

# UNIT-II

## 1. Introduction to Express

**Ans:** Express is a framework for Node.js that provides additional features to build web applications. It is built on top of Connect, which is another Node.js framework, and uses its middleware architecture. Express allows developers to include modular HTML template engines, extend the response object to support various data format outputs, implement a routing system, and much more. By using Express, you can structure your project in a better way, configure your application properly, and break your application logic into different modules. Express also offers features like managing sessions and a routing scheme that can simplify your web application development process.

## 2. Installing express.js

**Ans:** When developing a Node.js application, you will likely use many third-party modules to help with different functionalities. Manually installing all of these modules can become tedious and difficult to manage as your application grows.

**Traditional method:** Create package.json file that includes a name, version, and a dependencies property that specifies that the Express module should be installed, when you run the **npm install** command in your application's directory, npm will read the package.json file and install all of the listed dependencies.

**Ex:**

```
{
  "name" : "MEAN",
  "version" : "0.0.3",
  "dependencies" : { "express" : "4.14.0"}
}
```

**Modern method:** To address this, Node.js provides a package manager called npm, which allows you to create a package.json file that lists your application's dependencies. This file serves as a manifest for your application, making it easy for you or other developers to install and manage the dependencies. Run **npm install express** in the command prompt.

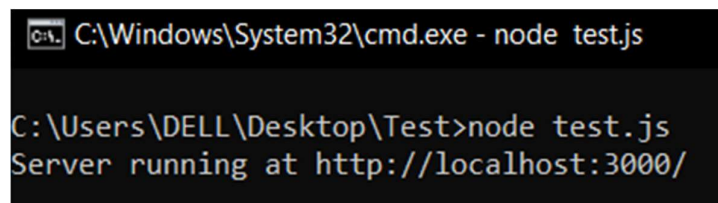
## 3. Creating your first Express application

**Ans:**

**Program:**

```
const express = require('express');
const app = express();
app.use('/', (req, res) => {
  res.send('Hello World');
});
app.listen(3000);
console.log('Server running at http://localhost:3000/');
module.exports = app;
```

**Output:**



```
C:\Windows\System32\cmd.exe - node test.js

C:\Users\DELL\Desktop\Test>node test.js
Server running at http://localhost:3000/
```



#### 4. request and response objects

**Ans:** In Express, when a request is made to the server, a request object and a response object are created and passed to the appropriate route handler.

##### The request Object:

It contains all of the information about the incoming request, including the URL, query parameters, request headers, and more. The following are some of the key properties and methods:

- **req.query:** This is a property that contains the parsed query-string parameters.
- **req.params:** This is a property that contains the parsed routing parameters.
- **req.body:** This is a property that's used to retrieve the parsed request body.
- **req.path:** It is used to retrieve the current request path.
- **req.cookies:** This is a property used in conjunction with the `cookieParser()` middleware to retrieve the cookies sent by the user agent.

##### The response Object:

It contains all of the information that will be sent back to the client in response to the request. The following are some of the key methods:

- **res.status(code):** sets the HTTP status code of the response.
- **res.status([status]).send([body]):** sends a non-streaming response with the specified status code and optional body.
- **res.status([status]).json([body]):** sends a JSON response with the specified status code and optional body.
- **res.set(field, [value]):** sets the value of a specified HTTP header field.
- **res.cookie(name, value, [options]):** sets a cookie in the response with the given name and value.
- **res.redirect([status], url):** redirects the client to a different URL.
- **res.render(view, [locals], callback):** renders a view using an HTML template engine and sends the result as an HTML response.

#### 5. External middleware

**Ans:** In Express, you can use external middleware functions to add functionality to your application. External middleware is a separate module that you can install and require in your Express application. These middleware extend Express to provide a better framework support and can help you shorten your development time. Some of the popular middleware are:

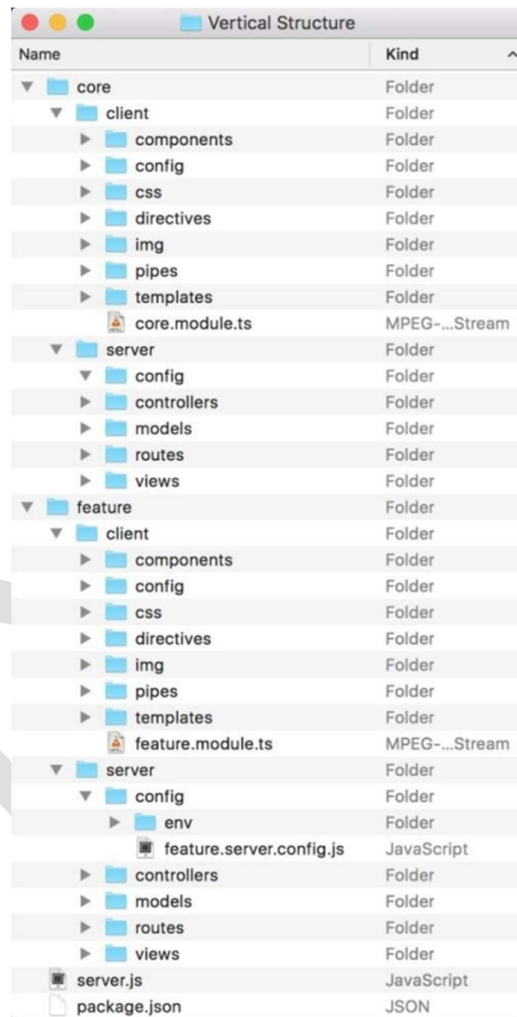
- **morgan:** It's a logger middleware that logs HTTP requests.
- **body-parser:** It's a middleware used to parse the request body.
- **method-override:** It's a middleware that provides HTTP verb support like PUT or DELETE, in places where the client doesn't support it.
- **compression:** It's a middleware used to compress the response data using GZIP/deflate.
- **express.static:** It's a middleware used to serve static files.
- **cookie-parser:** It's a middleware used to parse cookies and populates the `req.cookies` object.
- **Session:** It's a middleware used to support persistent sessions.

#### 6. The application folder structure

**Ans:** When building an Express application, it's important to organize your project files into logical units of code. JavaScript and Express are agnostic about the structure of your application, which means you can place all your code in a single file. However, it's recommended to organize your project to make it easier to manage as it grows in complexity. There are two major approaches to project structure: a horizontal structure for smaller projects and a vertical structure for feature-rich applications. A horizontal structure is simpler and easier to manage, while a vertical structure is more complex and better suited for larger applications with many features and a bigger team working on the project. It's important to choose the appropriate structure based on the estimated complexity of your application.

## 7. Vertical folder structure.

**Ans:**



The advantage of a vertical folder structure is that it allows for better organization and easier navigation of the application code. It also makes it easier to work on specific features or modules without having to wade through unrelated code. This approach is especially useful for larger applications with many features and a bigger team working on the project. However, it can be more complex to set up and maintain, and it may not be necessary for smaller projects.

## 8. File-naming conventions

**Ans:** When developing a MEAN application, you may end up with many files having the same name. This is because MEAN applications often have a parallel MVC structure for both the Express and Angular components. To address this issue, a recommended solution is to use a naming convention that adds the functional role and execution destination to the filename. For example, a feature controller file can be named `feature.controller.js`, a feature model file can be named `feature.model.js`, and so on. This

helps in quickly identifying the role and execution destination of your application files. However, it is important to note that this is just a best practice convention and you can replace the keywords with your own.

## 9. Implementing the horizontal folder structure

**Ans:** To implement the horizontal folder structure, create a new folder and make some subfolders inside it. Then, create a package.json file in the root folder and include the Express module as a dependency.

**Ex:**

```
{
  "name" : "MEAN",
  "version" : "0.0.3",
  "dependencies" : { "express" : "4.14.0" }
}
```



In the app/controllers folder, create a file called index.server.controller.js and add some code to it. This code defines a function called render() that sends a "Hello World" message when called. This file is an example of a controller, which is a module that defines the logic for handling HTTP requests. To use this controller, you will need to use an Express routing feature.

