WEB TECHNOLOGIES

Express JS –

Introductions to Web Services and REST APIs

Aisha Begam

Department of Computer Science and Engineering.

Introductions to Web Services and REST APIs

- A Web service is a service that lives on the Web.

- The **Web** is a huge information space filled with interconnected resources.

- A **service** is an application that can be consumed by software.

- An interface hides the complexities of the internal system.

- One service can support many applications.

- A programmable application component that is accessible via standard Internet protocols.

- W3C definition: *Software system designed to support interoperable machine-to-machine interaction over a network.*

- A Web service is an application with a Web API.

Characteristics of Web Services
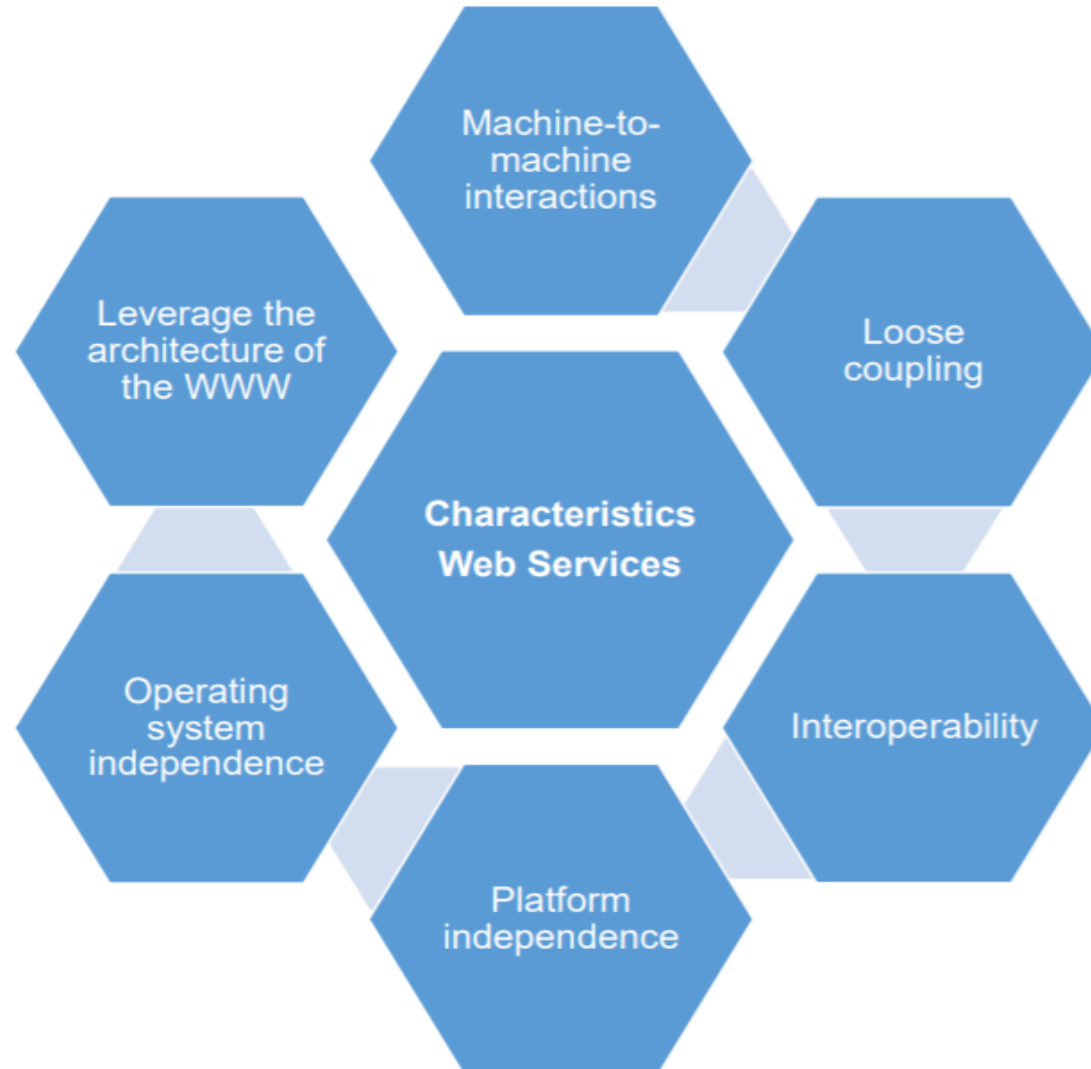
- REST is about resources and how to represent resources in different ways.

- REST is about client-server communication.

- REST is about how to manipulate resources.

- REST offers a simple, interoperable and flexible way of writing web services that can be very different from other techniques.

- **Re**presentational **S**tate **T**ransfer

- REST is an architecture all about the Client-Server communication .

- Guided by REST constraints (design rules).

- Based on Resource Oriented Architecture.

- A network of web pages where the client progresses through an application by selecting links.

- Requests/responses relate to representations of states of a resource.

- When client traverses link, accesses new resource (i.e., transfers state).

- Uses simple HTTP protocol.

- Client **request**s a specific **resource** from the server.

- The server **respond**s to that request by delivering the requested resource.

- Server does not have any information about any client.

- So, there is no difference between the two requests of the same client.

- A model which the representations of the resources are transferred between the client and the server.

- The Web as we know is already in this form!

Resources

- Resources are consistent mappings from an identifier [such as a URL path] to some set of views on server-side state.

- Every resource must be uniquely addressable via a URI.

- "If one view doesn't suit your needs, then feel free to create a different resource that provides a better view. "

- "These views need not have anything to do with how the information is stored on the server. They just need to be understandable (and actionable) by the recipient." *Roy T. Fielding*

RESTful Design Specifications (Constraints)

Client-Server
- Separation of concerns - user interface vs data storage
- Client and server are independent from eachother

Uniform Interface
- All resources are accessed with a generic interface (HTTP-based) which remains same for all clients.

Stateless
- Each request from client to server must contain all of the information
- No client session data or any context stored on the server

Layered System
- Allows an architecture to be composed of hierarchical layers
- Each component cannot "see" beyond the immediate layer.

Cacheable
- Specify data as cacheable or non cacheable
- HTTP responses must be cacheable by the clients

Code On-Demand
- REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts.

HTTP Methods supported by REST:

GET – Requests a resource at the request URL

- Should <u>not</u> contain a request body, as it will be discarded.

- <u>May</u> be cached locally or on the server.

- May produce a resource, but should not modify on it.

POST – Submits information to the service for processing. Should typically return the new or modified resource.

PUT – Add a new resource at the request URL

DELETE – Removes the resource at the request URL

OPTIONS – Indicates which methods are supported

HEAD – Returns meta information about the request URL

Express JS
REST APIs

Table 5-1. *CRUD Mapping for Collections*

Operation	HTTP Method	Resource	Example	Remarks
Read – List	GET	Collection	GET /customers	Lists objects (additional query string can be used to filter)
Read	GET	Object	GET /customers/1234	Returns a single object (query string may be used to filter fields)
Create	POST	Collection	POST /customers	Creates an object, and the object is supplied in the body.
Update	PUT	Object	PUT /customers/1234	Replaces the object with the object supplied in the body.
Update	PATCH	Object	PATCH /customers/1234	Modifies some attributes of the object, specification in the body.
Delete	DELETE	Object	DELETE /customers/1234	Deletes the object

RESTful Design Considerations : Steps for designing RESTful Web Service

- Identifying resources the service will expose over the network.

- Designing the URI Templates – map URIs to resources

- Applying the Uniform HTTP Interface – options available on each resource for different user groups.

- Security Considerations – Authentication and authorization

RESTful Design Considerations : Steps for designing RESTful Web Service

- Designing the Resource Representations – XML/JSON.

- Supporting alternate Representations – XML or JSON based on filters

- Providing Resource Metadata – Ability to discover resources and options

RESTful Design Considerations : RESTful Service Implementation Considerations

- Parse the incoming request to
    - Use URI to identify the resource.
    - Identify URI variables (and map them to resource variables)
    - HTTP method used in the request (and whether it's allowed for the resource).
    - Read the resource representation

- Authenticate and authorize the user.

RESTful Design Considerations : RESTful Service Implementation Considerations

- Use all of this information to perform the underlying service logic.

