

Node.js Web Application Week 6 Lecture

**COMMONWEALTH OF
Copyright Regulations 1969
WARNING**

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Outline

- Node.js execution
- Node.js application structure
- Express.js basics
 - Routing
 - Middleware
 - Template Engines

Node.js Motivation

- One of the design goals of Node.js and also the reason for its popularity is *performance*
- Node.js proposed a very different server architecture to most existing web server/application servers
- Any web server is expected to handle a large number of user requests concurrently
 - In early days, each request is handled by a process
 - Most modern servers handle request by thread: each request is handled by a separate thread
 - Server usually maintains a thread pool to avoid expensive thread creation/destroy operation
 - Each thread occupy a thread, which is allocated fraction system resources for sometime.
 - Server is restricted by the maximum thread number it can have concurrently
 - Node.js server runs on a single thread, very light footprint on system resource.

Single Threaded Execution

- Node.js server handles all requests in a **single thread**
- JavaScript is designed to run in a single thread, both in browser and on server side
 - It relies on event loop and callback mechanisms to achieve non-blocking, asynchronous execution
 - For longer running processes that do not occupy the main JavaScript resources, there is at least an event signaling the end of the that process, developers can write callback function for that “end” event
 - E.g.
 - Disk IO is long running and is done by OS’s file management component.
 - Database call is also long running and is executed mainly on database server
 - Downloading post request data may take long time and mainly uses network resources.

