Introduction

- Webserver like IIS / Apache serves static files(like html files) so that a browser can view them over the network.
- We need to place the files in a proper directory(like wwwroot in IIS), so that we can navigate to it using http protocol. The web server simply knows where the file is on the computer and serves it to the browser.
- Node offers a different paradigm than that of a traditional web server i.e. it simply provides the framework to build a web server.
- Interestingly building a webserver in node is not a cumbersome process, it can be written in just a few lines, moreover we'll have full control over the application.

HTTP module in Node.js

- We can easily create an HTTP server in Node.
- To use the HTTP server and client one must require('http').
- The HTTP interfaces in Node are designed to support many features of the protocol which have been traditionally difficult to use. In particular, large, possibly chunk-encoded, messages.
- Node's HTTP API is very low-level. It deals with stream handling and message parsing only. It parses a message into headers and body
- HTTP response implements the Writable Stream interface and request implements Readable Stream interface.

Creating HTTP Server

```
/* Loading http module*/
var http = require('http');

/* Returns a new web server object*/
var server = http.createServer(function(req,res){

        /* Sends a response header to the request.*/
        res.writeHead(200,{'content-type':'text/html'});

        /*sends a chunk of the response body*/
        res.write('<h1>Hello KLFS</h1>')

         /* signals server that all the responses has been sent */
        res.end('<b>Response Ended</b>')

});

/* Accepting connections on the specified port and hostname. */
server.listen(3000);

console.log('server listening on localhost:3000');
```

Routing

➢ **Routing refers to the mechanism for serving the client the content it has asked**

```
var http = require('http');
var server = http.createServer(function(req,res){
var path = req.url.replace(/\/?(?:\?.*)?$/, '').toLowerCase();
switch(path) {
        case '' :
                        res.writeHead(200, {'Content-Type': 'text/html'});
                        res.end('<h1>Home Page</h1>');
                        break;
        case '/about' :
                        res.writeHead(200, {'Content-Type': 'text/html'});
                        res.end('<h1>About us</h1>');
                        break;
        default:
                        res.writeHead(404, { 'Content-Type': 'text/plain' });
                        res.end('Not Found');
                        break;
        }
});
server.listen(3000);
```

Accept data through form form.html

```
<!doctype html>
    <html>
    <body>
      <form action="/calc" method="post">
        <input type="text" name="num1" /><br />
        <input type="number" name="num2" /><br />

        <button type="submit">Save</button>
      </form>
    </body>
    </html>
```

Accept data from user through form

```
var http=require("http");
var fs=require("fs");
var url=require("url");
var query=require("querystring");
//add user defined module
var m1=require("./formaddmodule");
//function to handle request and response
function process_request(req,resp)
{
//to parse url to separate path
  var u=url.parse(req.url);
  console.log(u);
//response header
  resp.writeHead(200,{'Content-
Type':'text/html'});
```

Continued

```
//routing info
switch(u.pathname){
   case '/’:
               // read data from html file
               fs.readFile("form.html",function(err,data){
                 if(err){
                              resp.write('some error');
                              console.log(err);

                 }else{
                 // write file data to response
                 resp.write(data);
                 //send response to client
                 resp.end();}
   });
break;
case '/calc':
              var str="";
              //data event handling while receiving data
              req.on('data',function(d){
              str+=d;});
              //end event when finished receiving data
              req.on('end',function(){
                console.log(str);
                          var ob=query.parse(str);
                          //calling user defined function from user module
                          var sum=m1.add(ob.num1,ob.num2);
                          resp.end("<h1>Addition : "+sum+"</h1>");

              });

 }

}
var server=http.createServer(process_request);
server.listen(3000);
```

Forms add module

Formaddmodule.js

```
exports.add=function(a,b){

   return parseInt(a)+parseInt(b);

}
```

Introduction

- **If we try to create apps by only using core Node.js modules we will end up by writing the same code repeatedly for similar tasks such as**
  - Parsing of HTTP request bodies
  - Parsing of cookies
  - Managing sessions
  - Organizing routes with a chain of if conditions based on URL paths and HTTP methods of the requests
  - Determining proper response headers based on data types
- **Developers have to do a lot of manual work themselves, such as interpreting HTTP methods and URLs into routes, and parsing input and output data.**
- **Express.js solves these and many other problems using abstraction and code organization.**

