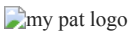

Q1: What Type Of Web Application Can Built Using Express Js?

you can build single-page, multi-page, and hybrid web applications.

Q2: What Are Core Features Of Express Framework?

- Allows to set up middlewares to respond to HTTP Requests.
- Defines a routing table which can works as per HTTP Method and URL.
- Dynamically render HTML Pages.

Q3: Why I Should Use Express Js?

Express 3.x is a light-weight web application framework to help organize your web application into an MVC architecture on the server side.

Q4: What Function Arguments Are Available To Express.js Route Handlers?

The arguments available to an Express.js route handler function are:
req - the request object
res - the response object
next (optional) - a function to pass control to one of the subsequent route handlers
The third argument may be omitted, but is useful in cases where you have a chain of handlers and you would like to pass control to one of the subsequent route handlers, and skip the current one.

Q5: How To Config Properties In Express Application?

var host = process.env.APP_HOST app.set('host', host);
logger.info('Express server listening on http://' + app.get('host'));

With Requireis: Create a file called 'config.json' inside a folder called 'config' inside the project folder. Add config properties in config.json.

"env":"development", "apiur1":"http://localhost:9080/api/v1/" }

Use require to access the config.json file.

var config = require('./config/config.json');

Q6: How To Allow Cors In Expressjs? Explain With An Example?

In an ExpressiS Application, we can config properties in following two ways:
With Process.ENV:
Create a file with name '.env' inside the project folder. Add all the properties in '.env' file. In server.js any of the properties can be used as:

app.allr, function(req, res, next) (res.set('Access-Control-Allow-Origin', "");
res.set('Access-Control-Allow-Methods', 'GET, POST, DELETE, PUT');
res.set('Access-Control-Allow-Headers', 'X-Requested-With, Content-Type');
if ('OPTIONS' == req.method) return res.send(200);
next();

Q7: How To Redirect 404 Errors To A Page In Expressjs?

In server.js add the following code to redirect 404 errors back to a page in our ExpressiS App:

app.use(function(req, res, next) (
res.status(404).
json((errorCode: 404, errorMsg: "route not found"}); });

Q8: How To Enable Debugging In Express App?

In different Operating Systems, we have following commands:
On Linux the command would be as follows:
$ DEBUG=express:* node index.js
On Windows the command would be:
set DEBUG=express:* & node index.js

expressInterviewsQuestions
staticFiles
morgan
BuildingAPIS
testing

What is Express.js

Home

Express is a fast, assertive, essential and moderate web framework of Node.js. You can assume express as a layer built on the top of the Node.js that helps manage a server and routes. It provides a robust set of features to develop web and mobile applications.
Let's see some of the core features of Express framework:

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

Why use Express

- Ultra fast I/O
- Asynchronous and single threaded
- MVC like structure
- Robust API makes routing easy

Express.js Request Object

Express.js Request and Response objects are the parameters of the callback function which is used in Express applications. The express.js request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.
app.get('/', function (req, res))

Express.js Request Object Properties

Properties	Description
req.app	This is used to hold a reference to the instance of the express application that is using the middleware.
req.baseurl	It specifies the URL path on which a router instance was mounted.
req.body	It contains key-value pairs of data submitted in the request body. By default, it is undefined, and is populated when you use body-parsing middleware such as body-parser.
req.cookies	When we use cookie-parser middleware, this property is an object that contains cookies sent by the request.
req.hostname	It contains the hostname from the "host" http header.
req.params	An object containing properties mapped to the named route ?parameters?. For example, if you have the route /user/:name, then the "name" property is available as req.params.name. This object defaults to .
req.path	It contains the path part of the request URL.
req.route	The currently-matched route, a string.
req.secure	A Boolean that is true if a TLS connection is established.

