Develop a Reliable Backend with Node and Express

Getting Started with Express.js

Learning Objectives

By the end of this lesson, you will be able to:

- Analyze the working of express.js for enabling a comprehensive understanding of its underlying mechanisms
- Assess MVC structure and modules for gaining insights into how they streamline development and promote code maintainability
- Handle GET and POST data for illustrating proficiency in extracting and managing data from different HTTP request types
- Elaborate on express.js static files for demonstrating the practical implementation of handling static resources in web applications using express.js

A Day in the Life of a MERN Stack Developer

Mr. Robin is working as a MERN stack developer in an organization looking to establish and maintain consistency in a product's performance and functional attributes with great database management. The goal is to determine how to improve the performance of the backend and build RESTful APIs and web applications.

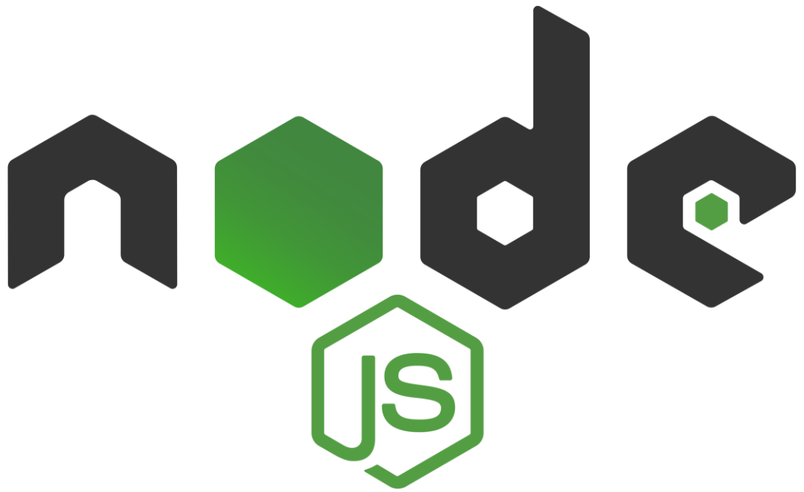
To achieve the above, along with some additional concepts, Robin will be learning a few concepts in this lesson that will help him create a backend web application framework.

These concepts collectively empower him to not only optimize backend performance and build RESTful APIs but also to create a resilient and adaptable backend framework that meets the organization's goals for consistency and scalability.

Introduction to Express.js

What Is Express.js?

Express.js is a lightweight framework that enhances Node.js web server capabilities by streamlining the existing APIs.



Middleware and routing help Node.js render dynamic HTTP objects easily.

What Is Express.js?

Express.js is a Node.js backend framework that is straightforward, quick, and like Sinatra.

- Offers powerful capabilities and tools for scalable backend
- Leverages the routing system and streamlined features to expand the framework

What Is Express.js?

The framework offers a set of tools for developing and delivering the following:

- Middleware
- Enterprise-ready online applications
- Routing
- HTTP requests and replies

Features of Express.js

Defines a routing table that is utilized to carry out various operations

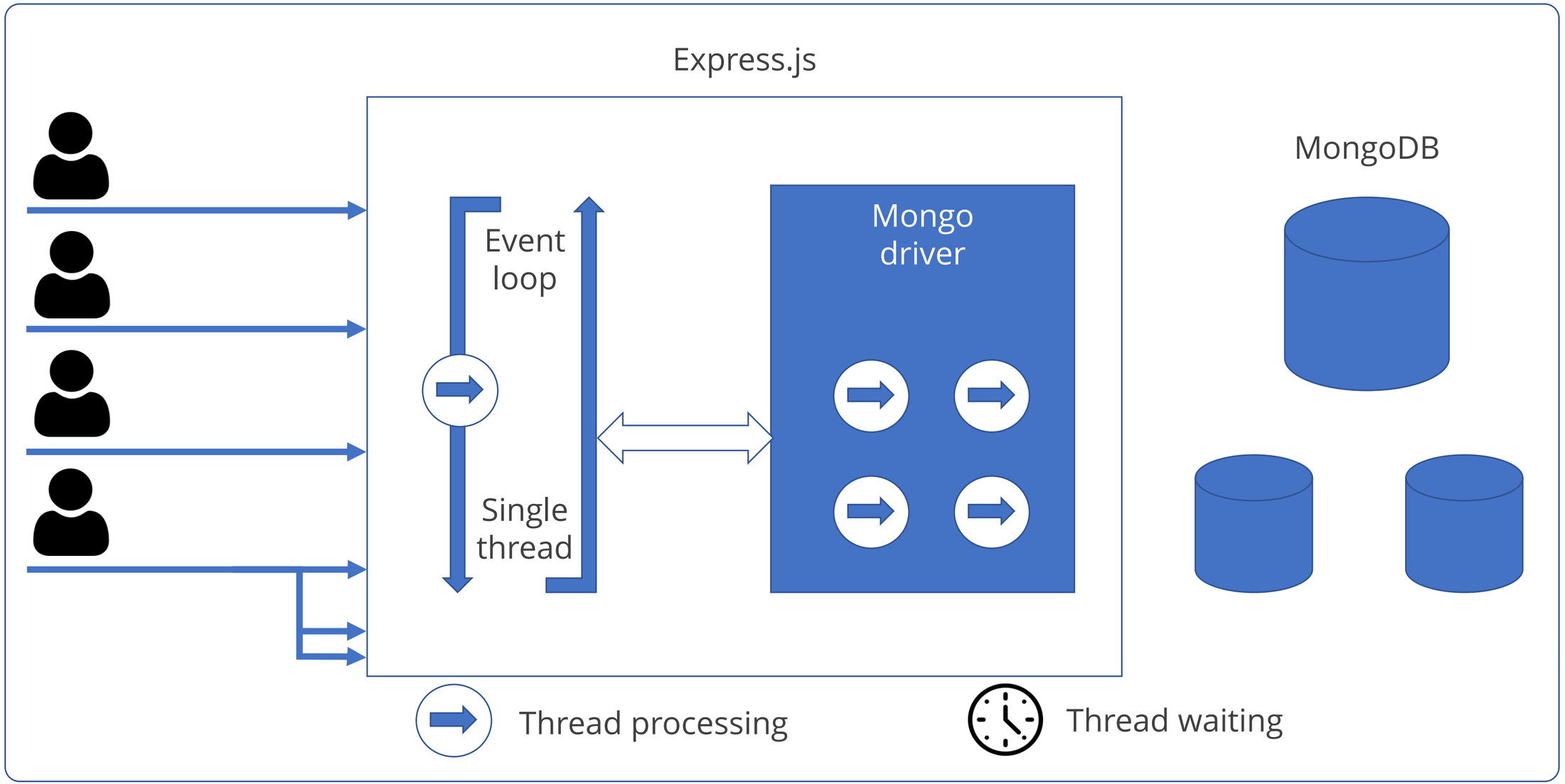
Allows the creation of single-page, multi-page, and hybrid web programs

Enables the dynamic rendering of HTML pages

Enables middleware configuration for HTTP request responses

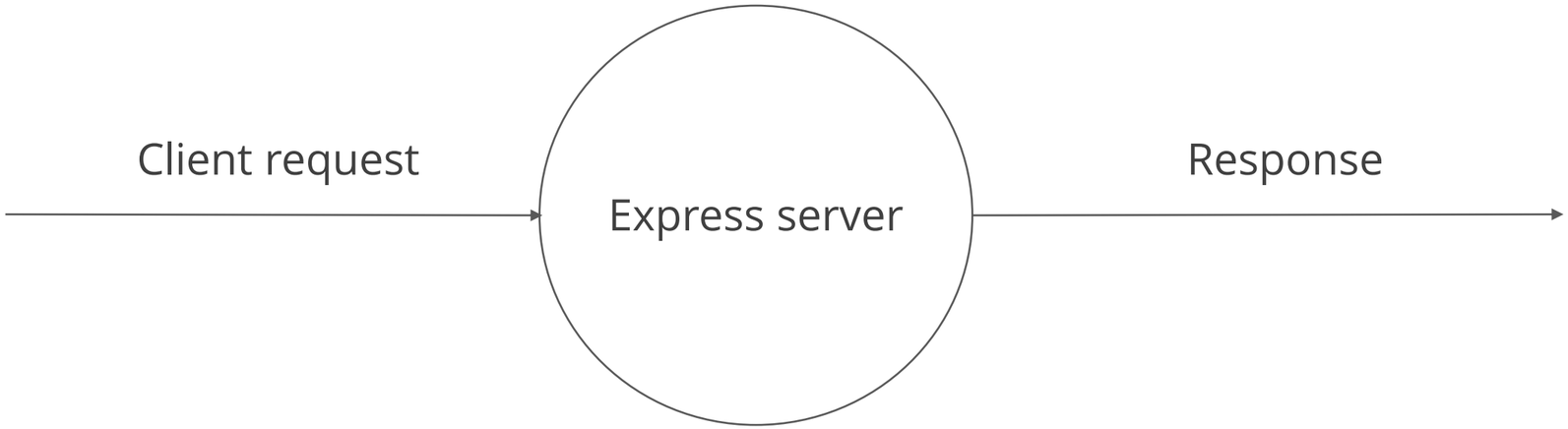
How Express.js Works

The following diagram depicts the working and architecture of Express.js:



How Express.js Works

Express.js includes many middleware features. The Express.js server takes the client request, processes the request, and sends back the response to the client.



How Express.js Works

Express.js accepts the following two parameters:

Path

Specifies the path when calling a middleware function

Callback

Calls the next middleware function in case of no termination

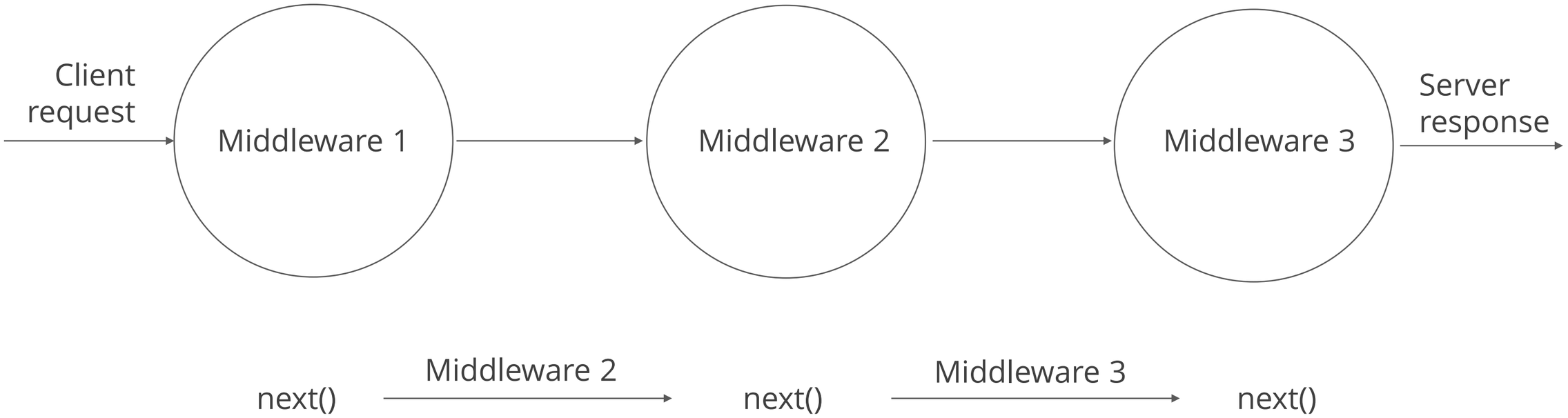
How Express.js Works

Syntax:

```
app.use(path,(req,res,next))
```

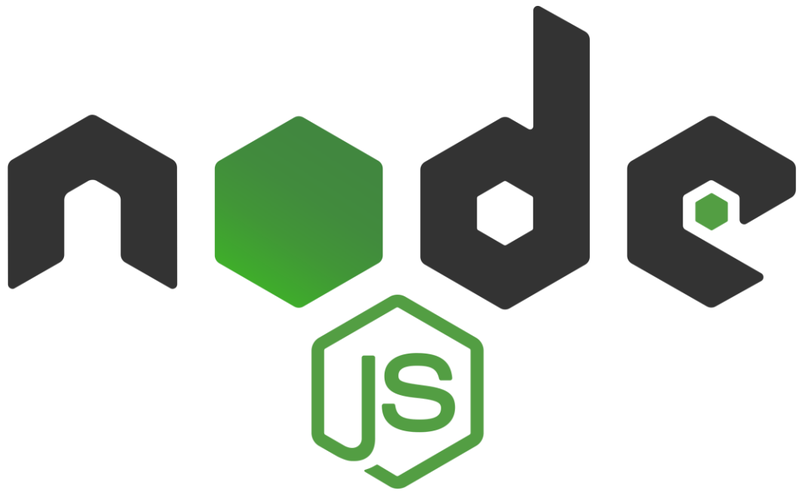
How Express.js Works

The following diagram depicts the multi-middleware cycle of Express.js:



CLI

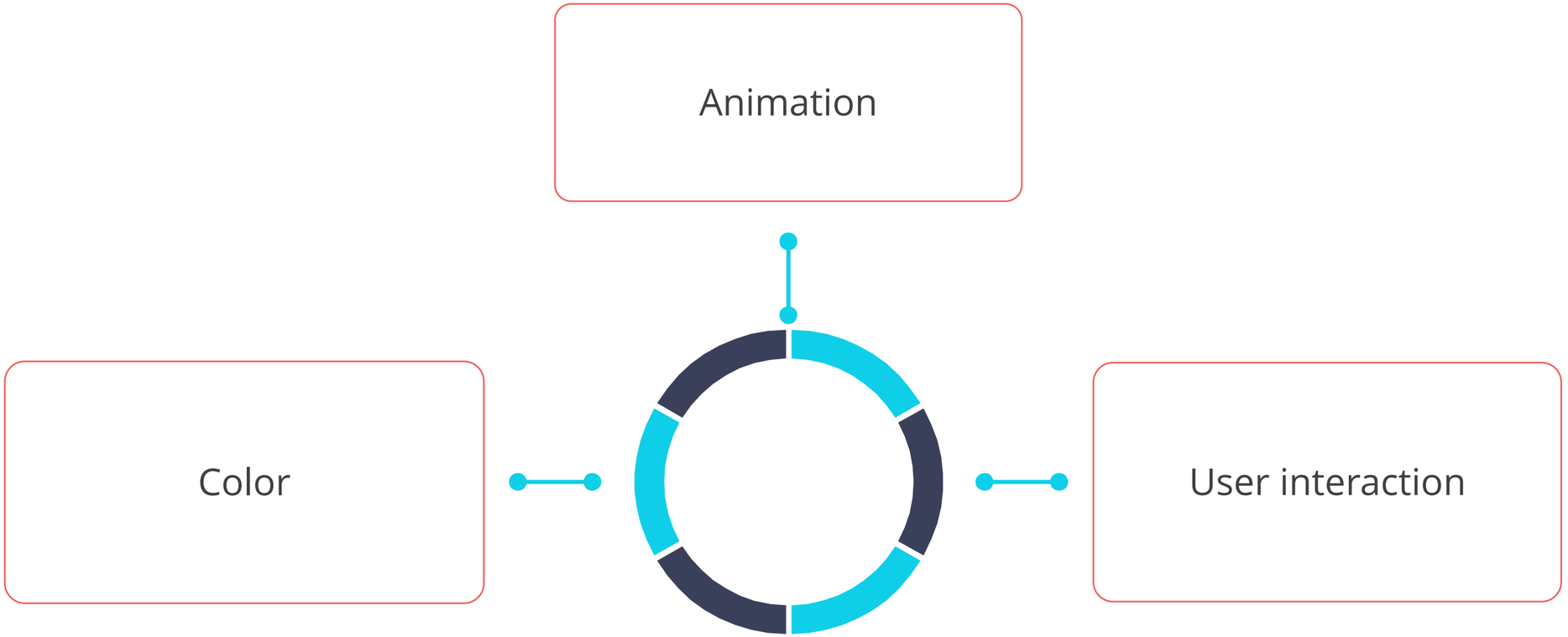
By using the Node.js ecosystem and the built-in command-line interfaces (CLIs), repetitive processes can be automated.



The **commander** package is a popular Node.js package used to create CLIs which simplifies the process of defining and parsing command-line arguments and options for Node.js applications.

