# NodeJS

Markus Veijola
October 2014

opiframe

## Introduction

* Node.js® is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications.
* Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices. (NodeJS web page).

## Introduction

* Node JS offers you two basic things:
    * Runtime environment (Google V8 VM)
    * Libraries for building the application
* You can download and install Node JS from here: http://nodejs.org/
* You can find also API documentation from the same place (please take a time to read the documentation)

## Introduction

- NodeJs Offers also:
  - Asynchronous I/O
  - Core done with C++, rest with JS
  - Handle several concurrent requests in a single process (low memory and CPU overhead).

## Introduction

- NodeJS is designed to be a server side scripting language: you can build RESTful web services easily and fast.
- Still it is not restricted to server side, you can also do client side scripting with NodeJS (like sockets).

## Installing

- Simply go to http://nodejs.org/ press the install button and run the installation application.
- After installation open command prompt (cmd) and execute next command to see that NodeJS installation was success:

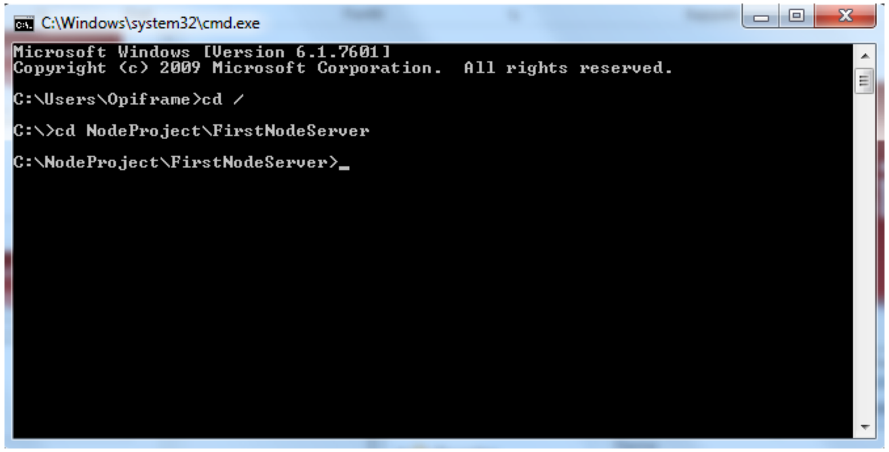
  ***node –version***
- If you see node version printed out you are ready to go.

# Node Package Manager

* **npm** (Node Package Manager) is the default package manager for Node.js. It comes with node installation.
* With package manager you can:
  * Install node applications that are available on the npm registry.
  * Manage your application dependencies.
* You can find npm registry from here: https://www.npmjs.org/

# Using npm & node

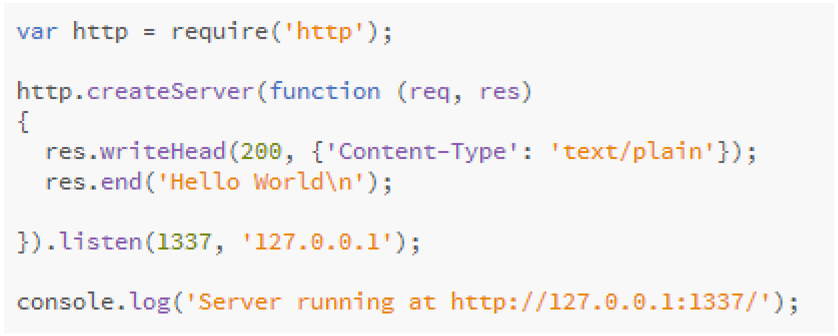
- **npm** is also a command line tool, so you have to open command prompt to use it.
- Before you do anything, create a workspace for you node project i.e. **c:\MyNodeProjects\FirstNodeServer**
- Then open CMD and browse to your workspace folder….


C:\Windows\system32\cmd.exe

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Opiframe>cd /

C:\>cd NodeProject\FirstNodeServer

C:\NodeProject\FirstNodeServer>_
```

# Using npm & node

- Open your favorite JavaScript editor (like brackets) and create app.js file with next content in your working directory…



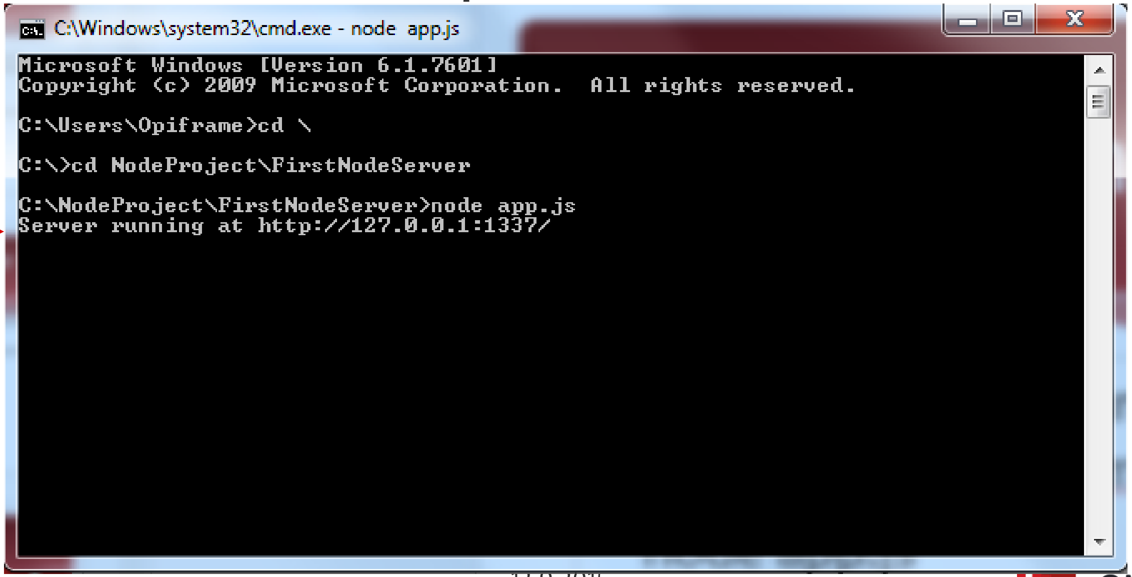
```
var http = require('http');

http.createServer(function (req, res)
{
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');

}).listen(1337, '127.0.0.1');

console.log('Server running at http://127.0.0.1:1337/');
```

# Launch The Server

- Now we can test our simple server we just created. Write next command in cmd…

  node app.js

- You should see that next text is printed in cmd and server is running…

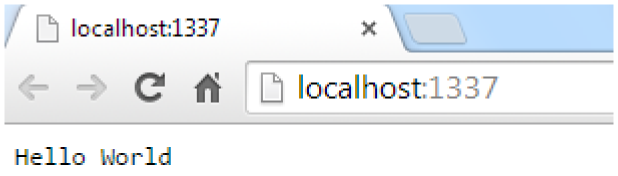


C:\Windows\system32\cmd.exe - node app.js

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Opiframe>cd \

C:\>cd NodeProject\FirstNodeServer

C:\NodeProject\FirstNodeServer>node app.js
Server running at http://127.0.0.1:1337/
```

# Launch The Server

- Now to test the application we made, open your favourite browser (chrome, firefox etc..) and write next url: http://localhost:1337/
- You should see that browser renders the string hello world as expected if you look our node code…



```
Hello World
```

opiframe

## Stop The Server

- Now if you make some modification in the code you ALWAYS must restart the server before modifications have any affect.
- To shut down our server activate the command prompt and press **Ctrl - c**

# nodemon

- To start and restart the server all the time can be overwhelming (at least if you have lots of bugs).
- To avoid this we can use npm to install utility software to help us. One of them is ***nodemon***.
- nodemon is  application that automatically load ANY changes you make in your application and reloads the server. This means that you don't' have to restart the server after every change. All you have to remember is SAVE you changes before they make any affect.