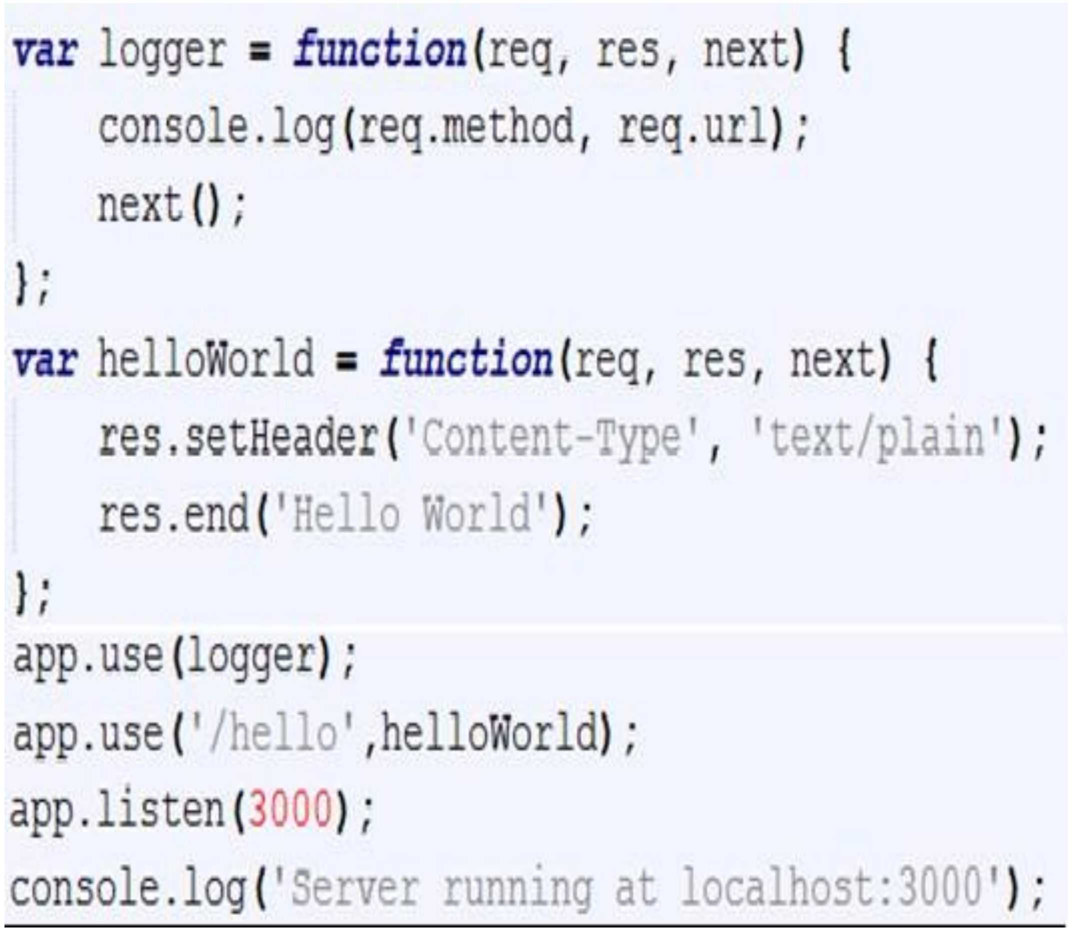
Working with Express framework

Introduction toExpress.js

- Express.js is a web framework based on the core Node.js http module and Connect components
- Express.js framework provides a model-view-controller-like structure for your web apps with a clear separation of concerns (views, routes, models)
- Express.js systems are highly configurable, which allows developers to pick freely whatever libraries they need for a particular project
- Express.js framework leads to flexibility and high customization in the development of web applications.
- In Express.js we can define middleware such as error handlers, static files folder, cookies, and other parsers.
- Middleware is a way to organize and reuse code, and, essentially, it is nothing more than a function with three parameters: request, response, and next.

Connect module

- **Connect is a module built to support interception of requests in a modular approach.**



```
var logger = function(req, res, next) {
    console.log(req.method, req.url);
    next();
};
var helloWorld = function(req, res, next) {
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World');
};
app.use(logger);
app.use('/hello',helloWorld);
app.listen(3000);
console.log('Server running at localhost:3000');
```

Express.js Installation

- **The Express.js package comes in two flavors:**
- ***express-generator*: a global NPM package that provides the command-line tool for rapid app creation (scaffolding)**
- ***express*: a local package module in your Node.js app’s node_modules folder**

Package.json

```
{  "name": "myapp",
 "version": "1.0.0",
  "description": "This is description",
"main": "index.js",
 "scripts": {    "test": "echo \"Error: no test specified\" && exit 1"  },
"dependencies":{      "express":"",        "body-parser":""  },
  "devDependencies":{    },
"keywords": [    "asd",    "sdjl",    "ljdfljsd",    "kjdflkjlkj"  ],
 "author": "Kishori",
"license": "ISC"}
```

To install if package.json file in myapp folder

```
C:/…../myapp>npm install
```

app.js

- **App.js is the main file in Express framework. A typical structure of the main Express.js file consists of the following areas**
  - 1. Require dependencies
  - 2. Configure settings
  - 3. Connect to database (optional)
  - 4. Define middleware
  - 5. Define routes
  - 6. Start the server

- **The order here is important, because requests travel from top to bottom in the chain of middleware.**

app.js

- **Routes are processed in the order they are defined. Usually, routes are put after middleware, but some middleware may be placed following the routes. A good example of such middleware, found after a routes, is error handler.**
- **The way routes are defined in Express.js is with helpers app.VERB(url, fn1, fn2, ..., fn), where fnNs are request handlers, url is on a URL pattern in RegExp, and VERB values are as follows:**
  - all: catch every request (all methods)
  - get: catch GET requests
  - post: catch POST requests
  - put: catch PUT requests
  - del: catch DELETE requests

app.js

- **Finally to start the server, using express object**

```
var express=require("express");
var app=express();
app.listen(3000, function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Handling post request

POST request

```
var express=require("express");
var app=express();
var bodyparser=require("body-parser");
        // num1=12&num2=13
        // ({extended:false}) ------- their uses querystring to parse data
       // ({extended:false}) ------- their uses qs module  to parse data
app.use(bodyparser.urlencoded({extended:false}));
app.use(function(req,resp,next){
                console.log(req.method+"------"+req.url);
                next();
});

app.get("/",function(req,resp){
  resp.sendFile("form.html",{root:__dirname});
});
app.post("/calc",function(req,resp){
              resp.send("<h1>"+req.body.num1+"------"+req.body.num2+"</h1>");
});

app.listen(2000);
```

```
//import the express module
var express = require('express');

//store the express in a variable
var app = express();

//allow express to access our html (index.html) file
app.get('/index.html', function(req, res) {
     res.sendFile(__dirname + "/" + "index.html");
  });
```

Handling GET request in express

Handling Get Request

```
app.get('/user', function(req, res){
      response = {
         first_name : req.query.first_name,
         last_name : req.query.last_name,
         gender: req.query.gender
         };
```

User-info.html

```
<html>
  <head>
    <title>Basic User Information</title>
  </head>
  <body>
      <form action="/user" method="GET">
            First Name: <input type="text" name="first_name"> <br>
            Last Name: <input type="text" name="last_name"> <br>
               Gender: <select name="gender">
                          <option value="Male">Male</option>
                          <option value="Female">Female</option>
                        </select>
                  <input type="submit" value="submit">
      </form>
  </body>
</html>
```

```
//route the GET request to the specified path, "/user".
//This sends the user information to the path
app.get('/user', function(req, res){
    response = {
      “first_name” : req.query.first_name,
      last_name : req.query.last_name,
      gender: req.query.gender
      };

    //this line is optional and will print the response on the command prompt
    //It's useful so that we know what infomration is being transferred
    //using the server
    console.log(response);