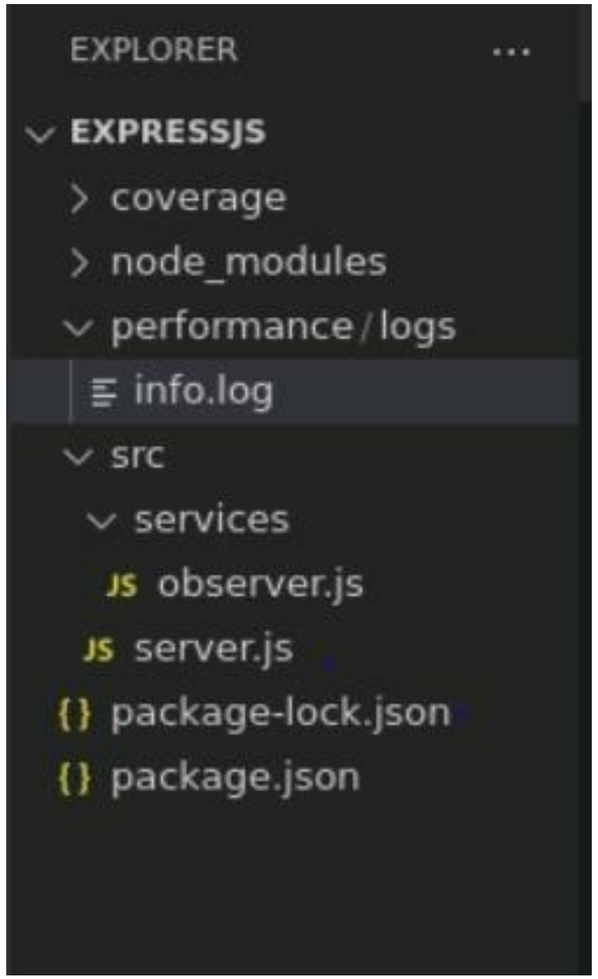
CLI

CLI is a text-based UI used to run programs, manage files, and communicate with machines.
It is possible to manage:



Watching for File Changes

In the logs folder, performance will append a line at the end of the info.log file.



Watching for File Changes

Use the following to track the changes in the file:

fs.watch

fs.watchFile

Watching for File Changes

The chokidar.watch() is used to take the path of the file to be watched. The following code depicts logic for chokidar.watch():

```
const chokidar = require('chokidar');

const watcher = chokidar.watch('path/to/folder', { persistent:
true });

watcher
  .on('add', path => log(`File ${path} has been added`))
  .on('change', path => log(`File ${path} has been changed`))
  .on('unlink', path => log(`File ${path} has been removed`));

watcher
  .on('addDir', path => log(`Directory ${path} has been
added`))
  .on('unlinkDir', path => log(`Directory ${path} has been
removed`))
  .on('error', error => log(`Watcher error: ${error}`))
  .on('ready', () => log('Initial scan complete. Ready for
changes'))
```

Watching for File Changes

The **readLastLines** function is used to read the last N lines of a file and watch the changes of the file:

```
const readLastLines = require('read-last-lines');

readLastLines.read('path/to/file', 34) // read last 34 lines
    .then((lines) => console.log(lines));
```

Watching for File Changes

Chokidar and read-last-lines are the two packages that must be installed.

```
npm install chokidar read-last-lines
```

Watching for File Changes

Create observer.js file:

```
const chokidar = require('chokidar');
const EventEmitter = require('events').EventEmitter;
const readLastLines = require('read-last-lines');

class Observer extends EventEmitter {
  constructor() {
    super();
  }

  watchFile(targetFile) {
    try {
      console.log(
        `[${new Date().toLocaleString()}] Watching
for file changes on: ${targetFile}`
      );
var watcher = chokidar.watch(targetFile, {
persistent: true });
```

```
 watcher.on('change', async filePath => {
        console.log(
          `[${new Date().toLocaleString()}]
${filePath} has been updated.`
        );
var updateContent = await
readLastLines.read(filePath, 1);

this.emit('file-updated', { message: updateContent
});
      });
    } catch (error) {
      console.log(error);
    }}}
```

Watching for File Changes

Creating server.js:

```
const Obserser = require('./services/observer');
var obserser = new Obserser();
const file = 'Performance/logs/info.log';
obserser.on('file-updated', log => {
  console.log(log.message);
});
obserser.watchFile(file);
```

Watching for File Changes

From the project root folder, issue the following command:

```
node src/server.js
```

Watching for File Changes

The console shows the following output:

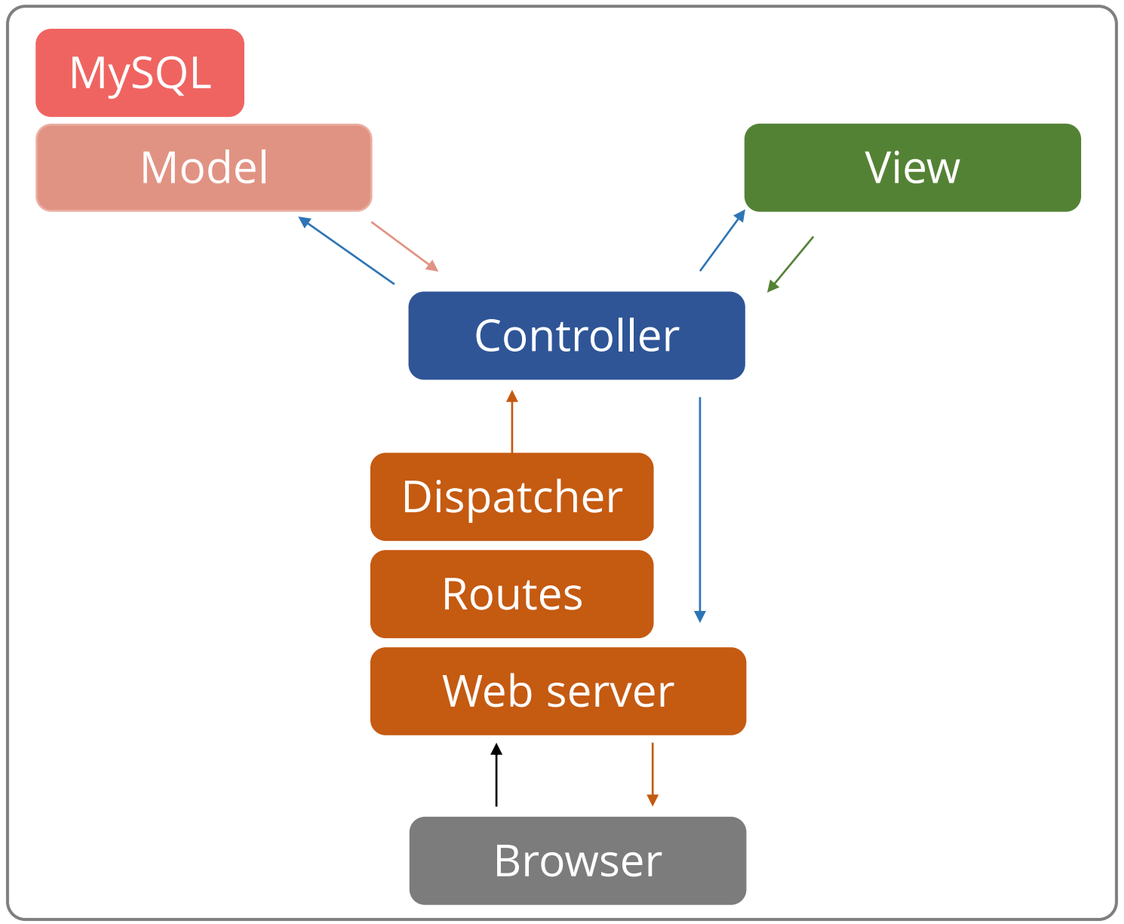
```
[[2/22/2023, 11:44:56PM] Watching for file changes on: Performance/logs/info.log
```

The console shows the following output once a new line is added to the info.log file:

```
[2/22/2023, 11:44:56PM] Performance/logs/info.log has been updated. New information here
```

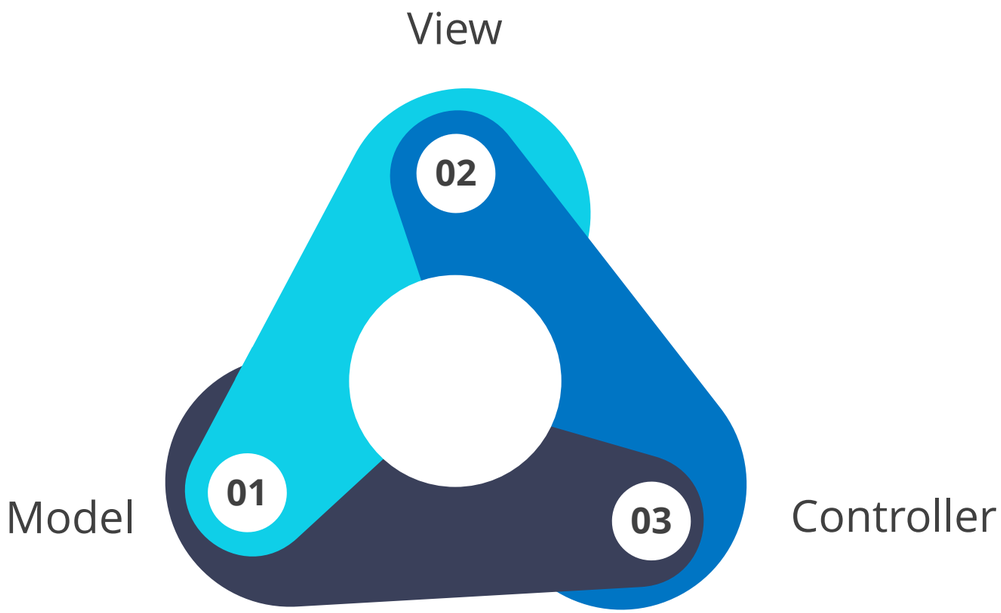
MVC Structure and Modules

The following diagram depicts the MVC infrastructure, which divides an application into three primary logical components: the model, the view, and the controller.



MVC Structure and Modules

MVC is used to create web servers to streamline development by dividing the web server into three components:



Modules

A module contains chunks of code that interact with external applications based on the shared functionality.

01 Consist of related files and folders

02 Have the ability to break down a complex code into manageable chunks

03 Can reuse the code

MVC Structure and Modules

The following are the three types of modules:

02

Local modules

01

Core modules

03

Third-party modules

Installation of Express.js

Problem Statement:

Duration: 20 min.

You have been assigned a task to install and create an Express app.

ASSISTED PRACTICE

Assisted Practice: Guidelines

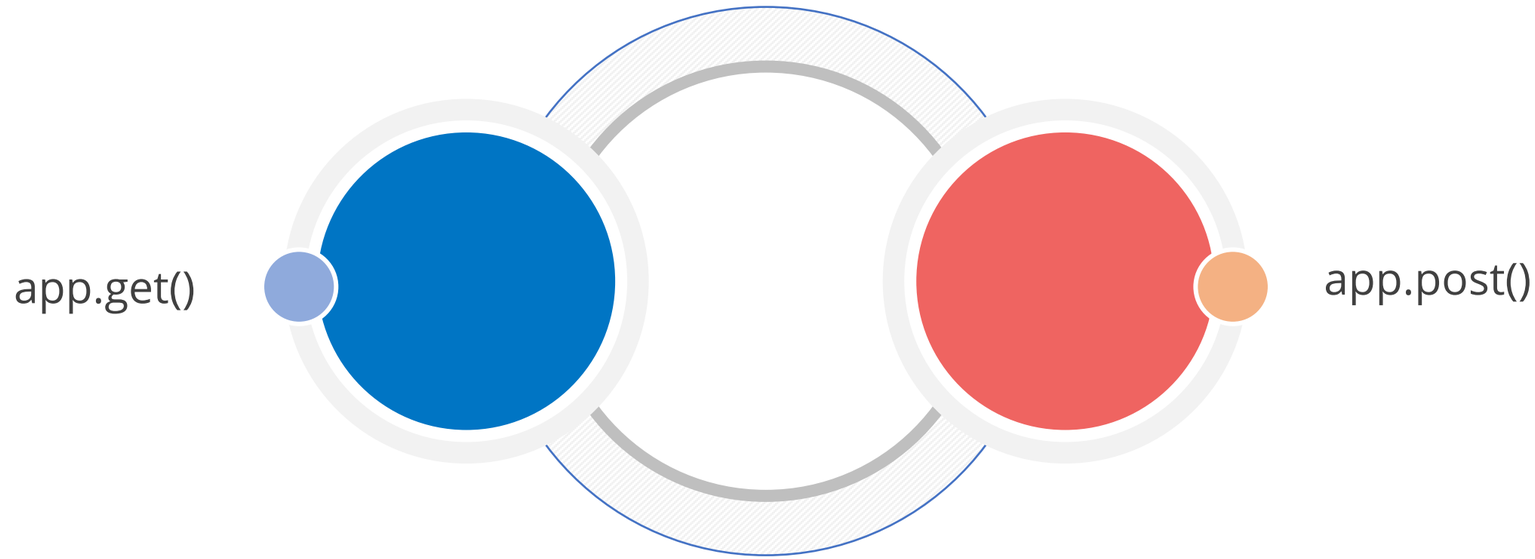
Steps to be followed:

1. Check whether Node.js is installed or not
2. Install Express.js
3. Create a Hello World Express app

Working with Express.js

Routing

Routing describes how URI endpoints in an application respond to client requests and how callback function works.



Routing

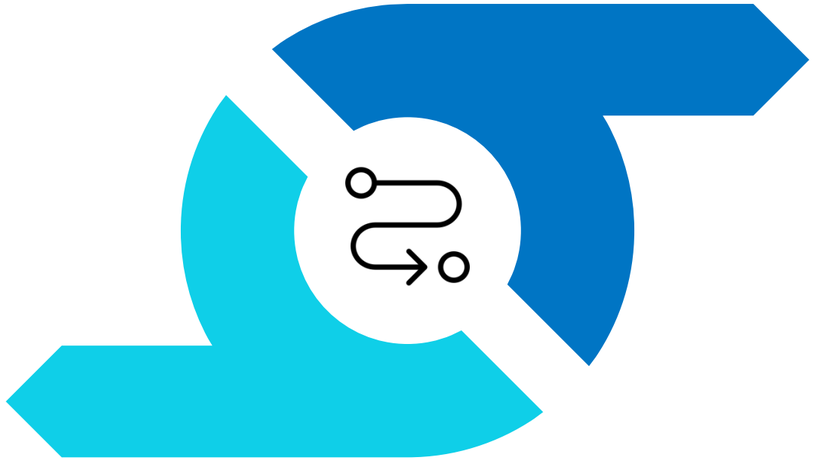
When the program receives a request to the route and HTTP method, it calls the callback function. This can be done in multiple ways.

Routing

There are multiple callback functions that are input to the routing method.

Routing methods consist of the following functions:

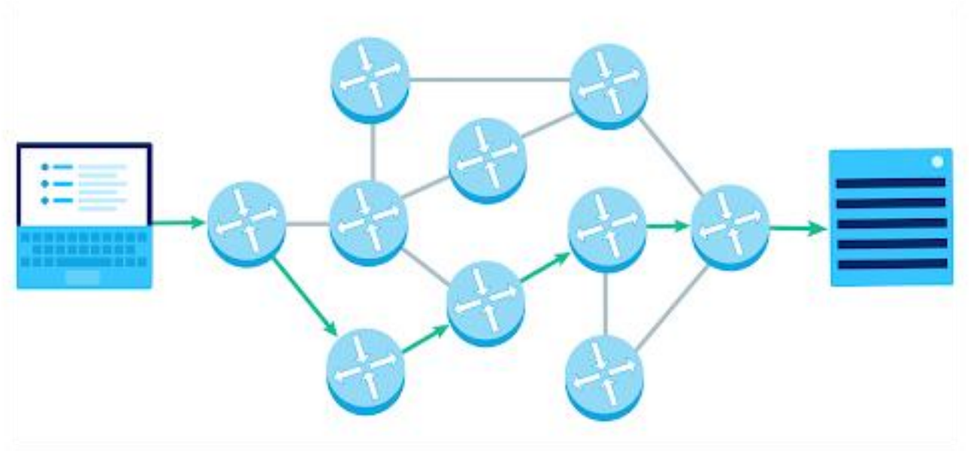
Passing an argument to the callback function



Calling next() inside the function body

Routing Techniques

When the program receives a request to the provided route (endpoint) and HTTP method, it calls the callback function specified by these routing methods (also known as **handler functions**).



The express class is given a routing method, which is derived from one of the HTTP methods.

Routing Techniques

Example of a route defined for the GET and POST methods to the app's root:

```
// GET method route
app.get('/', (req, res) => {
  res.send('GET request to the homepage')
})

// POST method route
app.post('/', (req, res) => {
  res.send('POST request to the homepage')
})
```

All HTTP request methods are supported by Express.js, including GET, POST, and others.

