

---

## **Express JS**

### **Introduction to Web services:**

The term “web service” is a general term that means any application programming interface (API) that’s accessible over HTTP.

### **What Is a Web Service?**

A Web service is an application that provides a **Web API**. A Web API is an API that lets the applications communicate using XML and the Web. The basic concept is: Web services use the Web to perform application-to-application integration.

### **Why Web Services?**

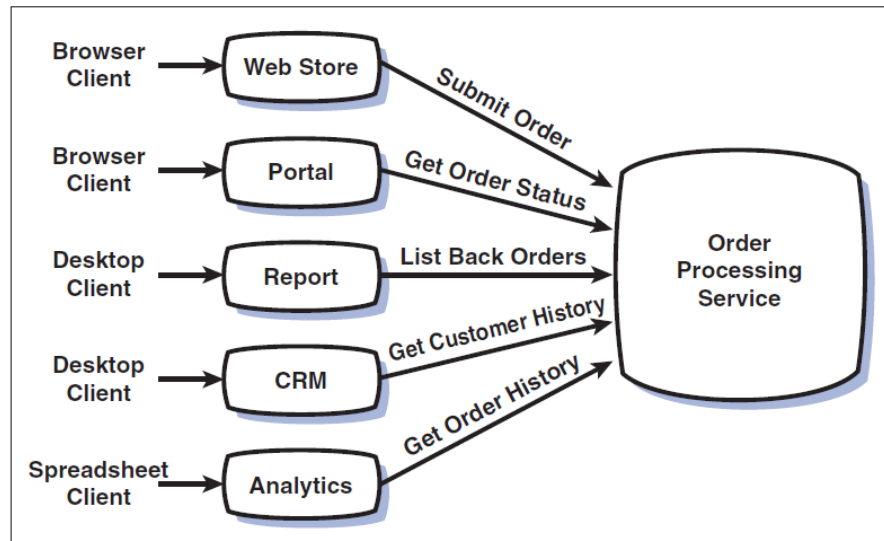
- Web services help you integrate applications
- Web services represent a new form of middleware based on XML and the Web.
- Web services support heterogeneous interoperability
- Web services are platform and language independent. You can develop a Web service using any language, and you can deploy it on any platform, from the tiniest device to the largest supercomputer.
- Web services simplify the process of making applications talk to each other. Simplification results in lower development cost, faster time to market, easier maintenance, and reduced total cost of ownership
- A Web service interface provides a layer of abstraction between the client and server. A change in one doesn’t necessarily force a change in the other. The abstract interface also makes it easier to reuse a service in another application. Loose coupling reduces the cost of maintenance and increases reusability.

### **Defining “Web” and “Service”**

- *A Web service is a service that lives on the Web*
- The Web is an immensely scalable information space filled with interconnected resources.
- The architecture for the Web has been developed and standardized by the World Wide Web Consortium (**W3C**).
- A **Web resource** is any type of named information object such as a word processing document, a digital picture, a Web page, an e-mail account, or an application—that’s accessible through the Web.