## 10. Handling request routing

**Ans:** When a client makes an HTTP request to an Express server, the server needs to know what to do with that request. Routing is the process of defining how an application responds to a client request to a particular endpoint or URL. In other words, routing is determining the code that should be executed when the client requests a particular URL. Express provides two methods to handle routing. The first method is to use the **app.VERB(path, callback)** method, where VERB is an HTTP verb such as GET, POST, PUT, or DELETE. For example, if the client sends a GET request to the root path "/", Express will execute the middleware function associated with that route, which might send back a response to the client. The second method is to use the **app.route(path).VERB(callback)** method, which allows you to chain multiple HTTP verbs for a single route. This method can make your code more concise and easier to read. Express also allows you to chain multiple middleware functions for a single route. Middleware functions are functions that are executed before the final request handler is called. Middleware functions can be used to validate requests, authenticate users, or perform other tasks before sending a response to the client. The order in which middleware functions are called can be important, as some middleware functions may need to be executed before others.

## 11. Adding the routing file

**Ans:** Suppose you have an Express app with a single route that serves a simple "Hello World!" message when you visit the root URL ("/"). Now let's say you want to add more routes to your app, but you don't want to clutter the main server.js file with too much code. You can create a separate file for handling the routes and then mount that file onto the main Express app using the app.use() method.

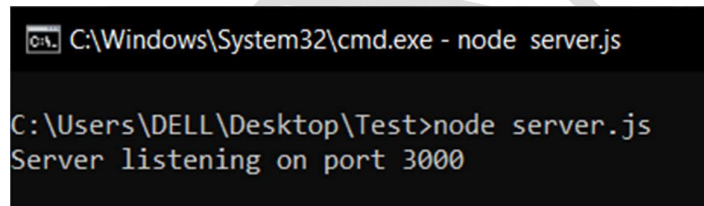
**Program:**

```
//server.js
```

```
const express = require("express");
const app = express();
app.get("/", (req, res) => {
  res.send("Hello World!");
});
const routes = require("./routes");
app.use("/", routes);
app.listen(3000, () => {
  console.log("Server listening on port 3000");
});
```

```
//routes.js
const express = require("express");
const router = express.Router();
router.get("/about", (req, res) => {
  res.send("This is the about page!");
});
router.get("/contact", (req, res) => {
  res.send("This is the contact page!");
});
module.exports = router;
```

### Output:



```
C:\Windows\System32\cmd.exe - node server.js

C:\Users\DELL\Desktop\Test>node server.js
Server listening on port 3000
```



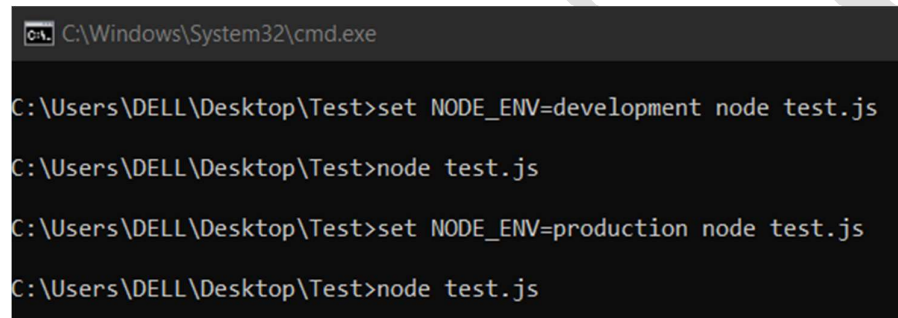
## 12. Configuring an Express application

**Ans:** Express has a configuration system that allows you to add functionalities to your application. You can customize it by changing the predefined configuration options or adding your own. You can also configure your application based on the environment it's running on, such as development or production, using the `process.env` property. The `NODE_ENV` environment variable is commonly used for environment-specific configurations. By using external middleware, you can further enhance your application's functionality, but you need to download and install them as dependencies first.

**Program:**

```
const express = require('express');
const morgan = require('morgan');
const compression = require('compression');
module.exports = function() {
  const app = express();
  if (process.env.NODE_ENV === 'development') {
    app.use(morgan('dev'));
  } else if (process.env.NODE_ENV === 'production') {
    app.use(compression());
  }
  return app;
};
```