- Generate an appropriate HTTP response, including
  - Proper status code
  - Description
  - Outgoing resource representation in the response entity body (if any)

REQUEST

GET /news/ HTTP/1.1

Host: example.org

Accept-Encoding: compress, gzip

User-Agent: Python- httplib2

Here is a **GET** request to «http://example.org/news/»

Method = **GET**

RESPONSE

HTTP/1.1 200 Ok

Date: Thu, 07 Aug 2008 15:06:24 GMT

Server: Apache

ETag: "85a1b765e8c01dbf872651d7a5"

Content-Type: text/html

Cache-Control: max-age=3600

<!DOCTYPE HTML>

- The request is to a resource identified by a **URI**
- (**URI** = **U**nified **R**esource **I**dentifier).

- In this case, the resource is «http://example.org/news/»

- Resources, or addressability is very important.

- Every resource is URL-addressable.

- To change system state, simply change a resource.

WEB TECHNOLOGIES

Express JS

Express is a fast, assertive, essential and moderate web framework of Node.js. Acts as a layer built on the top of the Node.js that helps manage a server and routes. It provides a robust set of features to develop web and mobile applications.

Why Express JS ?

- It can be used to design single-page, multi-page and hybrid webapplications.
- It allows to setup middlewares to respond to HTTP Requests.
- It defines a routing table which is used to perform different actions based on HTTP method and URL.
- It allows to dynamically render HTML Pages based on passing arguments to templates.

WEB TECHNOLOGIES

Express JS

Firstly, you have to install the express framework globally to create web

application using Node terminal. Use the following command to install

express framework globally.

npm install -g express

Use the following command to install express:

npm install express --save

The above command install express in node_module directory and create a

directory named express inside the node_module.

WEB TECHNOLOGIES

Express JS

You should install some other important modules along with express.

Following is the list:

- **body-parser:** This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data. npm install body-parser --save

- **cookie-parser:** It is used to parse Cookie header and populate req.cookies with an object keyed by the cookie names. npm install cookie-parser --save

- **multer:** This is a node.js middleware for handling multipart/form-data. npm install multer --save

Install Express JS

Basic Server Syntax
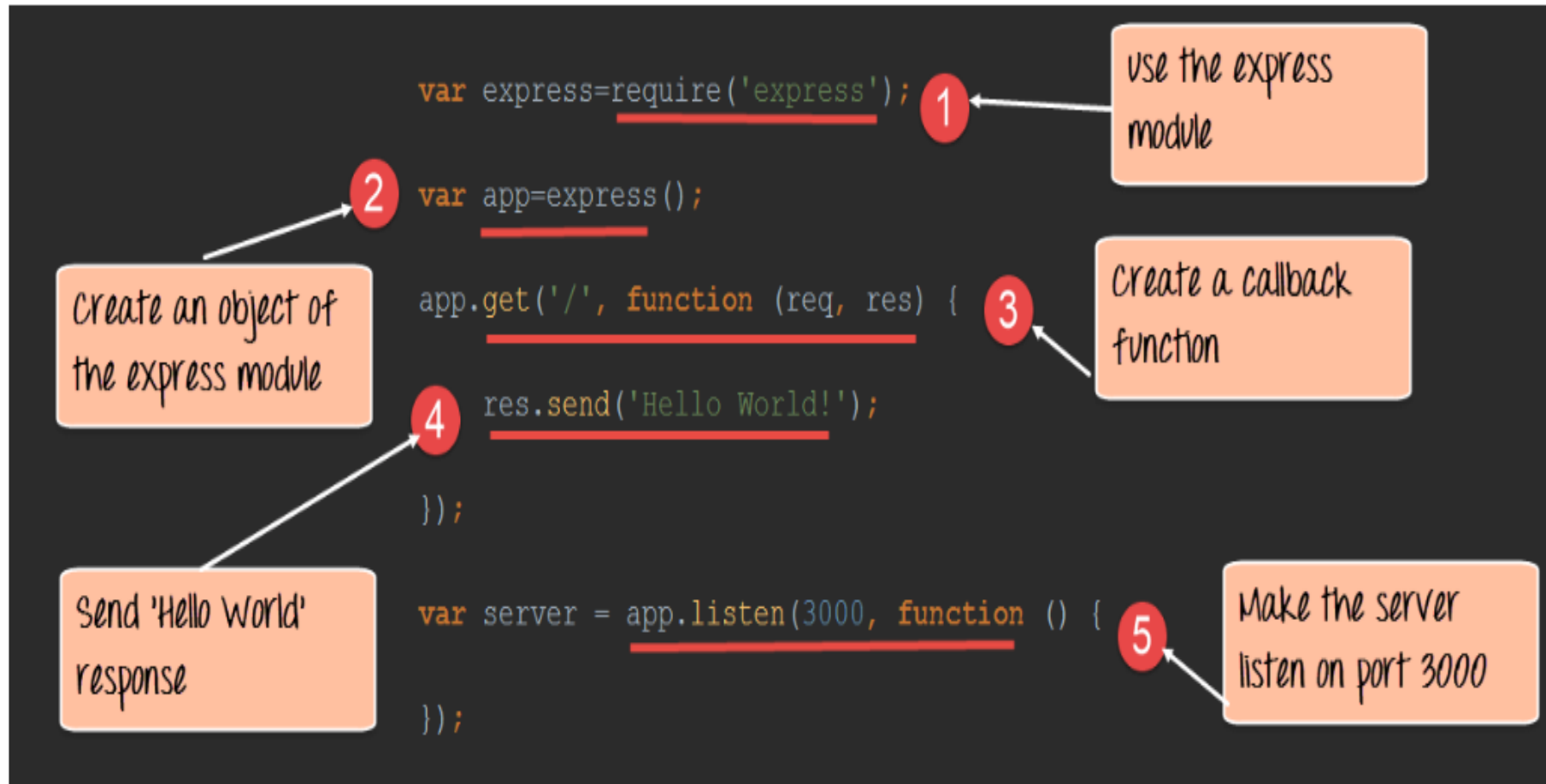
WEB TECHNOLOGIES
Express JS

Basic Route Handling

- Handling requests/route is **simple**

- **app.get(), app.post(), app.put(), app.delete() etc.**

- Access to params, querystring, url parts etc

- **Express has routers** so we can store routes in separate files and export

- We can parse incoming data with **Body Parser.**

```
app.get ('/', function(req, res) {
  // Fetch from database
  // Load pages
  // Return JSON
  // Full access to request & response
});
```

Building Application Stack

Building Application Stack

- What is MERN Stack?

- A stack is the mixture of technologies used to create Web applications.

- Any web application will be made utilizing various technologies like (frameworks, libraries, databases).

- The MERN stack is a JavaScript stack that is intended to make Application Development process smoother.

WEB TECHNOLOGIES
Building Application Stack

MERN Stack Components:

• **MongoDB:** A document-oriented, No-SQL database used to store the application data.

• **NodeJS**: The JavaScript runtime environment. Used to run JavaScript on a machine rather than in a browser.

• **ExpressJS**: A framework layered on top of NodeJS, used to build the backend of a site using NodeJS functions and structures.

• **ReactJS**: A library created by Facebook. It is used to build UI components that create the user interface of the single page web application.

Building Application Stack

Benefits of MERN Stack

• Java script is the programming language utilized both for client side and server-side.

•For tech stack with different programming languages, developers need to find out how to interface them together. With the JavaScript stack, developers should be proficient in JavaScript and JSON.

•Using the MERN stack enables developers to build highly efficient web applications.

EXPRESSJS: ROUTING

Routing

- Routing refers to the mechanism for serving the client the content it has asked for.

- It is the most important aspects of your website or web services.

- Routing in Express is simple, flexible, and robust.

- A route specification consists of
  - An HTTP method (GET, POST, etc.),
  - A path specification that matches the request URI,
  - And the route handler.

Routing

- The handler is passed in a request object and a response object.

- The request object can be inspected to get the various details of the request, and

- The response object's methods can be used to send the response to the client

EXPRESSJS: ROUTING

Routing

•Create an application using its root-level exported function

• *const app = express();* Set up routes using this app.