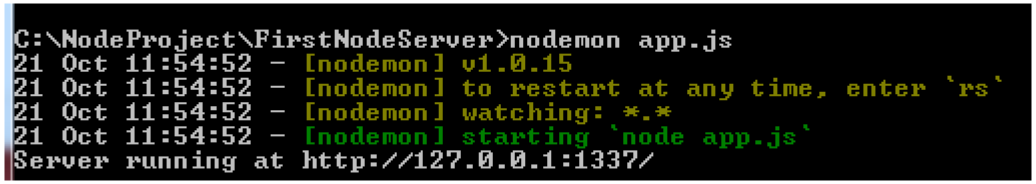
# Installing nodemon

* Again activate your command prompt and write next command to install nodemon:

  ***npm install nodemon***

* After the installation is complete start the server with next command:

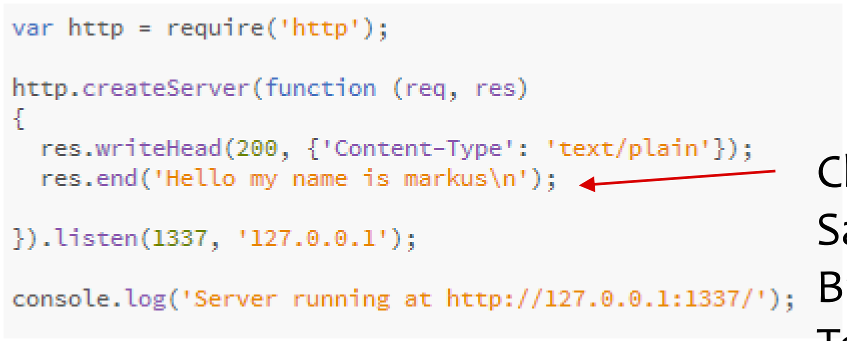
  ***nodemon app.js***



```
C:\NodeProject\FirstNodeServer>nodemon app.js
21 Oct 11:54:52 - [nodemon] v1.0.15
21 Oct 11:54:52 - [nodemon] to restart at any time, enter `rs`
21 Oct 11:54:52 - [nodemon] watching: *.*
21 Oct 11:54:52 - [nodemon] starting `node app.js`
Server running at http://127.0.0.1:1337/
```

# nodemon

* Now make a little change in code save the file and then refresh the browser to see if the change can be seen…



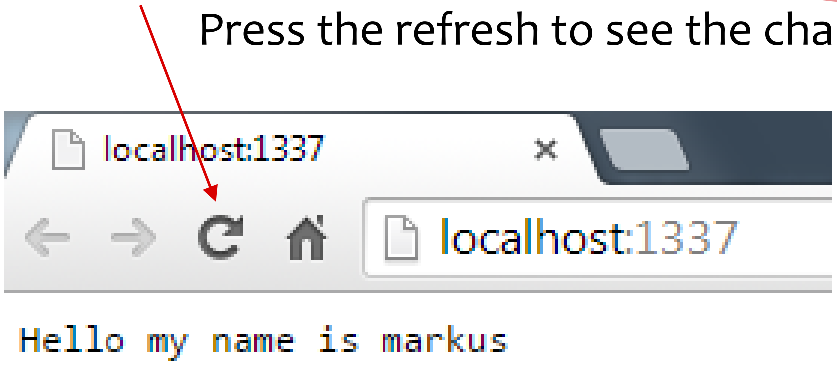
```
var http = require('http');

http.createServer(function (req, res)
{
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello my name is markus\n');

}).listen(1337, '127.0.0.1');

console.log('Server running at http://127.0.0.1:1337/');
```

Change the text in here.
Save the changes. Press refresh on
Browser to see if you can see the same
Text there…

opiframe

# nodemon

Press the refresh to see the changes….



```
Hello my name is markus
```

12.9.2015

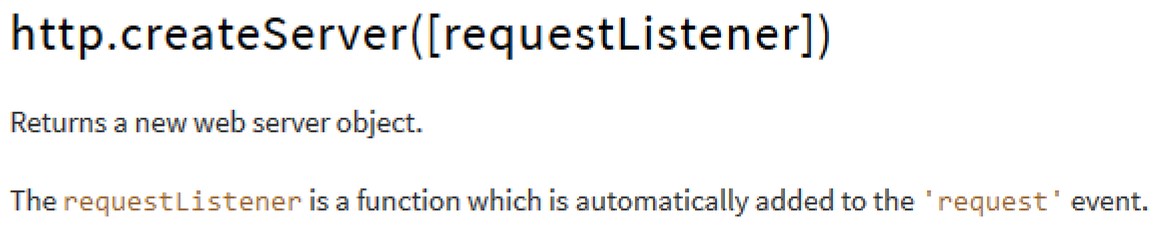
## The Code

* It is time to walkthrough the code. The first line you see in our application is `var http = require('http');`
* The require() function is used to load something called **modules** in node. You pass a module name as an argument for this function. The require function return the module object, which we store in our variable called ‘http’.
* The modules in node (and any other framework) contains some crucial functionality you need when implementing server side applications.
* The ‘http’ module for example contains a function called createServer(), which allows us to create a basic http server functionality.

opiframe

## The Code

* You can find the module documentation from here: [http://nodejs.org/api/](http://nodejs.org/api/)
* If you open the HTTP module documentation you can find the createServer() function…



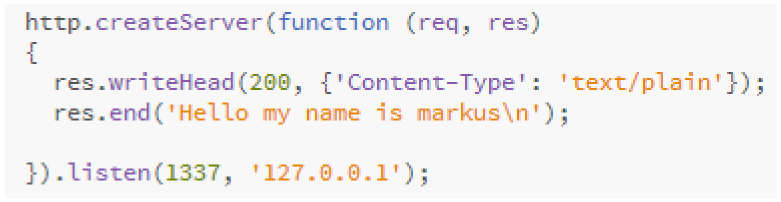
```
http.createServer([requestListener])
```

Returns a new web server object.

The `requestListener` is a function which is automatically added to the `'request'` event.

# The Code

- The [requestListener] is the callback function containing the req and res (request and response) objects. The req object contains everything the client send to our server (in our case the browser) and the res object is the object you can use to send something back to client. In our case we use res object to send “hello my name is markus” string to browser.