**Output:**



```
C:\Windows\System32\cmd.exe

C:\Users\DELL\Desktop\Test>set NODE_ENV=development node test.js

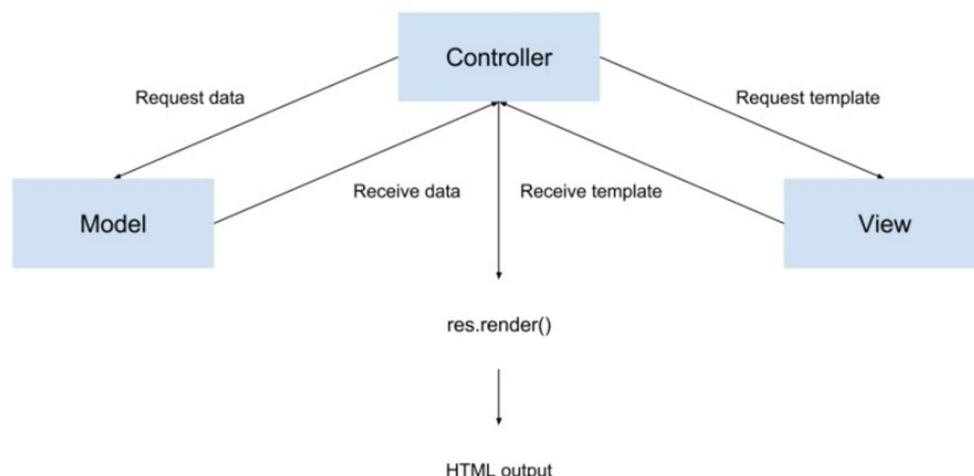
C:\Users\DELL\Desktop\Test>node test.js

C:\Users\DELL\Desktop\Test>set NODE_ENV=production node test.js

C:\Users\DELL\Desktop\Test>node test.js
```

### 13. Rendering views

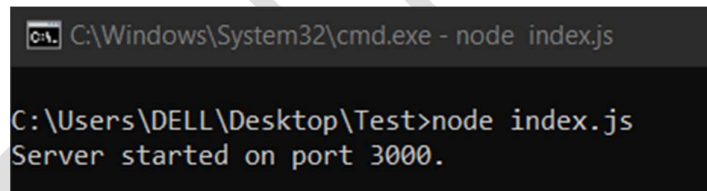
**Ans:** Rendering views is a common feature in web frameworks where data is passed to a template engine that renders the final view in HTML. In the MVC pattern, the controller uses the model to get the data and the view template to render the HTML output. Express allows the usage of various Node.js template engines like EJS template engine. Express has two methods to render views - `app.render()` and `res.render()`. The `app.render()` method renders the view and passes the HTML to a callback function, while the more commonly used `res.render()` method renders the view locally and sends the HTML as a response.



**Program:**

```
//index.js
const express = require('express');
const app = express();
app.set('view engine', 'ejs');
app.get('/', (req, res) => {
  const data = { title: 'Welcome to my website!', message: 'This is a
sample message.' };
  res.render('index', data);
});
app.listen(3000, () => {
  console.log('Server started on port 3000.');
```

```
// ./views/index.ejs
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
  </head>
  <body>
    <h1><%= message %></h1>
  </body>
</html>
```

**Output:**

```
C:\Windows\System32\cmd.exe - node index.js

C:\Users\DELL\Desktop\Test>node index.js
Server started on port 3000.
```

**14. Serving static files**

**Ans:** In a web application, it is common to have static files like images, CSS, and JavaScript that need to be served to the client. Express has a built-in middleware called `express.static()` that can be used to serve static files. To use this middleware, you need to add it to your application by including the line `app.use(express.static('./public'))` in your code. This tells Express to serve the static files located in the public folder. It is important to note that the order of middleware is important. The `express.static()` middleware should be placed after the routing middleware because if it is placed before, Express will first try to look for HTTP request paths in the static files folder, which can slow down the response.