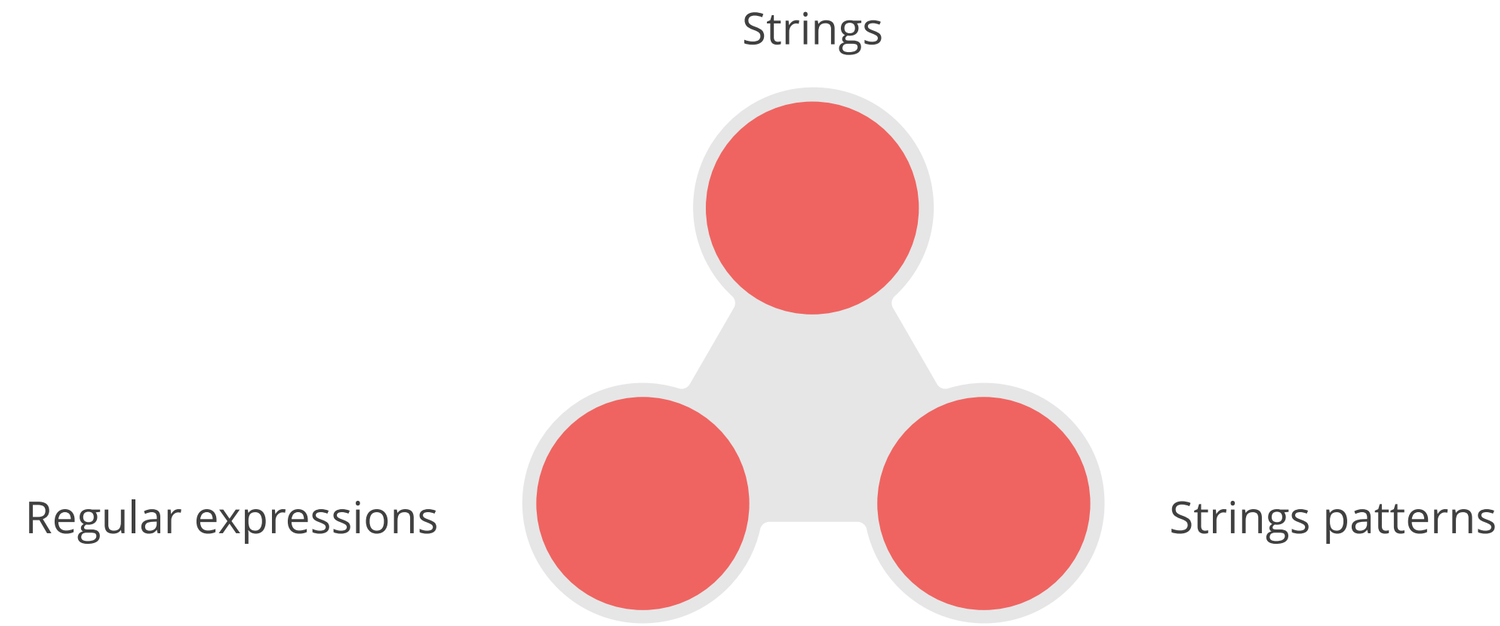
Routing Techniques

The custom routing method app.all() is used to load middleware routines at a path for all HTTP request methods.

```
app.all('/secret', (req, res, next) => {
  console.log('Accessing the secret section')
  next()
})
```

Route Pathways

The endpoints at which requests may be made are defined by route pathways in conjunction with a request method. Strings, string patterns, or regular expressions can all be used as route pathways.



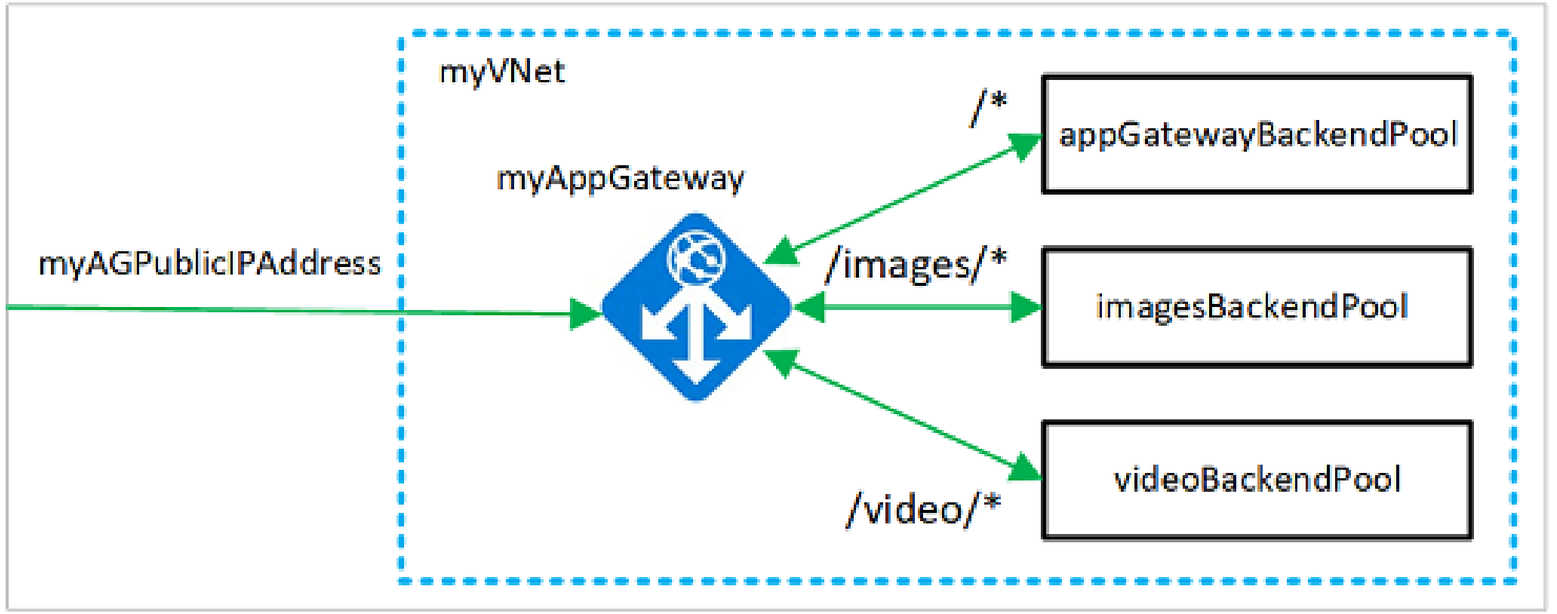
Route Pathways

The symbols ?, +, *, and () are subsets of their equivalents in regular expressions.

Use the dollar sign ($) in a path string and enclose it within quotation marks (‘ ’).

Route Specifics

The values supplied at their place in the URL are captured by route parameters, also known as URL segments.

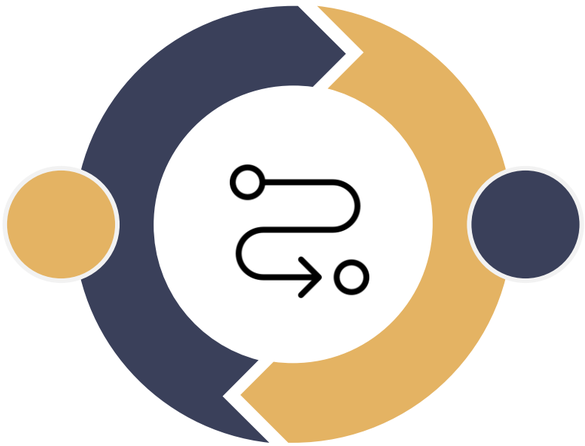


The names of the route parameters supplied in the path are used as the keys for the captured values in the req.params object.

Handlers of Routers

Several callback functions can be offered to act as middleware to process a request.

Establish preconditions on a route

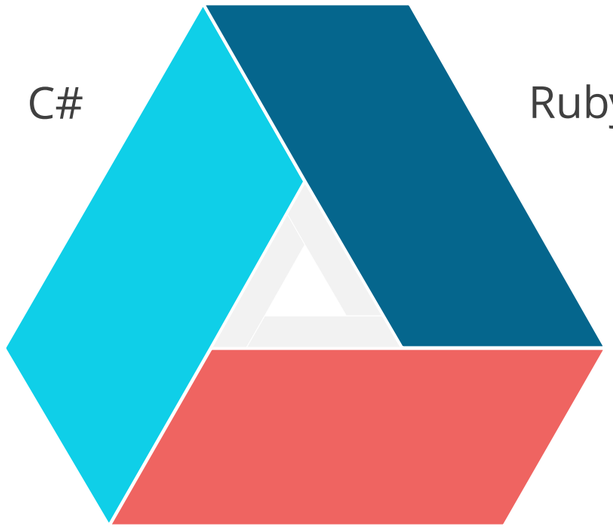


Transfer control to the following routes

These callbacks may use next('route') to skip the remaining route callbacks.

The Mode-View-Controller Pattern

An architectural or design pattern known as model-view-controller (MVC) divides an application into three basic logical components: model, view, and controller.



PHP framework

The Mode-View-Controller Pattern

MVC is an approach to designing software projects. One of the most popular and well accepted web development frameworks for building scalable and adaptable projects is VC.
The model-view-controller pattern:

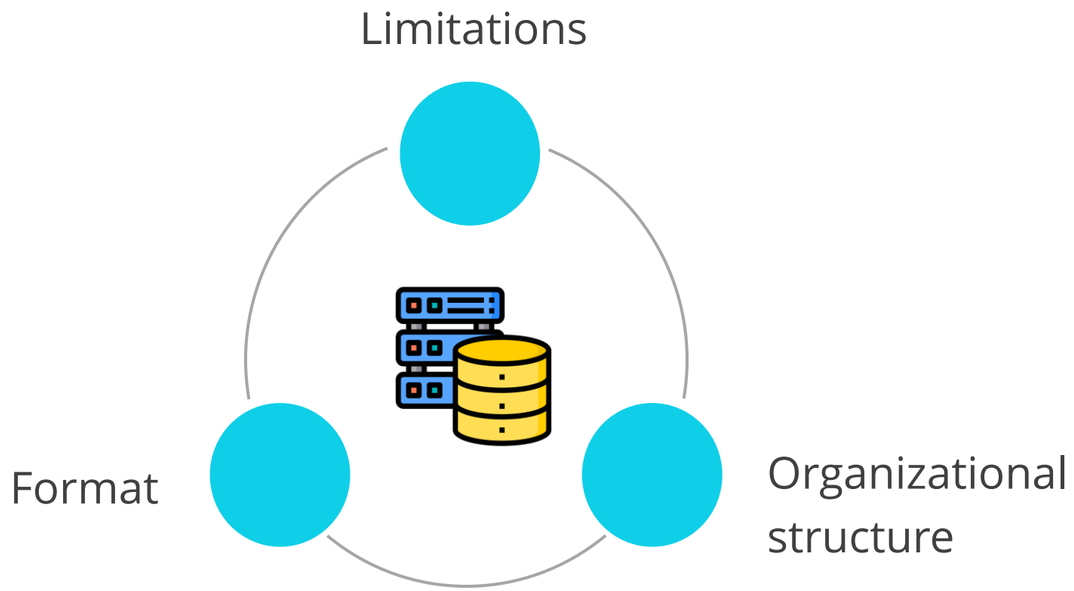
Helps to focus on a particular aspect of the application name

Enables effective code reuse and concurrent application development

With MVC, the program flow into and out of the application doesn't change even if the project structure deviates.

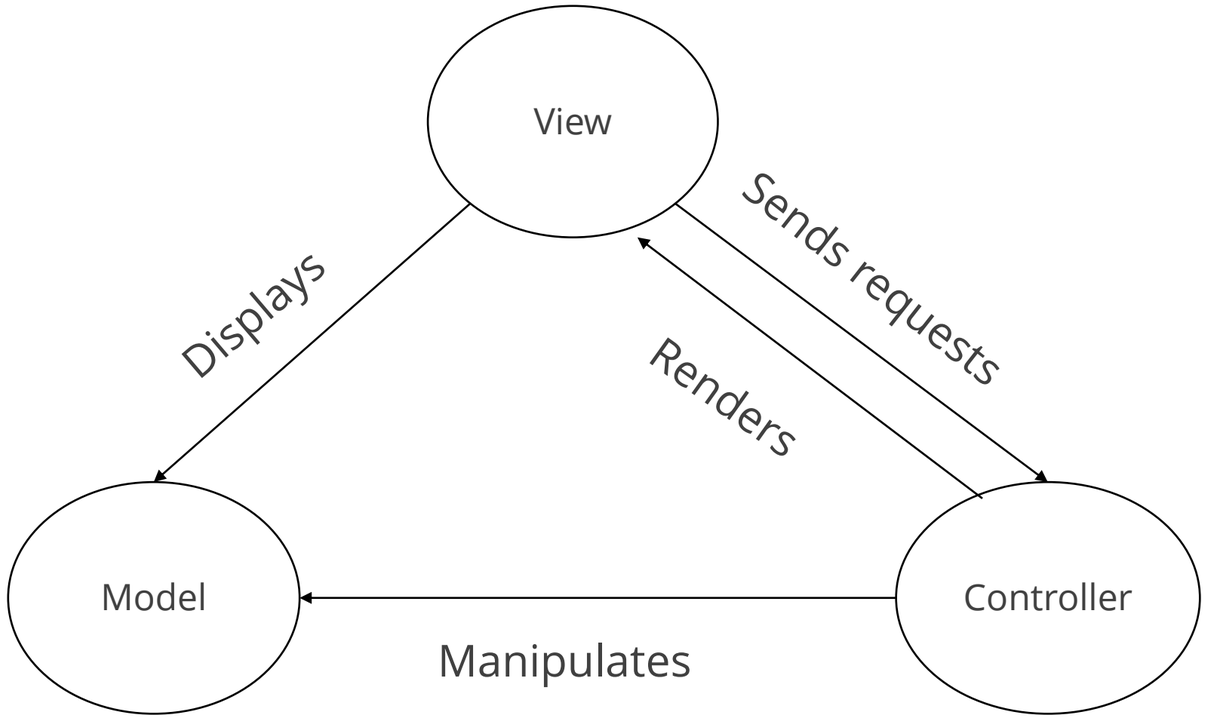
Model

The model component is responsible for representing all the user's data-related reasoning. This could be any other business logic-related data or the data that is being passed between the View and Controller components.



View

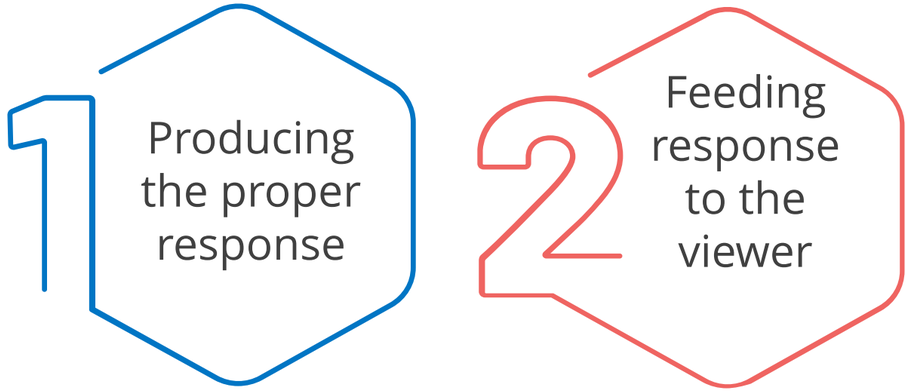
View is what the user is shown; they make use of the model and display data in the way the user desires. The data that is shown to the user may also be subject to modification.



They are made up of rendered or transmitted static and dynamic pages that the user requests.