• To set up a route, use a function to indicate which HTTP method; for e.g., to handle the GET,PUT, DELETE etc.,

•To this function, pass the pattern to match and a function to deal with the request if it does match.

```
const app = express();

app.get('/hello', (req, res) => {
    res.send('Hello World');
});
```

EXPRESSJS: ROUTING

Request Matching

•The request's method is matched with the route method (e.g., the get function was called on app, indicating it should match only GET HTTP methods)

•The request URL with the path spec matches ('/hello'), then the handler is called.

•In other words, "If you receive a GET request to the URL */hello*, then execute *this piece of code*."

```
const app = express();

app.get('/hello', (req, res) => {
    res.send('Hello World');
});
```

EXPRESSJS: ROUTING

Request Matching

• The method and the path need not be very specific. E.g., app.get(), app.post(), app.put(), etc.,

```
const app = express();

app.get('/hello', (req, res) => {
  res.send('Hello World');
});
```

• If you want to say "any method," you could use app.all().

• The path specification can also take regular expression-like patterns (like '/*.do') or regular expressions themselves.

• Regular expressions in paths are rarely used. Route *parameters in the path are used often.*

• They are named segments in the path specification that match a part of the URL.

EXPRESSJS: ROUTING

Route Parameters

• If a match occurs, the value in that part of the URL is supplied as a variable in the request object.

```
app.get('/customers/:customerId', …
```

• The customer ID will be captured and supplied to the handler function as part of the request in req.params, with the name of the parameter as the key.

• req.params.customerId can have any value for each of these URLs and can have multiple parameters.

• For e.g., /customers/:customerId/orders/:orderId, to match *a customer's order.*

WEB TECHNOLOGIES
EXPRESSJS: ROUTING

Route Lookup

- Multiple routes can be set up to match different URLs and patterns.

- The router tries to match all routes in the order in which they are installed.

- If two routes are possible matches to a request, it will use the first defined one.

- Can define routes in the order of priority.

EXPRESSJS: ROUTING

Route Lookup

When you add patterns its recommended to add more generic pattern *after the specific paths.*

• *For e.g, if you want to match everything that* goes under /api/, that is, a pattern like /api/*,

• Add this route only *after all the* specific routes that handle paths such as /api/issues.

EXPRESSJS: ROUTING

Handler Function

•Once a route is matched, the handler function is called.

•The parameters passed to the handler are a request object and a response object.

Request Objects

1. To access a parameter value we use **req.params.**

   req.param(name [, defaultValue])

EXPRESSJS: ROUTING

Request Objects

2*. req.query*: This holds a parsed query string.
- •It's an object with keys: as the query string parameters and
- •Values as the query string values.
- •Multiple keys with the same name are converted to arrays, and
- •Keys with a square bracket notation result in nested objects

(e.g., order[status]=closed can be accessed as req.query. order.status).

```
{ key: value }
```

```
?key1=value1&key2=value2&key3=value3
```

EXPRESSJS: ROUTING

Request Objects

3. *req.header, req.get(header)*:
- The get method gives access to any header in the request.
- The header property is an object with all headers stored as key-value pairs.

4. *req.path:*
The path for which the middleware function is invoked; can be any of:

- A string representing a path.
- A path pattern.
- A regular expression pattern to match paths.
- An array of combinations of any of the above.

Request Objects

5. *req.url, req.originalURL*:
- Contain the complete URL, including the query string.
- If any middleware modifies the request URL, originalURL retains the original URL as it was received, *before the modification.*

6. *req.body:*
- Contains key-value pairs of data submitted in the request body.
- By default, it is undefined, and is populated when you use body-parsing middleware is installed to read and optionally interpret or parse the body.

EXPRESSJS: ROUTING

Response Objects

The *res* object represents the HTTP response that an Express app sends when it gets an HTTP request.

```
res.send('<p>some html</p>')
```

1. **res.send(body)**: Sends the HTTP response.
   - If the body is an object or an array, it is automatically converted to a JSON string with an appropriate content type.

EXPRESSJS: ROUTING

Response Objects

The **res** object represents the HTTP response that an Express app sends when it gets an HTTP request.

2. **res.status(code)**: This sets the response status code.
- If not set, it is defaulted to 200 OK.
- One common way of sending an error is by combining the status() and send() methods in a single call like res.status(403).send("Access Denied").

```
res.status(400).send('Bad Request')
```

EXPRESSJS: ROUTING

Response Objects

3. **res.json(object)**: Sends a JSON response

- This method sends a response (with the correct content-type) that is the parameter converted to a JSON string using JSON.stringify().
- The parameter can be any JSON type, including object, array, string, Boolean, number, or null

```
res.json({ user: 'tobi' })
```

Response Objects

4. ***res.sendFile(path*)**:

- This responds with the contents of the file at path.
- The content type of the response is guessed using the extension of the file.

```
res.sendFile('/uploads/' + uid + '/' + file)
```

WEB TECHNOLOGIES
EXPRESSJS: Middleware

- An Express application is a series of middleware function calls.

- Router is nothing but a middleware function.

- Middleware functions have access to the request and response object (req,res), and the next middleware function in the application's request response cycle.

- The next middleware function is commonly denoted by a variable named **next**.

- **next** is the only built-in middleware (other than the router) available as part of Express.

WEB TECHNOLOGIES
Middleware

- Middleware can be at
    - The application level (i.e.,, applies to all requests) or
    - The router level (applies to specific request path patterns).

- The Middleware at the application level can be used like this:
    app.use(middlewareFunction);

- The static middleware that knows the location of static files to serve.

Middleware

•In order to use the same middleware in a route-specific way, you define as: **app.use('/public', express.static('static'));**

•This would have mounted the static files on the path /public and all static files would have to be accessed with the prefix /public,

•For e.g.,: /public/index.html.

- The **express.static** generator function generates one such middleware function.

- This middleware responds to a request by trying to match the request URL with a file under a directory specified by the parameter to the generator function.

- If a file exists,
    - It returns the contents of the file as the response; else
    - It chains to the next middleware function.

- The middleware is *mounted on the application using the **application's use()** method.*

MIDDLEWARE

Express JS

- The middleware generator takes the parameter static to indicate that this is the directory where all the static files reside.

```javascript
const express = require('express');

const app = express();
app.use(express.static('static'));

app.listen(3000, function () {
  console.log('App started on port 3000');
});
```

Express JS

- The **express.static** generator function generates one such middleware function.
- Helps to serve static files

```javascript
const express = require('express');
const path = require('path');

const app = express();

//set static folder
app.use(express.static(path.join (__dirname,'public')));

const PORT = process.env.PORT ||5000;

app.listen(PORT, () => console.log(`Server started on port:${PORT}`));
```

MIDDLEWARE

Express JS

- Created a public folder and added index.html and about.html.

- Parameter static to indicate that this is the directory where all the static files reside


```html
log  package.json  issues.js  new 1  myapp.js  index.html  about.html
<!doctype html>
<html lang="en">
  <head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, intial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/style.css"/>
    <title>My website</title>
  </head>
  <body>
    <h1>My Website!!!</h1>
    </body>
</html>
```

MIDDLEWARE

Express JS

- Created a public folder and added index.html and about.html.