Request Object Methods

req.accepts (types)
req.accepts('html');
req.accepts('text/html');

req.get(field)
This method returns the specified HTTP request header field.
req.get('Content-Type');
req.get('content-type');
req.get('Something');

req.is(type)
This method returns true if the incoming request's "Content-Type" HTTP header field matches the MIME type specified by the type parameter.
req.is('html');
req.is('text/html');
req.is('text/*');

req.param(name [, defaultValue])
This method is used to fetch the value of param name when present.
?name=sasha

req.param('name')
o/p => "sasha"

POST name=sasha
req.param('name')
o/p => "sasha"

/user/sasha for /user/:name
req.param('name')

Express.js Response Object

The Response object (res) specifies the HTTP response which is sent by an Express app when it gets an HTTP request.
What it does

- It sends response back to the client browser.
- It facilitates you to put new cookies value and that will write to the client browser (under cross domain rule).
- Once you res.send() or res.redirect() or res.render(), you cannot do it again, otherwise, there will be uncaught error.

Properties	Description
res.app	It holds a reference to the instance of the express application that is using the middleware.
res.headersSent	It is a Boolean property that indicates if the app sent HTTP headers for the response.
res.locals	It specifies an object that contains response local variables scoped to the request

Response Object Methods

Response Append method
This method appends the specified value to the HTTP response header field. That means if the specified value is not appropriate then this method redress that.

Express.js GET Request

GET and POST both are two common HTTP requests used for building REST API's. GET requests are used to send only limited amount of data because data is sent into header while POST requests are used to send large amount of data because data is sent in the body.
Express.js facilitates you to handle GET and POST requests using the instance of express.
Get method facilitates you to send only limited amount of data because data is sent in the header. It is not secure because data is visible in URL bar.

```
var express = require('express');
var app = express();
app.use(express.static('public'));
app.get('/index.html', function (req, res)
{ res.sendFile(__ dirname +"/" + "index.html" );
)
app.get('/process_get', function (req, res)
{ response = {
first_name:req.query.first_name,
last_name:req.query.last_name
console.log(response);
res.end(JSON.stringify(response));
)
var server = app.listen(8000, function ()
{ var host = server.address().address
var port = server.address().port
console.log("Example app listening at http://%s:%s", host, port)
)
```

Express.js POST Request

GET and POST both are two common HTTP requests used for building REST API's. POST requests are used to send large amount of data.
Express.js facilitates you to handle GET and POST requests using the instance of express.
Post method facilitates you to send large amount of data because data is send in the body. Post method is secure because data is not visible in URL bar but it is not used as popularly as GET method. On the other hand GET method is more efficient and used more than POST.

```
var express = require('express');
var app = express();
var bodyParser = require('body-parser');
// Create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended: false })
```

```
app.use(express.static({‘public'));
app.get('/index.html', function (req, res)
{ res.sendFile( dirname +"/" + "index.html" );
)
app.post('/process_post', urlencodedParser, function (req, res) {
// Prepare output in JSON format
response = {
first_name:req.body.first_name,
last_name:req.body.last_name
console.log(response);
res.end(JSON.stringify(response));
)
var server = app.listen(8000, function () {
var host = server.address().address
var port = server.address().port
console.log("Example app listening at http://%s:%s", host, port)
```

Express.js Routing

Routing is made from the word route. It is used to determine the specific behavior of an application. It specifies how an application responds to a client request to a particular route, URI or path and a specific HTTP request method (GET, POST, etc.). It can handle different types of HTTP requests.

```
var express = require(‘express');
var app = express();
app.get('/", function (req, res) {
console.log("Got a GET request for the homepage");
res.send('Welcome to JavaTpoint!'};
)
app.post('/', function (req, res) {
console.log("Got a POST request for the homepage");
res.send('lam Impossible! ‘);
)
app.delete(‘/del_student', function (req, res) {
console.log("Got a DELETE request for /del_student");
res.send(‘lam Deleted!'};
)
app.get('/enrolled_student', function (req, res) {
console.log("Got a GET request for /enrolled_student");
res.send(‘lam an enrolled student.');
)
// This responds a GET request for abcd, abxcd, ab123cd, and so on
app.get('/ab*cd', function(req, res) {
console.log("Got a GET request for /ab*cd");
res.send(‘Pattern Matched.');
) var server = app.listen(8000, function () {
var host = server.address().address
var port = server.address().port
console.log("Example app listening at http://%s:%s", host, port)
)
```