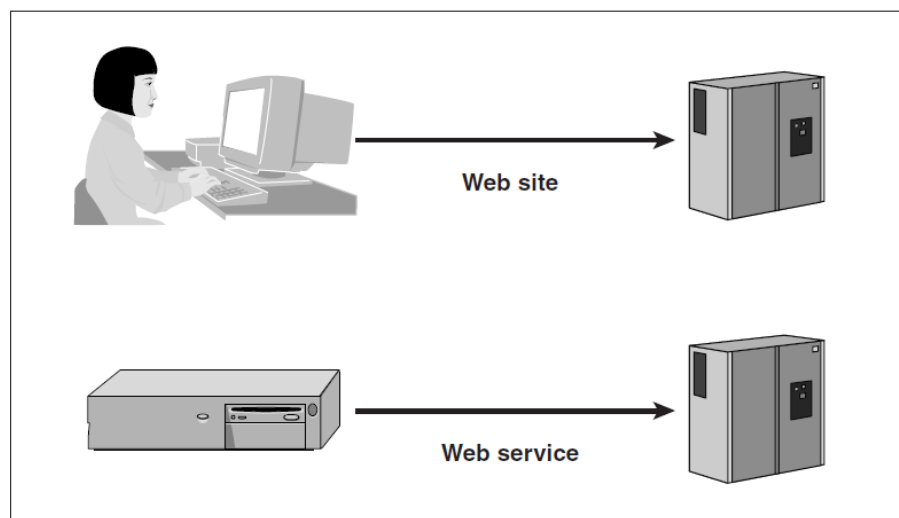
- All resources on the Web are connected via the Internet, and you access Web resources using standard Internet protocols.
- Any network-enabled application or device can access any resource in the Web.
- The Web solves one of your integration challenges: The Web is pervasive and provides universal connectivity.
- A **service** is an application that exposes its functionality through an application programming interface (API). In other words, a service is a resource that is designed to be consumed by software rather than by humans.
- The term “service” refers to something known as the **service-oriented architecture (SOA)**.
- A service exposes its functionality through an interface, and that interface hides the inner workings of the application. A client application doesn’t need to understand how the service actually performs its work. All it needs to understand is how to use the interface.
- A Web service possesses the characteristics of both a Web resource and a service.
- A service is a piece of software that does work for other software.
- A service can perform system functions or business application functions.
- All client/server technologies rely on this basic concept of a service. A service is the business or system application that plays the part of the server in a client/server relationship.
- An application server manages and coordinates the utilization of all resources available in a shared, multiprocessing environment, enabling optimized performance, scalability, reliability, and availability.

A service  
can be shared by  
many different  
applications.



- For example, as shown in Figure it's much simpler and easier to manage and maintain your order processing system if you have only one application service that actually processes orders. This one service can support all the different ways that you offer to place orders, inquire about order status, and generate reports about orders.
- Figure below summarizes the differences between a Web *site* and a Web *service*. A Web *site* represents a group of Web resources that are designed to be accessed by humans, and a Web *service* represents a group of Web resources that are designed to be accessed by applications.

**Figure 2-2:** A Web site is designed to be accessed by humans. A Web service is designed to be accessed by applications.



## Defining Characteristics of Web Services

---

A Web service exhibits the following defining characteristics:

- ❑ A Web service is a Web resource. You access a Web service using platform-independent and language-neutral Web protocols, such as HTTP. These Web protocols ensure easy integration of heterogeneous environments.
- ❑ A Web service provides an interface—a Web API—that can be called from another program. This application-to-application programming interface can be invoked from any type of application. The Web API provides access to the application logic that implements the service.
- ❑ A Web service is typically registered and can be located through a Web service **registry**. A registry enables service consumers to find services that match their needs. These service consumers may be humans or other applications.
- ❑ Web services support loosely coupled connections between systems. Web services communicate by passing XML messages to each other via a Web API, which adds a layer of abstraction to the environment that makes the connections flexible and adaptable.

### **REST API's:**

REST (short for representational state transfer) is an architectural pattern for application programming interfaces (APIs).

### **Resource Based**

- The APIs are resource based (as opposed to action based). APIs are formed by a combination of resources and actions.
- Resources are accessed based on a Uniform Resource Identifier (URI), also known as an *endpoint*. Resources are nouns (not verbs).
- Resources can also form a hierarchy. For example, the collection of orders of a customer is identified by `/customers/1234/orders`, and an order of that customer is identified by `/customers/1234/orders/43`.

### **HTTP Methods as Actions**

- To access and manipulate the resources, you use HTTP methods. While resources were nouns, the HTTP methods are verbs that operate on them. They map to CRUD (Create, Read, Update, Delete) operations on the resource. Table 5-1 shows commonly used mapping of CRUD operations to HTTP methods and resources.

**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

- Two important concepts about the HTTP methods are *safety* and *idempotency* of the methods.
- A safe method is one whose results can be cached. Thus, GET, HEAD, and OPTIONS are safe methods; you can call them any number of times and get the same results.
- An idempotent method is one that has the same effect when called multiple times. Note that it's not the same result; instead it's the effect on the resource.
- A safe method is always idempotent, but not the other way round.
- Only PUT and DELETE are idempotent, whereas POST and PATCH are not.
- If you use PUT multiple times, you continue to replace the same resource with the *same* new contents, thus the outcome is the same for each attempt. DELETE, if seen as "let the resource not exist,".
- PATCH and PUT are different, even though both can be used to update a resource.
- PATCH is used to modify a resource by adding to an array in the resource. PUT is used to completely replace the resource.

