# Τεχνολογίες Εφαρμογών Ιστού

Server-side programming
(NodeJS)

# Client-side and Server-side programming

- Dynamic web content creation technologies:
  - Client-side:
    - Javascript, VBScript, Java Applets, ActiveX controls
  - Server-side:
    - Common Gateway Interface, Active Server Pages, Java Servlets, Java Server Pages, PHP, Javascript frameworks, Ruby, ...

# Server-side programming advantages

- **No web browser compatibility issues**: Clients get simple HTML pages
- **Better security**: the code that generates the HTML page is not visible to the user
- **Less network overhead**: Only the necessary HTML code is transmitted.

# Web Servers

- Browsers speak HTTP and Web Servers speak HTTP
  - Browsers: send HTTP request and get HTTP responses
  - Web Server: get HTTP requests and send HTTP responses
- HTTP is layered on TCP/IP so a web server:
  - loop forever doing:
    - accept TCP connection from browser
    - read HTTP request from TCP connection
    - process HTTP request
    - write HTTP response to TCP connection
    - shutdown TCP connection (except if Connection: keep-alive)

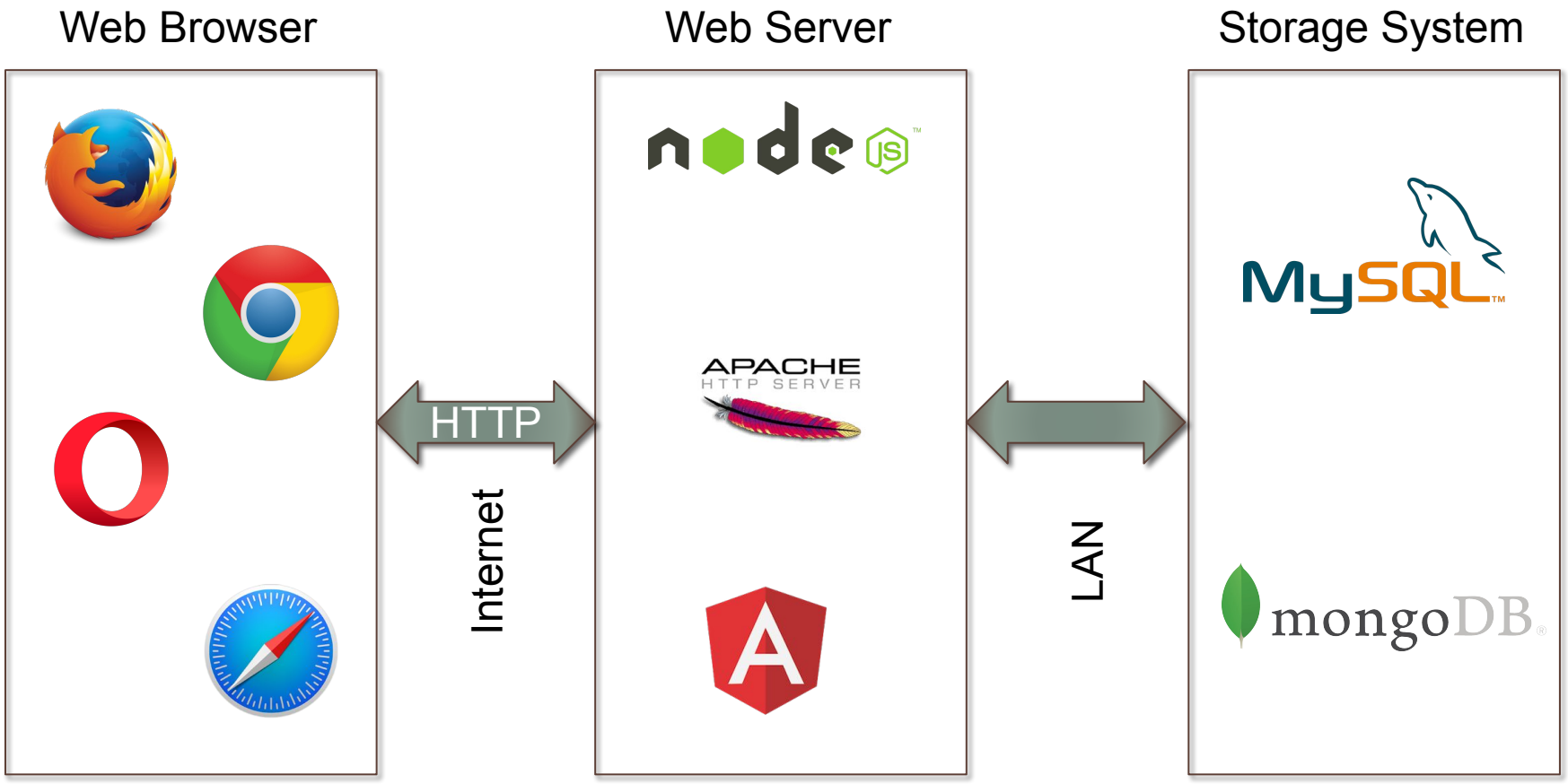
# Processing HTTP requests

- Process HTTP GET index.html
  - int fd = open("index.html");
  - int len = read(fd, fileContents, sizeOfFile(fd));
  - write(tcpConnection, httpResponseHeader, headerSize);
  - write(tcpConnection, fileContents, len);
- Note open and read may have to talk to a slow disk device
  - Can process requests concurrently by starting a new thread or a new process per request