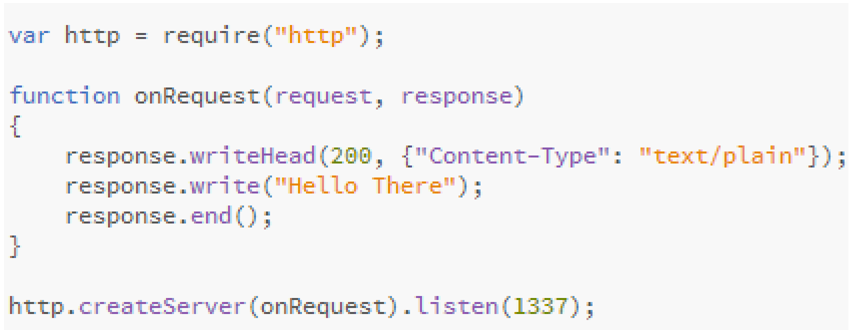
```
http.createServer(function (req, res)
{
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello my name is markus\n');

}).listen(1337, '127.0.0.1');
```

## The Code

- This is important to notice. As you can read from docs the createServer() function RETURNS new web server object. So after the function call you can see the '.' notation followed by the listen function call. This is called function 'chaining'. The createServer() function returns an object where we attach directly a function call listen() with '.' operator.
- The listen function begins accepting connections on the specified port and hostname. If the hostname is omitted, the server will accept connections directed to any IPv4 address (INADDR_ANY).
- In our case the port was '1337' and hostname was '127.0.0.1'
- The host name '127.0.0.1' aka 'loop back' is bind to name 'localhost' so that's why you write the address localhost:1337 to connect our node server.

# Conclusion

- Just writing a few lines of code you managed to create a HTTP server with NodeJS.
- Here you can see how powerful and easy NodeJS eventually is.
- As a bonus here is another way to create a same server…

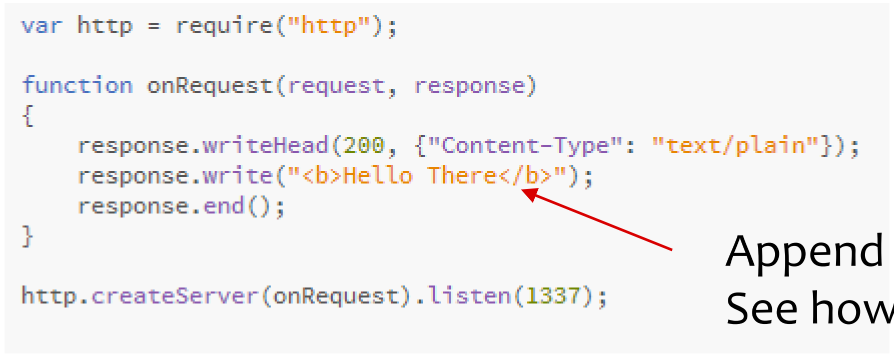


```
var http = require("http");

function onRequest(request, response)
{
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello There");
    response.end();
}

http.createServer(onRequest).listen(1337);
```

# Meaning of Headers

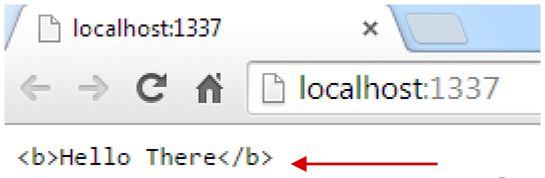
* Usually when you send response from server back to browser (or any client) you need to set a header for it.
* Headers are like instructions to browser or server: “What to do with the data” or “How to display the data” or “How to handle the data” etc.
* There are many headers in HTTP standard and you can find them all here: http://en.wikipedia.org/wiki/List_of_HTTP_header_fields
* In our example we set one header in our response “Content-Type” : “text/plain”. This header instructs the browser to handle the data as plain text. What this then means. Let’s change the code a little bit…

# Meaning of Headers



```
var http = require("http");

function onRequest(request, response)
{
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("<b>Hello There</b>");
    response.end();
}

http.createServer(onRequest).listen(1337);
```

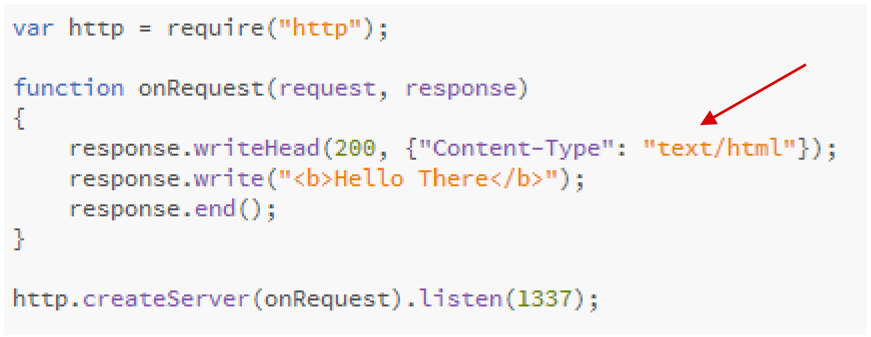
Append some HTML markup to our response to
See how browser handles it…



```
<b>Hello There</b>
```

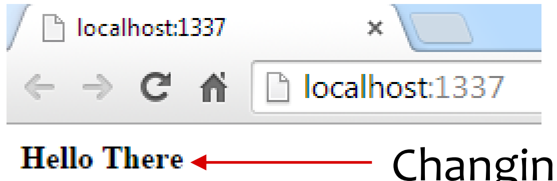
As you can see browser ignores the HTML tags and don't render the text in bold. This is because our server told it to do so by setting the content type to plain/text

# Meaning of Headers



```
var http = require("http");

function onRequest(request, response)
{
    response.writeHead(200, {"Content-Type": "text/html"});
    response.write("<b>Hello There</b>");
    response.end();
}

http.createServer(onRequest).listen(1337);
```

Change the content type to text/html…



**Hello There**

Changing the content type to text/html will tell the browser to look up the html tags from the content and process them (actually the text/html tells to browser that the content is html). As you can see the text is now bold as expected.

# Meaning of Headers

* So the headers are very important. Setting header wrong in response or request can make our application to look or work in wrong way.

# Express

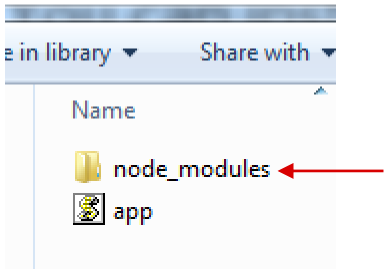
* Express is one of the modules created to make things even more simple than you have seen so far.
* Expression is fast and small server-side web development framework built on connect (see more https://github.com/senchalabs/connect).
* You can do basically the same things as with node.js, but with less code writing.

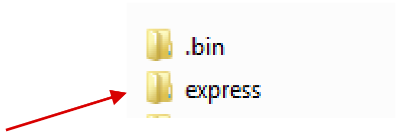
## Installing Express

- You can install the express module by using the following command:

  **npm install express --save**

- After the installation you should see that "node_modules" folder has appeared in your working directory



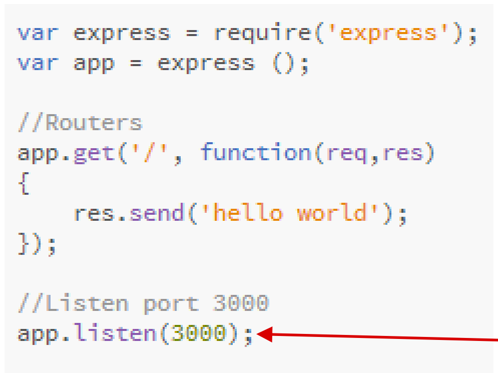


- And the node_modules folder contains the express folder…

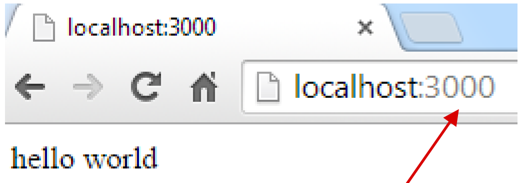
opiframe

## Express

* And after installing the module we can use it. Replace the previous code with the next code sample…



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

//Routers
app.get('/', function(req,res)
{
    res.send('hello world');
});

//Listen port 3000
app.listen(3000);
```



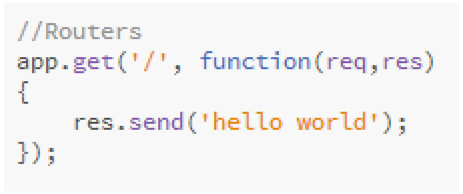
```
hello world
```

Note the port number!

opiframe

# Express

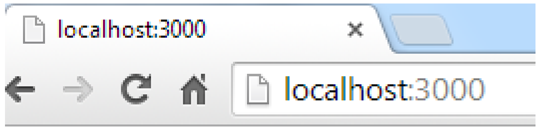
* The first line is familiar we just load the express module that we previously installed.
* The line `var app = express ();` creates the express server which we store in variable ‘app’.
* Now we can use app to create our routes, define the middleware's and configurations for our app.
* We create just one route using the get function… .