The IO Scaling Problem

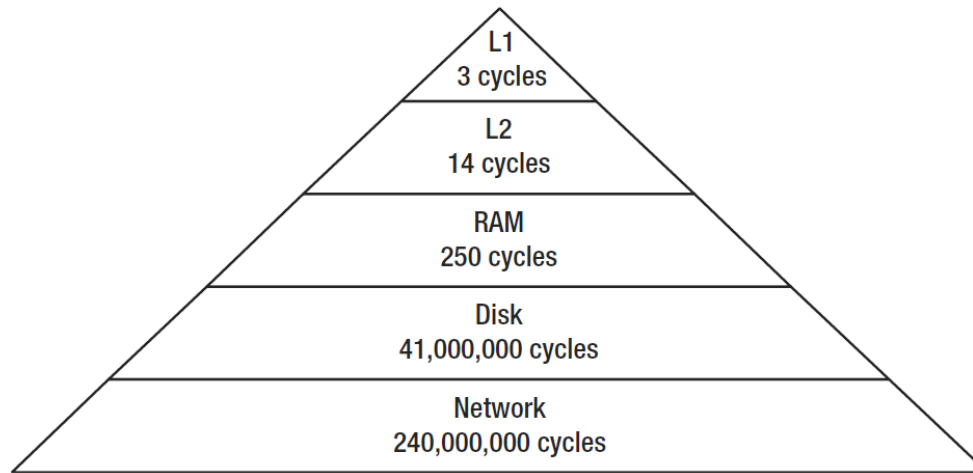


Figure 2-1. Comparing common I/O sources

Traditional Server using processes

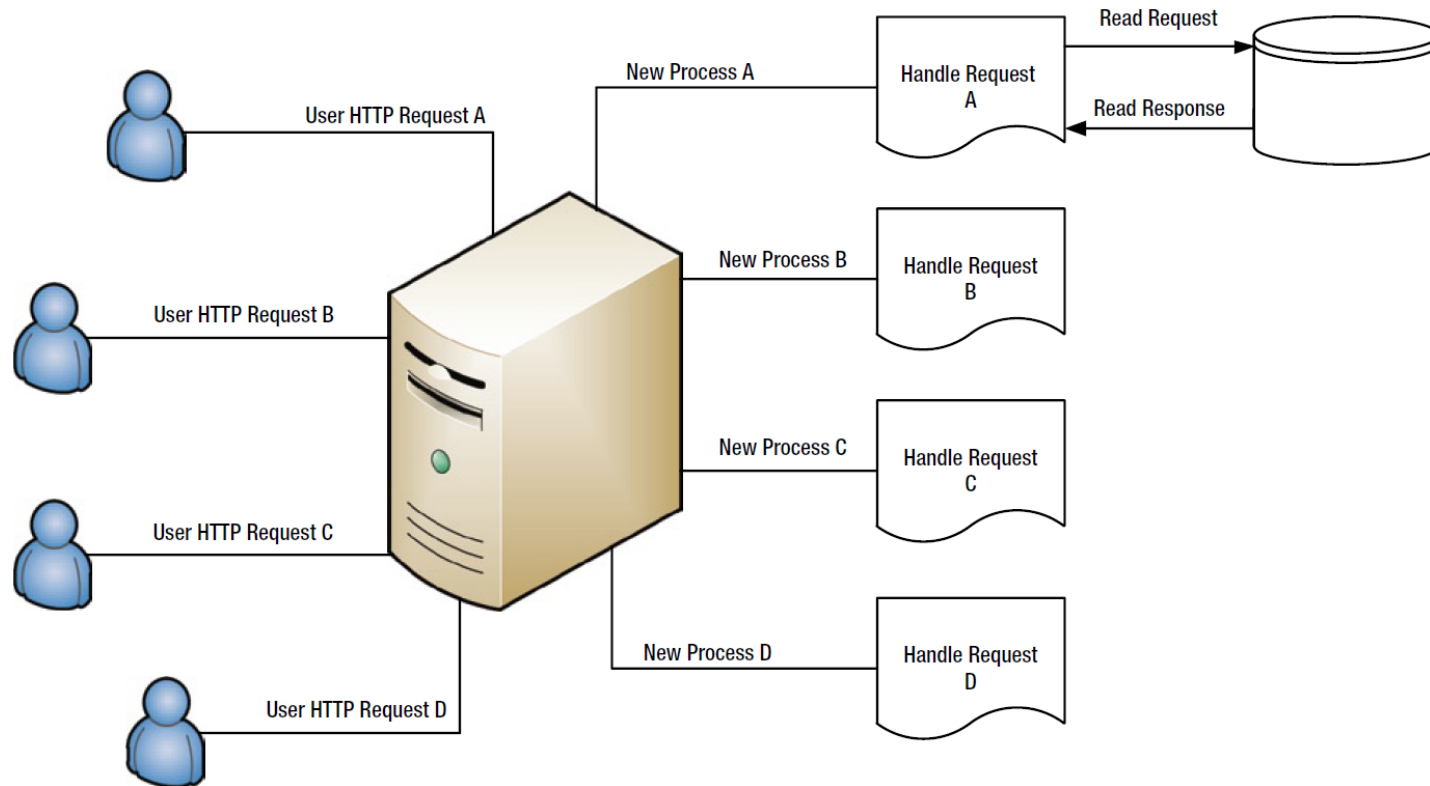
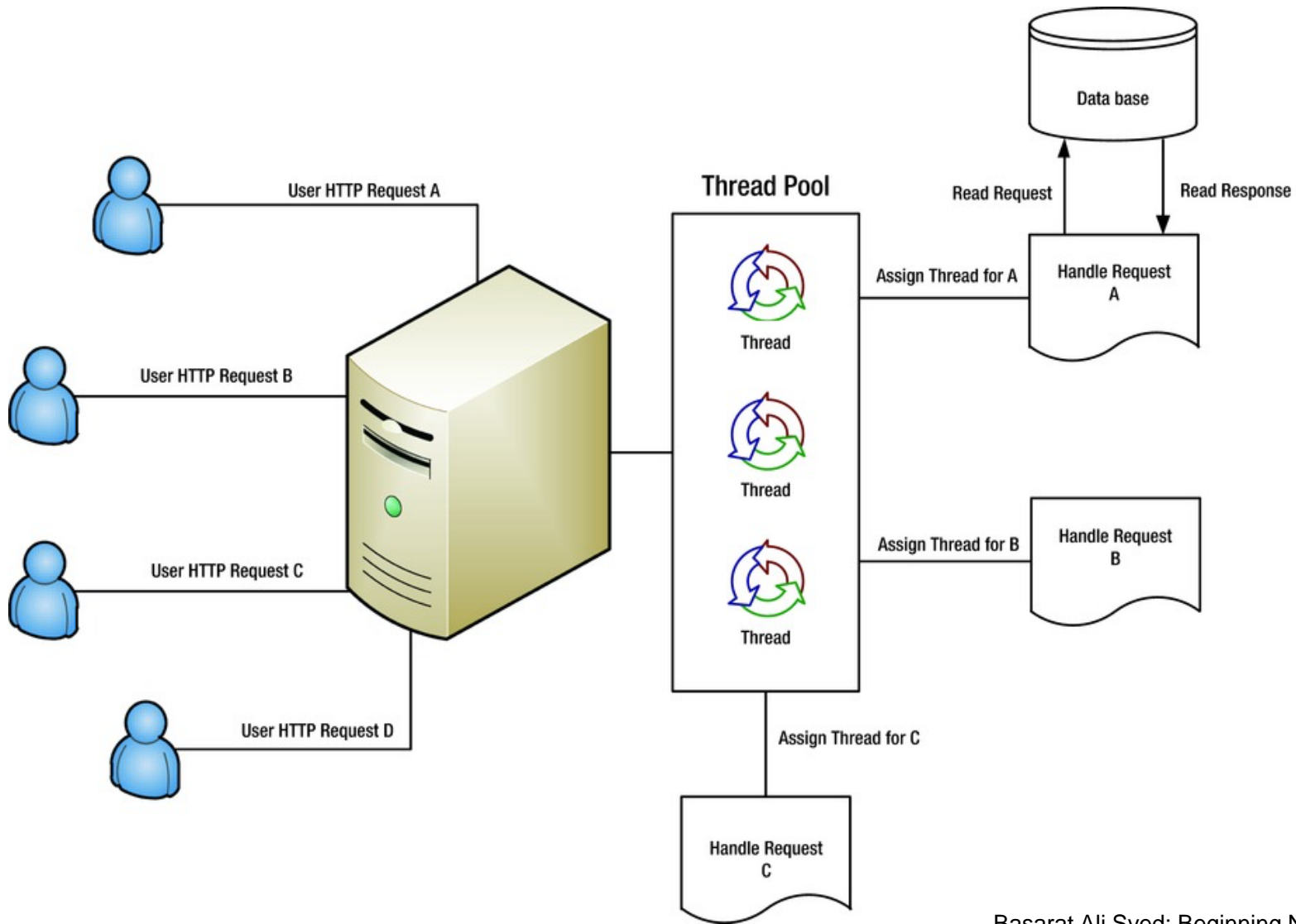