    //convert the response in JSON format
    res.end(JSON.stringify(response));
  });
```

```
//This piece of code creates the server
//and listens to the request at port 8888
//we are also generating a message once the
//server is created
var server = app.listen(8888, function(){
      var host = server.address().address;
      var port = server.address().port;
      console.log("Example app listening at http://%s:%s", host, port);
   });
```

Steps for creating Express.js Application

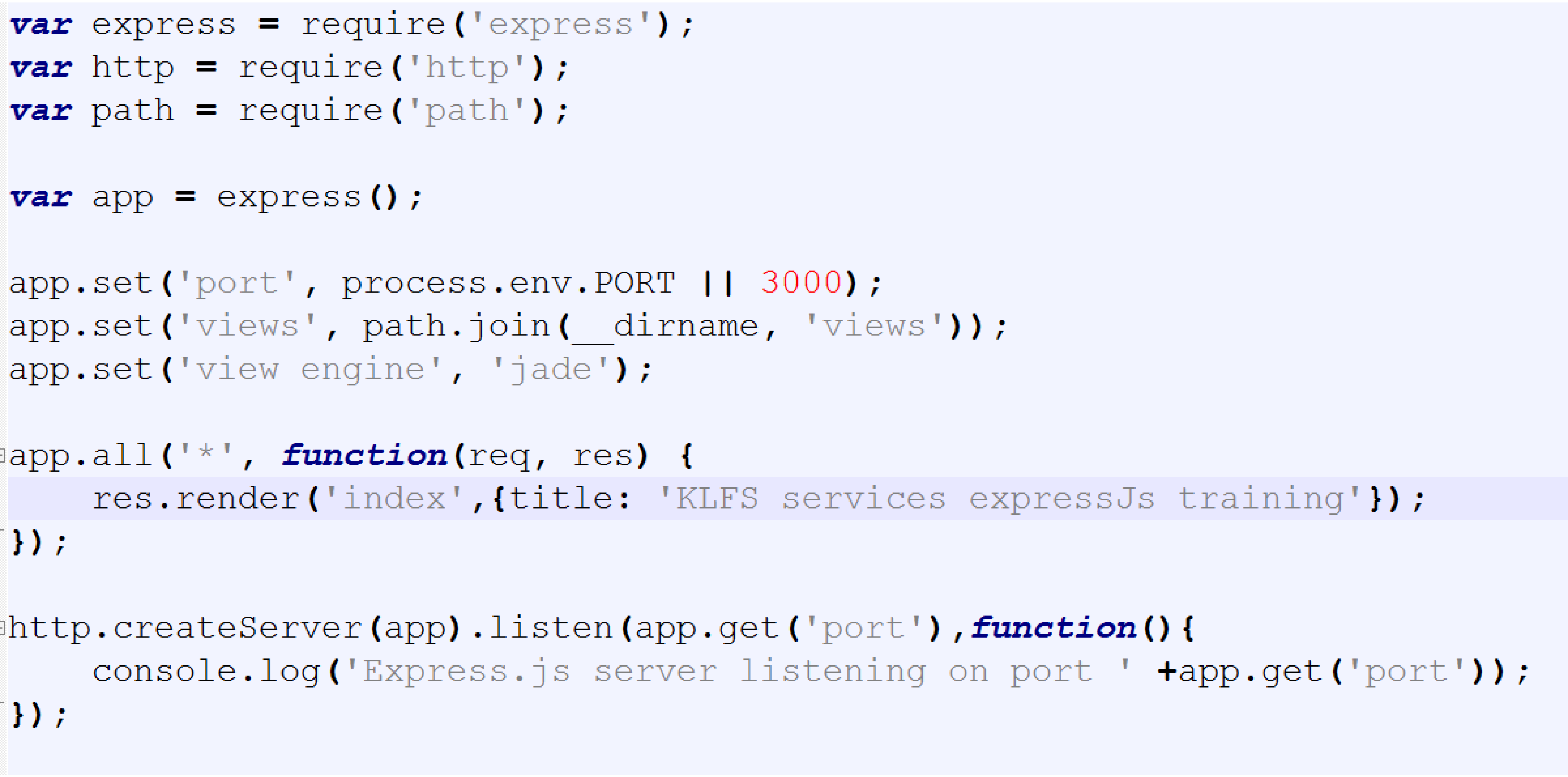
- **Step – 1 : Create a folder named MyApp**
- **Step – 2 : Create package.json with the following schema**

```
{
   "name": "MyApp",
   "version": "1.0.0",
   "description": "My first express js Application",
   "main": "app.js",
   "dependencies": {
     "body-parser": "1.10.1",
     "cookie-parser": "1.3.3",
     "express": "4.10.7",
     "jade": "1.8.2"
   },
    "scripts": {
     "start": "node app"
   },
   "author": "Kishori",
   "license": "ISC"
}
```

Steps for creating Express.js Application

- **Step – 3 : Install the dependencies using npm install command**
  - D:\Rojrocks\NodeJS\Lesson02\SampleApp>npm install
- **Step – 4 : Create the following folders under MyApp folder**
  - **public** : All the static (front-end) files like HTML
  - **public/css** : Stylesheet files
  - **public/images :** images
  - **public/scripts** : Scripts
  - **db** : Seed data and scripts for MongoDB
  - **views** : Jade (or any other template engine) files
  - **views/includes** : Partial / include files
  - **routes :** Node.js modules that contain request handlers
- **Step – 5 : Create the main file named app.js**

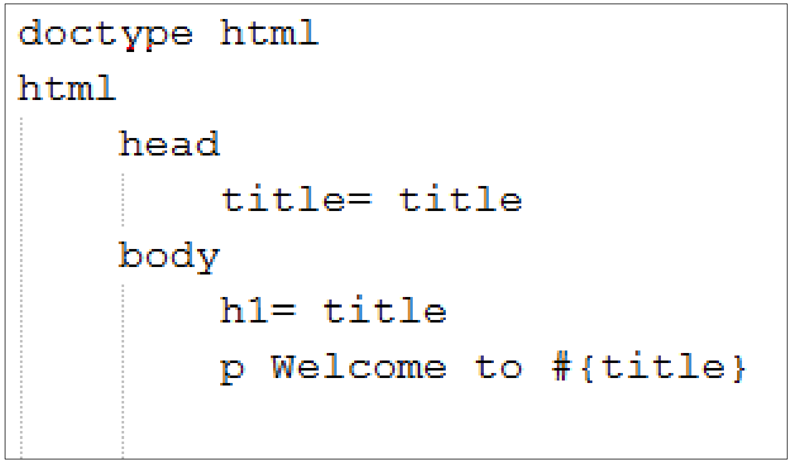
Steps for creating Express.js Application



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

var app = express();

app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.all('*', function(req, res) {
    res.render('index',{title: 'KLFS services expressJs training'});
});

http.createServer(app).listen(app.get('port'),function(){
    console.log('Express.js server listening on port ' +app.get('port'));
});
```