```
//Routers
app.get('/', function(req,res)
{
    res.send('hello world');
});
```

# What are Routes?

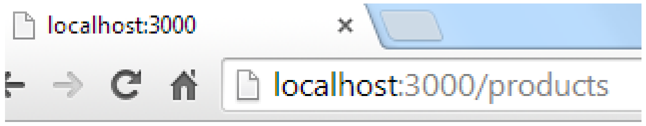
- Routes are URL schema, **which describe the interfaces for making requests to your web app.**
- Combining an HTTP request method (a.k.a. **HTTP verb**) and a path pattern, you define URLs in your app.
- For example, next URL has a path '/' (also called the root path) .



localhost:3000

localhost:3000

- If you write next url then the path is '/products'

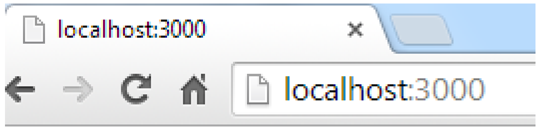


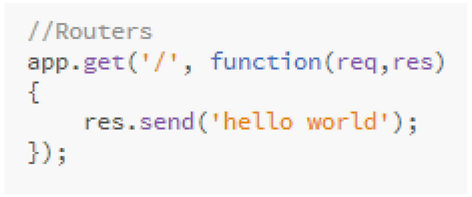
localhost:3000

localhost:3000/products

# What are Routes?

- For every path, you must have a corresponding handler for that in your server side. In ur case we have just one for the url that has a root path '/'





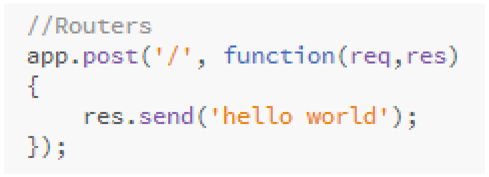
```
//Routers
app.get('/', function(req,res)
{
    res.send('hello world');
});
```

- So the first argument for get() function is the path it handles, the second one is the function called when this path is handled. The req and res objects are the same as explained previously.

opiframe

# Why get() function?

* There are many methods a browser or other client can use to send an http request. These are: GET, POST, PUT, DELETE, HEAD, TRACE, OPTIONS, CONNECT and PATCH.
* Each HTTP method has a corresponding method in express server. For example if browsers sends something to your server using POST method (i.e. sending a form data) you must handle it in your server using the post method instead of get.



```
//Routers
app.post('/', function(req,res)
{
    res.send('hello world');
});
```

# Express

- Normally when browser send a request it uses the GET method i.e. for fetching some .html page from server.
- Other methods are used to send, update or delete the data from server. In web applications usually the person implementing the app will use these methods internally. Examples will follow don’t worry if this is a little bit confusing.

# Express & Static files

* To understand what static files means let’s look a simple index.html markup code and the content of mystyle.css file.

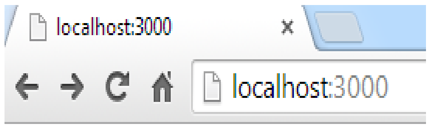
```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <script src="myscript.js"></script>
        <link href="mystyle.css" type="text/css" rel="stylesheet">
    </head>
    <body>
        <div>
            <h1>Hello World</h1>
        </div>
    </body>
</html>
```

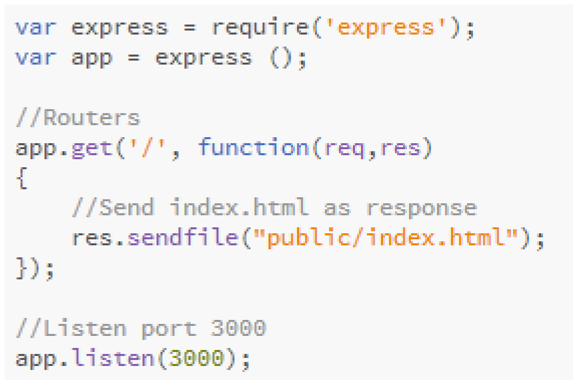
```
h1{
    color: red;
}
```

opiframe

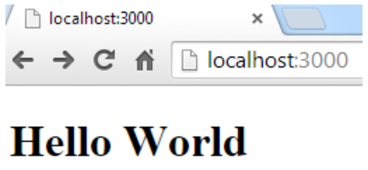
# Express & Static files

* Sending an .html file as a response to browser from our express server is simple…





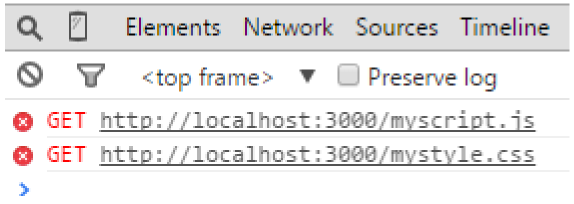
```
var express = require('express');
var app = express ();

//Routers
app.get('/', function(req,res)
{
    //Send index.html as response
    res.sendfile("public/index.html");
});

//Listen port 3000
app.listen(3000);
```



opiframe

## Express & Static files

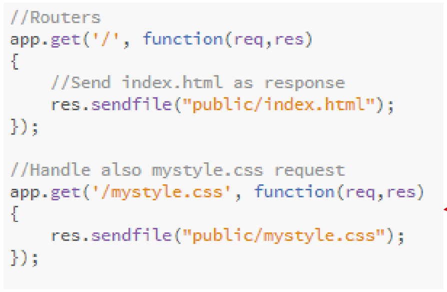
- But as you can see the styles we defined had no affect, the color of Hello World should be red, but instead it is black…and if you look the JavaScript console from browser there is a next error…



```
GET http://localhost:3000/myscript.js
GET http://localhost:3000/mystyle.css
>
```

# Express & Static files

* The browser makes a separate GET requests to script, css, images etc. on to server. You can see from JavaScript console that when browser gets to this line parsing the .html file `<link href="mystyle.css" type="text/css" rel="stylesheet">` it makes a next request to our server… `GET http://localhost:3000/mystyle.css`. This expects that our server contains a router for the path /mystyle.css but we don't have one… of course you could do following fix to our server to get it work…



```
//Routers
app.get('/', function(req,res)
{
    //Send index.html as response
    res.sendfile("public/index.html");
});