Express.js Cookies Management

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

```
var express = require(‘express');
var cookieParser = require(‘cookie-parser');
var app = express();
app.use(cookieParser(});
app.get('/cookieset',function(req, res){
res.cookie(‘cookie_name’, ‘cookie_value');
res.cookie(‘company'’, ‘javatpoint');
res.cookie(‘name’, ‘sonoo'};
res.status(200).send(‘Cookie is set');
);
app.get('/cookieget', function(req, res) {
res.status(200).send(req.cookies);
);
```

```
app.get('/", function (req, res) {
res.status(200).send(‘Welcome to JavaTpoint!'};
);
var server = app.listen(8000, function () {
var host = server.address().address;
var port = server.address().port;
console.log(‘Example app listening at http://%s:%s', host, port);
);
```

Express.js File Upload

In Express.js, file upload is slightly difficult because of its asynchronous nature and networking approach.
It can be done by using middleware to handle multipart/form data. There are many middleware that can be used like multer, connect, body-parser etc.
Let's take an example to demonstrate file upload in Node.js. Here, we are using the middleware 'multer'.

```
var express = require("express");
var multer = require(‘multer');
var app = express();
var storage = multer.diskStorage({
destination: function (req, file, callback) {
callback(null, './uploads');
filename: function (req, file, callback) {
callback(null, file.originalname);
}
);
var upload = multer({ storage : storage}).single(‘myfile');
app.get('/",function(req,res){
res.sendFile(_ dirname +"/index.html");
);
app.post('/uploadjavatpoint', function(req,res){
upload(req,res,function(err) {
if(err) {
return res.end("Error uploading file.");
}
res.end("File is uploaded successfully!");
);
);
app.listen(2000,function(){
console.log("Server is running on port 2000");
);
```

Express.js Middleware

Express.js Middleware are different types of functions that are invoked by the Express.js routing layer before the final request handler. As the name specified, Middleware appears in the middle between an initial request and final intended route. In stack, middleware functions are always invoked in the order in which they are added.
Middleware is commonly used to perform tasks like body parsing for URL-encoded or JSON requests, cookie parsing for basic cookie handling, or even building JavaScript modules on the fly.

What is a Middleware function

Middleware functions are the functions that access to the request and response object (req, res) in request-response cycle.
A middleware function can perform the following tasks:

- It can execute any code.
- It can make changes to the request and the response objects.
- It can end the request-response cycle.
- It can call the next middleware function in the stack.

Express.js Middleware

- Application-level middleware
- Router-level middleware
- Error-handling middleware
- Built-in middleware
- Third-party middleware

```
var express = require(‘express');
var app = express();
app.get('/", function(req, res) {
res.send('Welcome to JavaTpoint!'};
);
app.get('/help', function(req, res) {
res.send(‘How can | help You?’);
);
var server = app.listen(8000, function () {
var host = server.address().address
var port = server.address().port
console.log("Example app listening at http://%s:%s", host,port)
)
```

Use of Express.js Middleware

```
var express = require(‘express');
var app = express();
app.use(function(req, res, next) {
console.log(%s %s', req.method, req.url);
next();
);
app.get('/", function(req, res, next) {
res.send('Welcome to JavaTpoint!'};
);
app.get('/help', function(req, res, next) {
res.send(‘How can | help you?’);
);
var server = app.listen(8000, function () {
var host = server.address().address
var port = server.address().port
console.log("Example app listening at http://%s:%s", host,port)
)
```

What is scaffolding

Scaffolding is a technique that is supported by some MVC frameworks.
It is mainly supported by the following frameworks:
Ruby on Rails,OutSystems Platform, Express Framework, Play framework, Django, MonoRail, Brail, Symfony, Laravel, CodeIgniter, Yii, CakePHP, Phalcon PHP, Model-Glue, PRADO, Grails, Catalyst, Seam Framework, Spring Roo, ASP.NET etc.
Scaffolding facilitates the programmers to specify how the application data may be used. This specification is used by the frameworks with predefined code templates, to generate the final code that the application can use for CRUD operations (create, read, update and delete database entries).
An Express.js scaffold supports candy and more web projects based on Node.js.

