ADITYA ENGINEERING COLLEGE (A)

Express JS

By

Dept of Computer Science and Engineering

Aditya Engineering College(A)

Surampalem.

Express.js is a small framework that works on top of Node.js web server functionality to simplify its APIs and add helpful new features. It makes it easier to organize your application's functionality with middleware and routing. It adds helpful utilities to Node.js HTTP objects and facilitates the rendering of dynamic HTTP objects.

Why Express ?

- Develops Node.js web applications quickly and easily.
- It's simple to set up and personalise.
- Allows you to define application routes using HTTP methods and URLs.
- Includes several middleware modules that can be used to execute additional requests and responses activities.
- Simple to interface with a variety of template engines, including Jade, Vash, and EJS.
- Allows you to specify a middleware for handling errors.

Installing Express:

We can install it with npm. Make sure that you have Node.js and npm installed.

Step 1: Creating a directory for our project and make that our working directory.

$ mkdir gfg

$ cd gfg

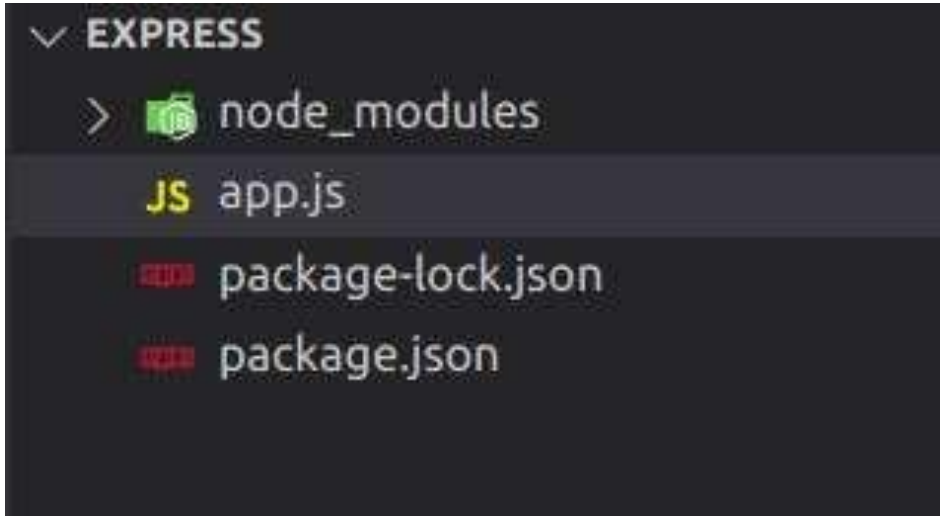
Step 2: Using npm init command to create a package.json file for our project.

$ npm init

Step 3: Now in your *gfg(name of your folder)* folder type the following command line:

$ npm install express --save

Project Structure: It will look like the following.



App.js:

```
var express = require('express');
var app  =  express();
app.get('/', function (req, res) {
res.send("This is Demo of express!");
});
app.listen(5000);
```

Step to run the application: Start the app by typing following command.

Node app.js

What is Routing?

Defining the endpoints of an application and the way the application responds to the incoming client requests are referred to as routing.

A route is nothing but a mixture of a URI, HTTP request method, and some handlers for that particular path.

Routing can be defined using two ways

- Application Instance
- Router class of Express

Method 1:

We can do routing using the application object. Look at the below code snippet which does the routing using the application object.

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

const app = express();

app.get('/', myController.myMethod);

app.get('/about', myController.aboutMethod);
```

Method 2

Let's use the **express.Router** class. Router class helps in grouping the route handlers together for a site and allows them to access them using a common route-prefix.

The below code explains how to define routes in an Express application.