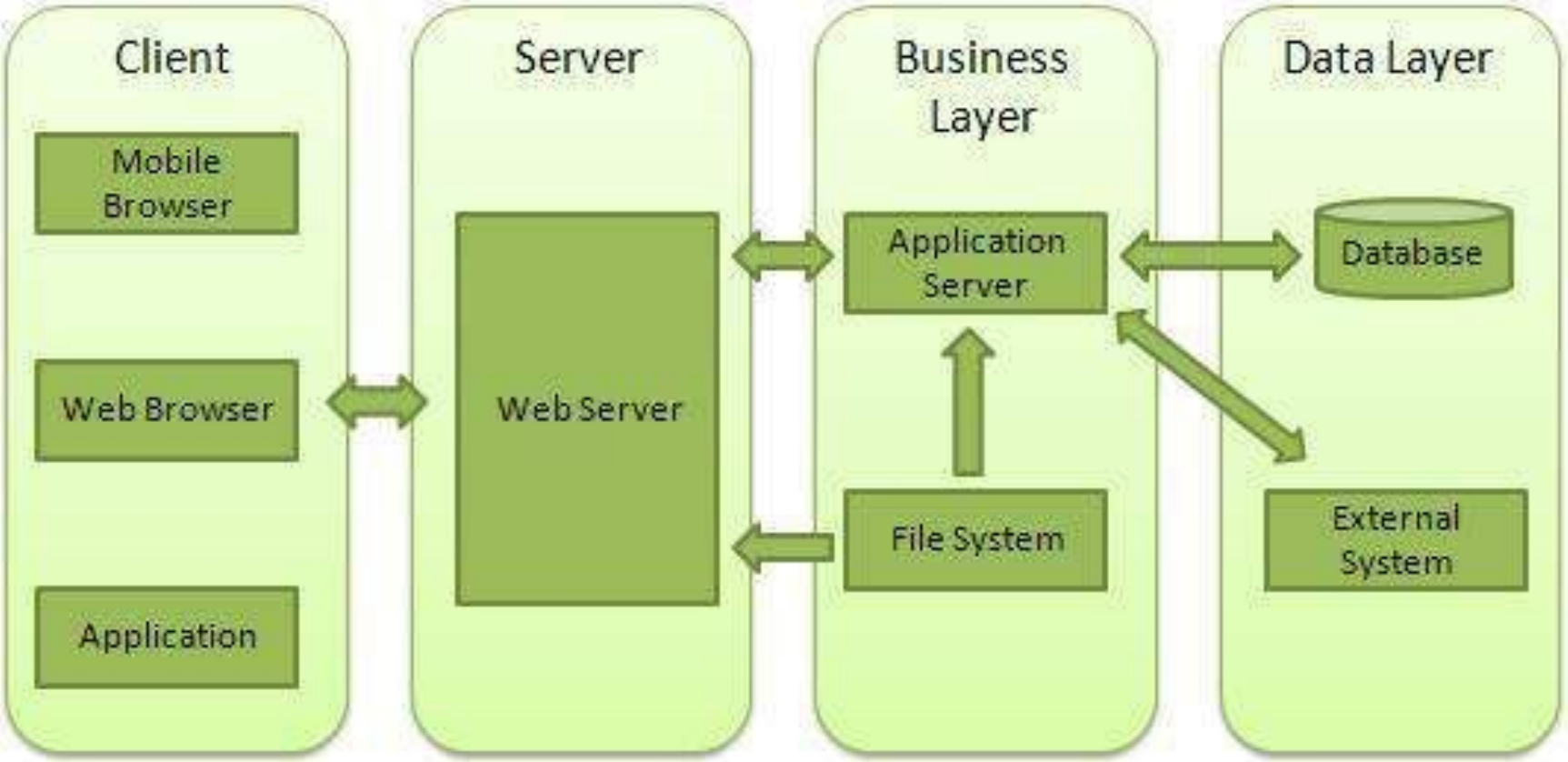
# Processing HTTP requests - cgi-bin

- Process HTTP GET of index.php
  - runProgramInNewProcess(tcpConnection);
- Template processing program fetches models from database system

# Web application architecture



# Web application architecture (detailed)



# 2nd Generation Web App Frameworks

- Web server runs a program per request - the controller:
  - Parse URL and/or HTTP request body to get parameters to view
  - Use parameters to fetch model data from DBMS (typically a SQL relational DBMS)
  - Run HTML view template with model data to generate the HTML
  - Send a HTTP response with the HTML back to the browser

# Web servers for JavaScript frameworks

- Most of the web app is simple static files - any web server speaking HTTP
  - View templates (HTML, CSS)
  - JavaScript files
- Remaining browser-server communication around model data
  - CRUD (Create Read Update Delete) of model data
  - Session info (e.g. login, etc.)
- Low requirements on web request processing
  - HTTP GET static files
  - Model data operation - mostly doing DBMS operations

# Progressive Web Apps

- Rationale
  - Progressively shift more functionality on the client side
    - Cache data on the client
    - Understand requirements and adapt on the client
- Benefits
  - Improved response times, personalization and thus, UX
- Baseline technologies
  - Service Workers
    - Like web workers but associated with the browser (the app) and not necessarily with a document
    - Mostly used for caching data and service client HTTP requests locally
  - Manifest
    - A way to describe a set of resources as a bundle and make it look as an independent (desktop, mobile, etc) app.

# NODE.JS

# Node js

- Use the javascript engine and javascript supporting code to run javascript custom code in the backend
  - Engine
    - Google’s V8 VM
  - Supporting Code
    - Nodejs modules
- NodeJS ::= Runtime environment & library

# Features

- Asynchronous and Event Driven
  - All APIs of Node.js library are asynchronous that is, non-blocking.
- Very Fast
  - Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.
- Single Threaded but Highly Scalable
  - uses a single threaded model with event looping.
- No Buffering
  - applications never buffer any data. These applications simply output the data in chunks.
- License
  - released under the MIT license

# Application

- A Node.js application consists of following three important parts
  - Required modules: We use require directive to load a Node.js module.
  - Server: A server which will listen to client's request similar to Apache HTTP Server.
  - Read request and return response: The server created in earlier step will read HTTP request made by the client and return the response.