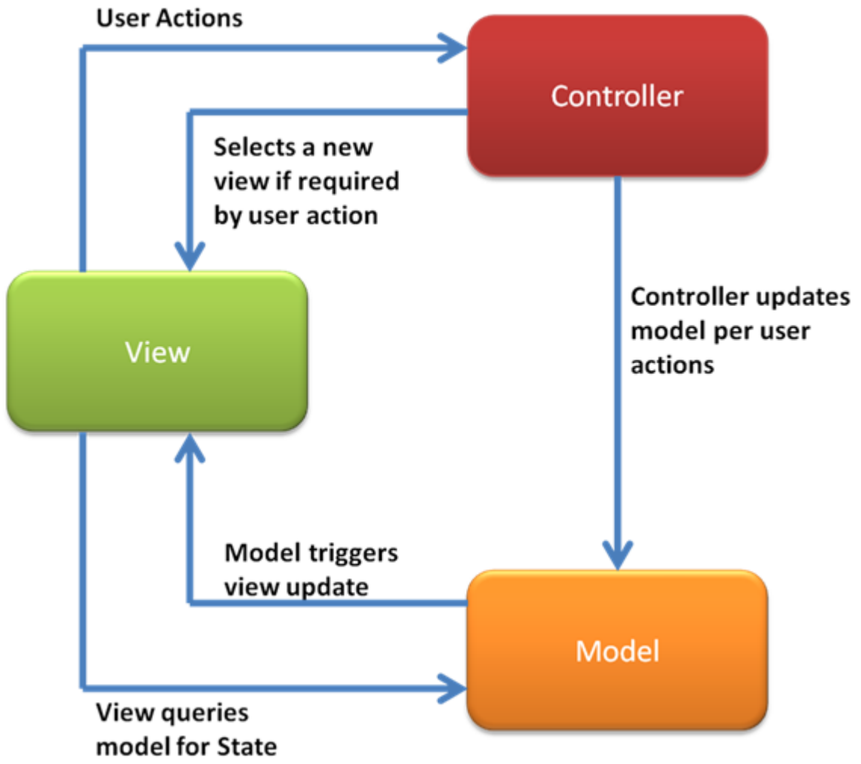
Controller

The controller functions as a mediator by enabling the link between the views and the model. It only needs to instruct the model and not worry about handling data logic. The controller is responsible for:



Controller

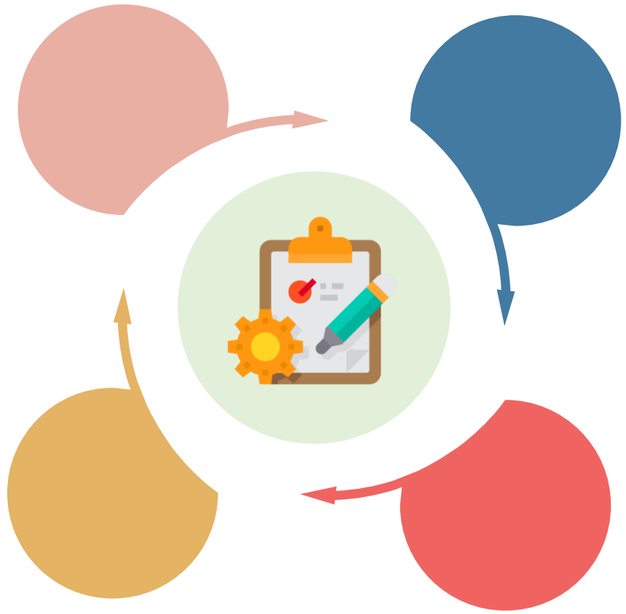
The data are entirely handled by the model, all the presentations are handled by the view, and the controller merely directs the model and view. This is how the MVC framework functions:



Configuring Express.js

Launch the terminal

Navigate to that directory



Make a directory for a project

Launch the project using the npm install command

Configuring Express.js

Insert the additional information requested:

```
Press ^C at any time to quit.
package name: (express-routing)
version: (1.0.0)
description: A simple app to implement Express Routing
entry point: (index.js)
test command:
git repository:
keywords:
author: Priyanka Yadav
license: (ISC)
```

Configuring Express.js

In this example, import the Express.js module and create an instance of the Express application:

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

Configuring Express.js

Employ the listen() method to build the server:

```
const port = 3000;
app.listen(3000, () => {
 console.log(`App running on port ${port}`);
});
```

Configuring Express.js

Launch the console and the module using nodemon app.js:

```
Priyanka@DESKTOP-B4MKO3A MINGW64 ~/Desktop/express-Routing (web-server)
$ nodemon app.js
[nodemon] 1.19.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node app.js`
App running on port 3000
```

Configuring Express.js

The request and the response are the only parameters that the callback function accepts.

```
app.<http header>("<route>",(req,res)=>{
<Callback function that specifies what we need to do when
the sepcific route is accessed>
});
```

Postman Configuration

Users are given the ability to generate, store, and read HTTP requests and their responses.

Distribute

Test

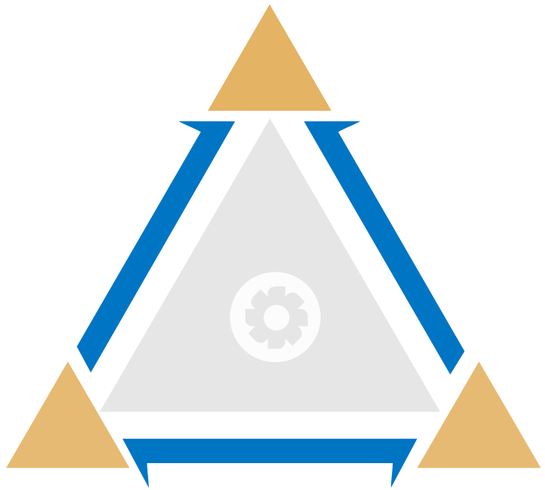
Create

Document APIs

Postman Configuration

To install a program known as Postman before writing the Express.js code:

Install the app

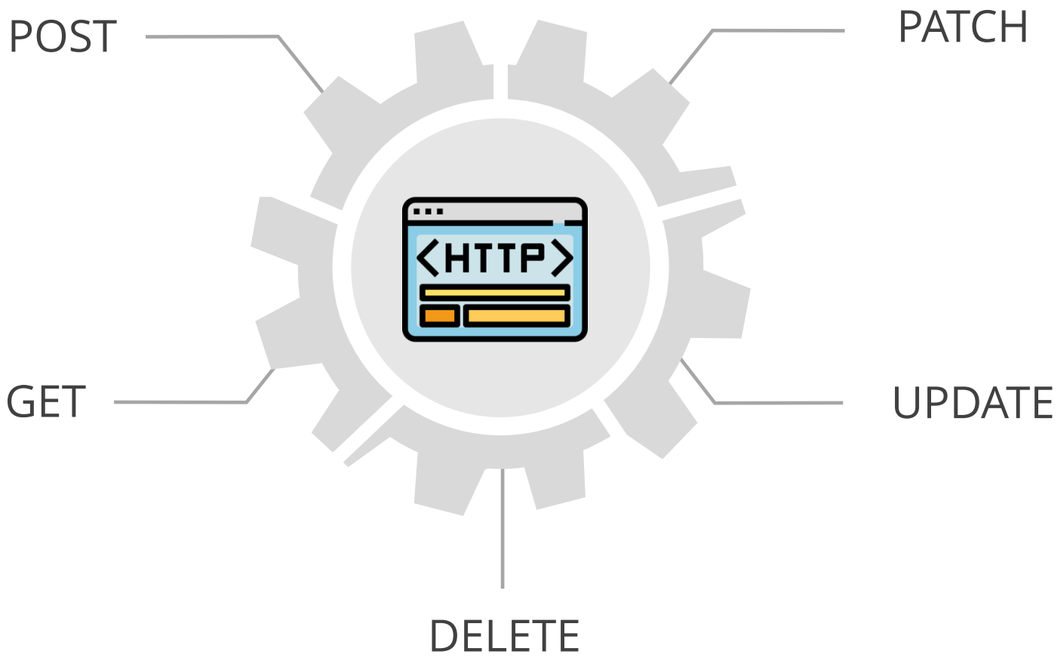


Get the app

Launch the app

Postman Configuration

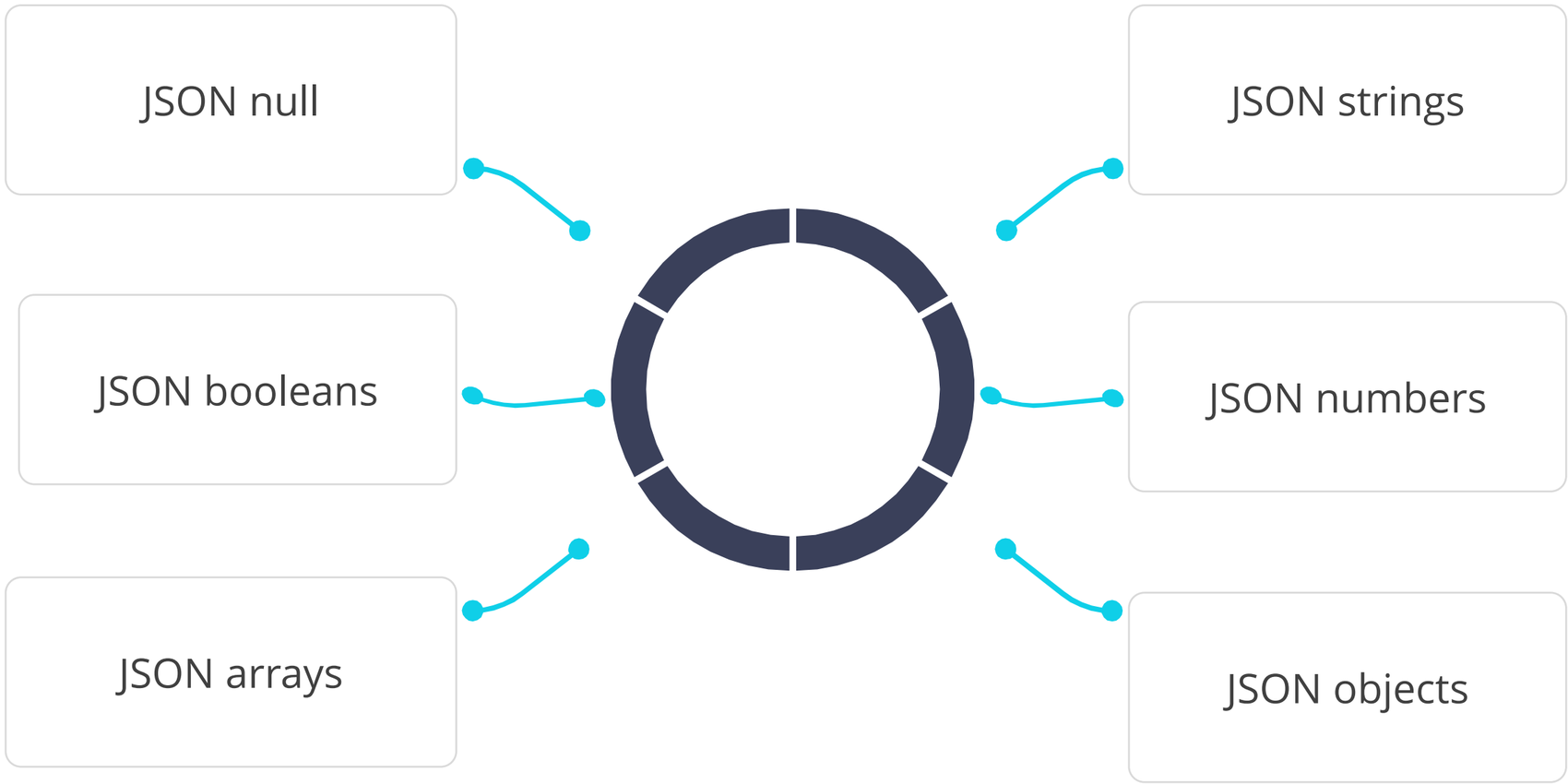
Organize the requests by saving them in collections and folders:



Postman offers the fundamental service required to create an API.

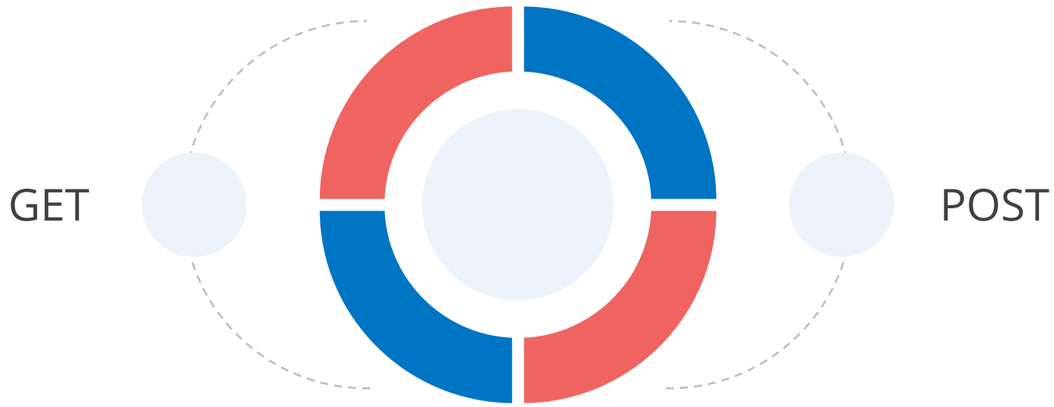
JSON Data

Values in JSON need to belong to one of the following data types:



Handle GET and POST Data

POST requests are used to submit data, while GET requests are used to retrieve data.



Express.js GET Request

The method GET must be called using an instance of Express.js.

When a user asks for that path, transmit the response from the callback.

Express.js GET Request

The Express.js POST request method is used to process POST requests.

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

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

app.listen(3000, () => {
  console.log('Example app listening on port 3000!')
})
```

It is not advised to use the GET method for sensitive data.

Express.js POST Request

A body parser is needed for Express.js to extract the incoming data from a POST request.

```
npm install body-parser
```

Express.js POST Request

This package needs to be imported into the project.

```
const express = require('express')
const bodyParser = require('body-parser')

const app = express()

app.use(bodyParser.json())
app.use(bodyParser.urlencoded({ extended: false }))

app.get('/', (req, res) => {
  res.send('Hello World!')
})
app.post('/', (req, res) => {
    let data = req.body;
    res.send('Data Received: ' + JSON.stringify(data));
})

app.listen(3000, () => {
  console.log('Example app listening on port 3000!')
})
```

Handling GET and POST Request

Problem Statement:

Duration: 15 min.

You have been assigned a task to demonstrate GET and POST request.

ASSISTED PRACTICE

Assisted Practice: Guidelines

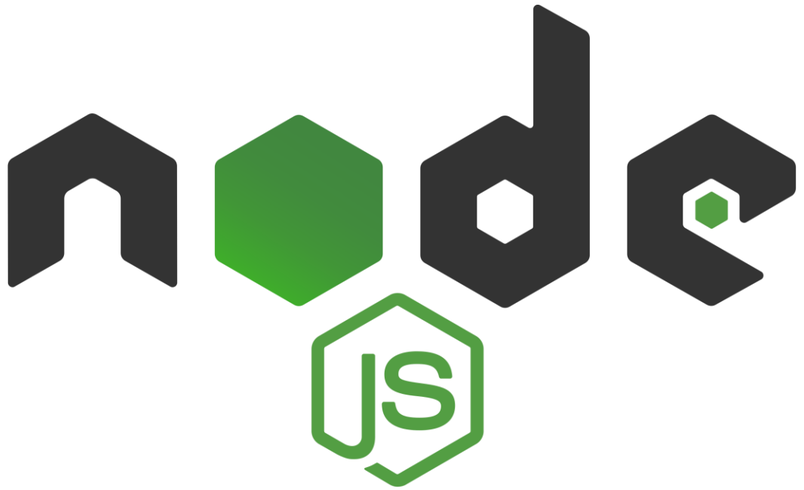
Steps to be followed:

1. Verify the Node.js installation
2. Create an Express app
3. Install and configure Postman
4. Handle GET and POST requests through Postman

Express.js Frameworks

Express.js Routers

Express.js Routers can define several routes or middleware to handle various incoming requests from clients.



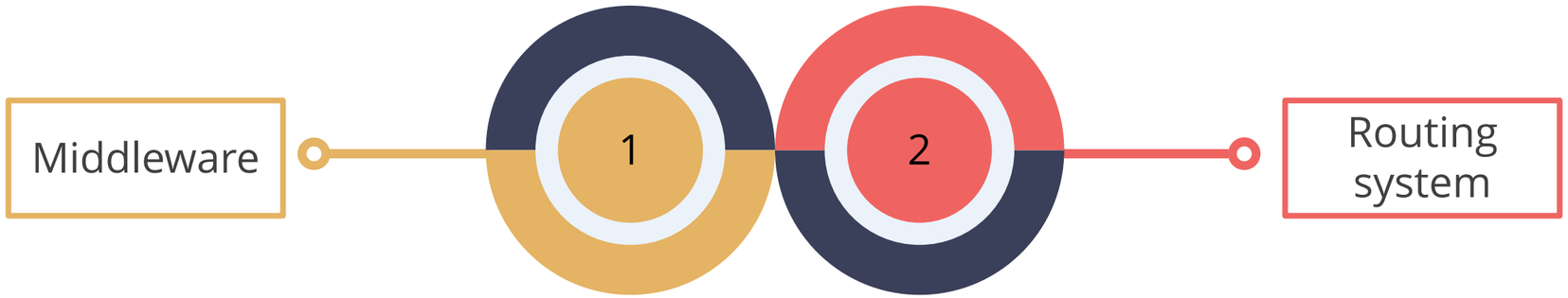
Express.js Routers

A new router object is made using the express.Router() function.