```
const express =  require('express');
const router = express.Router();
router.get('/', myController.myMethod);
router.get('/about', myController.aboutMethod);
module.exports = router;
```

Now to use the route module created above, we need to import it and then associate it with the application object. In **app.js** file include the below code:

```
const express = require('express');
const router = require('./routes/route');
const app = express();
app.use('/', router);
```

Defining a route:

A route can be defined as shown below

router.method(path,handler)

router: express instance or router instance

method: one of the HTTP verbs

path: is the route where request runs

handler: is the callback function that gets triggered whenever a request comes to a particular path for a matching request type

Route Method:

The application object has different methods corresponding to each of the HTTP verbs (GET, POST, PUT, DELETE). These methods are used to receive HTTP requests.

Below are the commonly used route methods and their description:

Method	Description
get()	Use this method for getting data from the server.
post()	Use this method for posting data to a server.
put()	Use this method for updating data in the server.
delete()	Use this method to delete data from the server.
all()	This method acts like any of the above methods. This can be used to handle all requests.

```
routing.get("/notes", notesController.getNotes);
routing.post("/notes", notesController.newNotes);
routing.all("*", notesController.invalid);
```

notesController is the custom js file created to pass the navigation to this controller file. Inside the controller, we can create methods like getNotes, newNotes, updateNotes

Handling Routes

Route handler can be defined as functions that get executed every time the server receives a request for a particular URL path and HTTP method.

The route handler function needs at least two parameters: **request object** and **response object**.

Request object.

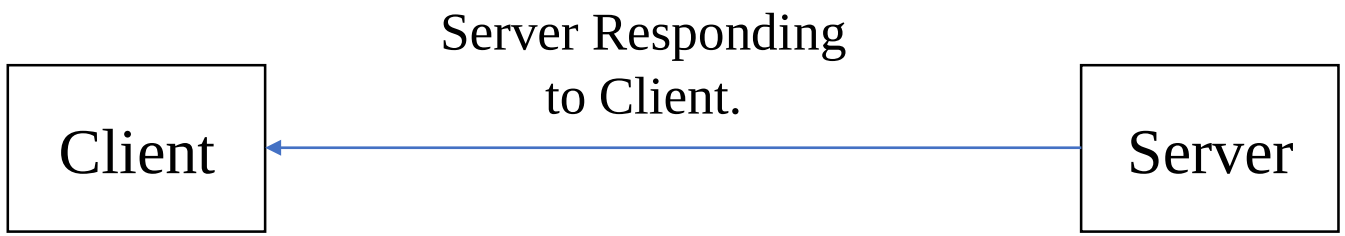
The HTTP request object is created when a client makes a request to the server. The variable named **req** is used to represent this object.

The request object in Express provides many properties and methods to access the request related information sent by the client to the server. Below are some commonly used methods of the request object.

Property	Description
req.app	Provides the Express application instance's reference.