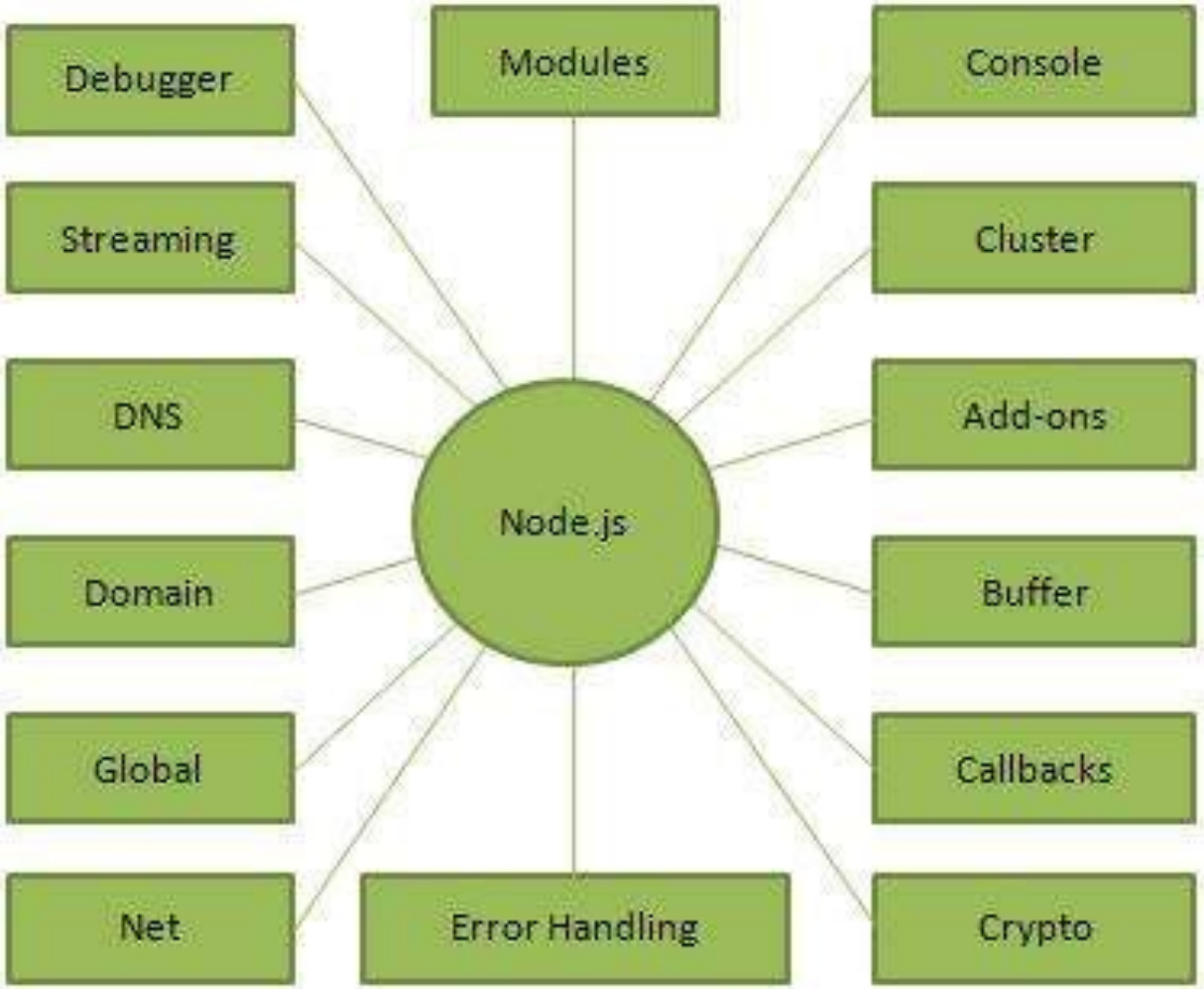
# Example

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

http.createServer(function (request, response) {

   // Send the HTTP header
   // HTTP Status: 200 : OK
   // Content Type: text/plain
   response.writeHead(200, {'Content-Type': 'text/plain'});

   // Send the response body as "Hello World"
   response.end('Hello World\n');
}).listen(8081);

// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

RESULT:

```
$ node main.js
Server running at http://127.0.0.1:8081/
```

# Drilling down to the core: Concepts



# Node Package Manager (npm)

- Node Package Manager (npm) provides following two main functionalities:
  - Online repositories for node.js packages/modules which are searchable on search.nodejs.org
  - Command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

# Callbacks

- Asynchronous equivalent for a function
- It is called at the completion of a given task
- Node makes heavy use of callbacks
- All APIs of Node are written is such a way that they supports callbacks.
- Any async function accepts a callback as a last parameter and the callback function accepts error as a first parameter.