Steps for creating Express.js Application

- **Step – 7 : Create index.jade under views folder and type the following contents**

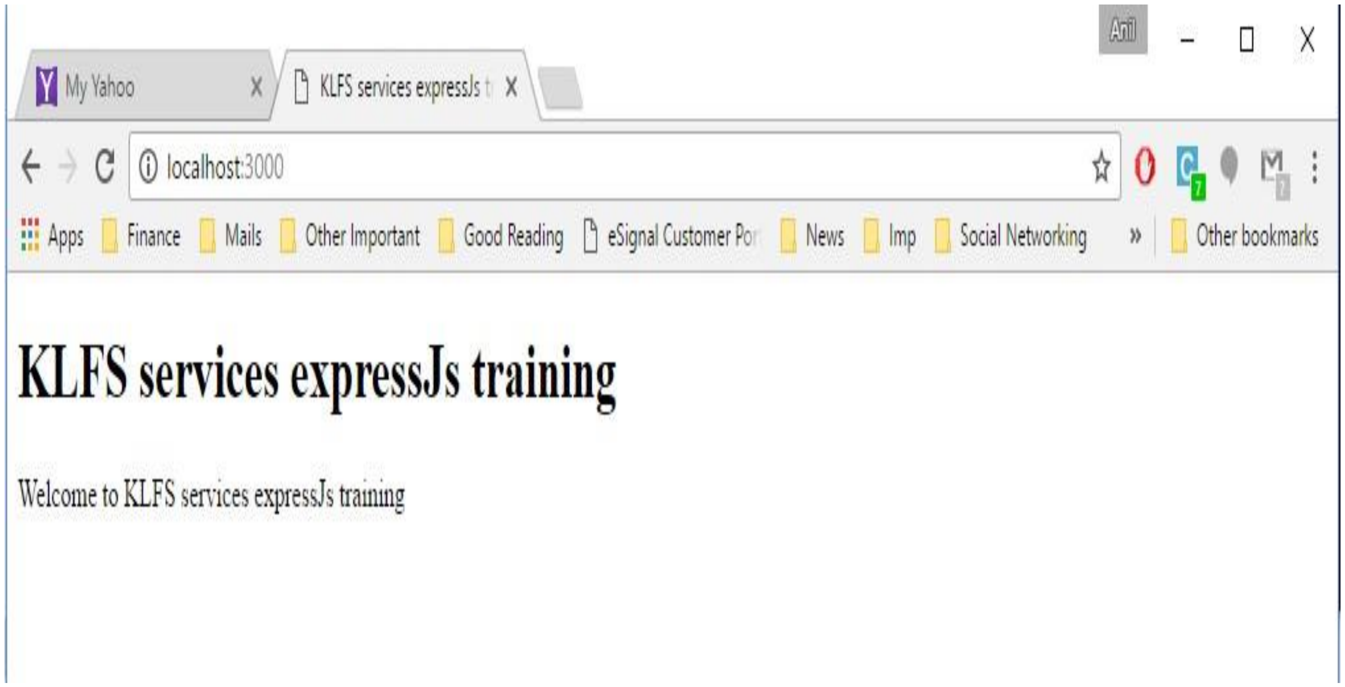


```
doctype html
html
    head
        title= title
    body
        h1= title
        p Welcome to #{title}
```