## JSON

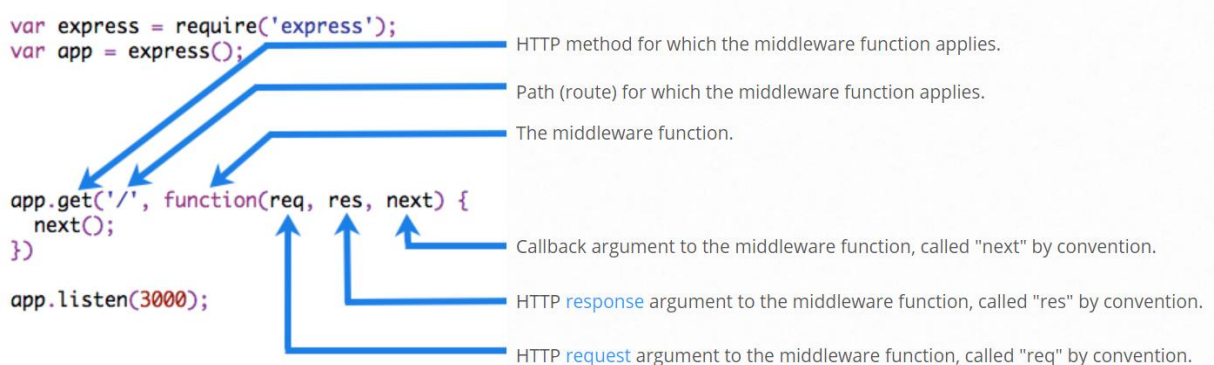
- JSON is used as the preferred encoding for the data, both in request and response bodies.
- Some REST specifications allow the caller to specify the format (JSON, XML, CSV, etc.),

## References:

1. TextBook: Vasan Subramanian - Pro MERN Stack\_ Full Stack Web App Development with Mongo, Express, React, and Node-Apress (2017).  
Page No 69-71.
2. Web Service basics.pdf

## Express Middleware

- Express middleware, which are custom pieces of code that can be inserted in any request/response processing path to achieve common functionality such as logging, authentication, etc.
- A middleware is a function that takes in an HTTP request and response object, plus the next middleware function in the chain.
- The function can look at and modify the request and response objects, respond to requests, or decide to continue with middleware chain by calling the next function.
- Middleware functions can perform the following tasks:
  - 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 function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function. Otherwise, the request will be left hanging.
- The following figure shows the elements of a middleware function call:



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

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

- This instantiates the application and then mounts a middleware.

- The middleware generator takes the parameter `static` to indicate that this is the directory where all the static files reside.
- The **`express.static`** generated middleware function is also smart enough to translate a request to `"/` (the root of the website) and respond by looking for any file in that directory.

### Example

Here is an example of a simple “Hello World” Express application middleware functions to the application: one called `myLogger` that prints a simple log message

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

### Middleware function `myLogger`

A simple example of a middleware function called “`myLogger`”. This function just prints “LOGGED” when a request to the app passes through it. The middleware function is assigned to a variable named `myLogger`.

```
var myLogger = function (req, res, next) {
  console.log('LOGGED')
  next()
}
```

To load the middleware function, call `app.use()`, specifying the middleware function.

For example, the following code loads the `myLogger` middleware function before the route to the root path (`/`).

```
var express = require('express')
var app = express()
var myLogger = function (req, res, next) {
  console.log('LOGGED')
  next()
}
```



---

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

Every time the app receives a request, it prints the message “LOGGED” to the terminal.

The order of middleware loading is important: middleware functions that are loaded first are also executed first.

If myLogger is loaded after the route to the root path, the request never reaches it and the app doesn’t print “LOGGED”, because the route handler of the root path terminates the request-response cycle.

The middleware function myLogger simply prints a message, then passes on the request to the next middleware function in the stack by calling the next() function.