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

var data = fs.readFileSync('input.txt');

console.log(data.toString());
console.log("Program Ended");
```

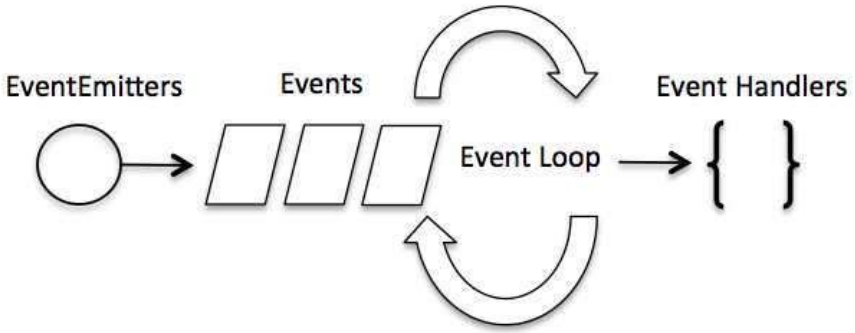
RESULT:
```
*File content here*
Program Ended
```

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

fs.readFile('input.txt', function (err,
data) {
    if (err) return console.error(err);
    console.log(data.toString());
});

console.log("Program Ended");
```

RESULT:
```
Program Ended
*File content here*
```

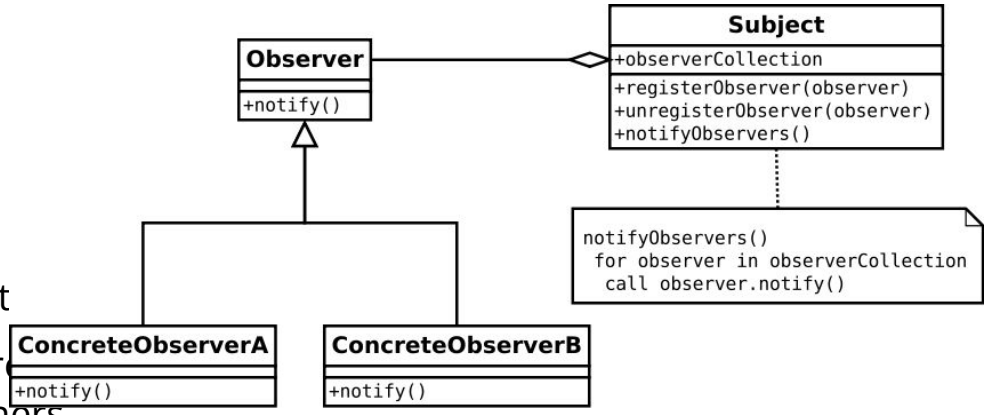
# Concurrency

- Supported via events and callbacks
- Event loop
  - a main loop that listens for events, and then triggers a callback function when one of those events is detected



# Callback Vs Events

- Callback functions are called when an async function returns
- Event handling follows the Observer Pattern
  - Subject informs the list of dependents (observers) of state changes by calling their methods
  - Listeners == Observers.
  - Whenever an event gets fired, its list
  - Multiple built-in events available thr
    used to bind events and event listeners



# Example

```
// Import events module
var events = require('events');
// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();

// Bind the connection event with a handler (anonymous function)
eventEmitter.on('connection', function(){
   console.log('connection succesful.');

   // Fire the data_received event
   eventEmitter.emit('data_received');
});

// Bind the data_received event with the anonymous function
eventEmitter.on('data_received', function(){
   console.log('data received succesfully.');
});

// Fire the connection event
eventEmitter.emit('connection');

console.log("Program Ended.");
```

# Managing binary data

- Octet streams handling for TCP streams or the file system
- Buffer class provides instances to store raw data similar to an array of integers
  - but corresponds to a raw memory allocation outside the V8 heap.

```
var buf = new Buffer('Testing');
var json = buf.toJSON(buf);
console.log(json);

RESULT:
$ node main.js
```

```
[ 84, 101, 115, 116, 105, 110, 103 ]
```

# Streams (1/2)

- Objects that enable the continuous reading and writing of data from a source
- There are four types of streams.
  - **Readable**: Streams that are used for read operations.
  - **Writable**: Stream that are used for write operations.
  - **Duplex**: Streams that can be used for both read and write operations.
  - **Transform**: A type of duplex stream where the output is computed based on input.