- **Step – 8 : Start the app by typing *npm start* in command prompt.**

Steps for creating Express.js Application

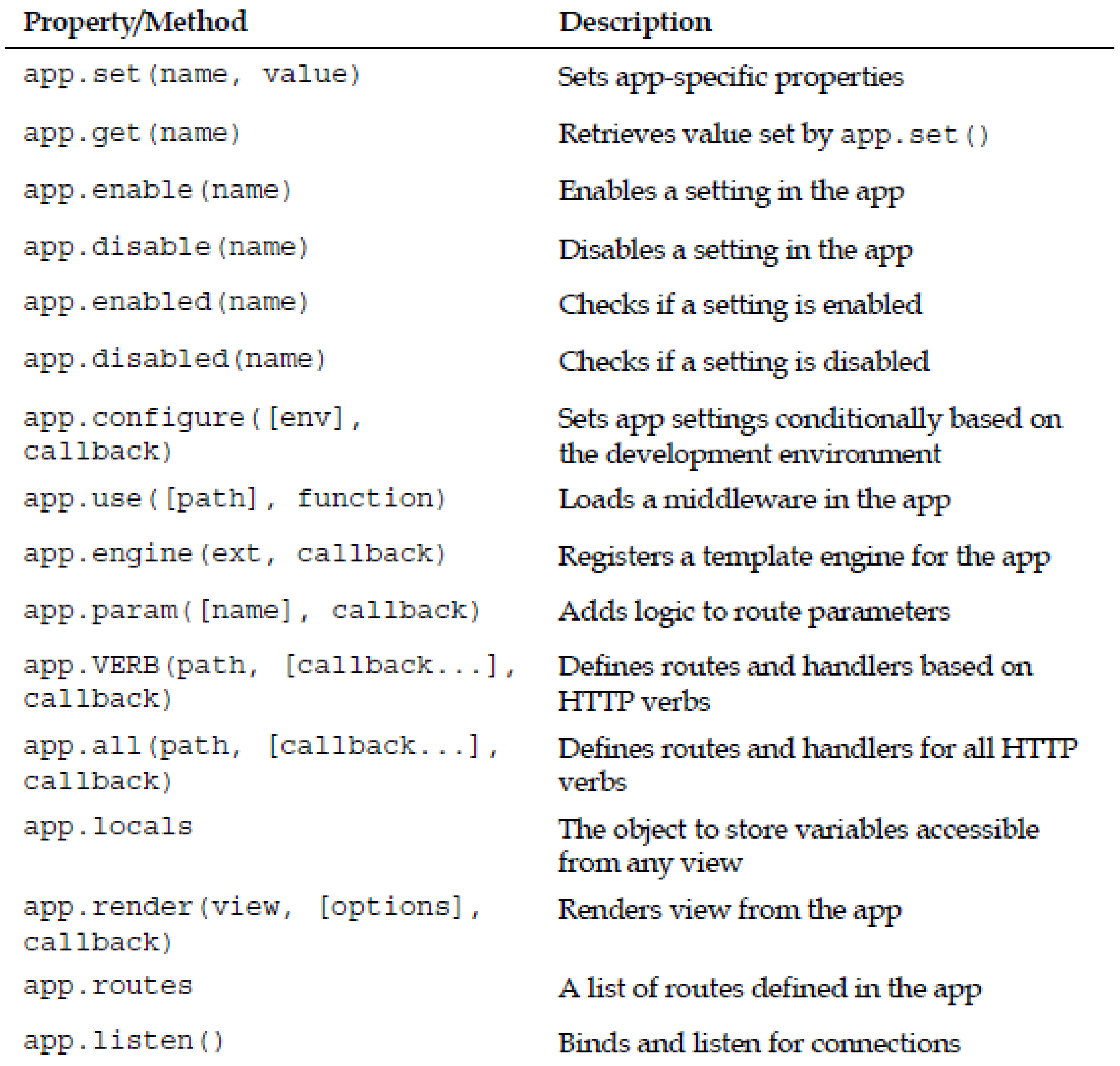
- **Step – 9 : Open browser and type *http://localhost:3000* to view the SampleApp**



KLFS services expressJs training

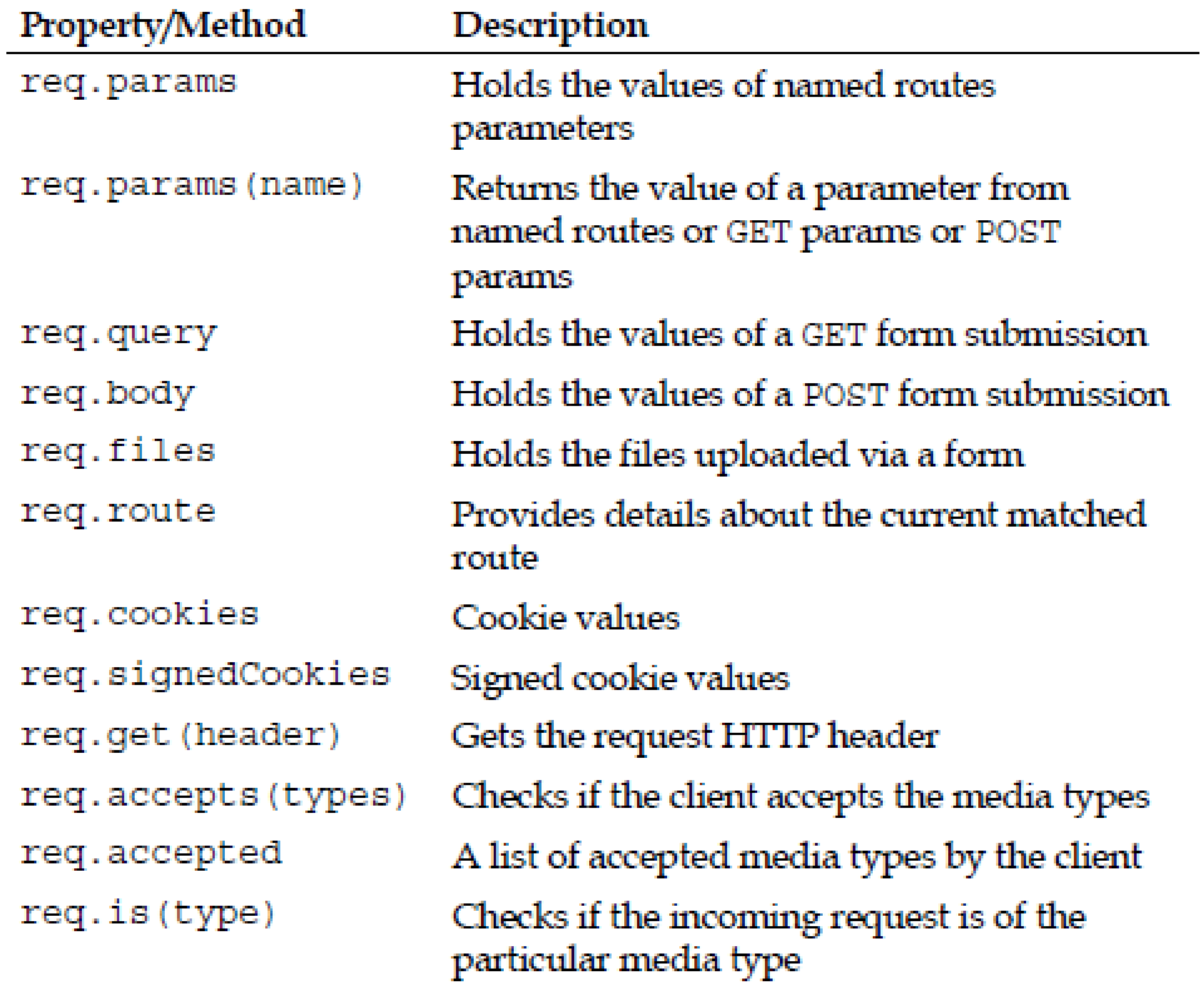
Welcome to KLFS services expressJs training