req.body	It contains the data sent as part of the body in HTTP post request as key-value pairs.
req.cookies	contains the information about the cookie sent with the request.
req.hostname	The host of the request HTTP header.
req.ip	The remote IP address of the request.
req.params	It contains the parameter send along with the request URL.
req.path	Provides the path from request URL.

Response object

The HTTP response object has information about the response sent from the server to the client. The response object is created along with the request object and is commonly represented by a variable named **res**.



The route handler function sends a response using res.send() method whenever a GET request is received.

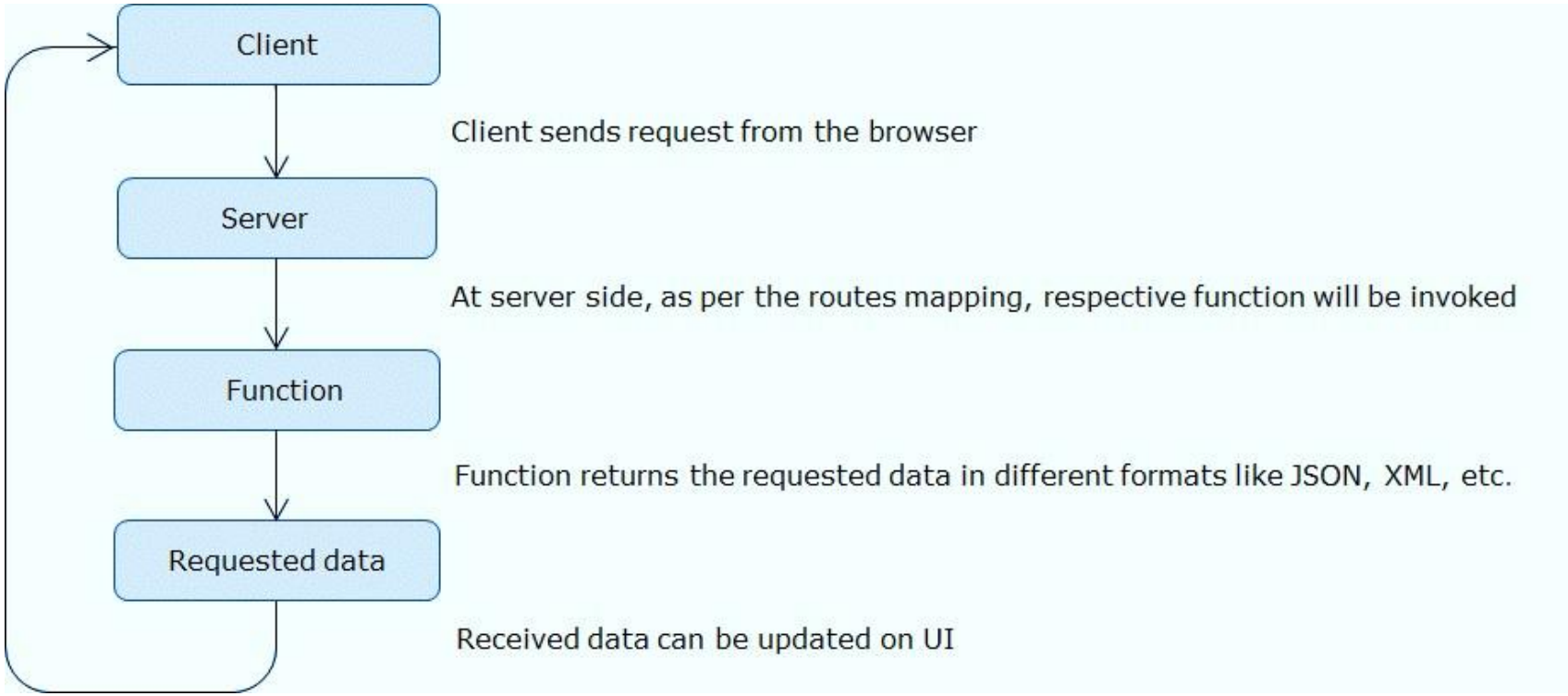
```
res.send('About us page');
```

The response object in Express also supports many methods for sending different types of responses.

Have a look at some commonly used methods of the response object.

Property	Description
res.download()	To download a file.
res.end()	To end the process of response.
res.json()	To send response in JSON format.
res.jsonp()	To send response in JSON format with JSONP support.
res.redirect()	To redirect a request to different URL.
res.render()	To render response through a view template.
res.send()	To send any type of response.
res.sendFile	To send a file as response.
res.sendStatus()	To set the status code of response

Basic architecture of Express routing:



Whenever a client sends a request to the server, that request is forwarded to the corresponding route handler function. The function process the request and then generates the response that is to be sent back to the client. The client will be able to see the response in the browser.

Route Parameters

Route parameters are useful for capturing the values indicated at certain positions in the URL. These are defined as named URL segments.

For example, in the URL, *'http://localhost:5000/user/Aditya'*, if we want to capture the portion of URL which contains "Smith", then

we can configure the route parameter as below:

router.get('/user/:username',myController.getMethod);

Here inside the getMethod of Controller, username is the route parameter. We access it using the below syntax.

req.params < parameter_name >

```
exports.getMethod = async (req, res) => {
const name = req.params.username;
res.send(`Welcome ${name}`);
};
```

If more than one parameter is passed as part of the request URL, then the required information can be extracted as shown below.