# Streams (2/2)

- Streams inherit EventEmitter Class
- At various times they throw events, such as:
  - data: This event is fired when there is data is available to read.
  - end: This event is fired when there is no more data to read.
  - error: This event is fired when there is any error receiving or writing data.
  - finish: This event is fired when all data has been flushed to the underlying system

# Example

```
var fs = require("fs");
var data = '';

// Create a readable stream
var readerStream = fs.createReadStream('input.txt');

// Set the encoding to be utf8.
readerStream.setEncoding('UTF8');

// Handle stream events --> data, end, and error
readerStream.on('data', function(chunk) {
   data += chunk;
});

readerStream.on('end',function(){
   console.log(data);
});

readerStream.on('error', function(err){
   console.log(err.stack);
});

console.log("Program Ended");
```

# Global Objects

- Available in all modules.
- modules, functions, strings and objects themselves
- E.g.
  - __filename represents the filename of the code being executed
  - __dirname represents the name of the directory that the currently executing script resides in
  - setTimeout(cb, ms) runs callback cb after at least ms milliseconds
  - etc

# Express Framework

- Minimal and flexible Node.js web application framework.
- Facilitates a rapid development of Node-based Web applications.
- Core features:
  - Allows to set up middlewares to respond to HTTP Requests.
  - Defines a routing table which is used to perform different action based on HTTP Method and URL.
  - Allows to dynamically render HTML Pages based on passing arguments to templates.

# Example

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

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

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)

})
```

# RESTful API

- Web-based standard
- REpresentational State Transfer
- Every component is a resource
- A resource is accessed by a common interface using HTTP standard methods
- Each resource is identified by URIs/ global IDs
- Uses various representation schemes to represent a resource, e.g.
  - Text
  - XML
  - JSON
- HTTP Methods
  - GET - This is used to provide a read only access to a resource.
  - PUT - This is used to create a new resource.
  - DELETE - This is used to remove a resource.
  - POST - This is used to update a existing resource or create a new resource.

# Example: JSON Data Representation

```
{
   "user1" : {
      "name" : "Name1",
       "password" : "password1",
       "profession" : "teacher",
       "id": 1
   },
   "user2" : {
      "name" : "Name2",
       "password" : "password2",
       "profession" : "librarian",
       "id": 2
   },
   "user3" : {
      "name" : "Name3",
       "password" : "password3",
       "profession" : "clerk",
       "id": 3
   }
}
```

# Example: RESTful API

| ID | URI | HTTP Method | POST body | Result |
|---|---|---|---|---|
| 1 | listUsers | GET | empty | Show list of all the users. |
| 2 | addUser | POST | JSON String | Add details of new user. |
| 3 | deleteUser | DELETE | JSON String | Delete an existing user. |
| 4 | :id | GET | empty | Show details of a user. |

# Example: listUsers

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

app.get('/listUsers', function (req, res) {
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       console.log( data );
       res.end( data );
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)

})
```

# Example: addUser

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

var user = {
   "user4" : {
      "name" : "Name4",
      "password" : "password4",
      "profession" : "teacher",
      "id": 4
   }
}

app.get('/addUser', function (req, res) {
   // First read existing users.
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       data = JSON.parse( data );
       data["user4"] = user["user4"];
       console.log( data );
       res.end( JSON.stringify(data));
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port
  console.log("Example app listening at http://%s:%s", host, port)

})
```

# Example: userDetails

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

app.get('/:id', function (req, res) {
   // First read existing users.
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       users = JSON.parse( data );
       var user = users["user" + req.params.id]
       console.log( user );
       res.end( JSON.stringify(user));
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port
  console.log("Example app listening at http://%s:%s", host, port)

})
```

# Example: deleteUser

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

var id = 2;

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

   // First read existing users.
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       data = JSON.parse( data );
       delete data["user" + 2];

       console.log( data );
       res.end( JSON.stringify(data));
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port
  console.log("Example app listening at http://%s:%s", host, port)