//Handle also mystyle.css request
app.get('/mystyle.css', function(req,res)
{
    res.sendfile("public/mystyle.css");
});
```

You can fix like this

opiframe

## Express & Static files

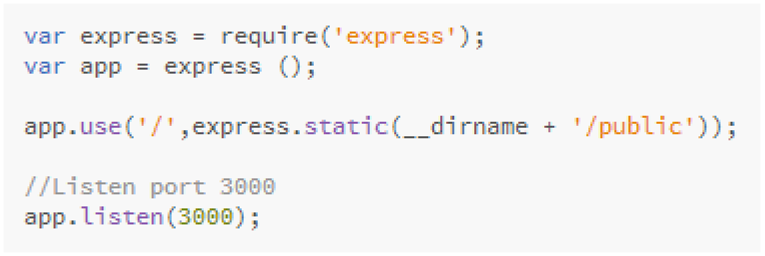
- But consider that you have 10 different .css and .js files. Then you have to make separate routes for each of them and that is not very handy dandy.
- To make this work much better we can tell express to use some folder as static context for our static files. The express automatically handles all the static files from that folder and for defined path…

## Middleware's

- Middleware’s are something you can take into use when ever some ‘extra’ service is needed from the module like express module.
- For  example: it might be a case that you don’t want to serve static files from your express server, then you don’t use the ‘static’ middleware from the express.
- Why middleware’s? Because of the performance. Every ‘extra’ service like serving static files, handling cookies etc. are arranged to different middleware parts. If you use them in your server, then you have to take a particular middleware in use.

## Middleware's

- You use ‘use()’ function to associate with some middleware. We want to take the ‘static’ middleware in use, so we need to add next line of code in our server app..



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

app.use('/',express.static(__dirname + '/public'));

//Listen port 3000
app.listen(3000);
```

- The first argument ‘/’ is the path. It says to express “when ever a request comes to path ‘/’ you handle it. The second argument defines the path for static files. I have stored all my .css, .js etc static files in folder ‘public’. The variable __dirname is a built-in variable that contains the full path to our application that is in my case C:\NodeProject\FirstNodeServer. So the full path for our static function is C:\NodeProject\FirstNodeServer\public

## Middleware's

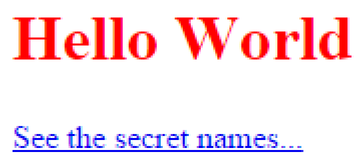
- No every static file you use in your index.html file is handled automatically by express if you just put it in your public folder.
- NOTE! Every middleware must be defined in use BEFORE any router is made!!!!

# More Routes!!

- Let's make an example where you really get into routes and how they work between the server and the browser.
- First we modify our index.html file for a little bit by appending a link (<a>) element to it. Pay attention to <a> elements 'href' attribute value.

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <script src="myscript.js"></script>
        <link href="mystyle.css" type="text/css" rel="stylesheet">
    </head>
    <body>
        <div>
            <h1>Hello World</h1>
            <a href="/names">See the secret names...</a>
        </div>
    </body>
</html>
```

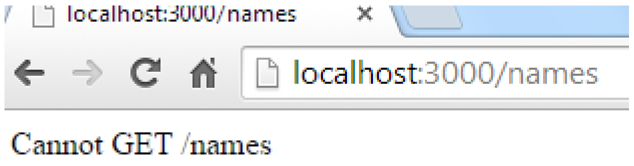
Route...


Hello World

See the secret names...

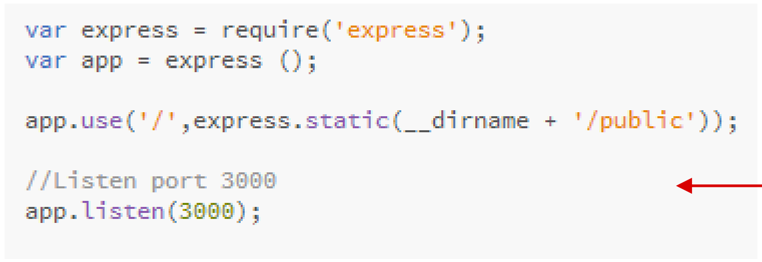
# More Routes!!

* Now if you click the link you see the next result…



```
Cannot GET /names
```

* This is because our server does not have a router for path /names.

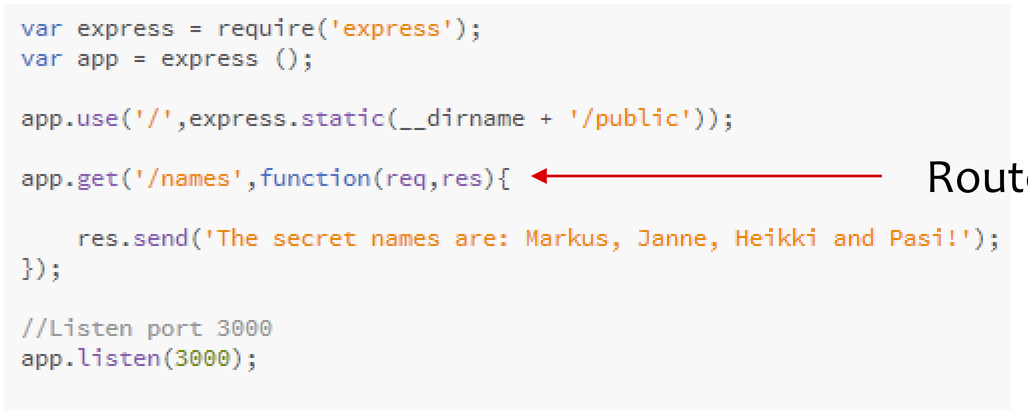


```
var express = require('express');
var app = express ();

app.use('/',express.static(__dirname + '/public'));

//Listen port 3000
app.listen(3000);
```

There is no routers at ALL!!!

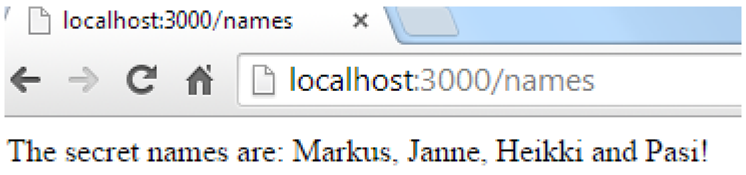
## More Routes!!

- To fix it you simply do the following….



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

app.use('/',express.static(__dirname + '/public'));

app.get('/names',function(req,res){

    res.send('The secret names are: Markus, Janne, Heikki and Pasi!');
});

//Listen port 3000
app.listen(3000);
```

Router for path "/name"



localhost:3000/names

localhost:3000/names

The secret names are: Markus, Janne, Heikki and Pasi!

opiframe

## Creating Modules

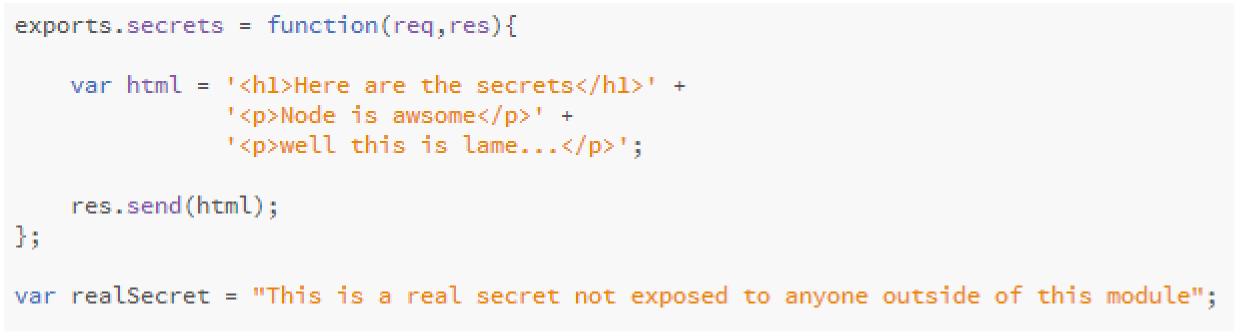
- These few very simple examples we dealt should give you a basic idea how  NodeJS and express framework works.
- It is not very hard to maintain a server that have just few pieces of functionality (like in our examples).
- But start to consider that you have hundreds of routers with very difficult logic in your server.
- You probably don’t want to put all that stuff in one file, instead you want to separate that logic in different modules.