```
router.get('/user/:username/:id',myController.getMethod)

exports.getMethod = async (req, res) => {
const username = req.params.username;
res.send(`Welcome ${username} with id ${req.params.id}`);
};
```

Why Middleware?

Consider the scenario where the following tasks are to be designed in our application.

- Authenticate or authorize requests
- Log the requests
- Parse the request body
- End a request-response cycle

The above jobs are not the core concerns of an application. But they are the cross-cutting concerns that are applicable to the entire application.

In the Express framework, these cross-cutting concerns can be implemented using **middleware**.

The middleware concept helps to keep the router clean by moving all the logic into corresponding external modules. The middleware is a great concept for organizing the code.

What is a Middleware?

A middleware can be defined as a function for implementing different cross-cutting concerns such as authentication, logging, etc.

The main arguments of a middleware function are the **request** object, **response** object, and the **next** middleware function defined in the application.

A function defined as a middleware can execute any task mentioned below:

- Any code execution.
- Modification of objects - request and response.
- Call the next middleware function.
- End the cycle of request and response.

Middleware functions are basically used to perform tasks such as parsing of requests body, cookie parsing for cookie handling, or building JavaScript modules.

How Middleware works?

In order to understand how middleware works, let us take a scenario where we want to log the request method and request URL before the handler executes. The route definition for which we want to add the middleware.

```
app.get('/login', myController.myMethod);
exports.myMethod = async (req, res, next) => {
res.send('/login');
};
```

The arguments of this function are:

- **req**: an object containing all the information about the request.
- **res**: an object containing all the information about the response sent from server to client.
- **next**: tells Express when the middleware is done with the execution.

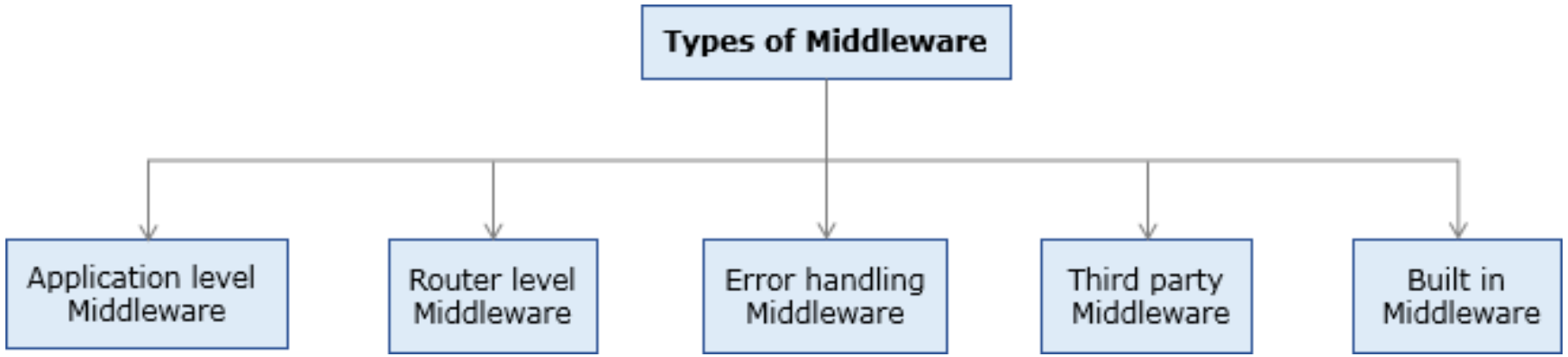
Inside the function, we have logic to log the request method and request URL along with the date. The **next()** method ensures that after the execution of middleware logic, the handler is executed. Now we can modify the route definition to add the middleware using **app.use()** method.

Chaining of Middleware

We can create a chain of middlewares before the request reaches the handler.
Consider a middleware which logs the request time and another middleware which logs the request URL as shown.

```
const logtime = async (req, res, next) =>{
  console.log('Request received at  ' + Date.now());
  next();
};
const logURL = async (req, res, next) =>{
  console.log('Request URL is ' + req.url);
  next();
};
app.use(logtime);
app.use(logURL);
```

Types of Middlewares



Connecting to MongoDB with Mongoose – Introduction

Organizations use various databases to store data and to perform various operations on the data based on the requirements.

Some of the most popular databases are:

- Cassandra
- MySQL
- MongoDB
- Oracle
- Redis
- SQL Server

A user can interact with a database in the following ways.

- Using query language of the respective databases' - eg: SQL
- Using an ODM(Object Data Model) or ORM(Object-Relational Model)

Introduction to Mongoose Schema

A schema defines document properties through an object, where the key name corresponds to the property name in the collection. The schema allows you to define the fields stored in each document along with their validation requirements and default values.