- Parameter static to indicate that this is the directory where all the static files reside

```
og ☒ | 💾 package.json ☒ | 💾 issues.js ☒ | 💾 new 1 ☒ | 💾 myapp.js ☒ | 💾 index.html ☒ | 💾 about.html ☒ |
<!doctype html>
<html lang="en">
  <head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, intial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/style.css"/>
    <title>About</title>
  </head>
  <body>
    <h1>About homepage</h1>
    </body>
</html>
```

WEB TECHNOLOGIES
MIDDLEWARE

Express JS

- Output in the browser
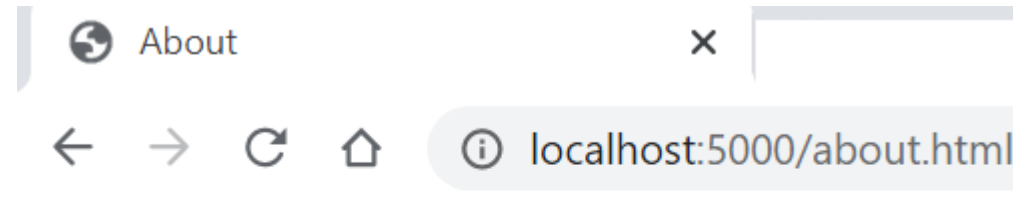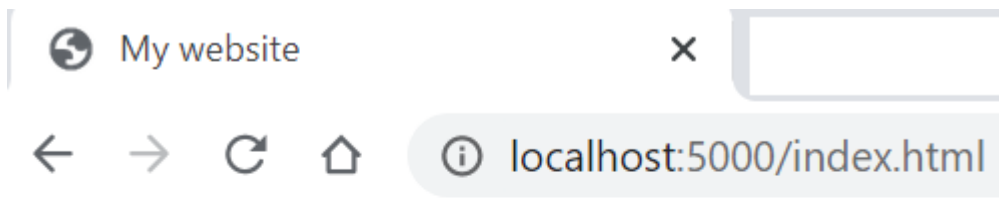
MIDDLEWARE

Express JS

- To use a middleware at the application level supply the function to the application,
  app.use(middlewareFunction);

- To use the same middleware in a route-specific way, you could have done the following:
  app.use('/public', express.static('static'));

- This would have mounted the static files on the path /public and all static files would have to be accessed with the prefix /public, for example, /public/index.html.

WEB TECHNOLOGIES
MIDDLEWARE

- The JSON parser middleware is created using ***bodyParser.json(),*** and is mounted at the application level using ***app.use().***

- The middleware ***bodyParser*** looks at the Content-Type header and determines if and how the body can be parsed.

- For JSON, the default content type is application/json.

- In the absence of this header, ***bodyParser*** does not parse the request body, and the variable ***req.body*** ends up as an empty object.

MIDDLEWARE

Express JS

- Initializing a middleware called logger which will print Hello in the console when a request is made.

```js
const express = require('express');
const path = require('path');

const app = express();

const logger=(req,res,next)=>{
    console.log('Hello');
    next();
};
//init middleware
app.use(logger);

//set static folder
app.use(express.static(path.join (__dirname,'public')));

//issues API routes..
app.use('/api/issues', require('./routes/api/issues'));

const PORT = process.env.PORT ||5000;

app.listen(PORT, () => console.log(`Server started on port:${PORT}`));
```

COOKIES

- To use cookies with Express, we will require the **cookie-parser**.

- cookie-parser is a middleware which parses cookies attached to the client request object.

- To use it, we will require it in our main file (**index.js)**; this can be used the same way as we use other middleware.

```
var cookieParser = require('cookie-parser’);
app.use(cookieParser());
```

Express JS

• Cookie-parser parses Cookie header and populates **req.cookies** with an object keyed by the cookie names.

• To set a new cookie, let us define a new route in your Express app like:

```
var express = require('express');
var app = express();
app.get('/', function(req, res){
//Sets name = express
res.cookie('name', 'express').send('cookie set');
});
app.listen(5000);
```

WEB TECHNOLOGIES
COOKIES

Adding Cookies with Expiration Time

- To add a cookie that expires, just pass an object with property 'expire' set to the time when you want it to expire.
- E.g., //Expires after 360000 ms from the time it is set.
- *res.cookie(name, 'value', {expire: 360000 + Date.now()});*

- Another way to set expiration time is using **'maxAge'** property.
- Using this property, we can provide relative time instead of absolute time.
- E.g.,//This cookie also expires after 360000 ms from the time it is set.
res.cookie(name, 'value', {maxAge: 360000});

Deleting Existing Cookies

- To delete a cookie, use the clearCookie function.
- For example, if you need to clear a cookie named **foo**, use the following code.

```
var express = require('express');
var app = express();
 app.get('/clear_cookie_foo', function(req, res){
res.clearCookie('foo');
res.send('cookie foo cleared'); });
app.listen(5000);
```

cookieParser(secret, options)

•Create a new cookie parser middleware function using the given secret and options.

•**secret** a string or array used for signing cookies.

•This is optional and if not specified, will not parse signed cookies.

• If a string is provided, this is used as the secret.

•If an array is provided, an attempt will be made to unsign the cookie with each secret in order.

•**options** an object that is passed to cookie.parse as the second option.

•**decode** a function to decode the value of the cookie

Express JS

- **cookieParser(secret, options)**
- The middleware will parse the Cookie header on the request and expose the cookie data as the property req.cookies and, if a secret was provided, as the property req.signedCookies.
- These properties are name value pairs of the cookie name to cookie value.
- When secret is provided, this module will unsign and validate any signed cookie values and move those name value pairs from req.cookies into req.signedCookies.

Express JS

- **cookieParser(secret, options)**
- A signed cookie is a cookie that has a value prefixed with s:.
- Signed cookies that fail signature validation will have the value false instead of the tampered value.
- In addition, this module supports special "JSON cookies".
- These are cookie where the value is prefixed with j:.
- When these values are encountered, the value will be exposed as the result of JSON.parse.
- If parsing fails, the original value will remain.

Express JS

cookieParser.JSONCookie(str)

Parse a cookie value as a JSON cookie.

This will return the parsed JSON value if it was a JSON cookie, otherwise, it will return the passed value.

cookieParser.JSONCookies(cookies)

Given an object, this will iterate over the keys and call JSONCookie on each value, replacing the original value with the parsed value.

This returns the same object that was passed in.

Express JS

cookieParser.signedCookie(str, secret)

•Parse a cookie value as a signed cookie.

•This will return the parsed unsigned value if it was a signed cookie and the signature was valid.

• If the value was not signed, the original value is returned.

• If the value was signed but the signature could not be validated, false is returned.

•The secret argument can be an array or string.

• If a string is provided, this is used as the secret.

• If an array is provided, an attempt will be made to unsign the cookie with each secret in order.

Express JS

cookieParser.signedCookies(cookies, secret)
•Given an object, this will iterate over the keys and check if any value is a signed cookie.
•If it is a signed cookie and the signature is valid, the key will be deleted from the object and added to the new object that is returned.
•The secret argument can be an array or string. If a string is provided, this is used as the secret.
•If an array is provided, an attempt will be made to unsign the cookie with each secret in order.

```
var express = require('express');
var cookieParser = require('cookie-parser');
var app = express();
app.use(cookieParser());


app.get('/cookieset',function(req, res){
res.cookie('cookie_name', 'cookie_value');
res.cookie('company', 'Expresscookie');
res.cookie('name', 'jj');
//  cookie with expiration date /res.cookie(name, 'value', {maxAge: 360000});
  //res.cookie('name', 'jj',{expire:360000 + Date.now()});
res.status(200).send('Cookie is set');
});
```

```
app.get('/cookieget', function(req, res) {
  res.status(200).send(req.cookies);
});
app.get('/', function (req, res) {
  res.status(200).send('Welcome to creating express cookies !');
});
app.get('/cookieclear',function (req, res) {
  res.status(200).send('cookie cleared !');
});
app.listen(3000, ()=>console.log("server is listening."))
```

SCAFFOLDING

- Scaffolding is a technique that is supported by some MVC frameworks.

- Scaffolding facilitates the programmers to specify how the application data may be used. This specification is used by the frameworks with predefined code templates, to generate the final code that the application can use for CRUD operations (create, read, update and delete database entries).

- An Express.js scaffold supports candy and more web projects based on Node.js.