Figure 2-2. Traditional web server using Processes

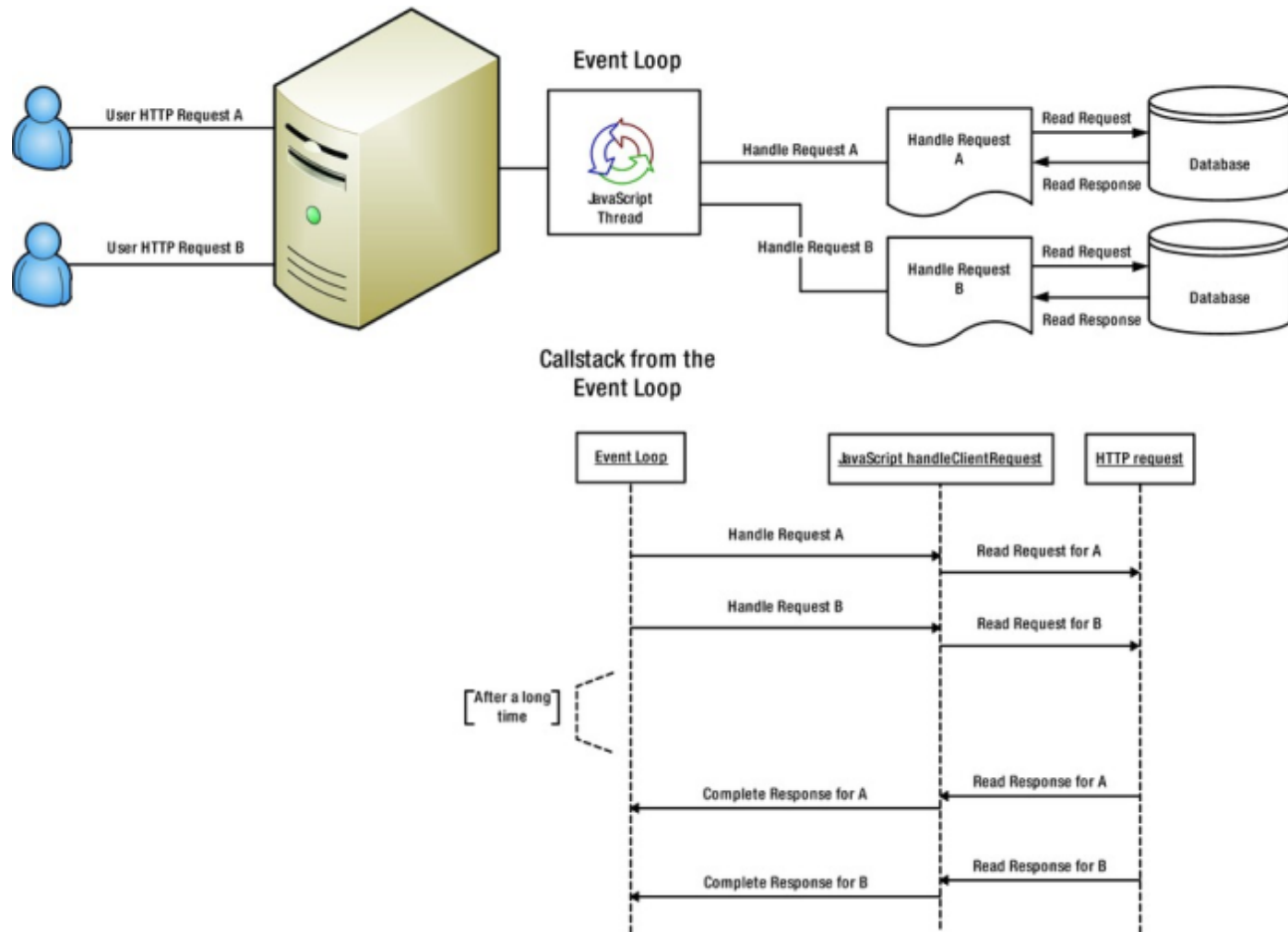
Basarat Ali Syed: Beginning Node.js, page 24