```
const mongoose = require('mongoose');
const schema = new mongoose.Schema({ property_1: Number,
property_2: String });
```

In a schema, we will define the data types of the properties in the document.
The most used types in the schema are:

- String : To store string values. For e.g: employee name
- Number: To store numerical values. For e.g: employee Id
- Date : To store dates. The date will be stored in the ISO date format.
- For e.g. 2018-11-15T15:22:00.
- Boolean

It will take the values true or false and is generally used for performing validations. We will discuss validations in the next resource.

- ObjectId

ObjectId is assigned by MongoDB itself. Every document inserted in MongoDB will have a unique id which is created automatically by MongoDB, and this is of type ObjectId.

- Array

To store an array of data, or even a sub-document array. For e.g: ["Cricket","Football"]

Validation through mongoose validator

Mongoose provides a way to validate data before you save that data to a database. Data validation is important to make sure that "**invalid**" data does not get persisted in your application. This ensures data integrity. A benefit of using Mongoose, when inserting data into MongoDB is its built-in support for data types, and the automatic validation of data when it is persisted.

Mongoose's validators are easy to configure. When defining the schema, specific validations need to be configured.

The following rules should be kept in mind while adding validation:

- Validations are defined in the Schema
- Validation occurs when a document attempts to be saved
- Validation will not be applied for default values
- Validators will not be triggered on undefined values. The only exception to this is the required validator
- Customized validators can be configured
- Mongoose has several built-in validators
- The **required** validator can be added to all SchemaTypes
- Number schema type has **min** and **max** validators
- Strings have **enum** and **match** validators

Models

Creating a model:

To use our schema definition, we need to wrap the **Schema** into a **Model** object we can work with.
The model provides an object which provides access to query documents in a named collection. Schemas are compiled into models using the **model()** method.

const Model = mongoose.model(name , schema)

The first argument is the singular name of the collection for which you are creating a Model.
The model() function makes a copy of the schema. Make sure that you have added everything you want to schema before calling the model().
Let us now create a model for our **myNotes** collection using the below code.

const NotesModel = mongoose.model("mynotes", myNotesSchema);

Here **NotesModel** is a collection object that will be used to perform CRUD operations.

Insert Document - Demo

Mongoose library offers several functions to perform various CRUD (Create-Read-Update-Delete) operations.

Inserting a document into the collection

To insert a single document to MongoDB, use the create() method. It will take the document instance as a parameter. Insertion happens asynchronously and any operations dependent on the inserted document must happen by unwrapping the promise response.

```
exports.newNotes = async (req, res) => {
  try {
    const noteObj = {
      notesID: 7558,
      name: 'Mathan',
      data: 'Mongo Atlas is very easy to configure and use.',
    };
    const newNotes = await NotesModel.create(noteObj);
    console.log(newNotes);
  } catch (err) {
    console(err.errmsg);
  }
```

CRUD Operations - Read

Documents can be retrieved through find, findOne, and findById methods.

Let us now retrieve all the documents that we have inserted into our myNotes Collection.

```
exports.getNotes = async (req, res) => {
  try {
    const notes = await NotesModel.find({}, { _id: 0,__v: 0 });
    if (notes.length > 0) {
      console.log(notes);
    }
  } catch (err) {
    console.log(err.errmsg);
  }
};
```

In line 1, a new async function getNotes is created with request and response objects as parameters.

In line 3, we are referring to the same NotesModel which we had created earlier and made use of the find() method to fetch the document(s) from the collection. The retrieved documents will be available in the const notes.

In line 4, we are ensuring the data by checking the length and displaying the returned data in the console.

Try catch block will ensure all the exceptions are handled in the code.

CRUD Operations - Update