```
express.Router( [options] )
```

Express.js Routers

A Router instance is frequently referred to as a **mini-app**, because it is both:



Express.js Routers

Following are the steps for building a basic Express.js application with several key functionalities:

A router is created as a module.

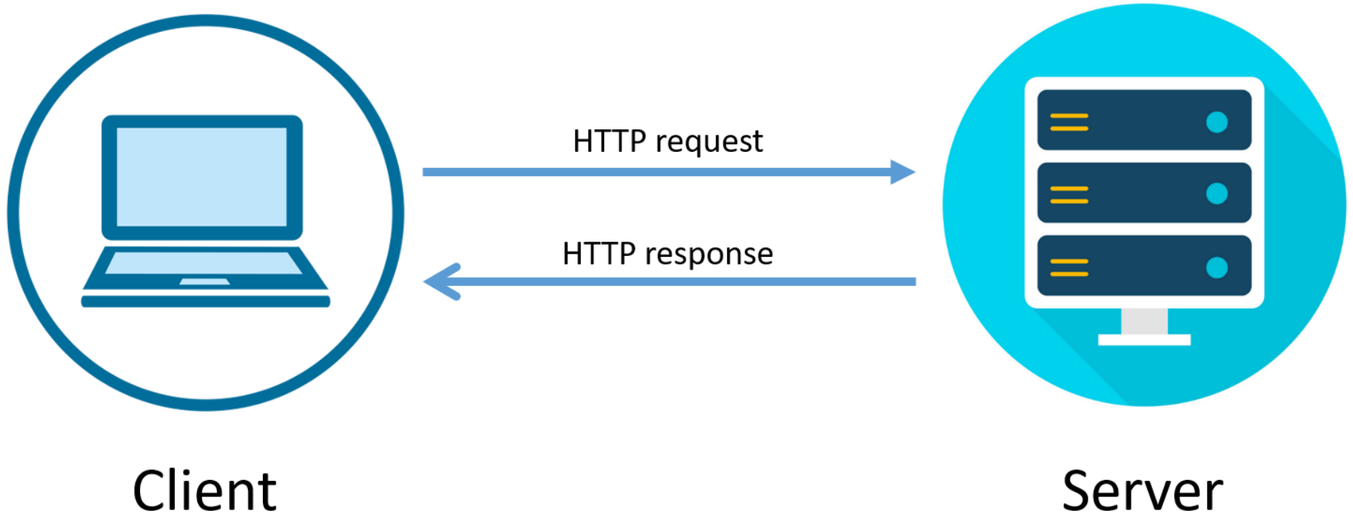
A middleware function is loaded.

The router module is mounted on a path.

Certain routes are defined.

Express.js Routers

Routing refers to choosing an application's response to a client request for a certain endpoint, which is a URI (or path) and an HTTP request type in particular.



When a route is matched, one or more handler functions for that route are called.

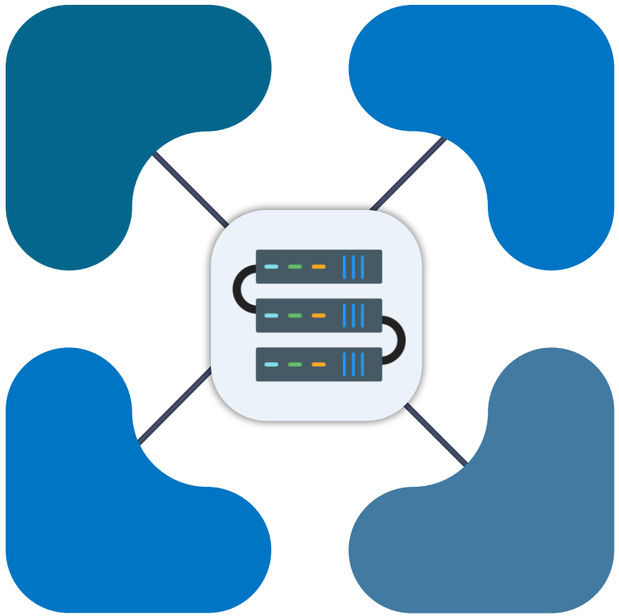
Express.js Routers

Route definition follows the form shown below:

```
app.METHOD(PATH, HANDLER)
```

In the above syntax:

Express uses the example of an app.



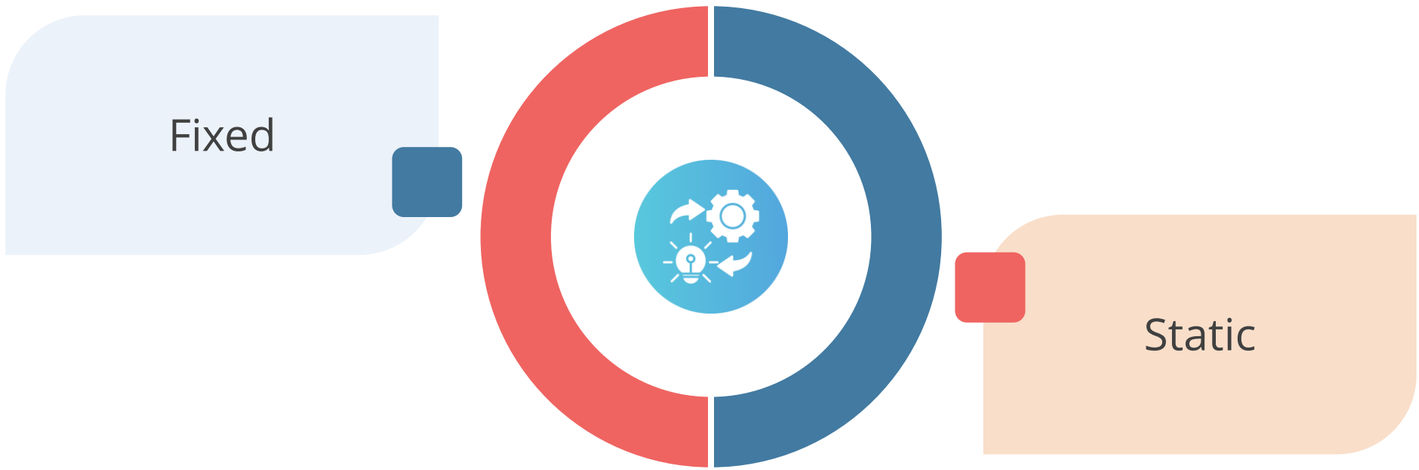
METHOD is the lowercase name of an HTTP request method.

When the route matches, the function HANDLER is called.

On the server, PATH is a path.

Express.js URL Building

It allows individuals to use dynamic routes by providing a range of route types.



It allows passing of parameters and process based on them by using dynamic routes.

Express.js URL Building

Illustration of a dynamic route in Express.js with the help of code:

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

app.get('/:id', function(req, res){
   res.send('The id you specified is ' + req.params.id);
});
app.listen(3000);
```

Express.js URL Building

The response will be as shown below:

The result will change if the user substitutes anything else for **123** in the URL.

Express.js URL Building

An advanced illustration of the routes is shown below:

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

app.get('/things/:name/:id', function(req, res) {
   res.send('id: ' + req.params.id + ' and name: ' +
req.params.name);
});
app.listen(3000);
```

Express.js URL Building

To access all the parameters the user passes in the URL, utilize the req.params object.

Users must define it separately if they wish to run code when they receive the value **/things**.

Pattern Matched Routes

Regex is another tool users may use to limit URL parameter matching.

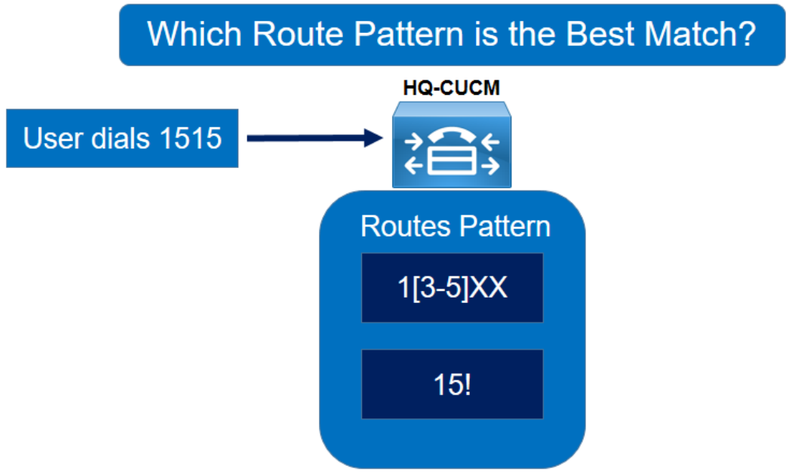
```
var express = require('express');
var app = express();

app.get('/things/:id([0-9]{5})', function(req, res){
   res.send('id: ' + req.params.id);
});

app.listen(3000);
```

Pattern Matched Routes

Keep in mind that only requests with a 5-digit long id will be matched by this. To match and validate the routes, one can use more sophisticated regexes.



One will receive a "Cannot GET your-request-route>" message as a response if none of the routes match the request.

Pattern Matched Routes

This message will be changed to a 404 error page.

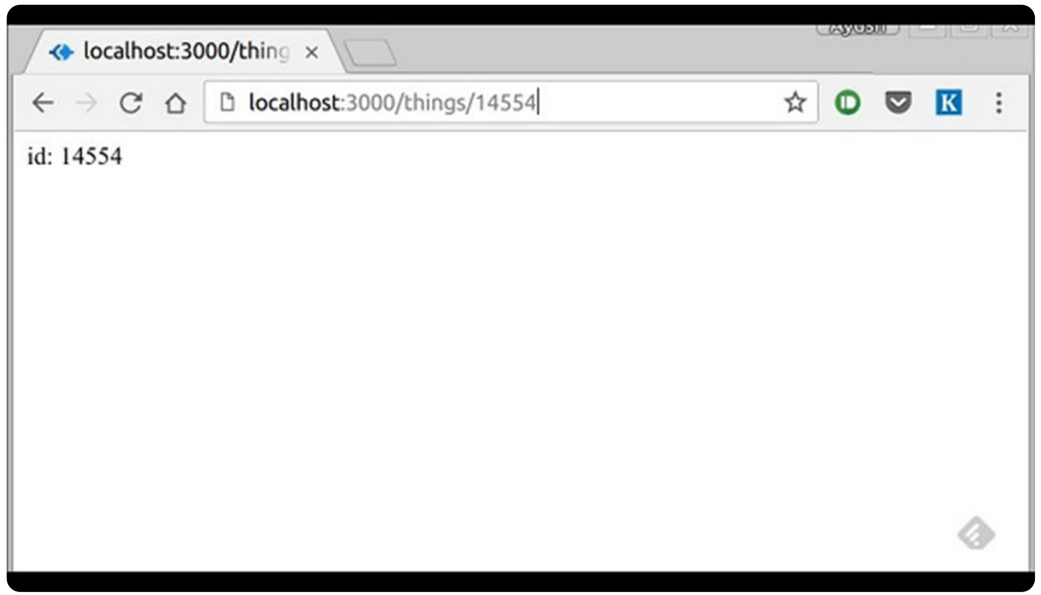
```
var express = require('express');
var app = express();

//Other routes here
app.get('*', function(req, res){
   res.send('Sorry, this is an invalid URL.');
});
app.listen(3000);
```

Since Express matches routes from the beginning to the finish of the index.js file, including the external routers needed, this should occur after all the routes because Express matches all the routes in the index.js file.

Pattern Matched Routes

The output is shown when requesting a valid URL.



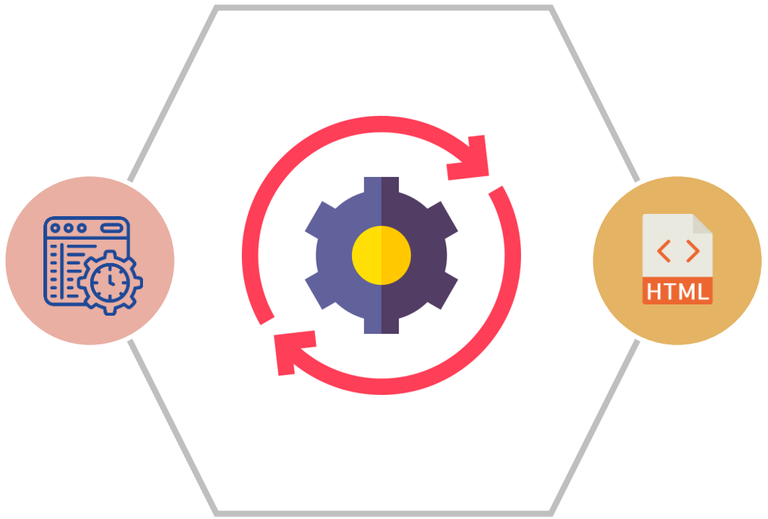
Pattern Matched Routes

The output for an erroneous URL request:

Express.js Templating

Express.js Templating is ideal for creating HTML pages quickly.

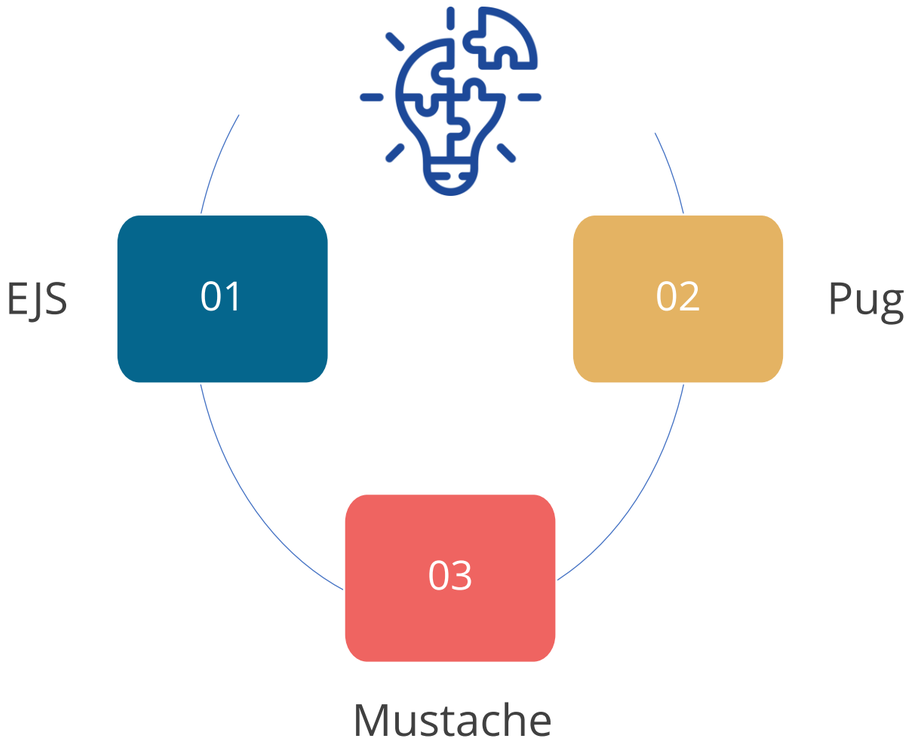
It changes variables in a template file into real values during runtime.



It converts the template into an HTML file that is provided to the client.

Express.js Templating

Examples of Templating Engines:



Express.js Templating

To avoid clogging up your server code with HTML, use templating engines to concatenate errant strings to preexisting HTML templates. It includes:

Express.js Templating

Establish Pug as the app's templating engine now that it has been deployed. Using the below command pug can be added to a project.

Here –-save is used to save pug inside dependencies in package.json file.

```
npm install --save pug
```

Express.js Templating

Making a new directory called **views** in Express.js.

Use below code as reference:

```
app.set('view engine', 'pug');
app.set('views','./views');
```

Express.js Templating

Create the first view.pug file inside of that and fill it with the information listed below:

```
doctype html
html
   head
      title = "Hello Pug"
   body
      p.greetings#people Hello World!
```

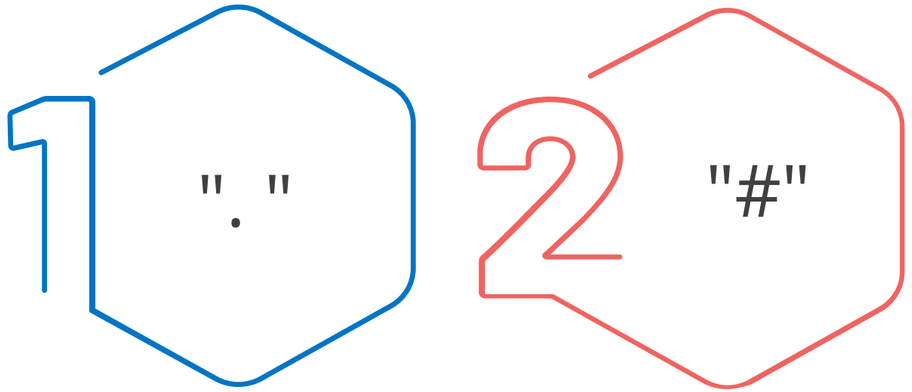
Express.js Templating

Add the following route to the app to run this page:

```
app.get('/first_template', function(req, res){
   res.render('first_view');
});
```

Express.js Templating

Instead of using the class and id keywords, you should specify them using the characters.



Express.js Templating

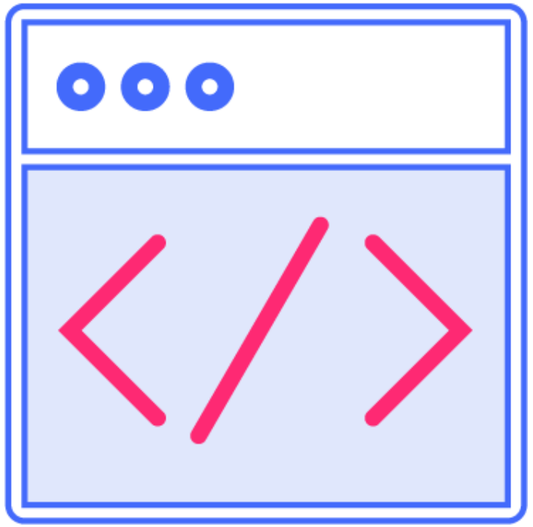
Pug can do much more than just making the HTML markup simpler.