application object properties & methods



Property/Method	Description
`app.set(name, value)`	Sets app-specific properties
`app.get(name)`	Retrieves value set by `app.set()`
`app.enable(name)`	Enables a setting in the app
`app.disable(name)`	Disables a setting in the app
`app.enabled(name)`	Checks if a setting is enabled
`app.disabled(name)`	Checks if a setting is disabled
`app.configure([env], callback)`	Sets app settings conditionally based on the development environment
`app.use([path], function)`	Loads a middleware in the app
`app.engine(ext, callback)`	Registers a template engine for the app
`app.param([name], callback)`	Adds logic to route parameters
`app.VERB(path, [callback...], callback)`	Defines routes and handlers based on HTTP verbs
`app.all(path, [callback...], callback)`	Defines routes and handlers for all HTTP verbs
`app.locals`	The object to store variables accessible from any view
`app.render(view, [options], callback)`	Renders view from the app
`app.routes`	A list of routes defined in the app
`app.listen()`	Binds and listen for connections

request object properties & methods



Property/Method	Description
`req.params`	Holds the values of named routes parameters
`req.params(name)`	Returns the value of a parameter from named routes or GET params or POST params
`req.query`	Holds the values of a GET form submission
`req.body`	Holds the values of a POST form submission
`req.files`	Holds the files uploaded via a form
`req.route`	Provides details about the current matched route
`req.cookies`	Cookie values
`req.signedCookies`	Signed cookie values
`req.get(header)`	Gets the request HTTP header
`req.accepts(types)`	Checks if the client accepts the media types
`req.accepted`	A list of accepted media types by the client
`req.is(type)`	Checks if the incoming request is of the particular media type

request object properties & methods

Property/Method	Description
`req.ip`	The IP address of the client
`req.ips`	The IP address of the client, along with that of the proxies it is connected through
`req.stale`	Checks if the request is stale
`req.xhr`	Checks if the request came via an AJAX request
`req.protocol`	The protocol used for making the request
`req.secure`	Checks if it is a secure connection
`req.subdomains`	Subdomains of the host domain name
`req.url`	The request path, along with any query parameters
`req.originalUrl`	Used as a backup for `req.url`
`req.acceptedLanguages`	A list of accepted languages by the client
`req.acceptsLanguage(langauge)`	Checks if the client accepts the language
`req.acceptedCharsets`	A list of accepted charsets by the client
`req.acceptsCharsets(charset)`	Checks if the client accepts the charset
`req.host`	Hostname from the HTTP header

response object properties & methods

Property/Method	Description
`res.status(code)`	Sets the HTTP response code
`res.set(field, [value])`	Sets response HTTP headers
`res.get(header)`	Gets the response HTTP header
`res.cookie(name, value, [options])`	Sets cookie on the client
`res.clearCookie(name, [options])`	Deletes cookie on the client
`res.redirect([status], url)`	Redirects the client to a URL, with an optional HTTP status code
`res.location`	The location value of the response HTTP header
`res.charset`	The charset value of the response HTTP header
`res.send([body|status], [body])`	Sends an HTTP response object, with an optional HTTP response code
`res.json([status|body], [body])`	Sends a JSON object for HTTP response, along with an optional HTTP response code