#### References:

1. TextBook: Vasan Subramanian - Pro MERN Stack\_ Full Stack Web App Development with Mongo, Express, React, and Node-Apress (2017). Page No 75-76.
2. <https://expressjs.com/en/guide/writing-middleware.html>
3. <https://expressjs.com/en/guide/using-middleware.html>

---

## Routing

- Router, takes a client request, matches it against any *routes* that are present, and executes the *handler* function that is associated with that route.
- The handler function is expected to generate the appropriate response.
- A route specification consists of an HTTP method (GET, POST, etc.), a path specification that matches the request URI, and the route handler.
- 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.

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

- To use Express, you first need to create an application using its root-level exported function. You do that using `const app = express ();`. You can now set up routes using this app.
- To set up a route, you use a function to indicate which HTTP method.
- For example, to handle the GET HTTP method, you use the app's `get ()` function. To this function, you pass the pattern to match and a function to deal with the request if it does match.

## Request Matching

- When a request is received, the first thing that happens is request matching.
- The request's method is matched with the route method (the `get` function was called on app, indicating it should match only GET HTTP methods), and
- The request URL with the path spec (`/hello` in the above example). If a request matches this spec, then the handler is called.
- The path specification can also take regular expression-like patterns (like `/*`. do') or regular expressions themselves

## Route Parameters

- Route parameters are named segments in the path specification that match a part of the URL.
- If a match occurs, the value in that part of the URL is supplied as a variable in the request object.
- This is used in the following form: `app.get('/customers/:customerId', ...)`.

### Route Lookup

- Multiple routes can be set up to match different URLs and patterns.
- The router does not try to find a best match; instead, it tries to match all routes in the order in which they are installed.
- The first match is used. So, if two routes are possible matches to a request, it will use the first defined one.
- If you want to match everything that goes under `/api/`, that is, a pattern like `/api/*`, you should add this route only *after* all the specific routes that handle paths such as `/api/issues`.

### 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

- **req.params:** To access a parameter value using `req.params`
- **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`).
- **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.
- **req.path:** This contains the path part of the URL, that is, everything up to any? That starts the query string. Usually, the path is part of the route if the route is not a pattern, but if the path is a pattern that can match

---

different URLs; you can use this property to get the actual path that was received in the request.

- **req.url, req.originalURL:** These properties contain the complete URL, including the query string.
- **req.body:** This contains the body of the request, valid for POST, PUT, and PATCH requests. Note that the body is not available (req.body will be undefined) unless a middleware is installed to read and optionally interpret or parse the body.

### Response Objects

The response object is used to construct and send a response to a request. If no response is sent, the client is left waiting.

**res.send (body):** responded with a string. This method can also accept a buffer (in which case the content type is set as application/ octet-stream as opposed to text/html in case of a string). If the body is an object or an array, it is automatically converted to a JSON string with an appropriate content type.

**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.json (object):** This is the same as res.send (), except that this method forces conversion of the parameter passed into a JSON, whereas res.send () may treat some parameters like null differently. It also makes the code readable and explicit, stating that you are indeed sending out a JSON.

**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.

### References:

1. TextBook: Vasan Subramanian - Pro MERN Stack\_ Full Stack Web App Development with Mongo, Express, React, and Node-Apress (2017).  
Page No 72-75.

---

## **Error Handling**

- **Error Handling** refers to how express catches and processes errors that occur both synchronously and asynchronously.
  - Express comes with a default error handler so you don't need to write your own to get started.
  - The success or failure of any REST API call is typically reflected in the HTTP status code.
  - The error message (a description of what went wrong) itself can be returned in the response body, again as a JSON string.
  - We will return an object with a single property called message that holds a readable as the description.
  - At the server, sending an error is simple; all you need to do is set the status using `res.status()` and send the error message as the response.
- 
- **Main HTTP Status Codes**
    - 200- OK; Standard response for successful HTTP requests
    - 201- Created; Request has been fulfilled. New resource created
    - 204- No Content; Request processed. No content returned
    - 301- Moved Permanently; This and all future requests directed to the given URI
    - 304- Not Modified; Resource has not been modified since last requested
    - 400- Bad Request; Request cannot be fulfilled due to bad syntax
    - 401- Unauthorized; Authentication is possible, but has failed
    - 403- Forbidden; Server refuses to respond to request
    - 404- Not Found; Requested resource could not be found
    - 500- Internal Server Error; Generic error message when server fails
    - 501- Not Implemented; Server does not recognize method or lacks ability to fulfill
    - 503- Service Unavailable; Server is currently unavailable
  - Express middleware is broken up into different types based on the number of arguments your middleware function takes.
  - A middleware function that takes 4 arguments is classified as "error handling middleware", and will only get called if an error occurs.  