# Creating Modules

* Modules are simply a separate JavaScript files constructed in particular way.
* An example would be fare enough to get you going with modules.
* First of all we need to create a external JavaScript file for it.
* The naming of this file is important, because it will be our module name. Be aware of this when designing modules.
* My module name will be “secret” so the JavaScript file name will be secret.js
* I will place my modules to ‘modules’ folder in my working directory.

# Creating Modules

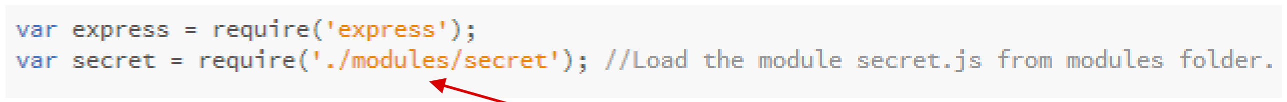
- The keyword to expose something from our module is 'exports'. Everything that is followed keyword exports comes as 'public' for other modules. Anything defined inside a module WITHOUT the keyword export can be considered as private.
- My secret.js contains these few line of code…



```
exports.secrets = function(req,res){

    var html = '<h1>Here are the secrets</h1>' +
               '<p>Node is awsome</p>' +
               '<p>well this is lame...</p>';

    res.send(html);
};

var realSecret = "This is a real secret not exposed to anyone outside of this module";
```

## Using the module

* The question is how we use that secret.js module. Well we use the require() function. I Made the next modification to my app.js file…



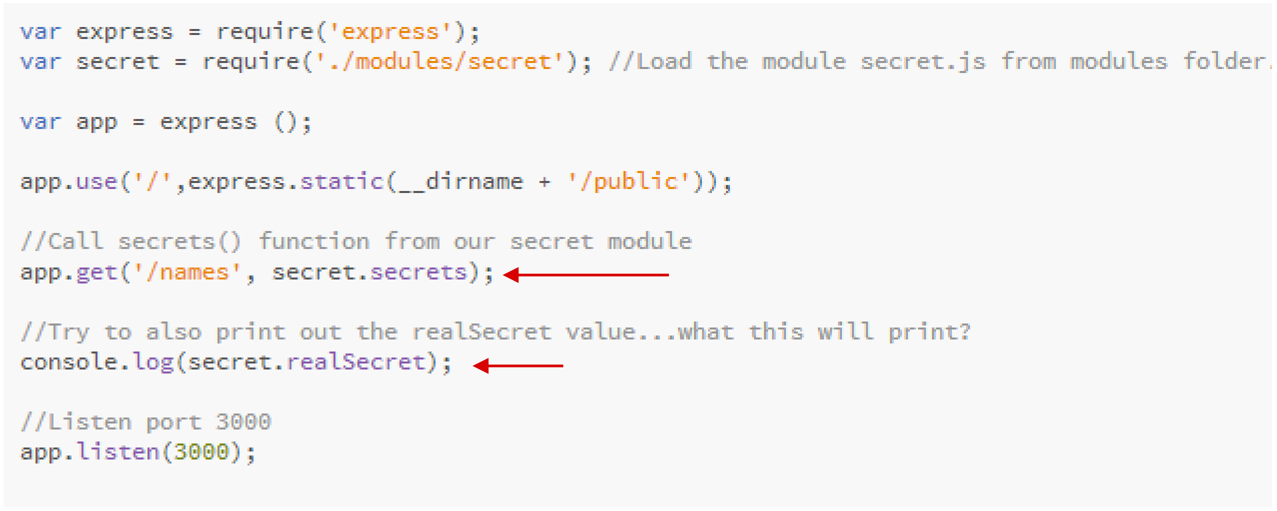
```
var express = require('express');
var secret = require('./modules/secret'); //Load the module secret.js from modules folder.
```

Load our module…

opiframe

# Using the module

* Then I use it…

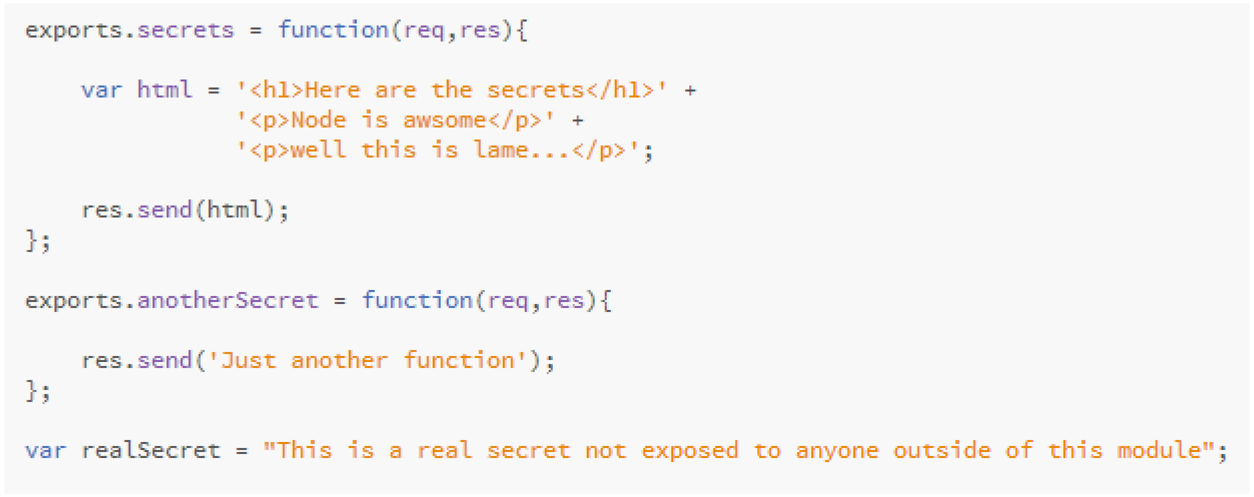


```
var express = require('express');
var secret = require('./modules/secret'); //Load the module secret.js from modules folder

var app = express ();

app.use('/',express.static(__dirname + '/public'));

//Call secrets() function from our secret module
app.get('/names', secret.secrets);

//Try to also print out the realSecret value...what this will print?
console.log(secret.realSecret);

//Listen port 3000
app.listen(3000);
```