Express is a relatively small framework that sits on top of Node.js’s web server func- tionality to simplify its APIs and add helpful new features. It makes it easier to orga- nize your application’s functionality with middleware and routing; it adds helpful utilities to Node.js’s HTTP objects; it facilitates the rendering of dynamic HTML views; it defines an easily implemented extensibility standard.

The JavaScript function that processes browser requests in your application is called a request handler. There’s nothing too special about this; it’s a JavaScript function that takes the request, figures out what to do, and responds. Node.js’s HTTP server handles the connection between the client and your JavaScript function so that you don’t have to handle tricky network protocols. In code, it’s a function that takes two arguments: an object that represents the request and an object that represents the response @ Node.js, a JavaScript platform typically used to run JavaScript on servers @Express, a framework that sits on top of Node.js’s web server and makes it easier to use.Middleware and routing, two features of Express Request handler functions

What is this Express?

Express is a relatively small framework that sits on top of Node.js’s web server func-tionality to simplify its APIs and add helpful new features. It makes it easier to orga-nize your application’s functionality with middleware and routing;
it adds helpfulutilities to Node.js’s HTTP objects;

it facilitates the rendering of dynamic HTML views;it defines an easily implemented extensibility standard.

What is Express?I don’t mean to tell you that Node.js is the fastest in the world because of its asynchro-nous capabilities. Node.js can squeeze a lot of performance out of one CPU core, butit doesn’t excel with multiple cores. Other programming

languages truly allow you to actively do two things at once.
To reuse the baking example:

other programming lan-guages let you buy more ovens so that you can bake more muffins simultaneously.Node.js is beginning to support this functionality but it's not as first-class in Node.js asit is in other programming languages. Personally, I don't believe that performance is the biggest reason to chooseNode.js. Although it's often faster than other scripting languages like Ruby or Python, I think the biggest reason is that it's all one programming language.

Often, when you're writing a web application, you'll be using JavaScript. Butbefore Node.js, you'd have to code everything in two different programming lan-guages. You'd have to learn two different technologies, paradigms, and libraries. WithNode.js, a back-end developer can jump into front-end code and vice versa. Personally, I think this is the most powerful feature of the runtime. Other people seem to agree: some developers have created the MEAN stack, whichis an all JavaScript web application stack consisting of MongoDB (a database controlledby JavaScript), Express, Angular.js (a front-end JavaScript framework), and Node.js.The JavaScript everywhere mentality is a huge benefit of Node.js. Large companies such as Wal-Mart, the BBC, LinkedIn, and PayPal are even gettingbehind Node.js.

What Express adds to Node.js

1.It adds a number of helpful conveniences to Node.js's HTTP server, abstractingaway a lot of its complexity. For example, sending a single JPEG file is fairly com-plex in raw Node.js (especially if you have performance in mind); Expressreduces it to one line.
2.It lets you refactor one monolithic request handler function into many smallerrequest handlers that handle only specific bits and pieces. This is more main-tainable and more modular

There are essentially two things going on here

1.Rather than one large request handler function, Express has you writingmany smaller functions (many of which can be third-party functions and notwritten by you). Some functions are executed for every request (for example,a function that logs all requests), and other functions are only executed some-times

2.Express adds two big features to the Node.js HTTP server: It adds a number of helpful conveniences to Node.js's HTTP server, abstractingaway a lot of its complexity. For example, sending a single JPEG file is fairly com-plex in raw Node.js (especially if you have performance in mind); Expressreduces it to one line.■It lets you refactor one monolithic request handler function into many smallerrequest handlers that handle only specific bits and pieces. This is more main-tainable and more modular.In contrast to figure 1.2, figure 1.3 shows how a request would flow through an Expressapplication.Figure 1.3 might look more complicated, but it's much simpler for you as the devel-oper. There are essentially two things going on here:■Rather than one large request handler function, Express has you writingmany smaller functions (many of which can be third-party functions and notwritten by you). Some functions are executed for every request (for example,a function that logs all requests), and other functions are only executed some-times (for example, a function that handles only the homepage or the 404page). Express has many utilities for partitioning these smaller request han-dler functions.■Request handler functions take two arguments: the request and the response.Node's HTTP server provides some functionality; for example, Node.js's HTTPserver lets you extract the browser's user agent in one of its variables. Expressaugments this by adding extra features such as easy access to the incomingrequest's IP address and improved parsing of URLs. The response object alsogets beefed up; Express adds things like the sendFile method, a one-line com-mand that translates to about 45 lines of complicated file code. This makes iteasier to write these request handler functions.