})
```

# Scaling Application

- Uses an event-driven paradigm to handle concurrency
- It also facilitates the creation of child processes
- Child processes always have three streams
  - child.stdin
  - child.stdout
  - child.stderr
- “child_process” module provides three major ways to create child process.
  - exec - child_process.exec method runs a command in a shell/console and buffers the output.
  - spawn - child_process.spawn launches a new process with a given command
  - fork - The child_process.fork method is a special case of the spawn() to create child processes.

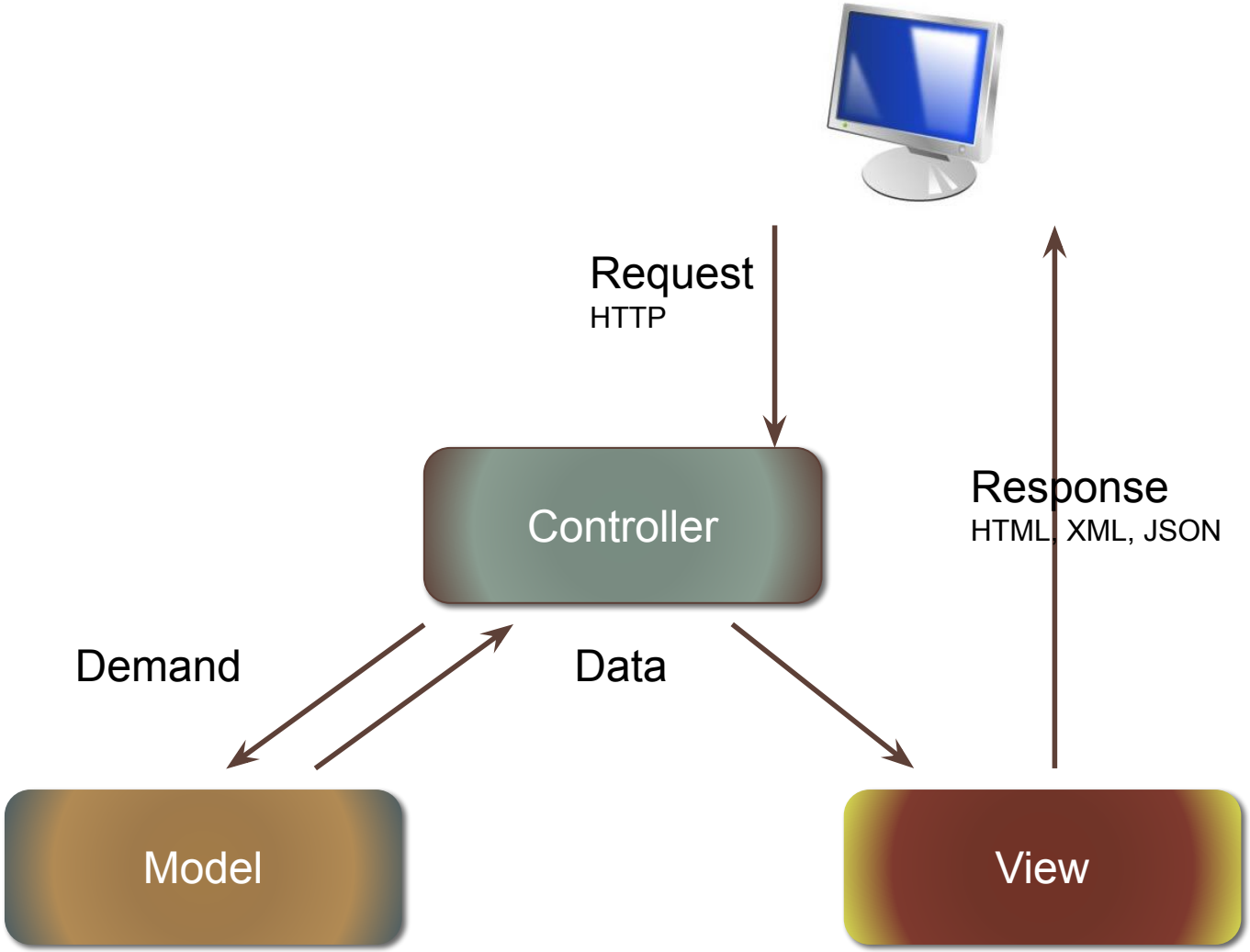
# Net Module

- Used to create both servers and clients
- Provides an aynchronous network wrapper
- Can be imported using following syntax:
  - `var net = require("net")`

# Sources

- http://www.tutorialspoint.com/nodejs

# MODEL-VIEW-CONTROLLER PATTERN


Request
HTTP
Controller
Response
HTML, XML, JSON
Demand
Data
Model
View

# Principles

- Separation of Concerns
  - 3 Concerns
- DRY (Don’t Repeat Yourself)