Multi Threaded Execution



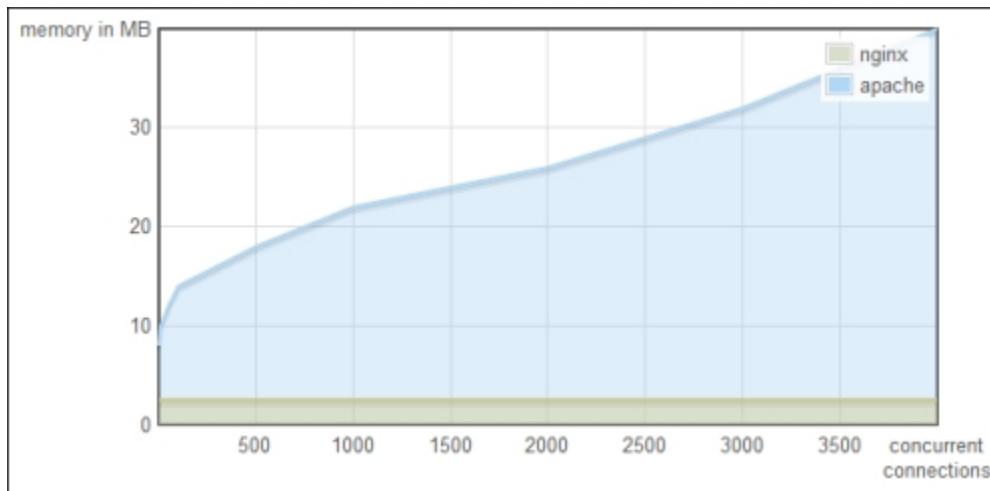
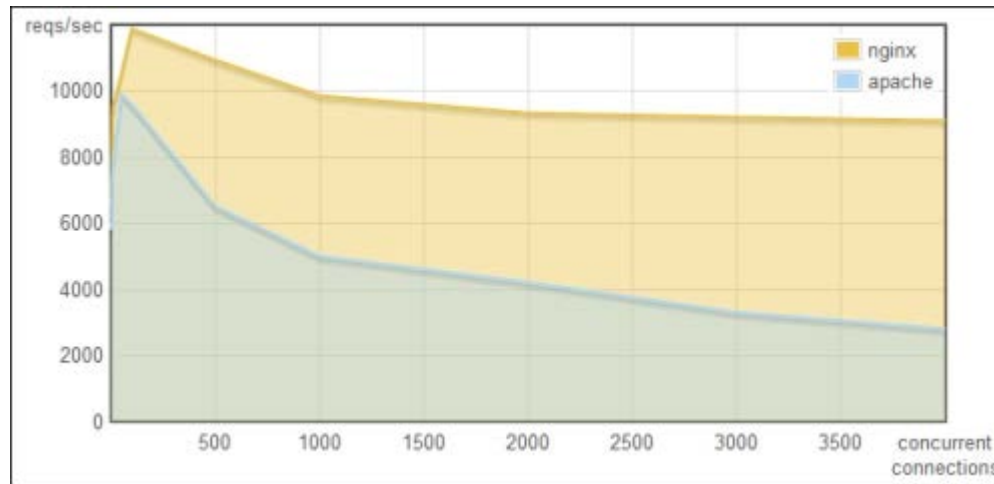
Basarat Ali Syed: Beginning Node.js, page 25

Asynchronous request handling



Basarat Ali Syed: Beginning Node.js, page 26

Performance comparison



Basarat Ali Syed: Beginning Node.js, page 26

Global 'things'

- JavaScript is a *world of globals*. Many APIs are globally defined.
- In OO language like Java, we define variables/methods inside class definition
- JavaScript code has many variables and functions that do not belong to a class or object
 - These are 'global' variable or functions
 - Belong to a global object
- Browser has a global object window
 - variables, functions not belonging to some object or class belong to the `window` object and has global scope
 - `document` object's fully qualified name is `window.document`
- In large application, with third party code or framework, there might be conflict in global namespaces.