Express has just four major features: middleware,routing, sub applications, and conveniencesWhat Express is used for

Express could be used to build any web application. It can process incoming requests and respond to them

First Programms

```
var express = require("express");
var app = express();
app.get("/", function(request, response) {
response.send("Hello, world!");
);
app. listen(3000, function() {
console.log("Express app started on port 3000.");
```

1

@ Middleware which is a way to break your app into smaller bits of behavior.Generally, middleware is called one by one, in a sequence.
@Routing similarly breaks your app up into smaller functions that are exe-cuted when the user visits a particular resource.
@Routers can further break up large applications into smaller, composablesubapplications.

Mustache.js

```
var Mustache = require("mustache");
var result = Mustache.render("Hi, {{first}} {{last}}!", {
first: "Nicolas",
```

```
last: "Cage" });
console.log(result);
```

Defining your own modules

```
var MAX = 100;
function randominteger() {
return Math.floor((Math.random() * MAX));
}
module.exports = randominteger;
@var randomint = require("./random-integer");
console.log(randomint()); // 12
console.log(randomint()); // 77
console.log(randomint()); // 3
```

The two most common external resources you'll deal with in Express are

1.Anything involving the filesystem—Like reading and writing files from your harddrive
2.Anything involving a network—Like receiving requests, sending responses, orsending your own requests over the internet

fs

```
var fs = require("fs");
var options = { encoding: "utf-8" };
fs.readFile("myfile.txt", options, function(err, data) {
if (err) {
console.error("Error reading file!");
return;
}
console.log(data.match(/x/gi).length +" letter X's"));
```

Adding a console.log after the asynchronous operations

```
var fs = require("fs");
vra options = { encoding: "utf-8" };
fs.readFile("myfile.txt", options, function(err, data) {
);
console.log("Hello world!");
@var express = require("express");
var http = require("http");
var app = express();
app.use(function(request, response) {
console.log("In comes a request to: " + request.url);
response.end("Hello, world!");
);
http.createServer(app).listen(3000);
```

middleware

```
var express = require(‘express');
var app = express();
app.get('/", function(req, res) {
res.send('Welcome to JavaTpoint!'};
);
app.get('/help', function(req, res) {
res.send(‘How can | help You?’);
);
var server = app.listen(8000, function () {
var host = server.address().address
var port = server.address().port
console.log("Example app listening at http://%s:%s", host,port)
)
```

middleware can also change the request or response objects

Adding fake authentication middleware

```
app.use(function(request, response, next) {
console.log("In comes a" + request.method +" to " + request.url);
next();});
app.use(function(request, response, next) {
var minute = (new Date(}).getMinutes();
```

```
if ((minute % 2) === 0) {
next();
telse {
response.statusCode = 403;
response.end("Not authorized.")};
}});app.use(function(request, response) {
response.end('Secret info: the password is "swordfish"!');
);
```

Third-party middleware librariesMORGAN LOGGING MIDDLEWARE

```
var express = require("express");
var logger = require("morgan");
var http = require("http");
var app = express();
app.use(logger("short"});
app.use(function(request, response) {
response.writeHead(200, { "Content-Type": "text/plain" });
response.end("Hello, world!");
);
http.createServer(app).listen(3000);
```

EXPRESS S STATIC MIDDLEWARE

```
var express = require("express");
var path = require("path");
var http = require("http");
var app = express();
var publicPath = path.resolve(__dirname, "public");
app.use(express.static(publicPath));
app.use(function(request, response) {
response.writeHead(200, { "Content-Type": "text/plain" });
response.end("Looks like you didn't find a static file.");
);
http.createServer(app).listen(3000);
```

Routing is a way to map requests to specific handlers depending on their URL and HTTP verb.