- https://www.youtube.com/watch?v=IgUjBWmrgjg (watch this video for reference)

WEB TECHNOLOGIES

SCAFFOLDING

npm install express-scaffold

After this step, execute the following command to install express generator:

npm install -g express-generator
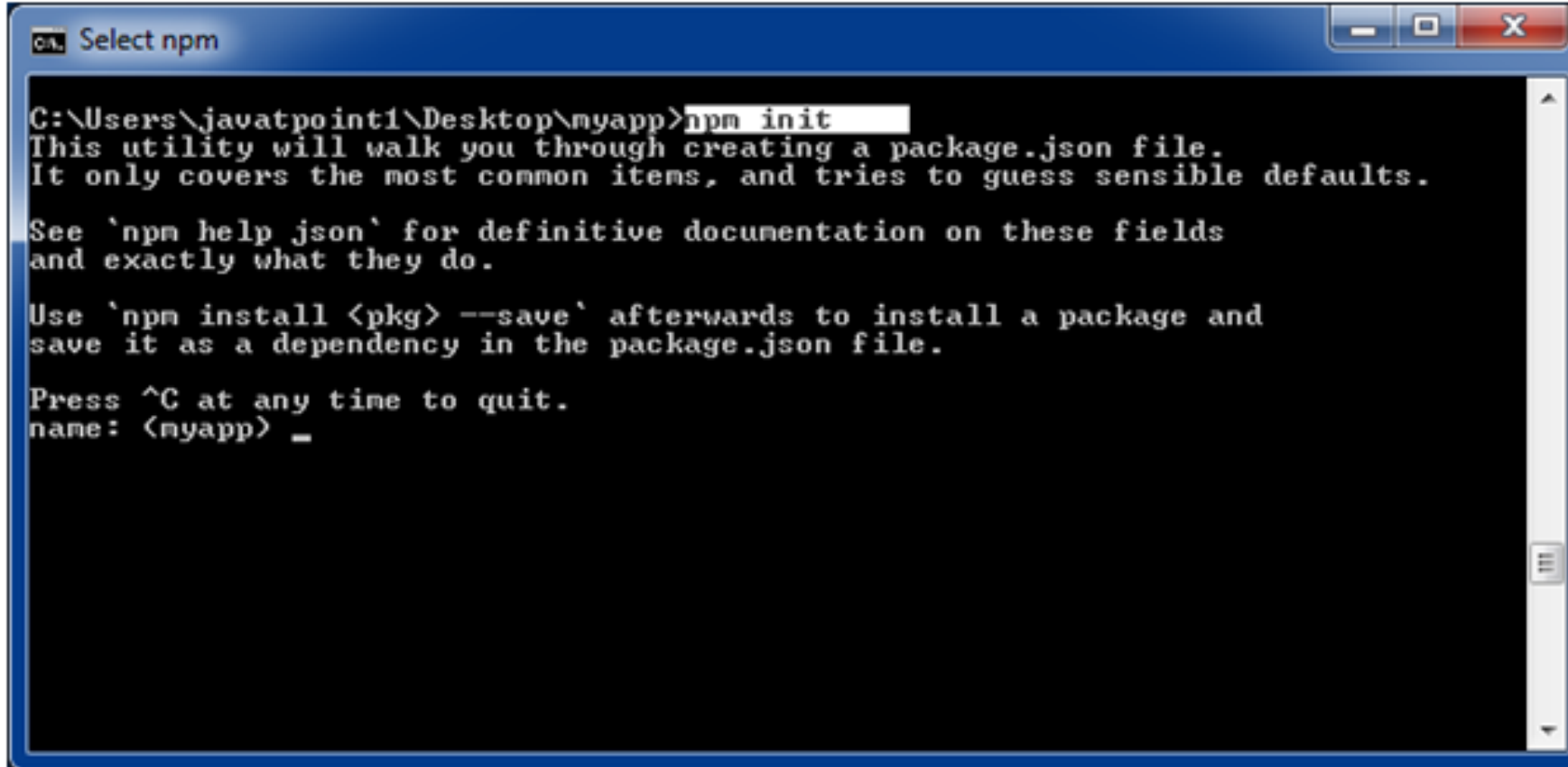
First create a directory named myapp. Create a file named app.js in the myapp directory

```
var express = require('express');
var app = express();
app.get('/', function (req, res) {
  res.send('Welcome to express scaffold')
});
app.listen(8000, function () {
  console.log('Example app listening on port 8000!');
});
```

WEB TECHNOLOGIES

SCAFFOLDING

- Open Node.js command prompt, go to myapp and run npm init command.
- Fill the entries and press enter.
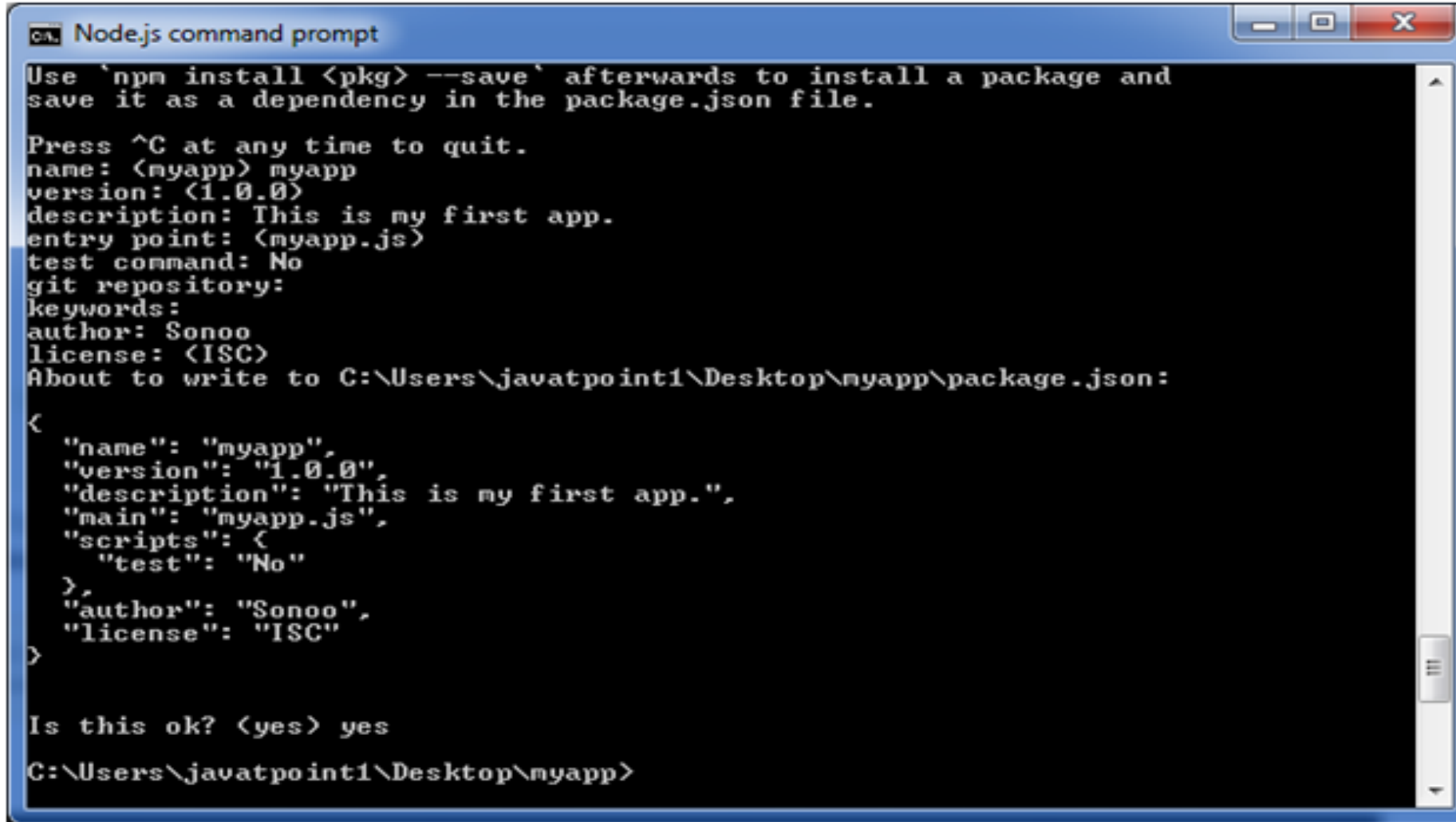
WEB TECHNOLOGIES

SCAFFOLDING

•Fill the entries and press enter.