opiframe

# Using the part of the module

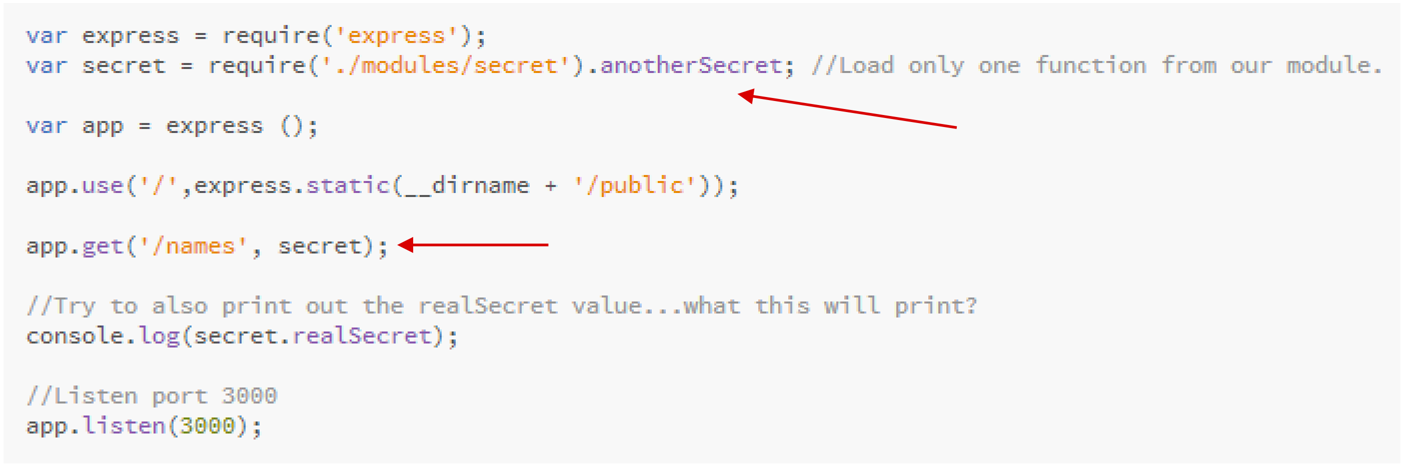
* Let’s say that I have several exported function in module, but I just need one of them. Can I just crap one of them? Answer is yes…
* Think that we have next module



```
exports.secrets = function(req,res){

    var html = '<h1>Here are the secrets</h1>' +
               '<p>Node is awsome</p>' +
               '<p>well this is lame...</p>';

    res.send(html);
};

exports.anotherSecret = function(req,res){

    res.send('Just another function');
};

var realSecret = "This is a real secret not exposed to anyone outside of this module";
```

# Using the part of the module

* We want to use just the function anotherSecret from that module in our app.js… how we can do that. Well see the next example…



```
var express = require('express');
var secret = require('./modules/secret').anotherSecret; //Load only one function from our module.

var app = express ();

app.use('/',express.static(__dirname + '/public'));

app.get('/names', secret);

//Try to also print out the realSecret value...what this will print?
console.log(secret.realSecret);

//Listen port 3000
app.listen(3000);
```

## Handling the POST method

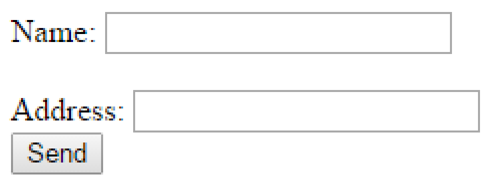
* One of the most common things to do in web apps is to post data from client to server. Mostly this happens via the HTML forms, but other ways are also possible (like AJAX and JSON).
* Next we see how we can handle this kind of communication between the client and the server.

## Handling the POST method

* First we need a form in our index.html file. Pay attention to ‘name’ attribute values in <input> elements. Those are important when we parse the data in server. Note also that form method is post and action is “/user_data”. Also note the input element which type is submit. This is important in forms if you want to send the data to server…

# Handling the POST method

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <script src="myscript.js"></script>
        <link href="mystyle.css" type="text/css" rel="stylesheet">
    </head>
    <body>
        <div>
            <form method="post" action="/user_data">
                Name: <input type="text" autofocus name="user_name"><br/><br/>
                Address: <input type="text" name="user_address"><br/>
                <input type="submit" value="Send">
            </form>
        </div>
    </body>
</html>
```

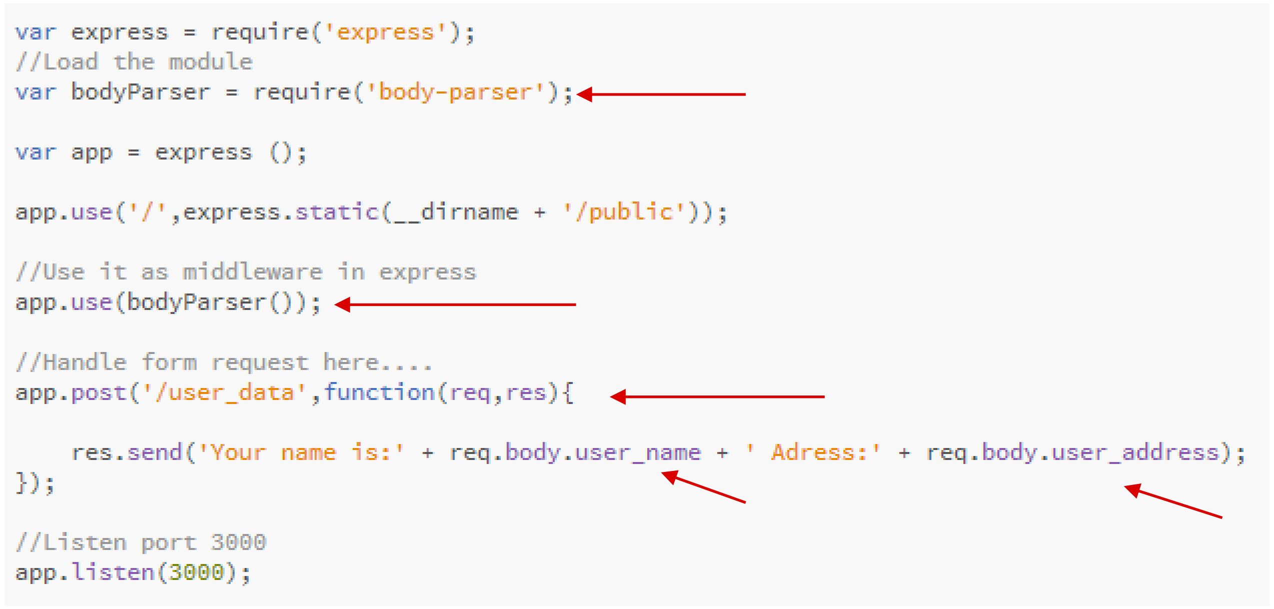


opiframe

## Handling the POST method

- Then the server side. To handle POST method in our server we need a middleware for that. The name of the middleware is bodyParser.
- Also we need a post router to handle the path ‘/user_data’
- The bodyParser has to be installed separately using npm tool: **npm install body-parser**

# Handling the POST method



```
var express = require('express');
//Load the module
var bodyParser = require('body-parser');

var app = express ();

app.use('/',express.static(__dirname + '/public'));

//Use it as middleware in express
app.use(bodyParser());

//Handle form request here....
app.post('/user_data',function(req,res){

    res.send('Your name is:' + req.body.user_name + ' Adress:' + req.body.user_address);
});