@The three calls to app.get are Express's magical routing system. They could alsobe app.post, which respond to POST requests, or PUT, or any of the HTTP verb.
@The first argumentis a path, like /about or /weather or simply /, the site's root. The second argument isa request handler function.

Extending request and response

Express augments the request and response objects that you're passed in everyrequest handler.
One nicety Express offers is the redirect method.

Using redirect

```
response.redirect("/hello/world"); response.redirect("http://expressjs.com");
```

sendFile

```
response.sendFile("/path/to/cool_song.mp3");
```

Let's use some of these things to build middleware that blocks an evil IP address.

```
var express = require("express");
var path = require("path");
var http = require("http");
var app = express();
var publicPath = path.resolve(__dirname, "public");
app.use(express.static(publicPath));
app.get("/", function(request, response) {
response.end("Welcome to my homepage!");
); app.get("/about", function(request, response) {
response.end("Welcome to the about page!");
);
app.get("/weather", function(request, response) {
response.end("The current weather is NICE.")
app.use(function(request, response) {
response.statusCode = 404;
```

```
response.end("404!");
);
http.createServer(app).listen(3000);
```

2

@Express has a middleware feature that allows you to pipeline a single request through a series of decomposed functions.
@Express's view-rendering features let you dynamically render HTML pages.
@Middleware—In contrast to vanilla Node.js, where your requests flow ■ through only one function, Express has a middleware stack, which is effec- tively an array of functions.
@Routing—Routing is a lot like middleware, but the functions are called ■ only when you visit a specific URL with a specific HTTP method. You could run a request handler only when the browser visits yourwebsite.com/ about, for example.
@Extensions—Express extends the request and response objects with extramethods and properties for developer convenience.
@Views—Views allow you to dynamically render HTML. This allows you to both change the HTML on the fly and write the HTML in other languages
@two objects(req, res) are passed through just one function. But in Express, these objects are passed through an array of functions, called the middleware stack.
@When next is called, Express will go on to the next function in the stack.
Eventually, one of these functions in the stack must call res.end, which will end the request. (In Express, you can also call some other methods like res.send or res.send- File, but these call res.end internally.) You can call res.end in any of the functions in the middleware stack, but you must only do it once or you'll get an error.

you call app.use to add a

function to your application's middleware stack. When a request comes in to this application, that function will be called.
@Run npm start and visit
localhost:3000 in your browser to see it. You'll see the request being logged into the console, and that's great news. But your browser will hang—the loading spinner will spin and spin and spin, until the request eventually times out and you get an error in your browser. That's not good! This is happening because you didn't call next. When your middleware function is finished, it needs to do one of two things:
It needs to finish responding to the request (with res.end or one of Express's convenience methods like res.send or res.sendFile).
It needs to call next to continue on to the next function in the middleware stack.
If you do both, only the first response finisher will go through and the rest will be ignored
@"Cannot GET /". Because you're never responding to the request yourself.
@The first thing you do in this function is use path.join to determine the path of the file. If the user visits /celine.mp3, req.url will be the string "/celine.mp3". There- fore, filePath will be something like "/path/to/your/project/static/celine.mp3".
@Next, you call fs.exists, which takes two arguments. The first is the path to check (the filePath you just figured out) and the second is a function. When Node has fig- ured out information about the file, it'll call this callback with two arguments.
@ You wouldn't need to have next anywhere. But because things are asynchronous, you need to manually tell Express when to continue on to the next middleware in the stack.
@If you visit a URL that doesn't have a corresponding file, you should still see the error message from before. This is because you're calling next and there's no more middleware in the stack.
@try moving the 404 handler. Make it the first middleware in the stack instead of the last. If you rerun your app, you'll see that you always get a 404 error no matter what. Your app hits the first middleware and never continues on. The order of your middleware stack is important—make sure your requests flow through in the proper order.

Morgan describes itself as "request logger middleware," which is exactly what you want.