```
exports.updateNotes = async (req, res) => {
  try {
    const noteObj = {
      name: 'Mathan',
      data: 'Updated notes',
    };
    const notes = await NotesModel.findOneAndUpdate(
      { notesID: 7555 },
      noteObj,
      {
        new: true, //to return new doc back
        runValidators: true, //to run the validators which specified in the
model
      }
    );
    if (notes != null) {
      console.log(notes);
    }
  } catch (err) {
    console.log(err.errmsg);
  }
};
```

In line 1, a new async function updateNotes is created with request and response objects as parameters.

In line 3, a new array of objects is created with the name noteObj with all the necessary keys that need to be updated.

In line 8, we are referring to the same Notes Model which we had created earlier and made use of the **findOneAndUpdate()** method to find and update one document in the collection. It will update the document in the collection based on the schema and will return the document which got updated to the const notes.

CRUD Operations - Delete

```
exports.deleteNotes = async (req, res, err) => {
  const delDet = await NotesModel.deleteOne({ notesID:
7555 });
  console.log(delDet);
};
```

In line 1, a new async function deleteNotes is created with the request, response, and error objects as parameters.
In line 2, we are referring to the same NotesModel which we had created earlier and made use of **deleteOne()** method to delete the document from the collection based on the condition. It will delete the document from the collection based on the condition and will return the details about the delete operation performed, which is initialized to the const delDet.
In line 3, on the successful deletion of the document, you will get the following output in the console.

API Development – https:

An application should be protected by HTTPS, even if it's not handling sensitive information for communications.

- The integrity of the application will be protected by HTTPS.
- The privacy and security of the data will be protected by HTTPS.

Generate the SSH keys in the server where the application will be running.

Configure the key and the certification and create the HTTPS server as shown in the below code:

```
const express = require('express');
const fs = require('fs');
const https = require('https');
const app = express();
app.get('/', function (req, res) {
res.send('Welcome to HTTPS');
});
https
  .createServer(
    {
      key: fs.readFile('sshServer.key'),
      cert: fs.readFile('sshServer.cert'),
    },
    app
  )
  .listen(3000, function () {
    console.log('Server running on
HTTPS');
  });
```

The application will be accessible using the URL, https://localhost:3000.

Why Session management?

Every user interaction with an application is an individual request and response. The need to persist information between requests is important for maintaining the ultimate experience for the user for any web application.

Session management is useful in the scenarios mentioned below:

- We need to maintain that a user has already been authenticated with the application.
- To retain various personalized user information that is associated with a session.

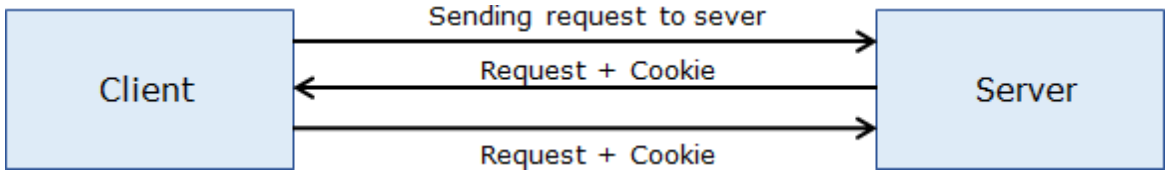
Let us now understand how we can set up sessions securely in our application to lessen risks like session hijacking.

Session Management is a technique used by the webserver to store a particular user's session information. The Express framework provides a consistent interface to work with the session-related data. The framework provides two ways of implementing sessions:

- By using cookies
- By using the session store

Cookies

Cookies are a piece of information sent from a website server and stored in the user's web browser when the user browses that website. Every time the user loads that website back, the browser sends that stored data back to the website server, to recognize the user.



Configuration

- The Express framework provides a **cookie API** using **cookieParser** middleware.
  The middleware **cookieParser** parses the cookies which are attached to the request object.
- To install the cookieParser middleware, issue the below command in the Node command prompt.

npm install cookie-parser

The above middleware can be configured in Express application as shown below:

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

The cookie-parser middleware is imported and then associated with the application object using the app.use() method.

Sessions

A session store can be used to store the session data in the back-end. Using this approach, a large amount of data can be stored, and also the data will be hidden from the user, unlike a cookie.

Configuration

Express provides a middleware called **express-session** for storing session data on the server-side.

To install **express-session** middleware, use npm.