```
<!DOCTYPE html>
<html>
   <head>
      <title>Hello Pug</title>
   </head>

   <body>
      <p class = "greetings" id = "people">Hello World!</p>
   </body>
</html>
```

Important Features of Pug

The body> tag was on the same indentation as the head> tag; they were siblings.



Indentation determines how tags are nested.

Important Features of Pug

Pug automatically closes tags for the user as soon as it encounters the tags at the following indentation levels:

Important Features of Pug

```
H1 Greetings from Pug
```

Space separated

```
Div
 | To add many lines of text,
| The pipe operator is available.
```

Piped text

Important Features of Pug

Explaining Blocking of text in pug.

```
div.
But, if you have a lot of text, that becomes tedious.
To indicate a block of text, add "." to the end of the
tag.
Enter tag in a new line with the appropriate indent to
place it inside this block.
```

Important Features of Pug

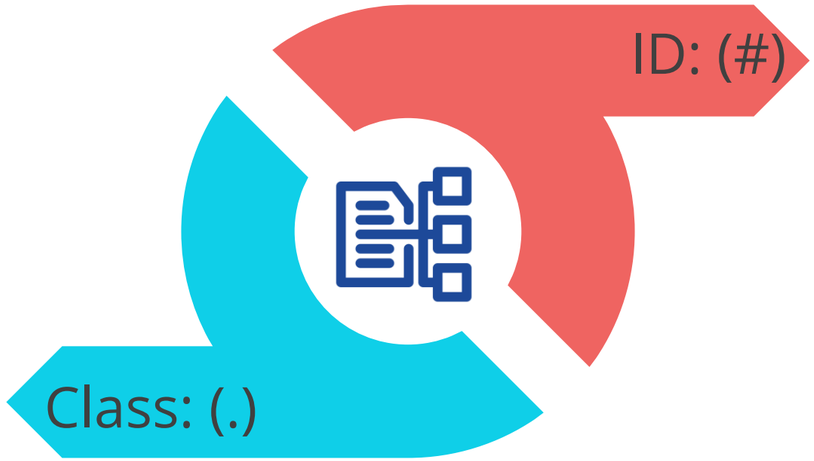
This comment is transformed into HTML comment using the syntax(<!--comment-->).

```
/A Pug-related comment
```

```
<!--This is a Pug comment-->
```

Important Features of Pug

User utilizes a parenthesized, comma-separated list of attributes to define them.
Attributes of the class and ID have unique representations.



Important Features of Pug

The definition of attributes, classes, and id for a certain html tag is covered in the following line of code:

```
div.container.column.main#division(width = "100",
height = "100")
```

```
<div class = "container column main" id =
"division" width = "100" height = "100"></div>
```

Important Features of Pug

Users can send a value from the route handler when rendering a Pug template.

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

app.get('/dynamic_view', function(req, res){
   res.render('dynamic', {
      name: "TutorialsPoint",
      url:"http://www.tutorialspoint.com"
   });
});

app.listen(3000);
```

Important Features of Pug

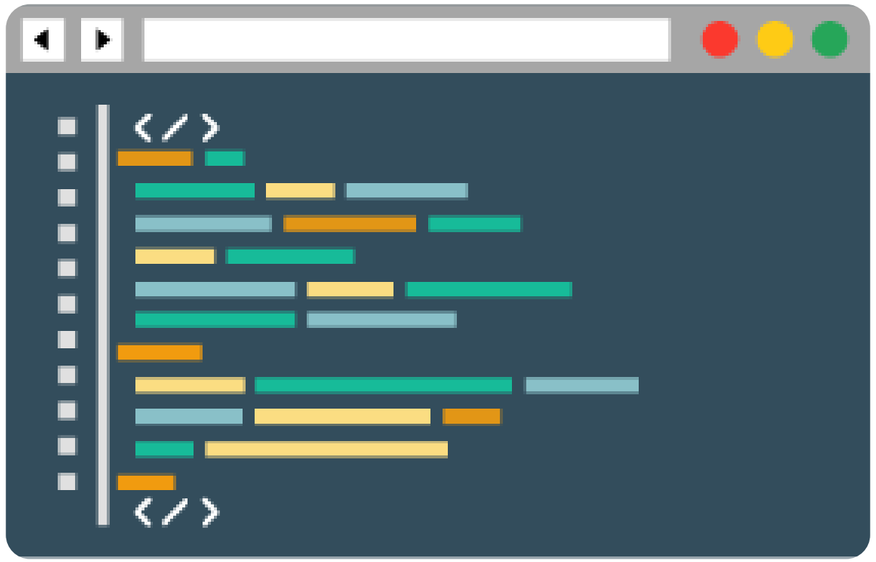
Example of dynamic pug : Add the code below to a new view file called dynamic.pug in the views directory.

```
html
    head
        title=name
    body
        h1=name
        a(href = url) URL
```

Open the browser to localhost:3000/dynamic view.

Important Features of Pug

With the #variableName syntax, passed variables can be inserted between the text of a tag.



Important Features of Pug

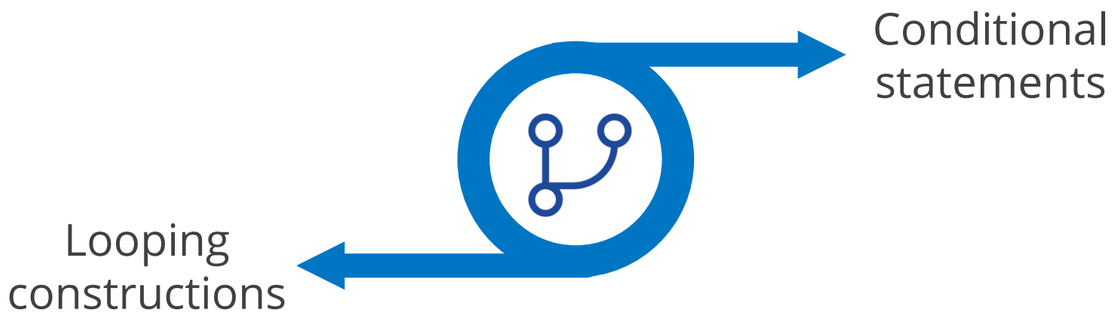
The value-using technique in pug to add greetings is explained through below code.

```
html
   head
      title = name
   body
      h1 Greetings from #{name}
      a(href = url) URL
```

Interpolation is the name of this value-using technique.

Important Features of Pug

Example of conditionals in pug: The page ought to say "Hello, User" if a user is signed in, and if not, the "Login/Sign Up" link should show up.



Important Features of Pug

Explaining straightforward template in pug using below code:

```
html
   head
      title Simple template
   body
      if(user)
         h1 Hi, #{user.name}
      else
         a(href = "/sign_up") Sign Up
```

Important Features of Pug

Users can send an object while rendering this using the routes.

```
res.render('/dynamic',{
   user: {name: "Ayush", age: "20"}
});
```

Important Features of Pug

Pug is a very user-friendly approach to building web page components.

Users may make use of the included feature rather than copying it into each new view they make.

Important Features of Pug

Creating 3 views using below code in pug.

```
div.header.
I am this website's header.
```

Header.pug

```
html
   head
      title Simple template
   body
      include ./header.pug
      h3 I'm the main content
      include ./footer.pug
```

Content.pug

```
div.footer.
I am this website's footer.
```

Footer.pug

Important Features of Pug

Code for constructing router using pug is shown below:

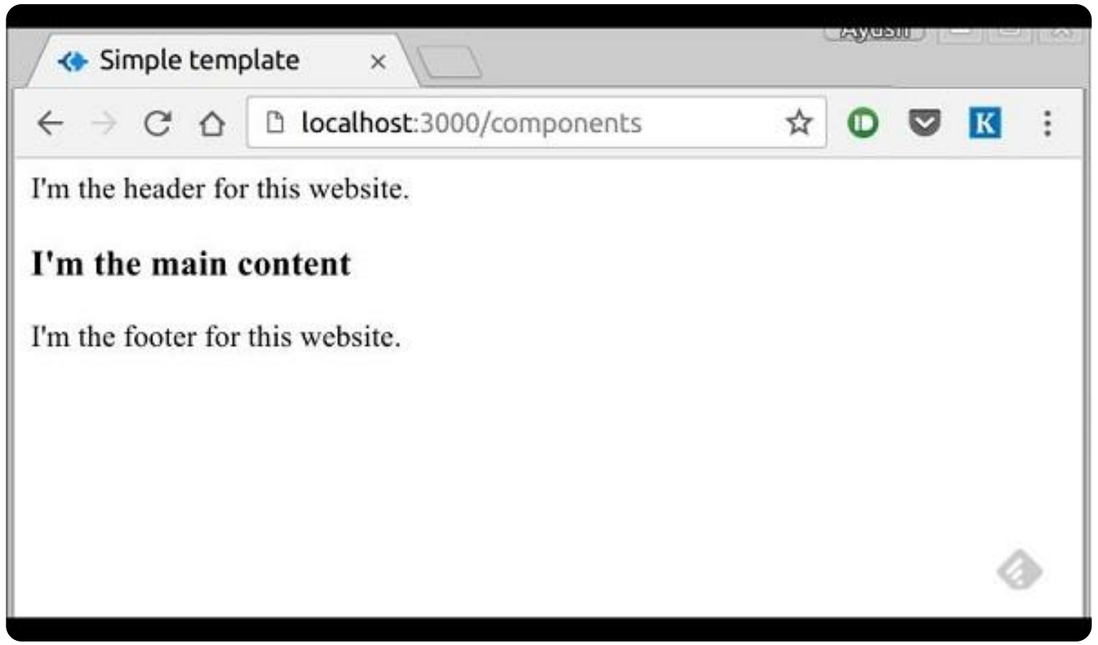
```
var express = require('express');
var app = express();

app.get('/components', function(req, res){
    res.render('content');
});

app.listen(3000);
```

Important Features of Pug

Plaintext, CSS, and JavaScript can all be included with the include command.



Express.js-StaticFiles

Express.js-StaticFiles

Express.js-Staticfiles uses serve-static as its foundation and serves static files.

```
express.static(root, [options])
```

The root argument specifies the root directory from which static assets will be served.

Express.js-StaticFiles

With this command, a user may install this package:

```
npm install express
```

Express.js-StaticFiles

Users may verify the express version.

```
npm version express
```

Express.js-StaticFiles

To execute this file, users must issue the aforementioned command.

```
node index.js
```

Express.js-StaticFiles

Example:

```
var express = require('express');
var app = express();
var path = require('path');
var PORT = 3000;

// Static Middleware
app.use(express.static(path.join(__dirname, 'public')))

app.get('/', function (req, res, next) {
    res.render('home.ejs');
})

app.listen(PORT, function(err){
    if (err) console.log(err);
    console.log("Server listening on PORT", PORT);
});
```

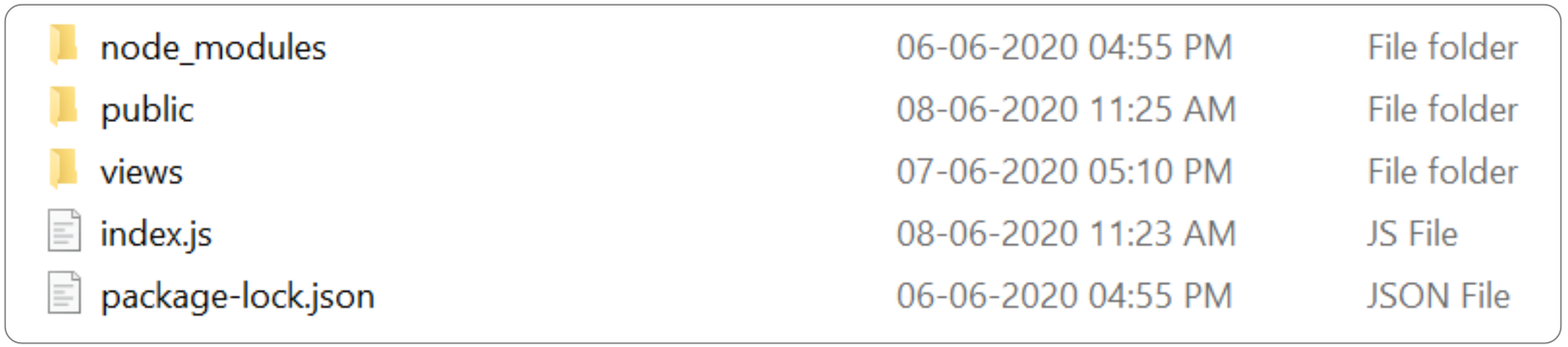
Express.js-StaticFiles

Add the code below to the home.ejs file in the views folder.

```
<!DOCTYPE html>
<html>
<head>
    <title>express.static() Demo</title>
</head>
<body>
<h2>Greetings from Skillcart</h2>
<img src="Demo.jpg" width="150" height="100" />
</body>
</html>
```

Express.js-StaticFiles

The public folder is now being provided to the server as static.



Express.js-StaticFiles

With the help of following commands:

```
npm install express
npm install ejs
```

Express and the ejs module are installed.

```
node index.js
```

Run the index.Js file.

Express.js- StaticFiles

Output:

```
Server listening on PORT 3000
```

Navigate to http://localhost:3000/ to see the output.

Express.js Form Data

Installing the body parser and multer middleware is the first step in getting forms up and running.

```
npm install --save body-parser multer
```

Express.js Form Data

Add the following code to the index.Js file:

```
var express = require('express');
var bodyParser = require('body-parser');
var multer = require('multer');
var upload = multer();
var app = express();

app.get('/', function(req, res){
   res.render('form');
});

app.set('view engine', 'pug');
app.set('views', './views');

// for parsing application/json
app.use(bodyParser.json());
```

```
// for parsing application/xwww-
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("recieved your request!");
});
app.listen(3000);
```

Express.js Form Data

Utilize the body parser for processing json and x-www-form-url encoded header requests.

Express.js Form Data

The code below will create a new view with the name form.pug.

```
html
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
```

Express.js Form Data

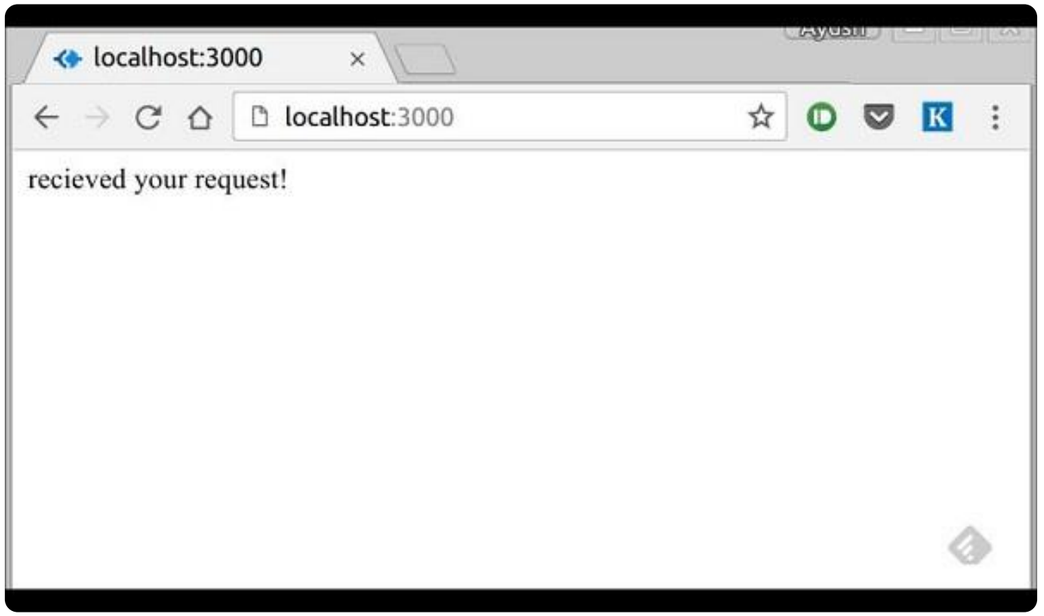
Run the server with the next methods.

```
nodemon index.js
```

Go to localhost:3000/ at this point, fill out the form as desired, and then submit it.

Express.js Form Data

The subsequent response will be shown:



Express.js Form Data

The user will see the body of the request displayed as a JavaScript object.

The parsed request content is contained in the req.body object.

Error Handling

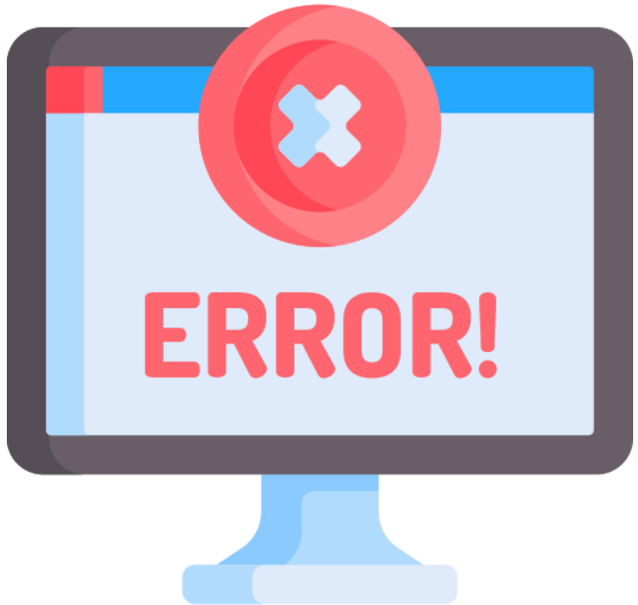
The term **Error Handling** describes how Express detects and handles the following issues:

Synchronous issues

Asynchronous issues

Catching Errors

It is crucial to make sure Express detects every mistake that happens while middleware and route handlers are being used.