Module System

- Module system is a way to organize namespaces defined in various script code in large JavaScript applications
 - Two different systems for browser and server version
- Node.js uses file-based module system (CommonJS)
 - Each file is its own module
 - Each file has access to the current module definition using the `module` variable.
 - The export of the current module is determined by the `module.exports` variable.
 - To import a module, use the globally available `require` function.

Outline

- Node.js execution
- **Node.js application structure**
- Express.js basics
 - Routing
 - Middleware
 - Template Engines

Node.js basic components

- **Node.js** has a few important globals
 - console
 - process
 - require
- **Node.js** is shipped with a few core modules
 - cli, fs, http
- There are lots of other useful modules that needs to be downloaded separately

Simple Web Application

require is a global function to import module .

createServer is a higher order function taking another function as paramter. It returns a server object.

The anonymous function with parameter **request** and **response** describe the application logic of the webserver.

```
var http = require('http')
var server = http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
});
server.listen(8888);
console.log("Web Server Running at http://localhost:8888 ... ");
```

console is a global object

Handling multiple requests

```
var http = require('http')
var server = http.createServer(function(request, response) {
  if (req.url == <some pattern>){
    //write response
  }else if (req.url == <some other pattern>){
    //write response
  }else if (req.url == <...>){
    //write response
  }else{ // a not recognizable url
    // send back an error status code and
    // error message
  }
});
server.listen(8888);
console.log("Web Server Running at http://localhost:8888 ... ");
```

All requests are handled by a single thread

Handling query string

Import another module with url parsing utilities

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

var server = http.createServer(function(request, response) {
  //targeting url like /sayHello?name=xxx
  if (req.url.indexOf('/sayHello') > -1){
    var query = url.parse(req.url,true).query;

    res.end('<body><h2> Hello `
      + query.name +
      '</h2></body></html>');

  }else{ // a not recognizable url
    // send back an error status code and
    // error message
  }
});

server.listen(8888);
console.log("Web Server Running at http://localhost:8888 ... ");
```


Handling POST request data

Post request contains a body part. The body may contain simple form data or large file in a uploading request

The request body is accessed through **data** event, so we listen to it, and save the received data in a variable.

When there is no more data, an **end** event fires. We listen to it to and call a function to generate the response.

Both handlers, defined as anonymous functions are able to access the local variable **body** of the outer function. This is called closure.

```
var http = require('http'),
    qs = require('querystring');

var server = http.createServer(function(request, response) {
  //targeting request like POST /sayHello with a body
  //name:xxx
  if (req.url.indexOf('/sayHello') > -1){
    var body = '';

    request.on('data', function(chunk){
      body += chunk;
    });

    request.on('end', function(){
      response.write('<body><h2> Hello ' +
        qs.parse(body).name +
        '</h2></body></html>');
    });
  } else { // a not recognizable url
    // send back an error status code and
    // error message
  }
});

