers to **M**ongoDB, **E**xpressJS, **A**ngularJS, and **N**ode.js

What is Express

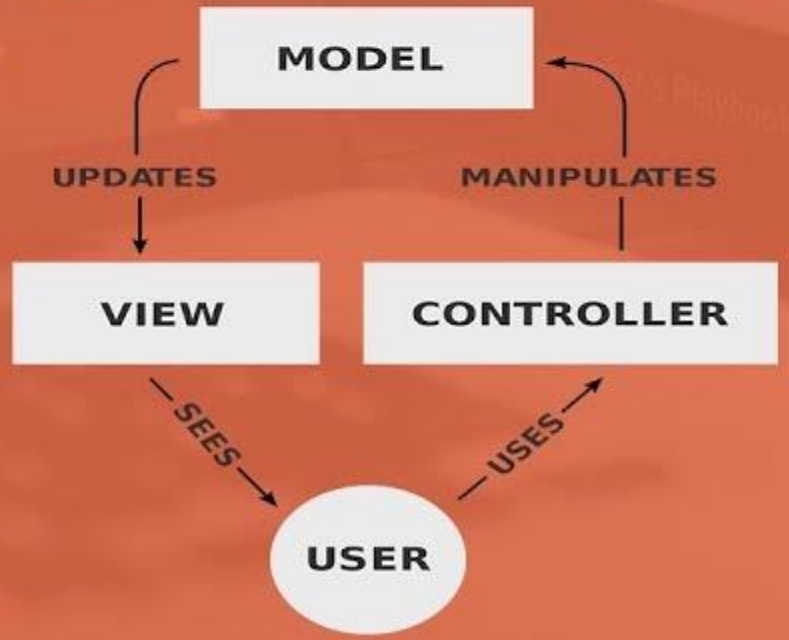
Express is a relatively small framework that sits on top of Node.js's web server functionality to simplify its APIs and add helpful new features.

- ✓It makes it easier to organize your application's functionality with middleware and routing
- ✓It adds helpful utilities to Node.js's HTTP objects
- ✓It facilitates the rendering of dynamic HTML views



What is The MVC?

Model - View - Controller



node JS

&

Express

Foundations of Express

Express is an abstraction layer on top of Node's built-in HTTP server. Express provides four major features

- ✓**Middleware**:where your requests flow through only one function, Express has a middleware stack, which is effectively an array of functions.
- ✓**Routing** : Routing is a lot like middleware, but the functions are called only when you visit a specific URL with a specific HTTP method.
- ✓**Extensions to request and response objects**: Express extends the request and response objects with extra methods and properties for developer convenience.
- ✓ **Views**:Views allow you to dynamically render HTML and Change HTML on the fly.

Middleware

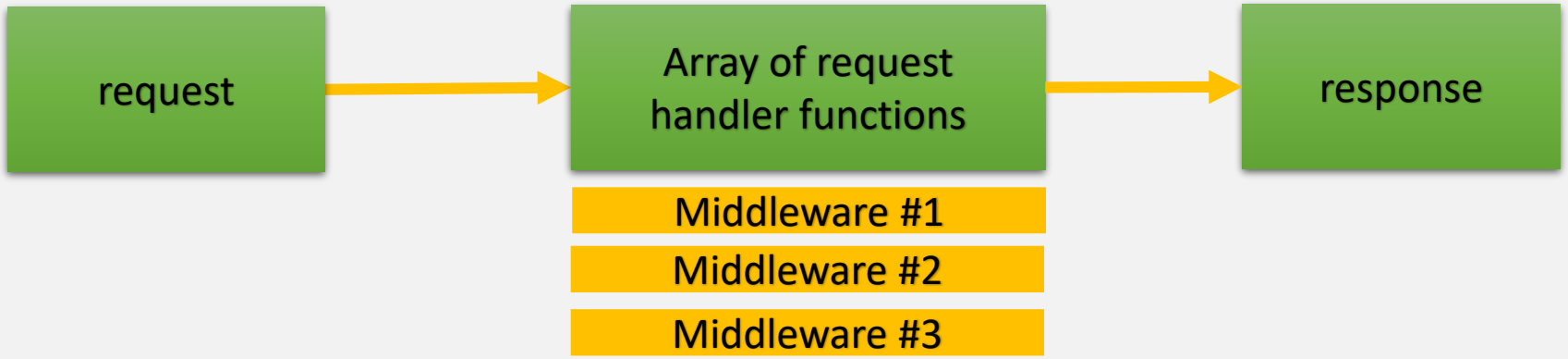
Middleware is very similar to the request handlers in Node.js, but middleware has one important difference: rather than having just one handler, middleware allows for many to happen in sequence.

```
let express=require("express");
let app=express();
//middleware
app.use((request,response)=>{
	console.log(request.url, request.method);
	response.end("Hello first Express app");
});
app.listen(8080,function(){
console.log("app starting listen to post 8080")
});
```

With middleware, rather than having your request pass through one function you write, it passes through an array of functions you write called a middleware stack.