Asynchronous code that throws an error will be caught and handled by Express.

Catching Errors

For instance:

```
app.get('/', (req, res) => {  throw new Error('BROKEN') //
Express will catch this on its own.})
```

Catching Errors

Users must pass errors from asynchronous functions called by middleware and route handlers to the next() function.

Catching Errors

Using below code for instance in catching errors:

```
app.get('/', (req, res, next) => {  fs.readFile('/file-does-not-
exist', (err, data) => {    if (err) {      next(err) // Pass errors
to Express.    } else {      res.send(data)    }  })})
```

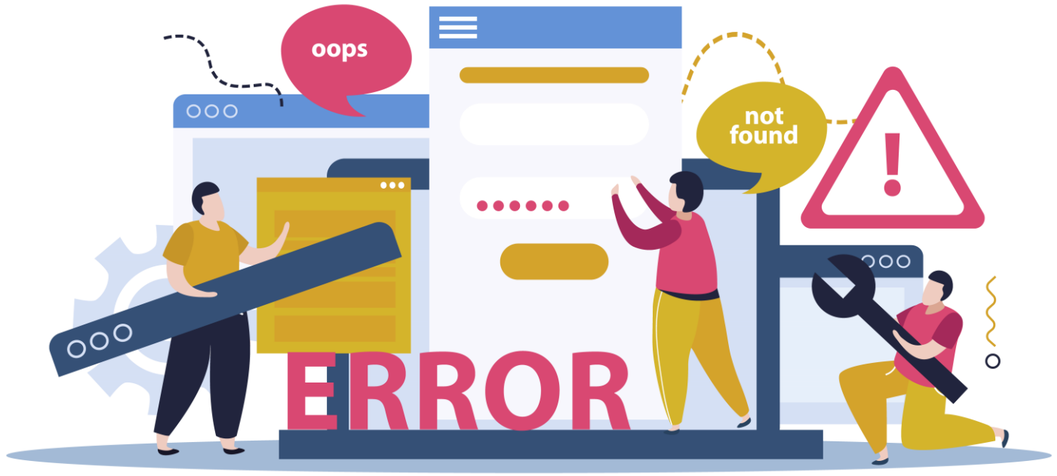
Catching Errors

Whenever they refuse a request or throw an error, they will automatically call next.

```
app.get('/user/:id', async (req, res, next) => {  const user = await
getUserById(req.params.id) res.send(user)})
```

Catching Errors

Next() will be called with a default Error object provided by the Express router if no rejected value is provided.



The current request is an error and skips all remaining non-error handling routing and middleware routines if you give anything to the next() function.

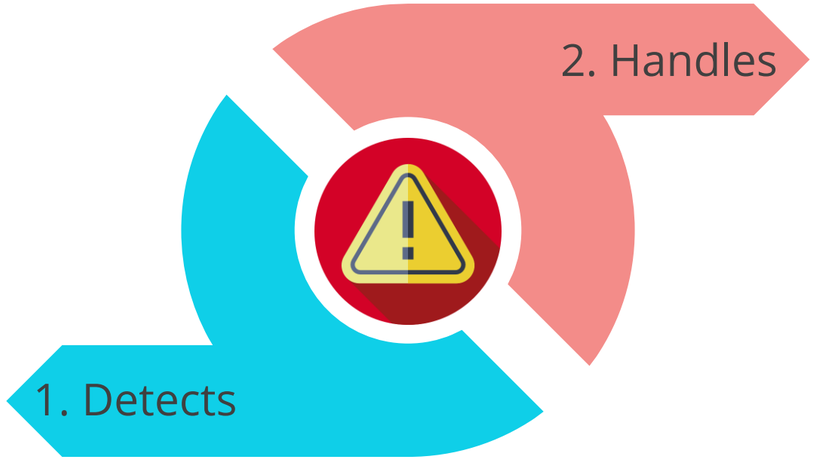
Catching Errors

Users can reduce this code as follows if the callback in a series returns only errors and no data:

```
app.get('/', [  function (req, res, next) {
fs.writeFile('/inaccessible-path', 'data', next)  },  function (req,
res) {    res.send('OK')  }])
```

Catching Errors

The second handler is executed if there is no error; otherwise, Express detects and handles the error.



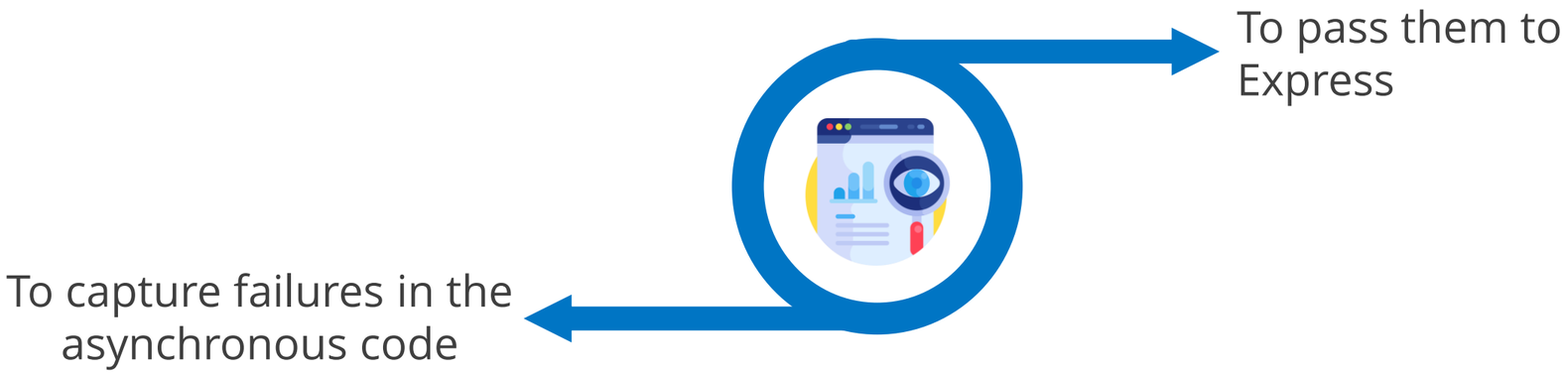
Catching Errors

Errors in asynchronous code that is called by route handlers must be caught and passed to express for processing.

```
app.get('/', (req, res, next) => {  setTimeout(() => {    try {
throw new Error('BROKEN')    } catch (err) {      next(err)    }  },
100) })
```

Catching Errors

Express would not detect the problem if the try...catch block was skipped because it is not a component of the synchronous handler function.



Catching Errors

If a function returns a promise, use promises instead of a try...catch block to avoid the overhead.

```
app.get('/', (req, res, next) => {  Promise.resolve().then(() => {
throw new Error('BROKEN')  }).catch(next) // Errors will be passed to
Express.})
```

Catching Errors

In JavaScript, when working with promises, the catch handler is used to handle any errors that might occur during the promise chain.

Synchronous errors

Rejected promises

Synchronous errors occur in regular, synchronous JavaScript code and are caught using try-catch, while rejected promises happen in asynchronous operations and are handled using .catch() or catch in promise chains.

Catching Errors

The asynchronous code can be as straightforward as possible, even when utilizing synchronous error handling by chaining multiple error handlers. For Instance:

```
app.get('/', [  function (req, res, next) {    fs.readFile('/maybe-
valid-file', 'utf-8', (err, data) => {      res.locals.data = data
next(err)    })  },  function (req, res) {    res.locals.data =
res.locals.data.split(',')[1]    res.send(res.locals.data)  }])
```

Framework for Catching Errors

A lightweight framework is designed to enhance Node.js web server capabilities by refining existing APIs. When discussing the readFile function within this context:

1. If readFile encounters a problem, it sends the error to Express.
2. If not, the next handler in the chain swiftly returns to the world of synchronous error handling.

Catching Errors

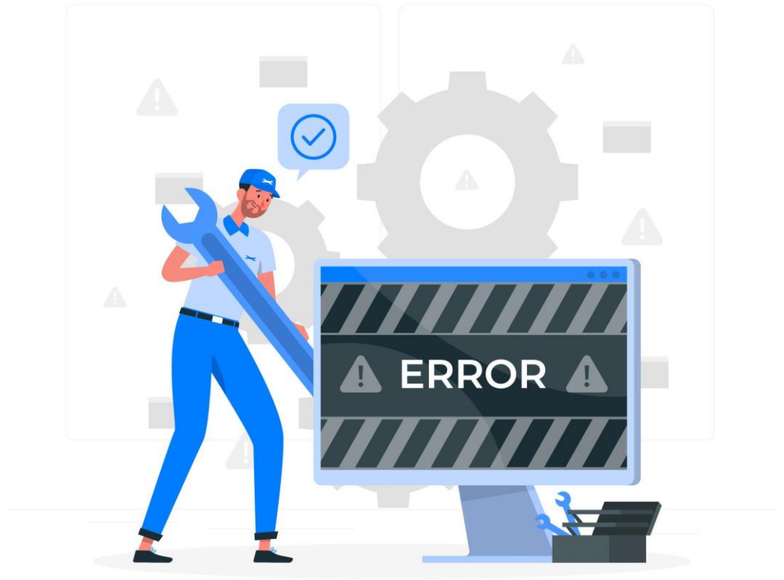
One must make sure that Express is informed of the error. Informing Express about encountered errors is crucial for the following reasons:.

Application continuity: Notifying Express avoids server crashes due to unhandled errors.

Error handling middleware invocation: Informed Express triggers custom error handling, preventing server disruptions.

The Default Error Handler

All issues that can arise in the app are handled by an error handler that is a part of Express.



The middleware function stack's final entry is this default error-handling function.

The Default Error Handler

The built-in error handler will take care of any errors you send to the next() without handling them in a custom error handler.

To run the app in production mode, set the environment variable from NODE_ENV to production.

The Default Error Handler

The err.status is used to set the res.statusCode (or err.statusCode).

The status code determines how to set the res.statusMessage.

The headers are listed in an object with the err.headers name.

The status code message's HTML body will be used; if not, err.Stack will be used.

The Default Error Handler

The Express default error handler shuts the connection and rejects the request if one calls the next() with an error after they have begun writing the response.

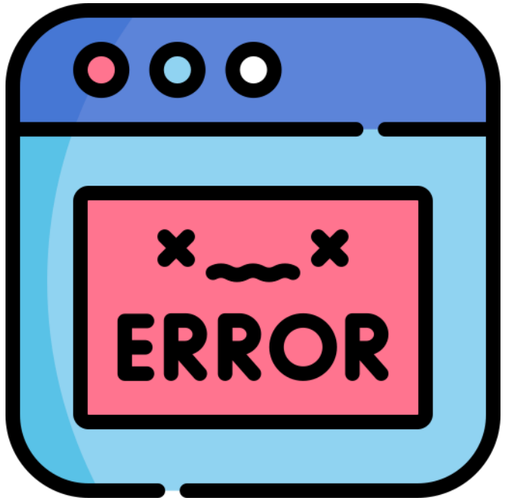
The Default Error Handler

Users must delegate to the default Express error handler when adding a custom error handler.

```
function errorHandler (err, req, res, next) {  if (res.headersSent) {
return next(err)  }  res.status(500)  res.render('error', { error: err
})}
```

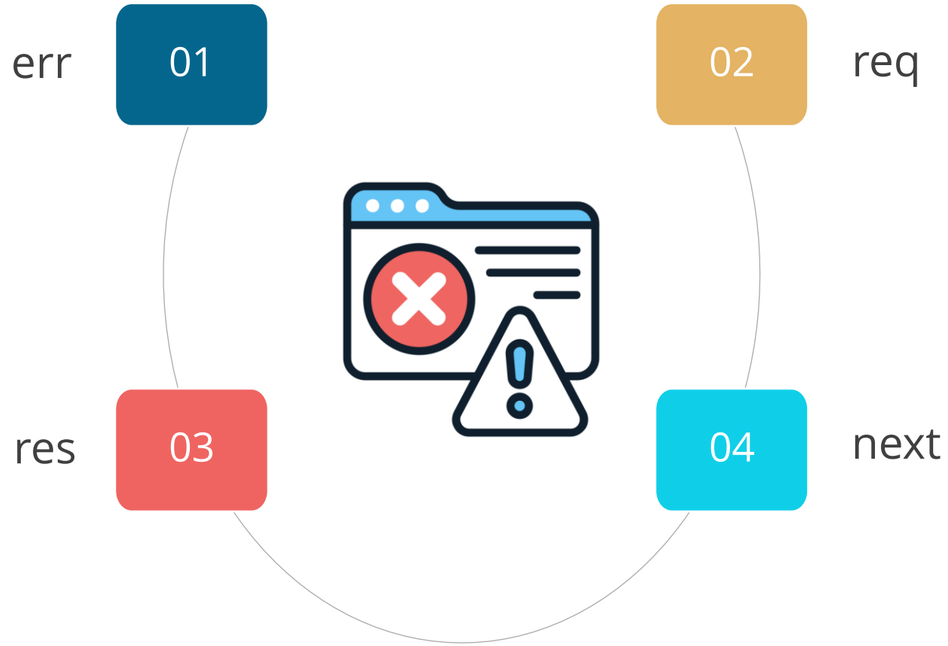
The Default Error Handler

The default error handler may be activated if one calls the next() more than once when the code contains an error.



Writing Error Handlers

Similar to previous middleware functions, but with four arguments as opposed to three, define error-handling middleware functions as follows:



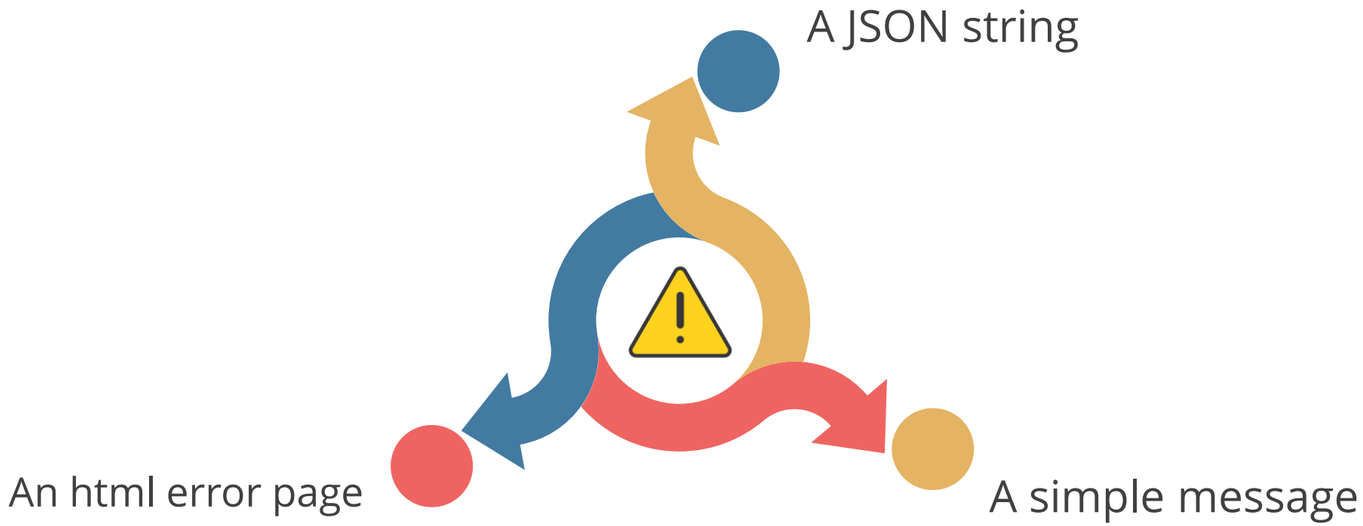
Writing Error Handlers

Using Error Handlers for instance:

```
app.use((err, req, res, next) => {  console.error(err.stack)
res.status(500).send('Something broke!')})const bodyParser =
require('body-parser')const methodOverride = require('method-
override') app.use(bodyParser.urlencoded({  extended:
true}))app.use(bodyParser.json())app.use(methodOverride())app.use((err
, req, res, next) => {  // logic})
```

Writing Error Handlers

Answers from a middleware function may come in any format.



Construct a number of error-handling middleware functions for the organization.