server.listen(8888);
console.log("Web Server Running at http://localhost:8888 ... ");
```

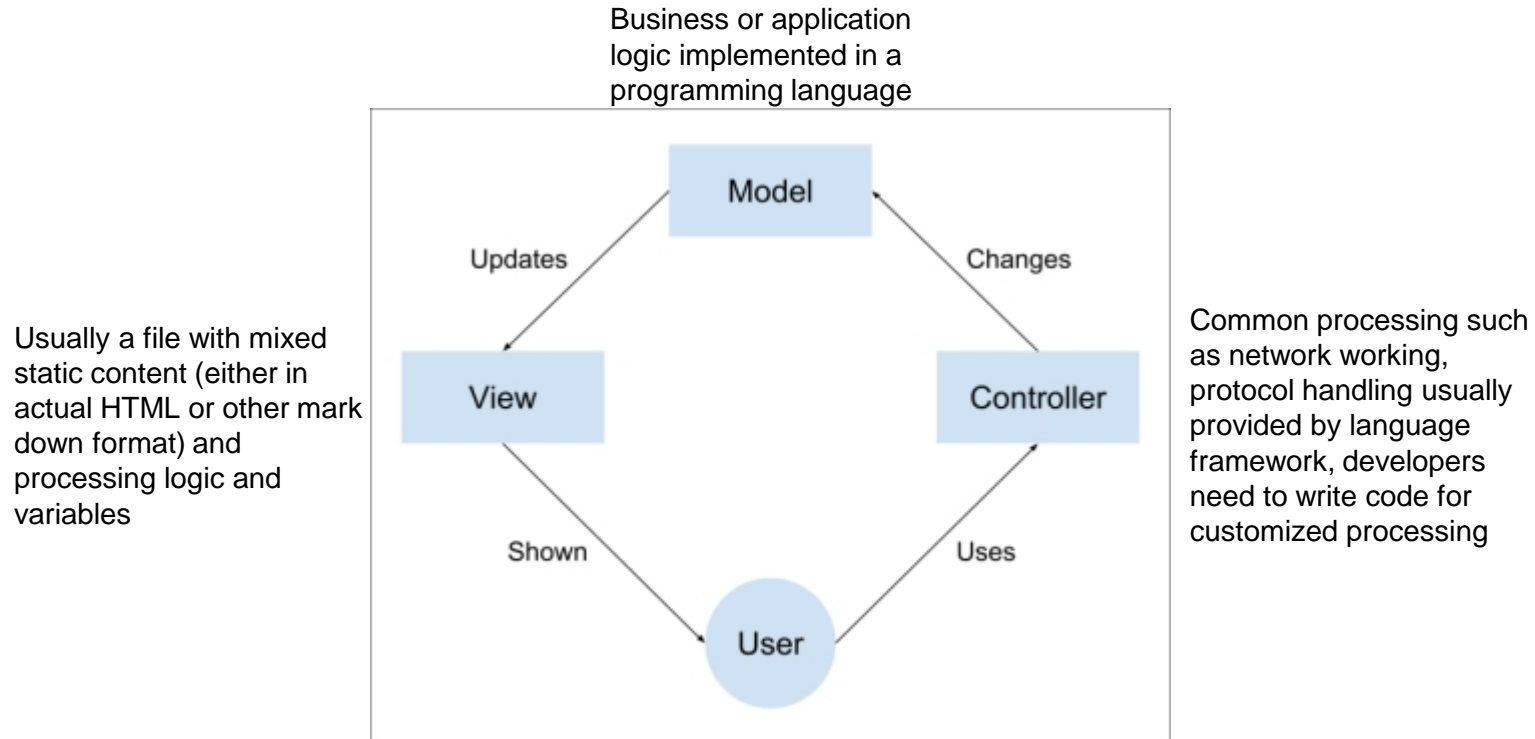
Outline

- Node.js execution
- Node.js application structure
- **Express.js basics**
 - Routing
 - Middleware
 - **Template Engines**

To write a real world application

- The MVC pattern
 - What is the view technology (something similar to early server page technology such as JSP)
 - Template language and template engines.
- Some framework to enable
 - URL / controller mapping
 - MVC wiring
 - Session management
 - Security management
 - ...
- **Express** is a popular framework for building MVC node.js application

Common MVC architecture



Introducing Express

- Express is the framework that implements lot of common tasks when writing web app
 - Server setup
 - Manage the request response paradigm
 - Defines directory structure
 - Routing URL to code
 - Talk to template engine(s) to convert template files into proper response HTML
 - Remembering visitors for session support

Routing

- Routing refers to the mapping between HTTP request (method, url) to the piece of code handling the request
- Express routing takes the following structure:
 - `app.METHOD(PATH,HANDLER)`
 - `app` is an instance of `express`.
 - `METHOD` is an HTTP request method, in lowercase.
 - `PATH` is a path on the server.
 - `HANDLER` is the function executed when the route is matched.
- Example routing code

```
app.get('/', function (req, res) {  
  res.send('Hello World!')  
})
```

```
app.post('/', function (req, res) {  
  res.send('Got a POST request')  
})
```

<https://expressjs.com/en/starter/basic-routing.html>

Express Hello World Example

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

var app = express()

app.get('/', function (req, res) {
    res.send('Hello World!')
})

app.listen(3000)
```

Middleware

- *Middleware* is basically any software (function) that sits between the application code and some low level API
- It has access to the request and response object
- It is used to implement common tasks on the request or response objects
 - If you have learned basic Java web app, middleware is like filter where you can stack a lot of them before or after servlet
- The overall process would be
 - Client sends request
 - Request arrives at server
 - Handled by middleware 1, passed to next
 - Handled by middleware 2, passed to next
 - route method, passed to next
 - Middleware 3, stop here <https://expressjs.com/en/guide/writing-middleware.html>