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

```
app.get('*', function(req, res, next) {  
  // This middleware throws an error, so Express will go straight to  
  the next error handler  
  throw new Error('woops');  
});
```

```
app.get('*', function(req, res, next) {  
  // This middleware is not an error handler (only 3 arguments),  
  // Express will skip it because there was an error in the previous  
  middleware  
  console.log('this will not print');  
});
```

```
app.use(function(error, req, res, next) {  
  // Any request to this server will get here, and will send an HTTP  
  response with the error message 'woops'  
  res.json({ message: error.message });  
});  
app.listen(3000);
```

- The only way to report errors to Express for use with error handlers is using the third argument to conventional middleware, the `next()` function.
- Route handlers (like `app.get('/User', function (req, res){})`) can also take a `next()` function as an argument
- Express middleware executes **in order**. You should define error handlers **last**, after all other middleware. Otherwise, your error handler won't get called:

```
const app = require('express')();  
  
app.use(function(error, req, res, next) {  
  // Will not get called. You'll get Express' default error  
  // handler, which returns error.toString() in the error body  
  console.log('will not print');  
  res.json({ message: error.message });  
});
```

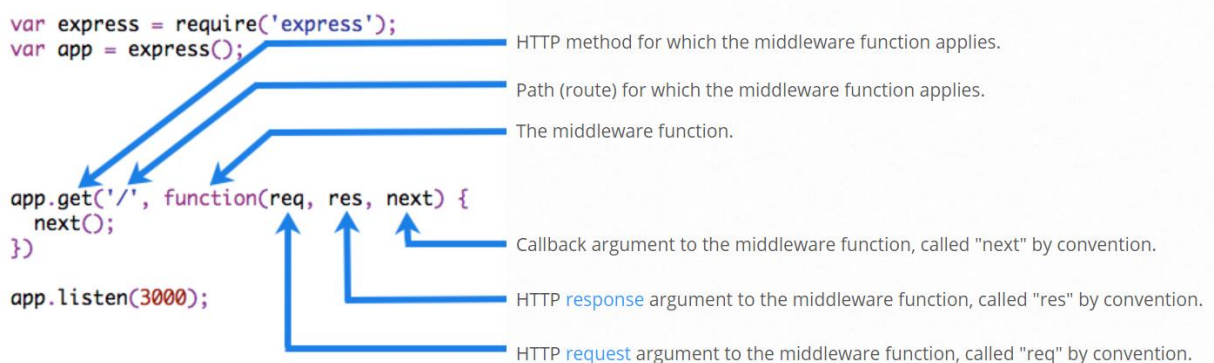
```
app.get('*', function(req, res, next) {  
  setImmediate(() => { next(new Error('woops')); });  
});  
  
app.listen(3000);
```

#### References:

1. TextBook: Vasan Subramanian - Pro MERN Stack\_ Full Stack Web App Development with Mongo, Express, React, and Node-Apress (2017).  
Page No:85-88
2. <https://expressjs.com/en/guide/using-middleware.html>
- 3.

## Express Middleware

- Express middleware, which are custom pieces of code that can be inserted in any request/response processing path to achieve common functionality such as logging, authentication, etc.
- A middleware is a function that takes in an HTTP request and response object, plus the next middleware function in the chain.
- The function can look at and modify the request and response objects, respond to requests, or decide to continue with middleware chain by calling the next function.
- Middleware functions can perform the following tasks:
  - 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 function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function. Otherwise, the request will be left hanging.
- The following figure shows the elements of a middleware function call:



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

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

- This instantiates the application and then mounts a middleware.