WEB TECHNOLOGIES
SCAFFOLDING

It will create a package.json file in myapp folder and the data is shown in JSON format.
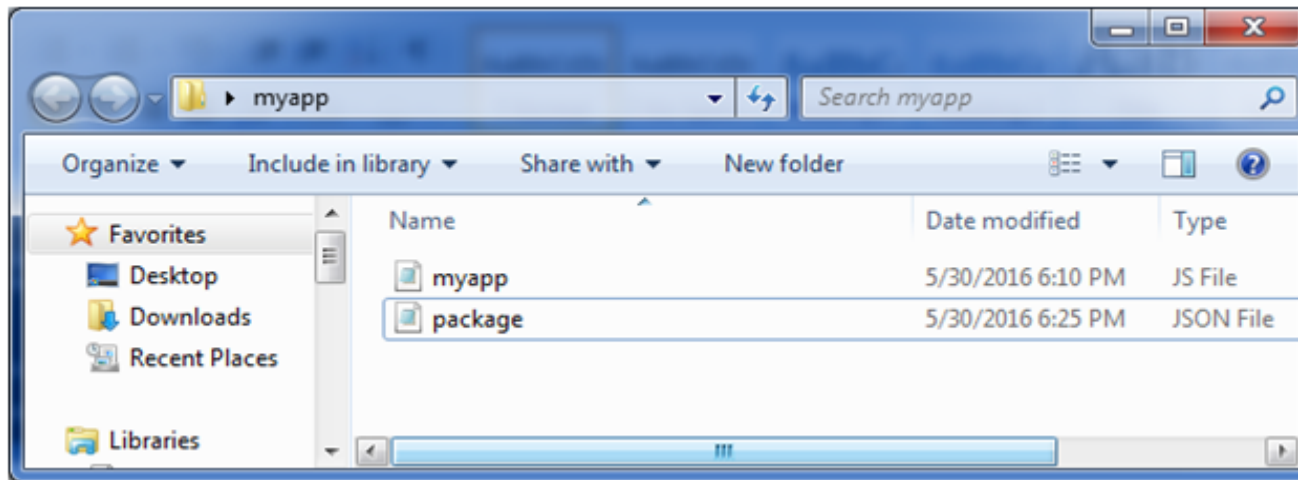
EXPRESS TEMPLATE

- Template engine allows you to use static template files in your applications.

- At runtime, the template engine

  - replaces variables in a template file with actual values, and

  - transforms the template into an HTML file sent to the client.

- This approach makes it easier to design an HTML page.

- Express does not have a template engine built in, but it supports any template engine of your choice such as pug, mustache and EJS

WEB TECHNOLOGIES
EXPRESS TEMPLATE

@ https://www.edureka.co/blog/expressjs-tutorial/

EXPRESS TEMPLATE
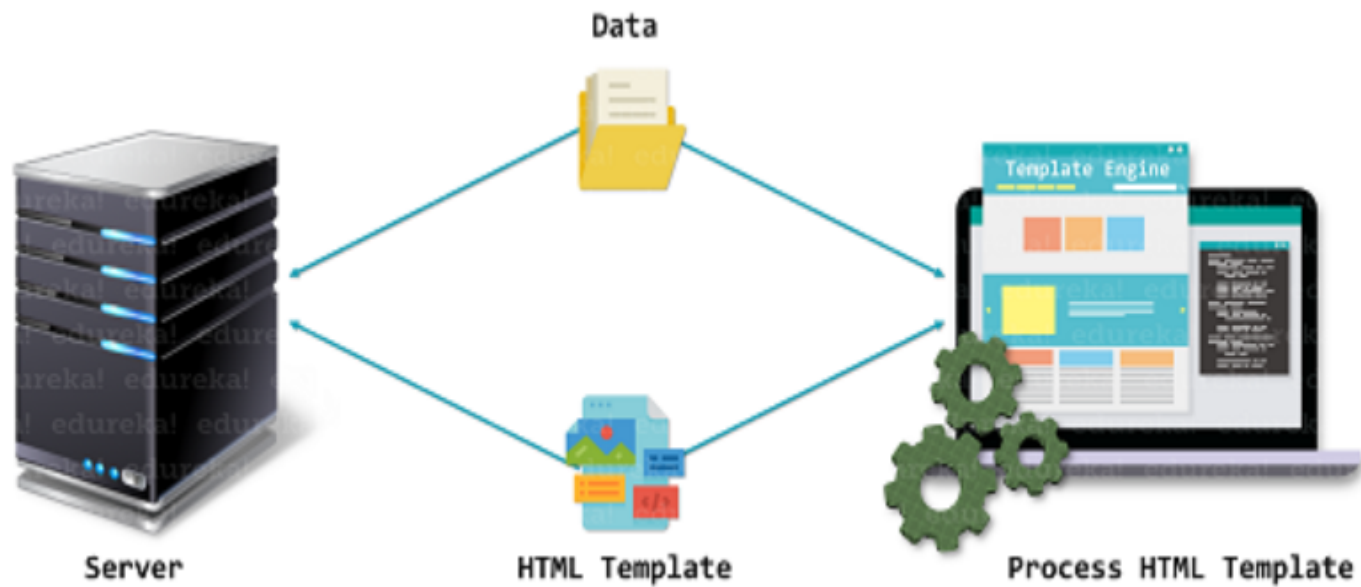
Template engine makes you able to use static template files in your application. To render template files you have to set the following application setting properties:

Views: It specifies a directory where the template files are located.

For example: app.set('views', './views').

view engine: It specifies the template engine that you use. For example, to use the Pug template engine: app.set('view engine', 'pug').

WEB TECHNOLOGIES

EXPRESS TEMPLATE

Pug Template Engine

Pug is a template engine for Node.js. Pug uses whitespaces and indentation as the part of the syntax.

npm install pug --save

Pug template must be written inside .pug file and all .pug files must be put inside views folder in the root folder of Node.js application.

By default Express.js searches all the views in the views folder under the root folder. you can also set to another folder using views property in express.

For example: app.set('views','MyViews').

Create a file named index.pug file inside views folder and write the following pug template in it:

```pug
doctype html
html
    head
        title A simple pug example
    body
        h1 This page is produced by pug template engine
        p some paragraph here..
```

WEB TECHNOLOGIES

Create a file as server.js

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

var app = express();

//set view engine

app.set("view engine","pug")

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

res.render('view.pug', index);

 res.render('index');

});

var server = app.listen(3000, function () {

    console.log('Node server is running..');

});
```

WEB TECHNOLOGIES
FILE UPLOAD

1. install nodejs,make workspace directory and run the commands

    Mkdir fileupload

    Cd fileupload

    Npm init –yes

2. Install rquired packages using npm

    npm install –save express multer body-parser ejs

   npm install –g nodemon(use to run app.ja automatiucally everytime changes are   done to the file)

3. Create app.js file and add the following code and run npm start/nodemon app

4. Define view engine with ejs /public path/view files path/bodyparser

5. Define and configure Multer storage

6. Define index path with '/' and HTML file

7. Define file upload function in app.js

8. Run a server and check in browser using npm start command

9. Define a image upload form with file input.

Express JS

- Simple express middleware for uploading files.
- **npm i express-fileupload**

```
C:\Users\Aisha Begum\express\myexpress>npm i express -fileupload
npm WARN myexpress@1.0.0 No repository field.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@2.1.3 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.1.3: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

+ express@4.17.1
updated 1 package and audited 180 packages in 7.902s

8 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities


C:\Users\Aisha Begum\express\myexpress>
```

Express JS

- When you upload a file, the file will be accessible from req.files

- It is similar to req.body and is turned on either by express.bodyParser() or express.multipart() middlewares.

- Express.js (and other modules behind the scene) process the request data (which is usually a form) and give us extensive information in the req.files.FIELD_NAME object.

Express JS

To illustrate how req.files works, let's add this route to the req project:

```
app.post('/upload', function(req, res){
console.log(req.files.archive);
//read req.files.archive.path
//process the data
//save the data
res.end();
})
```

Express JS

Example:

- You're uploading a file called **car.jpg**
- Your input's name field is **foo**: <input name="foo" type="file" />
- In your express server request, you can access your uploaded file from req.files.foo:

```
app.post('/upload', function(req, res) {
  console.log(req.files.foo); // the uploaded file object
});
```

Express JS

The **req.files.foo** object will contain the following:
req.files.foo.name: "car.jpg"

req.files.foo.mv: A function to move the file elsewhere on your server. Can take a callback or return a promise.

req.files.foo.mimetype: The mimetype of your file

req.files.foo.data: A buffer representation of your file, returns empty buffer in case useTempFiles option was set to true.

Express JS

The **req.files.foo** object will contain the following:

req.files.foo.tempFilePath: A path to the temporary file in case useTempFiles option was set to true.

req.files.foo.truncated: A boolean that represents if the file is over the size limit
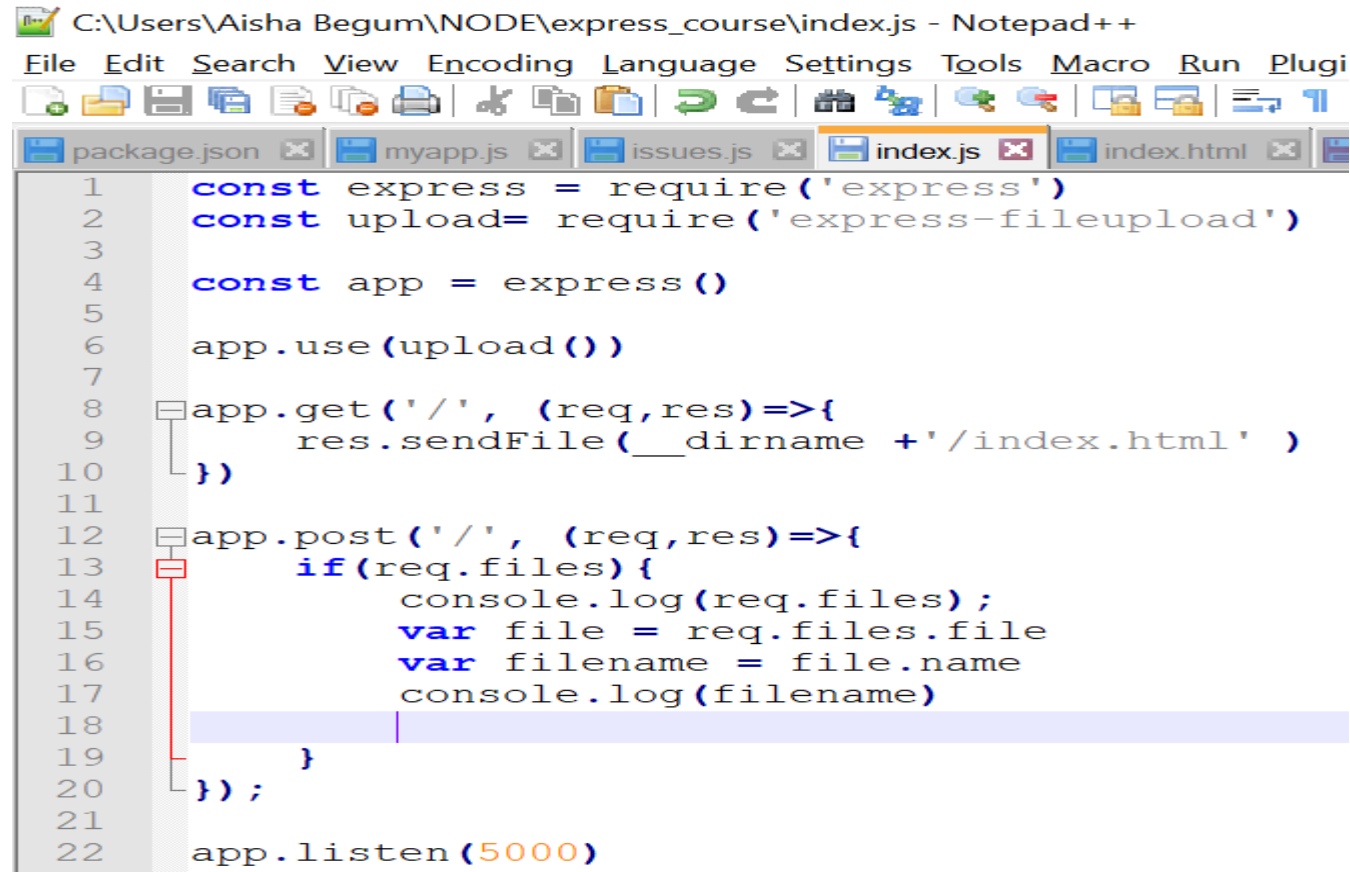
req.files.foo.size: Uploaded size in bytes

req.files.foo.md5: MD5 checksum of the uploaded file

WEB TECHNOLOGIES
FILE UPLOAD

Express JS

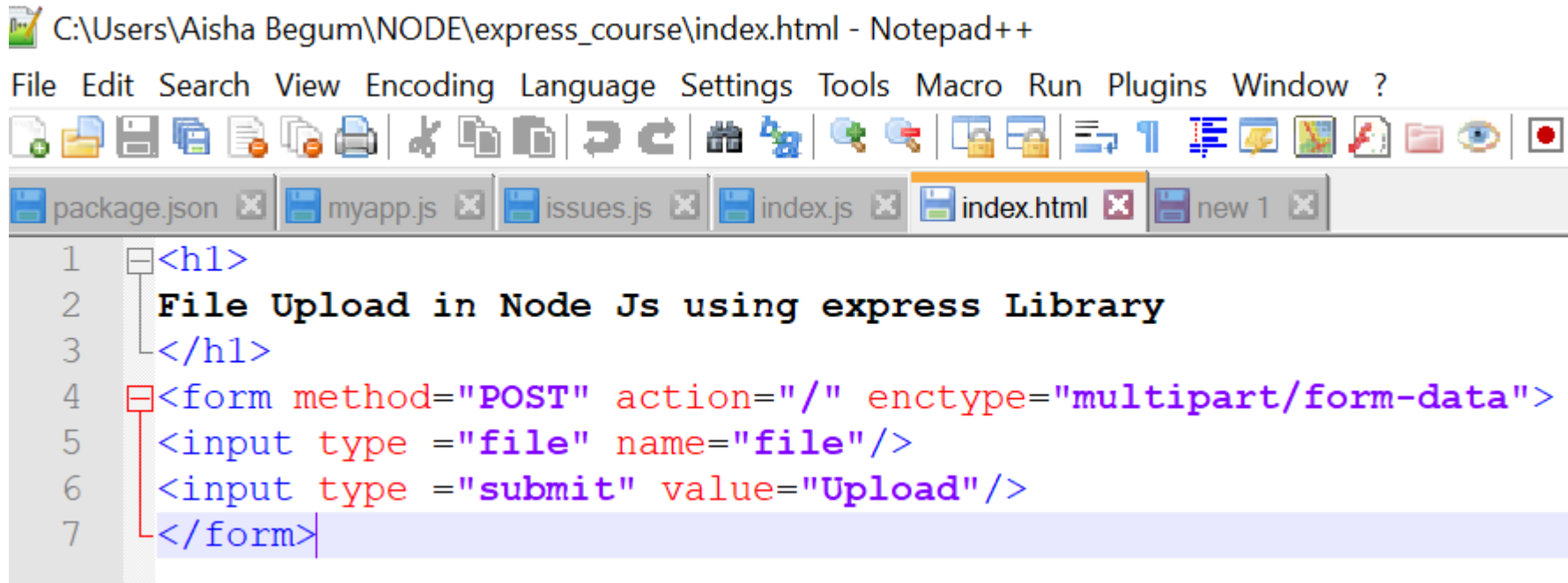Index.js for printing out the file name and details of uploaded file


```
C:\Users\Aisha Begum\NODE\express_course\index.js - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugi

package.json    myapp.js    issues.js    index.js    index.html

1    const express = require('express')
2    const upload= require('express-fileupload')
3
4    const app = express()
5
6    app.use(upload())
7
8    app.get('/', (req,res)=>{
9        res.sendFile(__dirname +'/index.html' )
10   })
11
12   app.post('/', (req,res)=>{
13       if(req.files){
14           console.log(req.files);
15           var file = req.files.file
16           var filename = file.name
17           console.log(filename)
18
19       }
20   });
21
22   app.listen(5000)
```

WEB TECHNOLOGIES
FILE UPLOAD

Express JS

Index.html: as simple file upload script shown as below:

```
C:\Users\Aisha Begum\NODE\express_course\index.html - Notepad++

File  Edit  Search  View  Encoding  Language  Settings  Tools  Macro  Run  Plugins  Window  ?