Middleware

Middle functions can do the following:

- Execute any code
- Make changes to request or response object
- End the request-response cycle
- Call next in the stack, middleware or route method

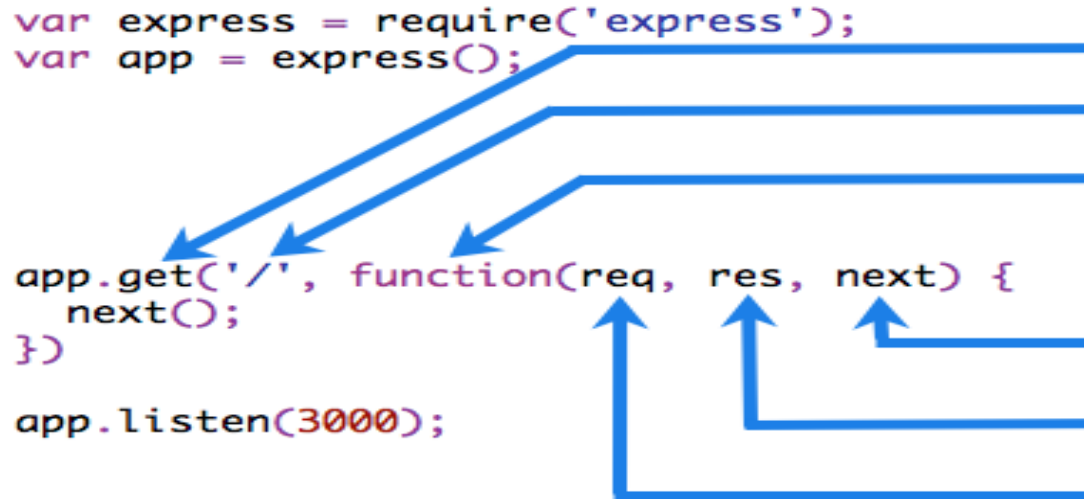
```
var express = require('express')
var app = express()

var myLogger = function (req, res, next) {
    console.log('LOGGED')
    next()
}
// This is a middleware function
// Call next() so the request-response cycle does not stop here

app.use(myLogger) // It will be called before the route method

app.get('/', function (req, res) {
    res.send('Hello World!')
})
app.listen(3000)
```

Middleware stack



Route method is also a middleware, most of the time we call it route method or handler. It can use the **next()** function to pass the control to a middleware running after the response is sent out

Useful middlewares

- Middlewares are commonly used for tasks like
 - Parsing POST body
 - Parsing cookies
 - And so on
- There are many third party middlewares for common processing
 - E.g. body-parser module consists of a few middleware functions for parsing form data

Template Engine

- Template engine represents the view technology
 - It defines a template format to blend static content with processing logic
 - Like classic server page JSP, or PHP
 - It is responsible for translating the template file into proper HTML file
 - Running processing logic
 - Replacing variable with values
- Many template engines that can work with Express
 - EJS
 - PUG (previously called JADE)
 - Mustache
- As expected, the oldest/simplest template engine (EJS) support the format of embedding JavaScript code in HTML

EJS -Embedded JavaScript template

- JavaScript code can be embedded nearly everywhere in a HTML file
 - The convention is very similar to early JSP
 - The control flow should be embedded with `<% %>`
 - Escaped output (expression) should be embedded with `<%= %>`
 - And others
- Example EJS

```
<% if (user) { %>  
    <h2><%= user.name %></h2>  
    <% } %>
```

<https://www.npmjs.com/package/ejs>

Simple Example

← → ↻ ⓘ localhost:3000

Please type in your name:

← → ↻ ⓘ localhost:3000/greeting?name=Jo

Welcome Jo

```
var express = require('express')
var path = require('path')

var app = express()
app.set('views', path.join(__dirname, 'views'));
app.get('/', function(req,res){
    res.render('greetingform.ejs')
});
app.get('/greeting', function(req,res){
    name=req.query.name
    res.render('greeting.ejs', {name:name})
});

app.listen(3000, function () {
    console.log('greeting app listening on port 3000!')
})
```