Writing Error Handlers

To specify a different error handler for requests made with and without XHR:

```
const bodyParser = require('body-parser')const methodOverride =
require('method-override') app.use(bodyParser.urlencoded({  extended:
true}))app.use(bodyParser.json())app.use(methodOverride())app.use(logE
rrors)app.use(clientErrorHandler)app.use(errorHandler)
```

Writing Error Handlers

In the following scenario, the general logErrors function might log request and error data to the stderr (standard error) stream.

```
function logErrors (err, req, res, next) {
console.error(err.stack)  next(err)}
```

Writing Error Handlers

It is crucial to compose the answer when the error-handling function does not call **next**.

```
function clientErrorHandler (err, req, res, next) {  if (req.xhr) {
res.status(500).send({ error: 'Something failed!' })  } else {
next(err)  }}
```

The requests will **hang** and won't be picked up for trash.

Writing Error Handlers

To **catch-all** errors, implement the error Handler function as follows :

```
function errorHandler (err, req, res, next) {  res.status(500)
res.render('error', { error: err })}
```

Writing Error Handlers

The route argument to jump to the next route handler if a route handler has numerous callback functions:

```
app.get('/a_route_behind_paywall',  (req, res, next) => {    if
(!req.user.hasPaid) {      // continue handling this request
next('route')    } else {      next()    }  }, (req, res, next) => {
PaidContent.find((err, doc) => {      if (err) return next(err)
res.json(doc)    })  })
```

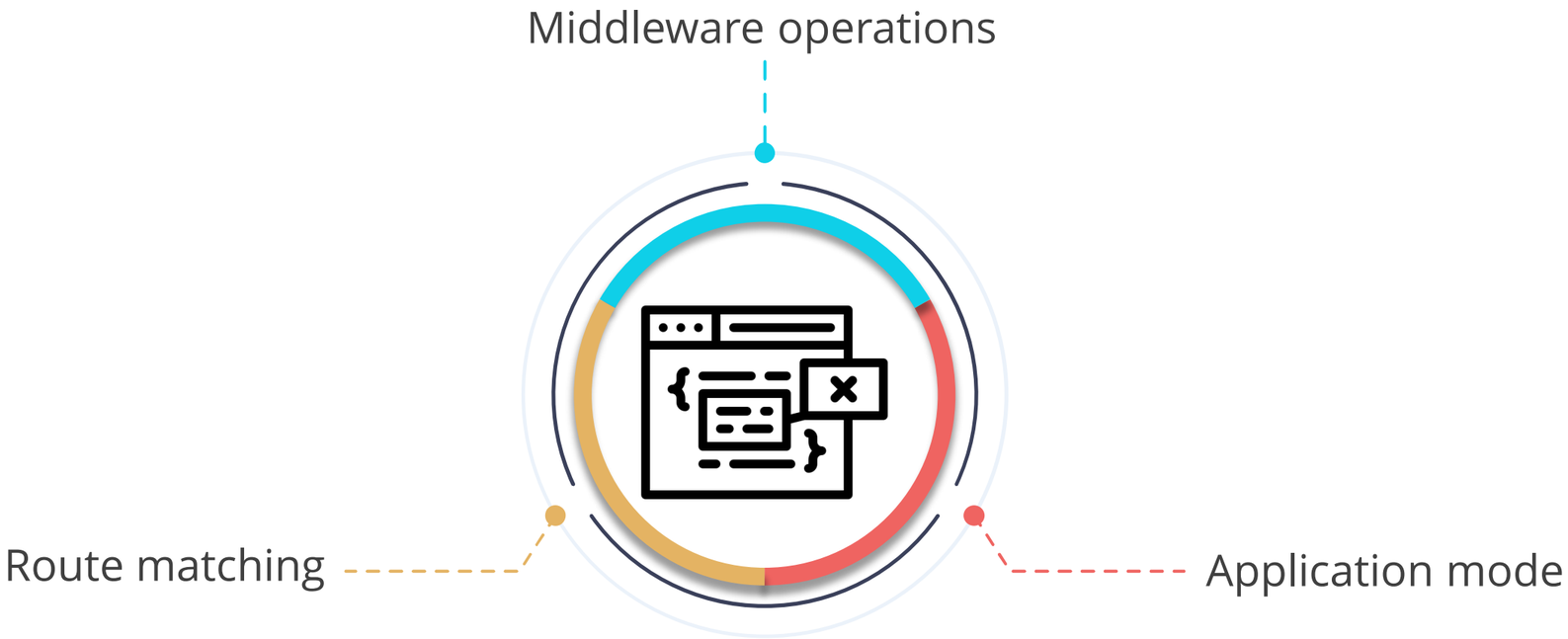
Writing Error Handlers

The getpaidcontent handler will not run, but any other handlers in the app for a route behind the paywall will run.

The completion and state of the current handler are indicated by calls to next() and next(err).

ExpressJS Module for Debugging

Express logs internal information.



ExpressJS Module for Debugging

The DEBUG environment variable to express:* to view all internal logs:

```
DEBUG = express:* node index.js
```

ExpressJS Module for Debugging

Following is the desired output:

```
ayushgp@swaggy:~/hello-world$ DEBUG=express:* node index.js
  express:application set "x-powered-by" to true +0ms
  express:application set "etag" to 'weak' +3ms
  express:application set "etag fn" to [Function: wetag] +2ms
  express:application set "env" to 'development' +0ms
  express:application set "query parser" to 'extended' +0ms
  express:application set "query parser fn" to [Function: parseExtendedQueryString] +0ms
  express:application set "subdomain offset" to 2 +0ms
  express:application set "trust proxy" to false +1ms
  express:application set "trust proxy fn" to [Function: trustNone] +0ms
  express:application booting in development mode +0ms
  express:application set "view" to [Function: View] +0ms
  express:application set "views" to '/home/ayushgp/hello-world/views' +0ms
  express:application set "jsonp callback name" to 'callback' +0ms
  express:router use / query +8ms
  express:router:layer new / +1ms
  express:router use / expressInit +1ms
  express:router:layer new / +0ms
  express:router use / jsonParser +0ms
  express:router:layer new / +0ms
  express:router use / urlencodedParser +2ms
  express:router:layer new / +0ms
  express:router use / multerMiddleware +1ms
  express:router:layer new / +0ms
  express:router:route new / +0ms
  express:router:layer new / +0ms
  express:router:route get / +1ms
  express:router:layer new / +0ms
  express:router:route new /:id([0-9]{3,}) +0ms
  express:router:layer new /:id([0-9]{3,}) +0ms
  express:router:route get /:id([0-9]{3,}) +0ms
```

ExpressJS Module for Debugging

These logs are useful when a part of the software isn't working properly.

The DEBUG variable can also be limited to a certain area to be logged.

ExpressJS Module for Debugging

Limit the logging to the application and router using the following code:

```
DEBUG = express:application,express:router node index.js
```

ExpressJS Module for Debugging

Debug is by default off, but it is automatically turned on in a production environment.

Debug can be customized to the user’s needs.

Applications Generated by Express+

The debug module is used by an application created by the express command.

Debug namespace is scoped to the application's name.

Applications Generated by Express+

If one uses the $ express sample app to create the app, one may activate the debug statements by using the below command:

```
$ DEBUG=sample-app:* node ./bin/www
```

Applications Generated by Express+

By providing a list of names separated by commas, one can create multiple debug namespaces:

```
$ DEBUG=http,mail,express:* node index.js
```

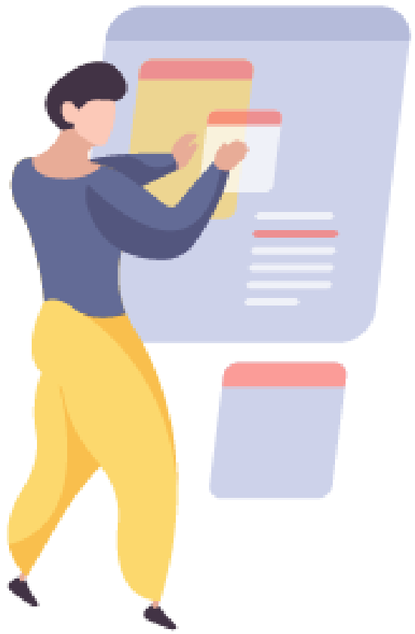
Advanced Options

A few environment variables that one can adjust when using Node.js to change how the debug logging functions:

Name	Purpose
DEBUG	Specifies which debugging namespaces to enable or disable
DEBUG_COLORS	Uses colors in the debug output is optional
DEBUG_DEPTH	Specifies the depth of object inspection
DEBUG_FD	Writes debug output to use a file descriptor
DEBUG_SHOW_HIDDEN	Reveals hidden characteristics of examined objects

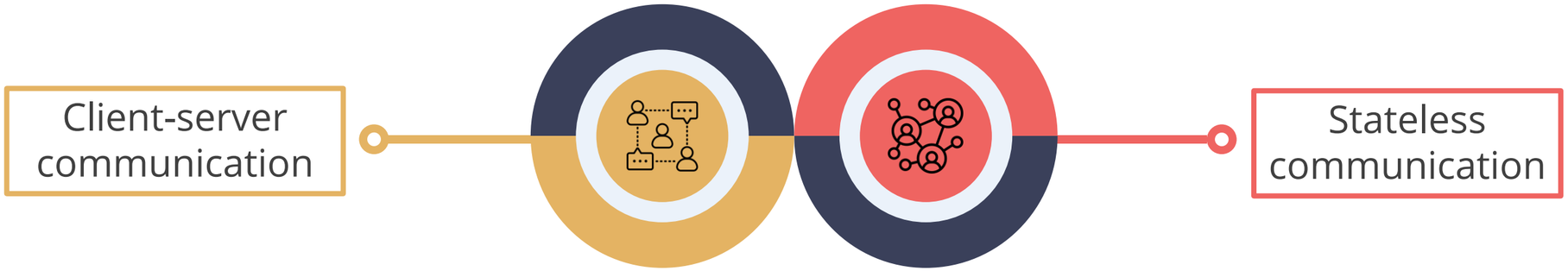
Advanced Options

Environment variables starting with DEBUG_ are transformed into an options object when used with %o/%O formatters.



REST API: What Is It?

Representational state transfer is referred to as REST.



It is typically used in conjunction with the HTTP protocol.

REST API: What Is It?

The following operations are performed by RESTful apps using HTTP requests:

All four CRUD activities in REST employ HTTP.

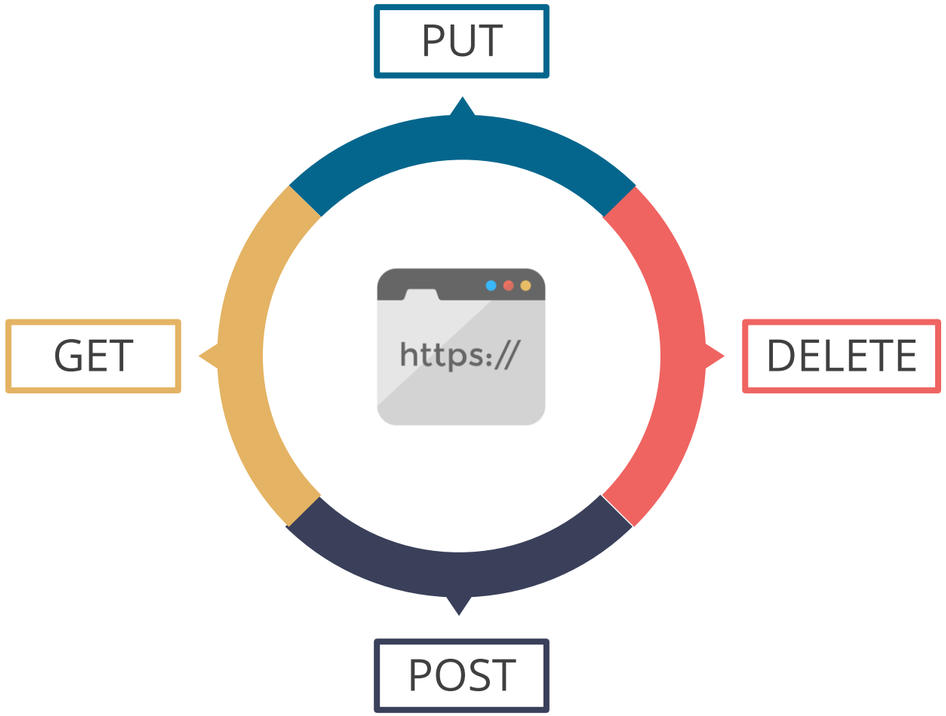
Describe JSON

JSON is a simple format for exchanging data.

Machines can easily parse and generate it.

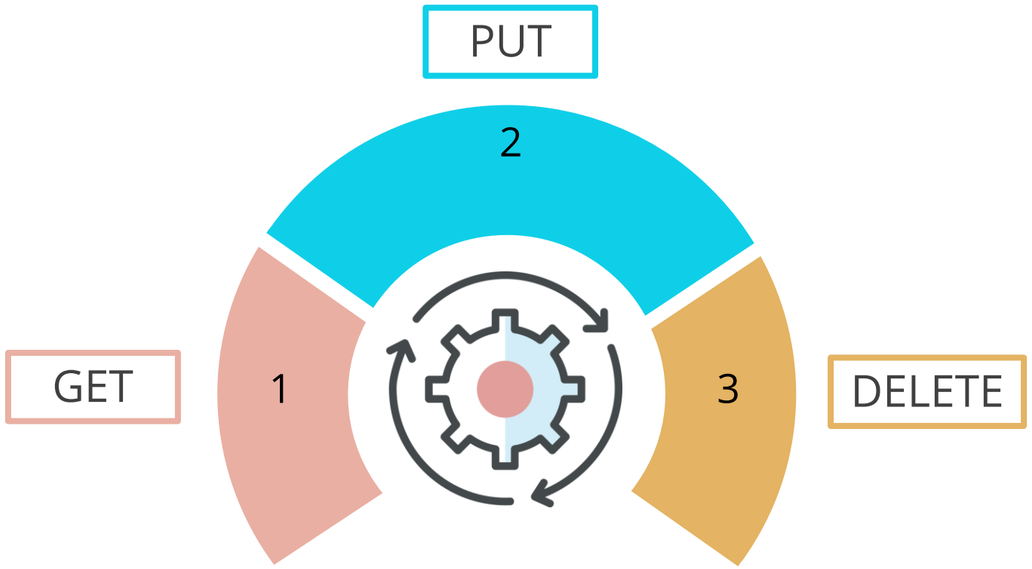
Protocols

HTTP enables the use of a variety of protocols for client-server communication.



Protocols

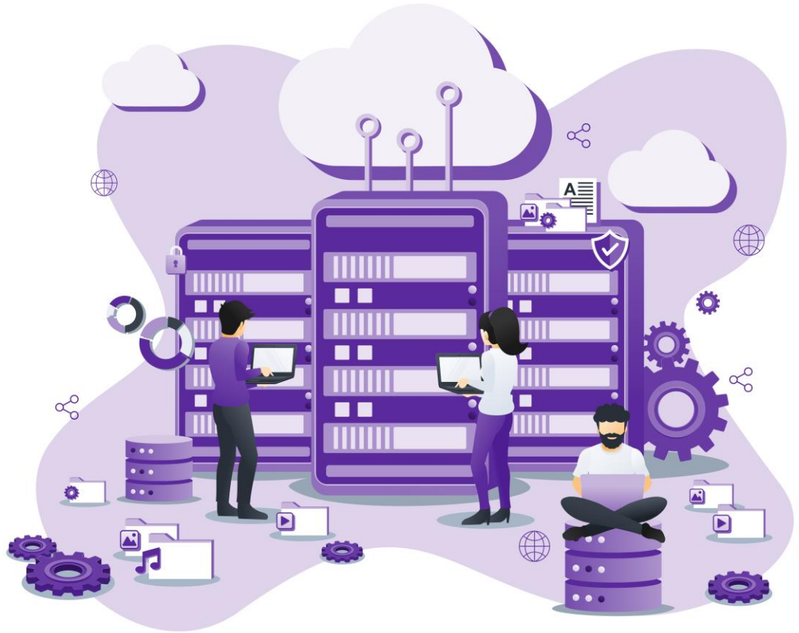
Implementing these given operations as idempotent methods is recommended.



POST shouldn't have idempotence.

GET

A query of some kind is carried out via GET.



It makes no demands to alter the system's current state in any way.

GET

This does not indicate that the server's status is not changing.

It indicates that the client did not make a request.

POST

The request to establish a new entity is known as POST.

The request body should contain the content of that object.

PUT

The following operations are similar:

PUT should either create a new object if one doesn't already exist or change an existing one.

DELETE

DELETE is a command that asks the server to remove a certain object.

Working with Express.js Framework

Problem Statement:

Duration: 30 min.

You have been assigned a task to demonstrate the working of Express.js framework.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Perform routing in Express.js
2. Build URL in Express.js
3. Templatize in Express.js
4. Create StaticFiles in Express.js
5. Handle Error States in Express.js
6. Debug in Express.js Module

Key Takeaways

- Express.js is a lightweight framework that enhances Node.js web server capabilities by streamlining existing APIs.
- Using the Node.js ecosystem and the built-in command-line interfaces (CLIs), repetitive processes can be automated.
- Routing describes how URI endpoints in an application respond to client requests.
- Express.js Routers can define several routes or middleware to handle various incoming requests from clients.

Thank You