package.json    myapp.js    issues.js    index.js    index.html    new 1

1  <h1>
2   File Upload in Node Js using express Library
3  </h1>
4  <form method="POST" action="/" enctype="multipart/form-data">
5  <input type ="file" name="file"/>
6  <input type ="submit" value="Upload"/>
7  </form>
```
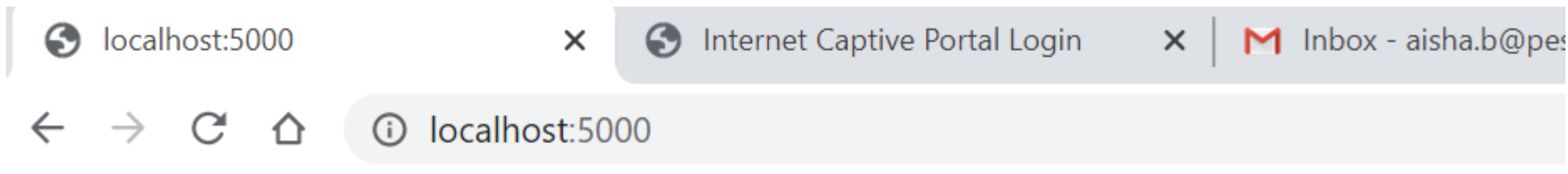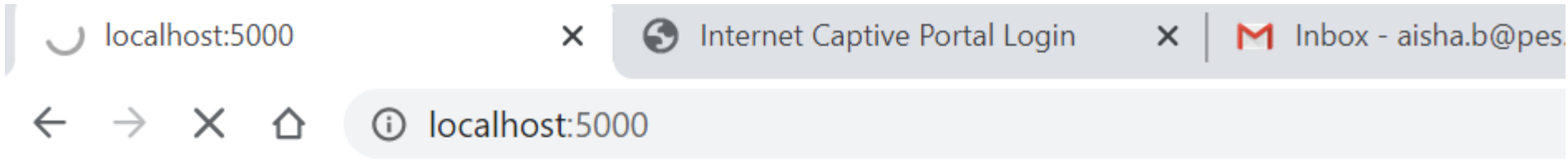
Express JS

In browser when we type the url: http://localhost:5000

Express JS

In browser when we type the url: http://localhost:5000 and choose file.

Here I have chosen images.jpg

Express JS

Upload a file then you can see the details in the command prompt about the file

```
C:\Users\Aisha Begum\NODE\express_course>node index
{
  file: {
    name: 'images.jpg',
    data: <Buffer ff d8 ff e0 00 10 4a 46 49 46 00 01 01 00 00 01 00 01 00 00 ff db 00 84 00 09 06 07 13 13 12 15 13 13 13 15 16 15 17 17 1d 1a 18 18 18 18 1b 1d 1d 18 ...
10523 more bytes>,
    size: 10573,
    encoding: '7bit',
    tempFilePath: '',
    truncated: false,
    mimetype: 'image/jpeg',
    md5: '0627265b42b7f89b5db7a4cb9925bd08',
    mv: [Function: mv]
  }
}
images.jpg
```

Express JS

Create a file for upload in the root folder.

The uploaded files will be saved in the created folder

Express JS

Using file.mv method we will be storing the file in the created folder

```
ckage.json   myapp.js   issues.js   index.js   index.html   new 1

const express = require('express')
const upload= require('express-fileupload')

const app = express()

app.use(upload())

app.get('/', (req,res)=>{
    res.sendFile(__dirname +'/index.html' )
})

app.post('/', (req,res)=>{
    if(req.files){
        console.log(req.files);
        var file = req.files.file
        var filename = file.name
        console.log(filename)
        file.mv('./uploads/' + filename,function(err){
            if(err){
                res.send(err)
            }else {
                res.send("File Uploaded")
            }
        })
    }
});

app.listen(5000)
```
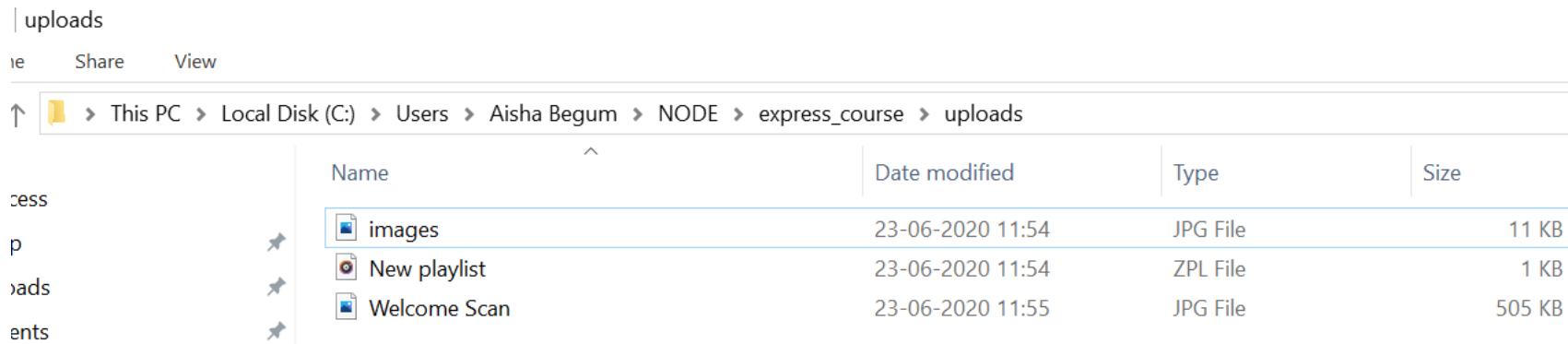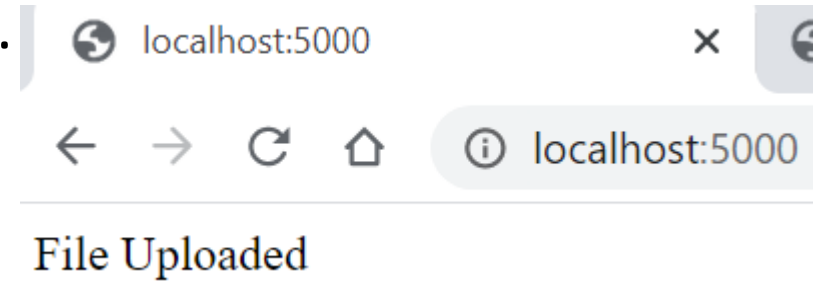
WEB TECHNOLOGIES
FILE UPLOAD

Express JS

In the browser you can see that file has been uploaded and when u check the uploads folder you can see that file has been uploaded. The file upload can be on any type

localhost:5000

File Uploaded

uploads

Share   View

This PC > Local Disk (C:) > Users > Aisha Begum > NODE > express_course > uploads

Name	Date modified	Type	Size
images	23-06-2020 11:54	JPG File	11 KB
New playlist	23-06-2020 11:54	ZPL File	1 KB
Welcome Scan	23-06-2020 11:55	JPG File	505 KB

EXPRESS ERROR HANDLING

Error handling in Express is done using middleware. But this middleware has special properties. The error handling middleware are defined in the same way as other middleware functions, except that error-handling functions **MUST have four arguments** instead of three – **err, req, res, next**.

EG - to send a response on any error, we can use –

```
app.use(function(err, req, res, next) {
    console.error(err.stack);
    res.status(500).send('Something broke!');
});
```

The error handling middleware allows us to separate our error logic and send responses accordingly. The next() method we discussed in middleware takes us to next middleware/route handler.

For error handling, we have the next(err) function. A call to this function skips all middleware and matches us to the next error handler for that route.

WEB TECHNOLOGIES
EXPRESS ERROR HANDLING

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

app.get('/', function(req, res){
   //Create an error and pass it to the next function
   var err = new Error("Something went wrong");
   next(err);
});


/*  other route handlers and middleware here */

//An error handling middleware
app.use(function(err, req, res, next) {
   res.status(500);
   res.send("Oops, something went wrong.")
});


app.listen(3000);
```

This error handling middleware can be strategically placed after routes or contain conditions to detect error types and respond to the clients accordingly.

THANK YOU

Aisha Begam

Department of Computer Science Engineering

aisha.b@pes.edu

+91 9741907626