```
npm install express-session
```

To use this module in the Express application, first, import it as shown below:

```
const session = require('express-session');
```

Why Security?

Attackers may attempt to exploit security vulnerabilities present in web applications in various ways. They may plan attacks like clickjacking, cross-site scripting, placing insecure requests and identifying web application frameworks etc.

It is important to follow secure coding techniques for developing secure, robust and hack-resilient applications.

What is Security?

Security is basically about protecting the application assets like the web page, the database, and so on.

Security depends on the following elements:

- **Authentication**: the process of exclusively identifying the clients of your applications and services.
- **Authorization**: a process that manages the resources and operations that the authenticated user is permitted to access.
- **Auditing**: process to assure that a user cannot deny performing an operation or initiating a transaction.
- **Confidentiality**: the process of making sure that data remains private and confidential.
- **Integrity**: the promise that data is protected from malicious modification.
- **Availability**: means that systems remain available for legitimate users.

Helmet Middleware

Express applications can be secured by using a middleware called helmet.

The helmet middleware is a set of 14 small middleware functions that help in setting up security-related HTTP headers with default values and also removes unnecessary headers which expose the application related infor n csp xpress application:

Below are the set of midd

Middleware	Description
crossdomain	Adds the Content-Security-Policy header which prevents attacks like cross-site scripting.
dnsPrefetchControl	Sets X-Permitted-Cross-Domain-Policies header. It prevents Adobe Flash and Adobe Acrobat from loading content on site.
expectCt	Sets X-DNS-Prefetch-Control header.
featurePolicy	Sets Expect-CT header to handle certificate Transparency.
hidePoweredBy	Sets Feature-Policy header. It can restrict certain browser features like fullscreen, notifications, etc.
hpkp	Removes the “X-Powered-By” header which exposes the information that application is run with Express server.
hsts	Sets Public Key Pinning headers which prevents attacks like man in the middle (MIM).
ieNoOpen	Sets Strict-Transport-Security header which enforces usage of HTTPS for connecting to the server.
noCache	Sets the X-Download-Options header which prevents Internet Explorer browser users from downloads.
noSniff	Enables Cache-Control, Pragma, Expires, and Surrogate-Control headers which prevents client caching site resources, which are old.
frameguard	Sets X-Content-Type-Options.
xssFilter	Sets the “X-Frame-Options” header which blocks attacks like clickjacking which prevents the web page to be accessed on other sites.
referrerPolicy	Sets the Cross-site scripting (XSS) filter latest web browsers.
	Sets Referrer-Policy header.

Middleware	Description
csp	Adds the Content-Security-Policy header which prevents attacks like cross-site scripting.
crossdomain	Sets X-Permitted-Cross-Domain-Policies header. It prevents Adobe Flash and Adobe Acrobat from loading content on site.
dnsPrefetchControl	Sets X-DNS-Prefetch-Control header.
expectCt	Sets Expect-CT header to handle certificate Transparency.
featurePolicy	Sets Feature-Policy header. It can restrict certain browser features like fullscreen, notifications, etc.
hidePoweredBy	Removes the “X-Powered-By” header which exposes the information that application is run with Express server.
hpkp	Sets Public Key Pinning headers which prevents attacks like man in the middle (MIM).
hsts	Sets Strict-Transport-Security header which enforces usage of HTTPS for connecting to the server.
ieNoOpen	Sets the X-Download-Options header which prevents Internet Explorer browser users from downloads.
noCache	Enables Cache-Control, Pragma, Expires, and Surrogate-Control headers which prevents client caching site resources, which are old.
noSniff	Sets X-Content-Type-Options.
frameguard	Sets the “X-Frame-Options” header which blocks attacks like clickjacking which prevents the web page to be accessed on other sites.
xssFilter	Sets the Cross-site scripting (XSS) filter latest web browsers.
referrerPolicy	Sets Referrer-Policy header.