- The middleware generator takes the parameter `static` to indicate that this is the directory where all the static files reside.
- The **`express.static`** generated middleware function is also smart enough to translate a request to `"/` (the root of the website) and respond by looking for any file in that directory.

### Example

Here is an example of a simple “Hello World” Express application middleware functions to the application: one called `myLogger` that prints a simple log message

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

### Middleware function `myLogger`

A simple example of a middleware function called “`myLogger`”. This function just prints “LOGGED” when a request to the app passes through it. The middleware function is assigned to a variable named `myLogger`.

```
var myLogger = function (req, res, next) {
  console.log('LOGGED')
  next()
}
```

To load the middleware function, call `app.use()`, specifying the middleware function.

For example, the following code loads the `myLogger` middleware function before the route to the root path (`/`).

```
var express = require('express')
var app = express()
var myLogger = function (req, res, next) {
  console.log('LOGGED')
  next()
}
```

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

Every time the app receives a request, it prints the message “LOGGED” to the terminal.

The order of middleware loading is important: middleware functions that are loaded first are also executed first.

If myLogger is loaded after the route to the root path, the request never reaches it and the app doesn’t print “LOGGED”, because the route handler of the root path terminates the request-response cycle.

The middleware function myLogger simply prints a message, then passes on the request to the next middleware function in the stack by calling the next() function.

#### References:

1. TextBook: Vasan Subramanian - Pro MERN Stack\_ Full Stack Web App Development with Mongo, Express, React, and Node-Apress (2017). Page No 75-76.
2. <https://expressjs.com/en/guide/writing-middleware.html>
3. <https://expressjs.com/en/guide/using-middleware.html>

---

## Express JS-Form data

Forms are an integral part of the web.

To get started with forms, we will first install the *body-parser*(for parsing JSON and url-encoded data) and *multer*(for parsing multipart/form data) middleware.

To install the *body-parser* and *multer*, go to your terminal and use –

### **npm install body-parser multer**

After importing the body parser and multer, we will use the body-parser for parsing json and x-www-form-urlencoded header requests, while we will use multer for parsing multipart/form-data.

Replace your **index.js** file contents with the following code –

```
var express = require('express');
var bodyParser = require('body-parser');
var multer = require('multer');
var upload = multer();
var app = express();
//Read form saved as form.pug
app.get('/', function(req, res){
  res.render('form');
});
//Creating a view
app.set('view engine', 'pug');
app.set('views', './views');

// for parsing application/json
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```

---

```
//form-urlencoded for parsing multipart/form-data
app.use(upload.array());
app.use(express.static('public'));