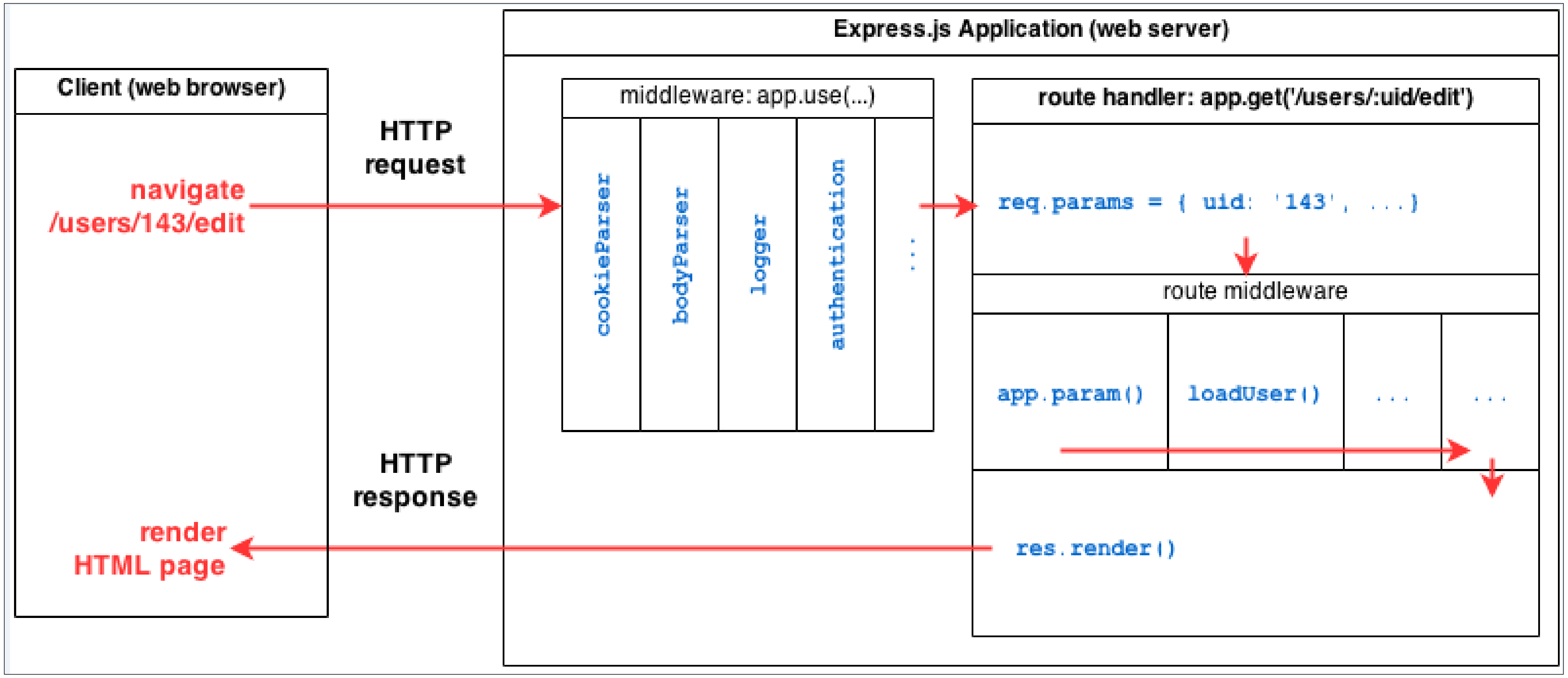
response object properties & methods

Property/Method	Description
`res.jsonp([status|body], [body])`	Sends a JSON object for HTTP response with JSONP support, along with an optional HTTP response code
`res.type(type)`	Sets the media type HTTP response header
`res.format(object)`	Sends a response conditionally, based on the request HTTP Accept header
`res.attachment([filename])`	Sets response HTTP header Content-Disposition to attachment
`res.sendfile(path, [options], [callback]])`	Sends a file to the client
`res.download(path, [filename], [callback])`	Prompts the client to download a file
`res.links(links)`	Sets the HTTP Links header
`res.locals`	The object to store variables specific to the view rendering a request
`res.render(view, [locals], callback)`	Renders a view

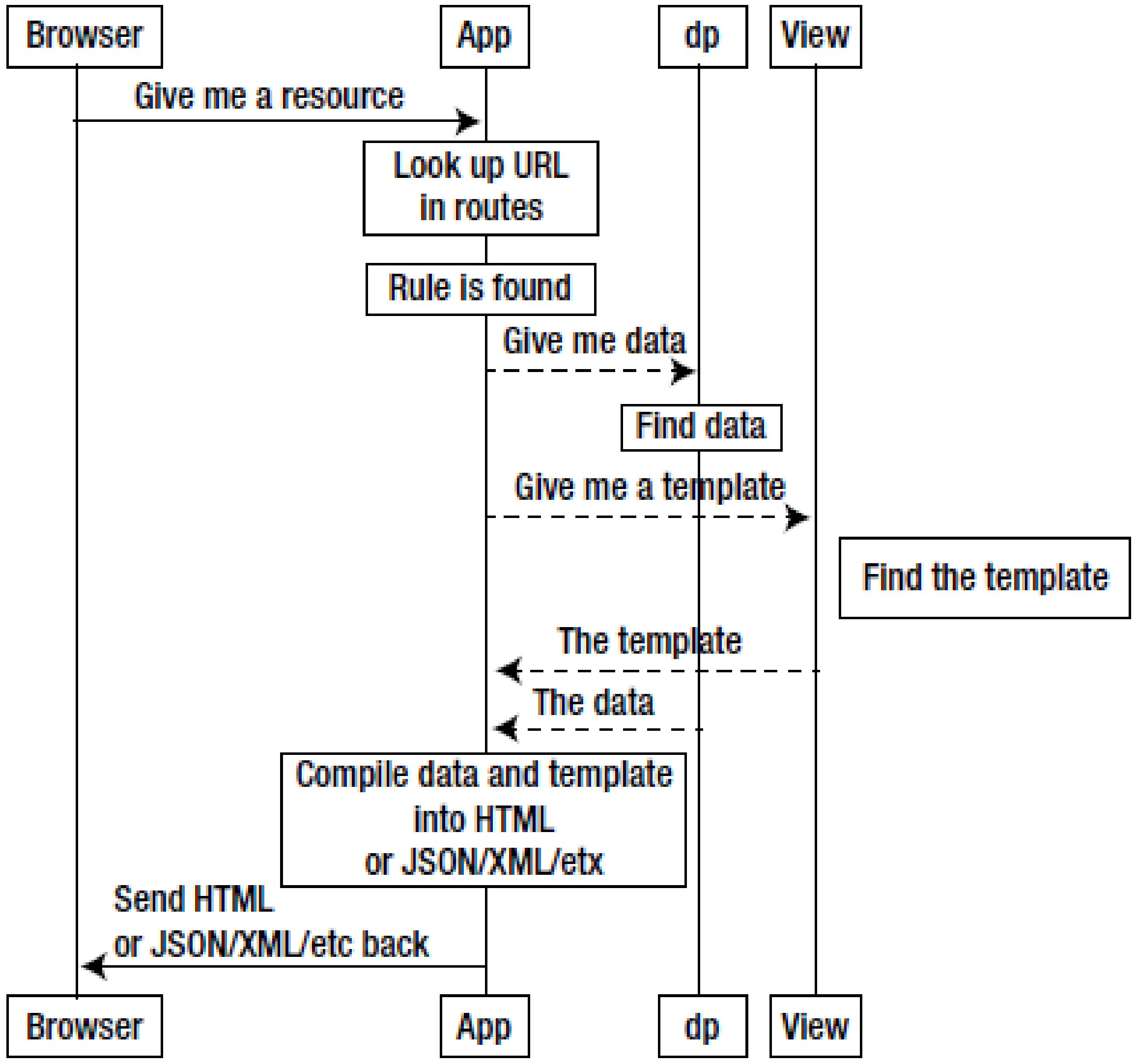
How Express.js works

- **Express.js usually has an entry point, a main file. Most of the time, this is the file that we start with the node command or export as a module. In the main file we do the following**
  - Include third-party dependencies as well as our own modules, such as controllers, utilities, helpers, and models
  - Configure Express.js app settings such as template engine and its file extensions
  - Connect to databases such as MongoDB, Redis, or MySQL (optional)
  - Define middlewares and routes
  - Start the app and Export the app as a module (optional)
- **When the Express.js app is running, it's listens to requests. Each incoming request is processed according to a defined chain of middleware and routes, starting from top to bottom.**

How Express.js works



Request flow in Express



Request flow in Express

- **In Express server the request flow will be :**
  - Route → Route Handler → Template → HTML
- **The route defines the URL schema. It captures the matching request and passed on control to the corresponding route handler**
- **The route handler processes the request and passes the control to a template.**
- **The template constructs the HTML for the response and sends it to the browser.**
- **The route handler need not always pass the control to a template, it can optionally send the response to a request directly.**