//Listen port 3000
app.listen(3000);
```

opiframe

## AJAX & JSON

* What if you want to send something from client (browser) to server, but don’t want to use form? How you can do it? There are two options: AJAX or Sockets.
* An example here would be nice eh?
* First we make all the needed stuff in client side. We need to write some client side JavaScript.

# AJAX & JSON

Index.html

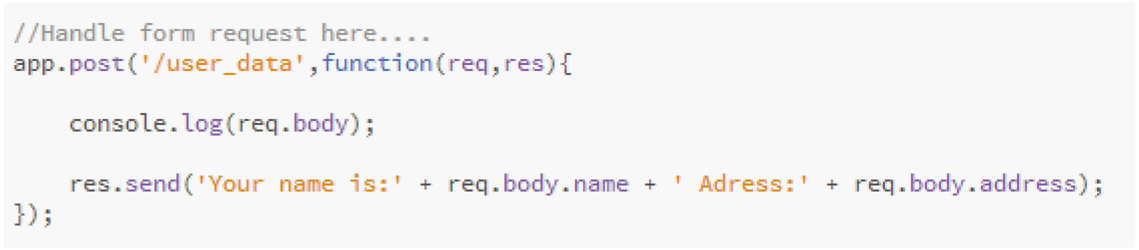
```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <script src="myscript.js"></script>
        <link href="mystyle.css" type="text/css" rel="stylesheet">
    </head>
    <body>
        <div>
            <p id="username">Markus Veijola</p>
            <p id="address">Rautatienkatu 40 as 30</p>
            <p id="email">markus.veijola@gmail.com</p>
        </div>
    </body>
</html>
```

Myscript.js

```
window.onload = function(event){

    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = responseReady;

    xhr.open('POST','/user_data',true);

    //Need to set this header because the content in json object
    //you need to set headers AFTER open function
    xhr.setRequestHeader("Content-type","application/json");

    var data = {

        name:username.innerHTML,
        address:address.innerHTML,
        email:email.innerHTML
    }

    xhr.send(JSON.stringify(data));

};

function responseReady(event){

    if(event.target.readyState == 4 && event.target.status == 200)
    {
        console.log(event.target.responseText);
    }
}
```

# AJAX & JSON

* Server side…



```
//Handle form request here....
app.post('/user_data',function(req,res){

    console.log(req.body);

    res.send('Your name is:' + req.body.name + ' Adress:' + req.body.address);
});
```

opiframe

## express.io == Sockets

- There are couple of ways to use sockets socket.io and express.io where the latter is based on socket.io.
- express.io makes it a little bit simpler to use sockets.
- First you need to install express.io using npm: **npm install express.io**
- Then make your express.io server….

# express.io == Sockets

* Server side code….

```
var express = require('express');
var bodyParser = require('body-parser');
var app = require('express.io')();


app.http().io();

app.use('/',express.static(__dirname + '/public'));

//Use it as middleware in express
app.use(bodyParser());

//Handle form request here....
app.post('/user_data',function(req,res){

    console.log(req.body);

    res.send('Your name is:' + req.body.name + ' Adress:' + req.body.address);
});

app.io.route('client_ready',function(req){
    console.log(req.data);
    req.io.emit('my_response',{message:'here you go ' + req.data.message});
});

//Listen port 3000
app.listen(3000);
```

## express.io == Sockets

* Client side…

Index.html

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <script src="/socket.io/socket.io.js"></script>
        <script src="myscript.js"></script>
        <link href="mystyle.css" type="text/css" rel="stylesheet">
    </head>
    <body>
        <div>
            <p id="username">Markus Veijola</p>
            <p id="address">Rautatienkatu 40 as 30</p>
            <p id="email">markus.veijola@gmail.com</p>
        </div>
    </body>
</html>
```

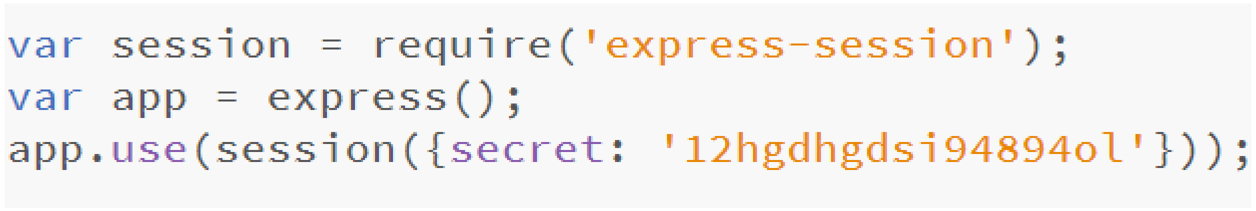
myscript.js

```
window.onload = function(event){

    io = io.connect();

    // Emit ready event.
    io.emit('client_ready',{message:'here you go'});

    io.on('my_response', function(data) {
        alert(data.message)
    });

};
```

## Session Handling

* Using sessions to keep track of users as they journey through your site is key to any respectable application.
* Luckily, as usual, using Express with your Node.js application makes it super simple to get sessions up and running.
* First of all you need to install session module with command: npm install express-session
* The do the following to use the module…

## Session Handling

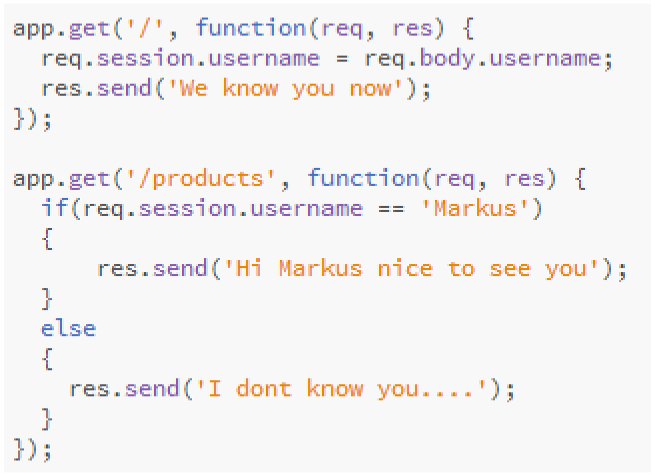


```
var session = require('express-session');
var app = express();
app.use(session({secret: '12hgdhgdsi94894ol'}));
```

## Session Handling

* We also provide a secret to the session initializer, which provides a little more security for our session data. Of course you might what to use a key that is a little more secure.
* Sessions are accessible through the request object in each route. You can get and set properties just like you would when handling an object normally. For example, lets set some session data in the awesome route.

# Session Handling



```
app.get('/', function(req, res) {
  req.session.username = req.body.username;
  res.send('We know you now');
});

app.get('/products', function(req, res) {
  if(req.session.username == 'Markus')
  {
      res.send('Hi Markus nice to see you');
  }
  else
  {
    res.send('I dont know you....');
  }
});
```

## package.json

* When you are about publishing your application to big audience (in some cloud environment for example) you have to create a package.json file for your project.
* This file is a project file, where you can define multiple things about your application. Most of all at least the dependencies.
* You can run command **npm init** to create a very basic “template” package.json file for your application.

## Example

```
{
  "name": "hellonode",
  "subdomain": "hellonode",
  "scripts": {
    "start": "server.js"
  },
  "dependencies": {
    "express": "*",
    "jade": "0.5.x",
    "express.io": "1.9.x"
  },
  "version": "0.0.0",
  "engines": {
    "node": "v0.8.x"
  }
}
```

## Where it is?

- You always place your package.json file in the root folder of your project.
- When you execute next command in your root folder containing the project file: **npm install**

  it will install automatically all the dependencies etc.
- There is also this ‘start’ attribute which is crucial when you want to publish your app in cloud. From this attribute the cloud environment knows which file of your program contains the “start up code” where you set up your server up and running.