**Program:**

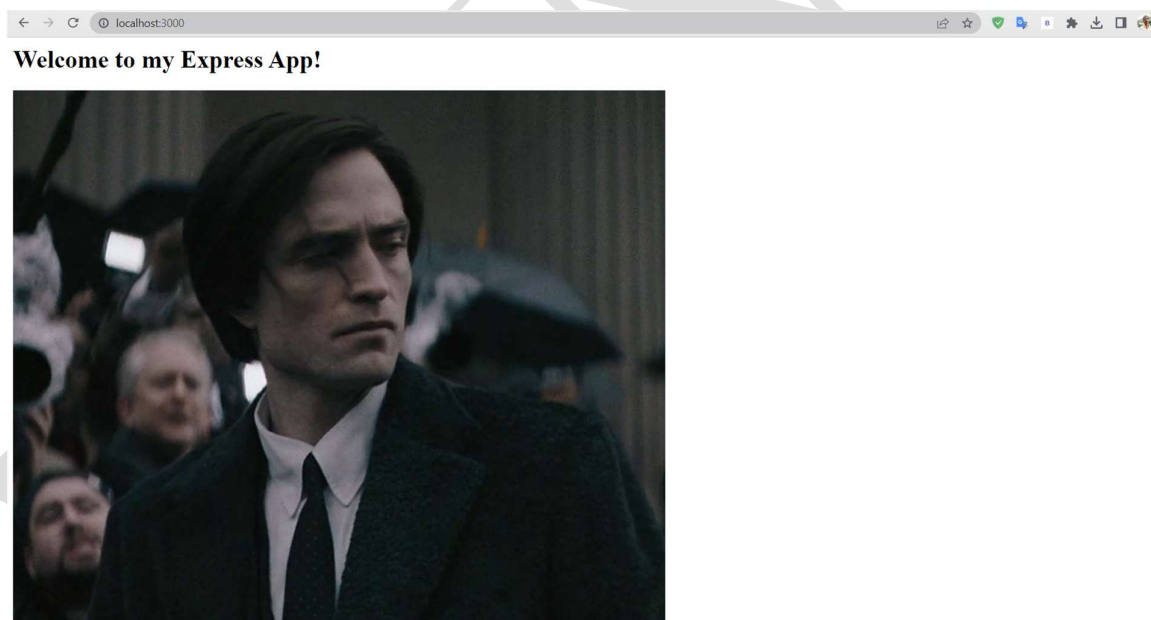
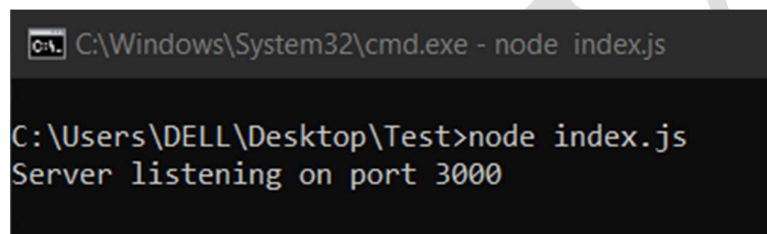
```
// index.js
const express = require('express');
const app = express();
app.use(express.static('public'));
app.listen(3000, () => {
```



```
    console.log('Server listening on port 3000');
  });

// ./public/index.html
<!DOCTYPE html>
<html>
  <head>
    <title>My Express App</title>
  </head>
  <body>
    <h1>Welcome to my Express App!</h1>
    
  </body>
</html>
```

### Output:



### 15. Configuring sessions

**Ans:** Sessions are an important feature for web applications, as they allow you to store user-specific data across requests. Express provides session middleware that you can use to create and manage sessions.

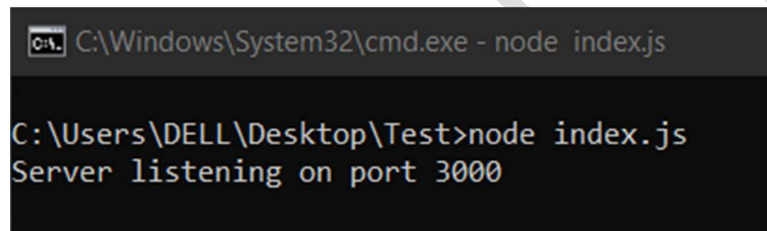
#### Program:

```
const express = require('express');
const session = require('express-session');
const app = express();
app.use(session({
  secret: 'your-secret-key',
```



```
    resave: false,
    saveUninitialized: true
  }));
app.get('/login', (req, res) => {
  req.session.username = 'john';
  res.send('Logged in');
});
app.get('/dashboard', (req, res) => {
  const username = req.session.username;
  if (!username) {
    res.redirect('/login');
  } else {
    res.send(`Welcome back, ${username}!`);
  }
});
app.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

**Output:**

A terminal window with a dark background. The title bar reads "C:\Windows\System32\cmd.exe - node index.js". The command prompt shows "C:\Users\DELL\Desktop\Test>node index.js" and the output is "Server listening on port 3000".

```
C:\Windows\System32\cmd.exe - node index.js

C:\Users\DELL\Desktop\Test>node index.js
Server listening on port 3000
```