@morgan is a function that returns a middleware function.
When you call it, it will return a function like the one you wrote previously; it'll take three arguments and call console.log.
Most third-party middleware works this way— you call a function that returns the middleware,
var morganMiddleware = morgan("short"); app.use(morganMiddleware);
Notice that you're calling Morgan with one argument: a string, "short". This is a Morgan-specific configuration option that dictates what the output should look like. There are other format strings that have more or less information: "combined" gives a lot of info; "tiny" gives a minimal output. When you call Morgan with different config- uration options, you're effectively making it return a different middleware function.
@express.static. It works a lot like the middleware we wrote, but it has a bunch of other features. It does several complicated tricks to achieve better security and performance, such as adding a caching mechanism.

Error-handling middleware

These middleware functions take four arguments instead of two or three. The first one is the error (the argument passed into next), and the remainder are the three from before: req, res, and next. You can do anything you want in this middleware. When you're done, it's just like other middleware: you can call res.end or next. Call- ing next with no arguments will exit error mode and move onto the next normal mid- dleware; calling it with an argument will continue onto the next error-handling middleware if one exists.

Express team maintains

a number of middleware modules: body-parser for parsing request bodies. For example, when a user submits a form. See more at https://github.com/expressjs/body-parser. cookie-parser for parsing cookies from users. It needs to be paired with

another Express-
supported middleware like express-session. Once you've done this, you can keep track of users, providing them with user accounts and other features. We'll explore this in greater detail in chapter 7. https://github.com/expressjs/cookie-session has more details. Compression for compressing responses to save on bytes. See more at https://github.com/expressjs/compression.

Express applications have a middleware stack.

When a request enters your ■ application, requests go through this middleware stack from the top to the bot- tom, unless they're interrupted by a response or an error. Middleware is written with request handler functions. These functions take two ■ arguments at a minimum: first, an object representing the incoming request; second, an object representing the outgoing response. They often take a func- tion that tells them how to continue on to the next middleware in the stack.
@Express applications have a middleware stack.

expressdocs
expressInterviewsQuestions
morgan
BuildingAPIS
testing

staticFiles.js

```
var express = require("express");
var path = require("path");
var fs = require("fs");
var app = express();
app.use(function(req, res, next) {
console.log("Request IP: "+ req.url);
console.log("Request date: " + new Date());

next();
});

app.use(function(req, res, next) {
var filePath = path.join(_dirnarne, "static", req.urI);
fs.stat(filePath, function(err, filelnfo) {

if (err) { next(); return;
}
if (fileInfoisFile()) {
res.sendFile(filePath);
}
else { next();
}
});
});
app.use(function(req, res) { res.status(404);
res.send("File not found!");
});
app.listen(3000, function() { console.log("App started on port 30001; }
```