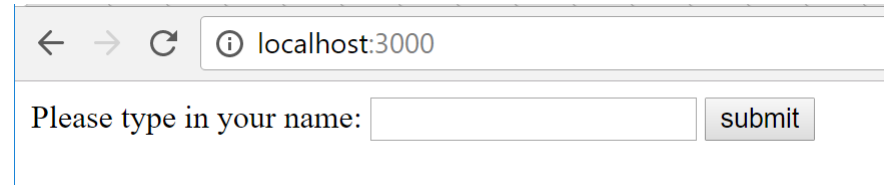
Simple Example – the form

Greetingform.ejs

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Greetings Input</title>
</head>
<body>
```

```
<form method="GET" action = "greeting">
Please type in your name: <input type = "text" name="name" ></input>
<input type = "submit" value = "submit"></input>
</form>

</body>
</html>
```

A screenshot of a web browser window. The address bar shows 'localhost:3000'. The page content displays the text 'Please type in your name:' followed by a text input field and a 'submit' button.

Please type in your name:

Simple Example – the response

Greetingform.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title>Customized Greeting!</title>
  </head>
  <body>
    Welcome <%= name %>
  </body>
</html>
```

← → ↻ ⓘ localhost:3000/greeting?name=Jo

Welcome Jo

Pug Template Engine

- Pug defines simple rules for writing static HTML content together with processing logic
 - HTML tags
 - Attributes
 - Code and Control Structure
 - Interpolation

Pug: HTML Tags

- By default, text at the start of a line (or after only white space) represents an html tag. Indented tags are nested, creating the tree like structure of html.

```
ul
  li Item A
  li Item B
  li Item C
```

```
<ul>
  <li>Item A</li>
  <li>Item B</li>
  <li>Item C</li>
</ul>
```

Pug: Attribute

- Attributes
 - Tag attributes look similar to html (with optional comma), but their values are just regular JavaScript.

```
a(href='google.com') Google
|
|
a(class='button' href='google.com') Google
|
|
a(class='button', href='google.com') Google
```

```
<a href="google.com">Google</a>
<a class="button" href="google.com">Google</a>
<a class="button" href="google.com">Google</a>
```

Pug: simple control

- Any JavaScript code can be included with a leading '-' character
- There are also first class conditional and iteration syntax

```
- for (var x = 0; x < 3; x++)  
  li item
```

```
<li>item</li>  
<li>item</li>  
<li>item</li>
```

```
- var user = {description:'foo bar baz'}  
- var authorised = false  
div#user  
  if user.description  
    h2.green Description  
    p.description= user.description  
  else if authorised  
    h2.blue Description  
    p.description.  
      User has no description,  
      why not add one...  
  else  
    h2.red Description  
    p.description User has no description
```

```
<div id="user">  
  <h2  
    class="green">Description</h2>  
    <p class="description">foo bar  
    baz</p>  
</div>
```

Pug: Interpolation

- JavaScript variables and expression can be included in various ways

```
- var title = "On Dogs: Man's Best Friend";  
- var author = "enlore";  
- var theGreat = "<span>escape!</span>";  
- var unescaped = "The tag <em> names </em> stays";
```

```
h1= title
```

```
p Written with love by #{author}
```

```
p This will be safe: #{theGreat}
```

```
p This is the unescaped example: !{unescaped}
```

```
<h1>On Dogs: Man's Best Friend</h1>
```

```
<p>Written with love by enlore</p>
```

```
<p>This will be safe: &lt;span&gt;escape!&lt;/span&gt;</p>
```

```
<p>This is the unescaped example: The tag <em> names </em> stays</p>
```

Pug: Interpolation

- Inline tags

- Inline tags, as opposite to block styled tags, do not start a new line.
- PUG provides an easy way of writing them to avoid unnecessary indent

p

This is a very long and boring paragraph that spans multiple lines. Suddenly there is a **#[strong strongly worded phrase]** that cannot be **#[em ignored]**.

p

And here's an example of an interpolated tag with an attribute: **#[q(lang="es") ¡Hola Mundo!]**

<p>This is a very long and boring paragraph that spans multiple lines. Suddenly there is a ****strongly worded phrase**** that cannot be ****ignored****.**</p>**

<p>And here's an example of an interpolated tag with an attribute: **<q lang="es">**¡Hola Mundo!**</q></p>**

Resources

- Rauch, Guillermo 2012, Smashing Node.js: JavaScript Everywhere,
 - e-book, accessible from USYD library,
 - Chapter 7 HTTP
- Basarat Ali Syed 2014, Beginning Node.js
 - E-book, accessible from USYD library
 - Chapter 2 and 3
- Haviv, Amos Q, MEAN Web Development
 - E-book, accessible from USYD library
 - Chapter 2 and 3