In Node.js , request and response object are passed to only **one** function. But in Express these objects are passed through an **array** of functions called the middleware stack.

Express will start at the first function in the stack and continue in order down the stack.



Every function in this stack takes three arguments. The first two are request and response from before. They're given to you by Node, The third argument to each of these functions is itself a function, conventionally called next. When next is called, Express will go on to the next function in the stack.

Eventually, one of these functions in the stack must call res.end, which will end the request.

```
let express=require("express");
let app=express();
//middlewares
app.use((request,response,next)=>{
        console.log(request.url,request.method);
        next();
});
app.use((request,response)=>{
        response.writeHead(200,"text/plain");
        response.end("passing form last middleware")
});
app.listen(8080);
```

Error handling Middleware

These middleware functions take **four** arguments instead of two or three. The first one is the error (the argument passed into next), and the remainder are the three from before: req, res, and next.
While not enforced, error-handling middleware is conventionally placed at the end of your middleware stack, after all the normal middleware has been added.
If no errors happen, it'll be as if the error-handling middleware never existed. To reiterate more precisely, "no errors" means "next was never called with any arguments." If an error does happen, then Express will skip over all other middleware until the first error-handling middleware in the stack.

```
app.use((request,response,next)=>{
        if(condition)
        {
                next( new Error("Bad data"));
        }
        else
        {
                next();
        }
});

app.use((request,response)=>{
        response.writeHead(200,"text/plian");
        response.end("passing form last middleware")
});

app.use((err,req,res,next)=>{
        //handling Error
});
```

Built-in Middlewares

- ✓ Morgan module : containing info about loggers
- ✓ Body-parse module :handles parsing HTTP request bodies
- ✓ Cookie-parser module : used to parse http cookies
- ✓ static file Express Middleware : web server configured to serve the asset files in the public directory.

Built-in Static File Middleware

express.static is a function that returns a middleware function. It takes one argument: the path to the folder you'll be using for static files(path.join).
If the file exists at the path, it will send it. If not, it will call next and continue on to the next middleware in the stack.

```
let express = require("express");
let path = require("path");
let app = express();
let staticPath = path.join(__dirname,"folderName");
app.use(express.static(staticPath));
```

Routing

Routing is a way to map requests to specific handlers depending on their URL and HTTP verbs (GET ,POST ,PUT and DELETE).
They work just like middleware; it's a matter of when they're called.

```
app.get("/about",function(req,res,next){
    res.end("welcome to about page");
});

app.post("/register",function(req,res){
   res.end(" welcome to register page ");
});
```

Grabbing parameters to routes

These routes can get smarter. In addition to matching fixed routes, they can match more complex ones (imagine a regular expression or more complicated parsing)

```
app.get("/hello/:who",(req,res)=>{
    res.send(req.params.who);
});
```

It's no coincidence that this who is the specified part in the first route. Express will pull the value from the incoming URL and set it to the name you specify.

visiting localhost:8080/hello/earth for the following message: Hello, earth. Note that this won't work if you add something after the slash. For example, localhost:8080/hello/entire/earth will give a 404 error.

Grabbing query arguments

Another common way to dynamically pass information in URLs is to use query strings.

```
app.get("/login",function(req,res){
        res.end(" welcome to register page "+req.query.name);
});
```

Using Routers

Express 4 added **routers**, a feature to help ease these growing pains. **Router** allow s you to chunk your big app into numerous mini-apps that you can later put together.

"A router is an isolated instance of middleware and routes."

```
let userRouter=express.Router();

userRouter.use((req,res,next)=>{
        console.log("authorized");
        next();
}).get("/login",(req,res)=>{
        res.send("get");
}).post("/login",(req,res)=>{
        res.send("post");
});
app.use("/users",userRouter);
```

Views

Views are dynamically generated html. A number of different view engines are available. There's **EJS** (Embedded JavaScript), Handlebars, Pug, and more. All of these have one thing in common: at the end of the day, they spit out HTML.

```
npm install ejs --save.
```

Then:

```
app.set("view  engine","ejs");
app.set("views",path.join(__dirname,"/views"));
```

Simple View rendering

Main app.js file

```
app.use("/welcome",(req,res)=>{res.render("welcome",{message:" ITI "});
});
```

Welcome.ejs file

```
<html>
<head> Welcome Page</head>
<body>
       welcome every one to <%=message %> website
</body>
</html>
```

Simple View rendering

Main app.js file

```
app.use("/welcome",(req,res)=>{ res.locals.message="iti";
                                res.render("welcome");
});
```

Welcome.ejs file

```
<html>
<head> Welcome Page</head>
<body>
      welcome every one to <%=message %> website
</body>
</html>
```

User Sessions

Sessions are used to keep user authenticated on website. Express session package is the best choice

```
npm install express-session --save.
```

```
let session=require("express-session");
app.use(session({
      secret:"secret key",
      resave:false,
      saveUninitialized:true
}));
```

Thank You!