expressdocs
expressInterviewsQuestions
staticFiles
BuildingAPIS
testing

staticFiles.js

```
var express = require("express");
var morgan = require("morgan");
var app = express();

app.use(morgan("short"));
var morganMiddleware = morgan("short");

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

expressdocs
expressInterviewsQuestions
staticFiles
morgan
testing

API meant to be consumed by code.At some level, all APIs sit on top of software interfaces.At a high level, APIs are ways for one piece of code to talk to another piece of code. This could mean a computer talking to itself or a computer talking to another computer over a network. Express API, like app.use or app.get. These are interfaces that you as a programmer can use to talk to other code. These APIs will take HTTP requests and respond with JSON data.You could build a simple web page that consumed this API. It might send AJAX requests to your server, parse the JSON, and display it in the HTML.
The funda- mentals of an Express API are pretty straightforward: take a request, parse it, and respond with a JSON object and an HTTP status code. You'll use middleware and rout- ing to take requests and parse them, and you'll use Express's conveniences to respond to requests.
API can use other data interchange formats like XML or plain text.HTTP verbs (also known as HTTP methods)

1.GET—The most common HTTP method anyone uses. As the name suggests, it ■ gets resources. When you load someone's homepage, you GET it. When you load an image, you GET it. GET methods shouldn't change the state of your app; the other methods do that.Idempotence is important to GET requests. Idempotent is a fancy word that means doing it once should be no different than doing it many times. If you GET an image once and refresh 500 times, the image should never change. The response can change—a page could change based on a changing stock price or a new time of day—but GETs shouldn't cause that change. That's idempotent.
2.POST—Generally used to request a change to the state of the server. You POST a ■ blog entry; you POST a photo to your favorite social network; you POST when you sign up for a new account on a website. POST is used to create records on servers, not modify existing records. POST is also used for actions, like buy this item. Unlike GET, POST is non- idempotent. That means that the state will change the first time you POST, and the second time, and the third time, and so on.
3.PUT—A better name might be update or change. If I've published (POSTed) a ■ job profile online and later want to update it, I would PUT those changes. I could PUT changes to a document, or to a blog entry, or to something else.
if you try to PUT changes to a record that doesn't exist, the server can (but doesn't have to) create that record.

Handling different HTTP verbs

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

app.get("/", function(req, res) {
res.send("you just sent a GET request, friend");
);

app.post("/", function(req, res) {
res.send("a POST request? nice");
);
app.put("/", function(req, res) {
res.send("i don't see a lot of PUT requests anymore");
);

app.delete("/", function(req, res) {
res.send("oh my, a DELETE??");
);

app.listen(3000, function() {
console.log({"App is listening on port 3000");
);
```

POST vs. PUT

Because PUT can create records just like POST can, you could say that PUT better corresponds to create. PUT can create and update records, so why not put it in both spots? Similarly, the PATCH method (which we haven't yet mentioned) sometimes takes the update role. To quote the specification, "the PUT method is already defined to over- write a resource with a complete new body, and cannot be reused to do partial changes." PATCH allows you to partially overwrite a resource. PATCH was only for- mally defined in 2010, so it's relatively new on the HTTP scene, which is why it's less used. In any case, some people think PATCH is better suited to update than PUT. Because HTTP doesn't specify this stuff too strictly, it's up to you to decide what you want to do.

API versioning

Let me walk you through a scenario. You design a public API for your time zone app and it becomes a big hit. People all over the world are using it to find times all across the globe. It's working well. But, after a few years, you want to update your API. You want to change something, but there's a problem: if you make changes, all of the people using your API will have to update their code. What do you do? Do you make the changes you want to make and break old users, or does your API stagnate and never stay up to date? There's a solution to all of this: version your API. All you have to do is add version information to your API. So a request that comes into this URL might be for version 1 of your API.
/v1/timezone
and a request coming into version 2 of your API might visit this URL:
/v2/timezone**app.js**

```
var express = require("express");
var apiVersion1 = require("./apil.js");
Requires and uses the router

var app = express();

app.use("/v1", apiVersion1);
app. listen(3000, function() {
console.log("App started on port 3000");
```

app2.js

```
var express = require("express");
var api = express.Router();


api.get("/timezone", function(req, res) {
res.send("API 2: super cool new response for /timezone");

module.exports = api;
```

main.js

```
var express = require("express");
var apiVersion1 = require("./apil.js");
var apiVersion2 = require("./api2.js");


var app = express();
app.use("/v1", apiVersion1);
app.use("/v2", apiVersion2);

app. listen(3000, function() { console.log("App started on port 3000"); 1);
```

Setting HTTP status codes

Every HTTP response comes with an HTTP status code.
HTTP status ranges in a nutshell:
1xx: hold on
2xx: here you go
3xx: go away
4xx: you messed up
5xx: I messed up
WHAT ABOUT HTTP 2? Most HTTP requests are HTTP 1.1 requests, with a handful of them still using version 1.0. HTTP 2, the next version of the stan- dard, is slowly being implemented and rolled out across the web. Luckily, most of the changes happen at a low level and you don't have to deal with them. HTTP 2 does define one new status code—421—but that shouldn't affect you much.

Setting the HTTP status code in Express

view engine

Express has a view system that can dynamically render HTML pages. You call ■ res.render to dynamically render a view with some variables. Before doing this, you must configure Express to use the right view engine in the right folder.