Using middleware

- **A middleware is a JavaScript function to handle HTTP requests to an Express app.**
- **It can manipulate the request and the response objects or perform an isolated action, or terminate the request flow by sending a response to the client, or pass on the control to the next middleware.**
- **A middleware can:**
  - Execute any code.
  - Make changes to the request and the response objects.
  - End the request-response cycle.
  - Call the next middleware in the stack.
- **If the current middleware does not end the request-response cycle, it must call next() to pass control to the next middleware, otherwise the request will be left hanging.**

Types of middleware

- **An Express application can use the following kinds of middleware:**
  - Application-level middleware
  - Router-level middleware
  - Built-in middleware
  - Third-party middleware

Express.js Scaffolding

- **To generate a application skeleton for Express.js app, we need to run a terminal command express [options] [dir | appname] the options for which are the following:**
  - -e, --ejs: add EJS engine support (by default, Jade is used)
  - -c <engine>, --css <engine>: add stylesheet <engine> support, such as LESS, Stylus or Compass (by default, plain CSS is used)
  - -f, --force: force app generation on a nonempty directory
  - C:\myapp> express –e -f

Unit Testing, Logging & Debugging

Introduction

- **Unit Testing is nothing but breaking down the logic of application into small chunks or 'units' and verifying that each unit works as we expect.**
- **Unit Testing the code provides the following benefits :**
  - Reduce code-to-bug turn around cycle
  - Isolate code and demonstrate that the pieces function correctly
  - Provides contract that the code needs to satisfy in order to pass
  - Improves interface design
- **Unit testing can be done in 2 ways**
  - Test-Driven Development
  - Test-After Development

Test-Driven Development

- **In Test Driven Development (TDD) automated unit tests are written before the code is actually written. Running these tests give you fast confirmation of whether your code behaves as it should.**
- **TDD can be summarized as a set of the following actions:**
  1. **Writing a test :** In order to write the test, the programmer must fully comprehend the requirements. At first, the test will fail because it is written prior to the feature
  2. **Run all of the tests and make sure that the newest test doesn't pass :** This insures that the test suite is in order and that the new test is not passing by accident, making it irrelevant.
  3. **Write the minimal code that will make the test pass :** The code written at this stage will not be 100% final, it needs to be improved at later stages. No need to write the perfect code at this stage, just write code that will pass the test.

Test-Driven Development

4. **Make sure that all of the previous tests still pass:** If all tests succeeds, developer can be sure that the code meets all of the test specifications and requirements and move on to the next stage.
5. **Refactor code :** In this stage code needs to be cleaned up and improved. By running the test cases again, the programmer can be sure that the refactoring / restructuring has not damaged the code in any way.
6. **Repeat the cycle with a new test :** Now the cycle is repeated with another new test.

- **Using TDD approach, we can catch the bugs in the early stage of development**
- **It works best with reusable code**

Test-After Development

- In Test-After Development, we need to write application code first and then write unit test which tests the application code.
- TAD approach helps us to find existing bugs
- It drives the design
- It is a part of the coding process
- Enables Continual progress and refactoring
- It defines how the application is supposed to behave.
- Ensures sustainable code

Behavior-Driven Development

- **Test-Driven Development (TDD) focus on testing where as Behavior-Driven Development (BDD) mainly concentrates on specification.**
- **In BDD we write test specifications, not the tests which we used to do in TDD i.e. no need focus anymore on the structure of the source code, focus lies on the behavior of the source code**
- **In BDD we need to follow just one rule. Test specifications which we write should follow *Given-When-Then* steps**
- **It is very easy to write / read and understand the test specifications, because BDD follows a common vocabulary instead of a test-centered vocabulary (like test suite, test case, test …)**
- **BDD is a specialized version of TDD that specifies what needs to be unit-tested from the perspective of business requirements.**

Behavior-Driven Development

Story: As a registered user, need to login in to the system to access home page. If username or password are invalid, they will stay in login page and the system will shows an error message.

Scenario: Valid Login
Given The user is in login page
And the user enters a valid username
And the user enters a valid password
When the user logs in
Then the user is redirected to Home page

Scenario: Enter an Invalid password
Given The user is in login page
And the user enters a valid username
And the user enters a invalid password
When the user logs in
Then the user is redirected to the Login Page
And the System shows the following message : "Invalid username or password"

Unit Testing in Node

- In Node there are numerous tools / testing frameworks are available to achieve the objective of having well tested quality code.
- Mocha, Chai and SuperTest integrates very well together in a natural fashion
- Mocha is a testing framework which follows Behavior-driven Development (BDD) interface makes our tests readable and make us very clear what is being tested.
- Mocha allows to use any assertion library like (chai, should.js, expect.js)
- Chai is a BDD / TDD assertion library for node and the browser that can be delightfully paired with any testing framework like Mocha.
- SuperTest is a HTTP testing library which has a bunch of convenience methods for doing assertions on headers, statuses and response bodies. If the server is not already listening for connections then it is bound to an ephemeral port, so there is no need to keep track of ports.

Mocha

- **Mocha is a mature and powerful testing framework for Node.js.**
- **It is more robust and widely used. NodeUnit, Jasmine and Vows are the other alternatives.**
- **Mocha supports both BDD interface's like (describe, it, before) and traditional TDD interfaces (suite, test, setup).**
- **We need explicitly tell Mocha to use the TDD interface instead of the default BDD by specifying *mocha -u tdd <testfile>***
  - **describe** *analogous to* **suite**
  - **it** *analogous to* **test**
  - **before** *analogous to* **setup**
  - **after** *analogous to* **teardown**
  - **beforeEach** *analogus to* **suiteSetup**
  - **afterEach** *analogus to* **suiteTeardown**
- **The mocha package can be installed via**
  - *npm install mocha --save-dev*

Steps to debug

- **Install node-inspector as a global module**
  - npm install –g node-inspector
- **There are two steps needed to get you up and debugging**
- **Step – 1: Start the Node Inspector Server**
  - ***node-inspector***
- **Step – 2: Enable debug mode in your Node process**
  - To start Node with a debug flag ***node --debug program.js***
  - To pause script on the first line ***node –debug-brk program.js***