app.post('/', function(req, res){
  console.log(req.body);
  res.send("Request is received!");
});
app.listen(3000);
```

Let us create an html form to test this out. Create a new view called form.pug with the following code –

html

head

title Form Tester

body

form(action = "/", method = "POST")

div

label(for = "say") Say:

input(name = "say" value = "Hi")

br

div

label(for = "to") To:

input(name = "to" value = "Express forms")

br

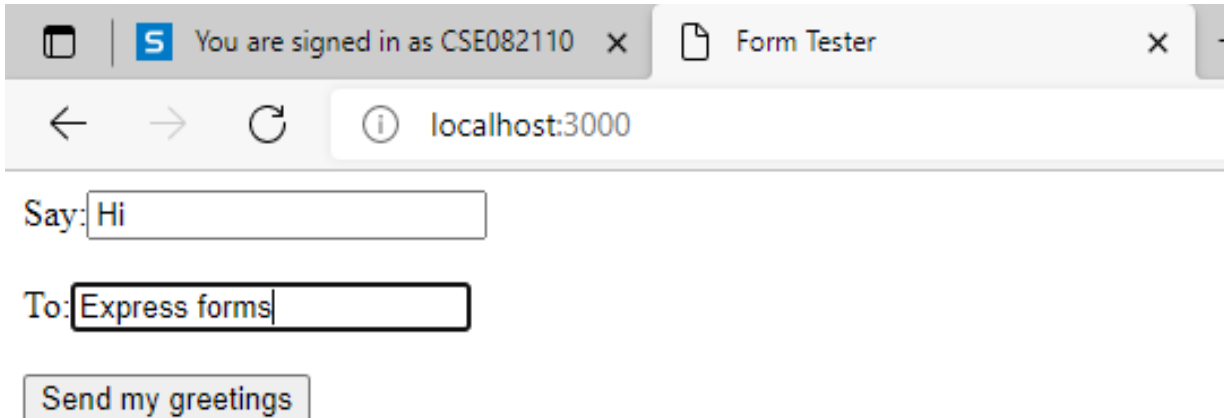
button(type = "submit") Send my greetings

---

Run your server using the following.

nodemon index.js

Now go to localhost:3000/ following is displayed in browser---



Browser tabs: You are signed in as CSE082110, Form Tester

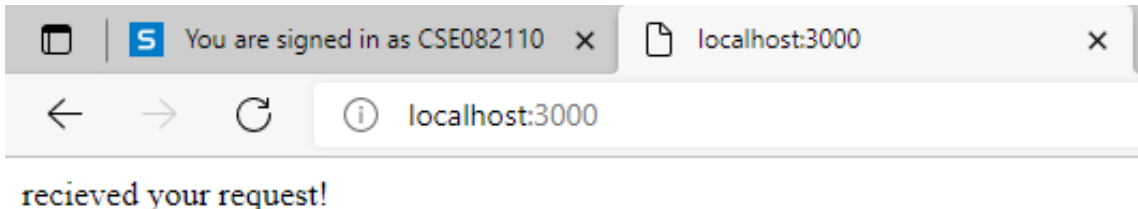
Address bar: localhost:3000

Form fields:

Say:

To:

Now, fill the form and submit the following response is generated.

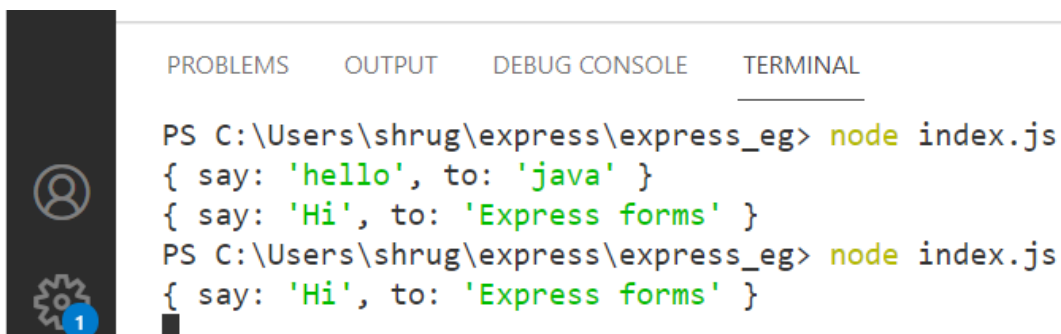


Browser tabs: You are signed in as CSE082110, localhost:3000

Address bar: localhost:3000

Response: recieved your request!

In console; it will show you the body of your request as a JavaScript object as in the following screenshot –



PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\shrug\express\express_eg> node index.js
{ say: 'hello', to: 'java' }
{ say: 'Hi', to: 'Express forms' }
PS C:\Users\shrug\express\express_eg> node index.js
{ say: 'Hi', to: 'Express forms' }
```

## Why pug?

Pug in node.js is a template engine that uses case sensitive syntax to generate html, in other words it returns a string of html rendered as per data specified in a pug file.

We can say that pug is the middleman who plays a role to convert the injected data and translate it into html syntax.

Pug is a template engine for Node and for the browser. It compiles to HTML and has a simplified syntax, which can make you more productive and your code more readable.

Pug makes it easy both to write reusable HTML, as well as to render data pulled from a database or API

---

---

## File Uploads in Express:

A large number of mobile apps and websites allow users to upload profile pictures and other files. Therefore, handling files upload is a common requirement while building a REST API with Node.js & Express.

How to handle single and multiple files uploads with Node.js and Express backend and save uploaded files on the server.

### **express-fileupload**

Simple express middleware for uploading files.

### **Install**

# With NPM

```
npm i express-fileupload
```

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

`req.files` is used to handle file uploads. 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.

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  
});
```

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.
- 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

For more information on express file uploads visit:

<https://attacomsian.com/blog/uploading-files-nodejs-express>

---