Hello world example

Make sure you've followed the instructions in Installing.

Now, you're going to create a very basic Express app.

> NOTE: This is essentially the simplest Express app you can create. It's a single file—***not*** what you'd get if you use the Express generator, which creates the scaffolding for a full app with numerous JavaScript files, Jade templates, and sub-directories for various purposes.

In the `myapp` directory, create a file named `app.js` and add the following code to it:

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

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

var server = app.listen(3000, function () {
  var host = server.address().address;
  var port = server.address().port;

  console.log('Example app listening at http://%s:%s', host, port);
});
```

The app starts a server and listens on port 3000 for connection. It will respond with "Hello World!" for requests to the root URL (`/`) or ***route***. For every other path, it will respond with a **404 Not Found**.

> The `req` (request) and `res` (response) are the exact same objects that Node provides, so you can invoke `req.pipe()`, `req.on('data', callback)`, and anything else you would do without Express involved.

Run the app with the following command.

```
$ node app.js
```

Then, load http://localhost:3000/ in a browser to see the output.

Routing

Routing refers to the definition of end points (URIs) to an application and how it responds to client requests.

A route is a combination of a URI, a HTTP request method (GET, POST, and so on), and one or more handlers for the endpoint. It takes the following structure `app.METHOD(path, [callback...], callback)`, where `app`is an instance of `express`, `METHOD` is an HTTP request method, `path` is a path on the server, and `callback` is the function executed when the route is matched.

The following is an example of a very basic route.

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

// respond with "hello world" when a GET request is made to the homepage
app.get('/', function(req, res) {
  res.send('hello world');
});
```

Route methods

A route method is derived from one of the HTTP methods, and is attached to an instance of `express`.

The following is an example of routes defined for the GET and the POST methods to the root of the app.

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

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

Express supports the following routing methods corresponding to HTTP methods: `get`, `post`, `put`, `head`, `delete`, `options`, `trace`, `copy`, `lock`, `mkcol`, `move`, `purge`, `propfind`, `proppatch`, `unlock`, `report`, `mkactivity`, `checkout`, `merge`, `m-search`, `notify`, `subscribe`, `unsubscribe`, `patch`, `search`, and `connect`.

> To route methods which translate to invalid JavaScript variable names, use the bracket notation. For example, `app['m-search']('/', function ...`

There is a special routing method, `app.all()`, which is not derived from any HTTP method. It is used for loading middleware at a path for all request methods.

In the following example, the handler will be executed for requests to “/secret” whether using GET, POST,

PUT, DELETE, or any other HTTP request method supported in the http module.

```
app.all('/secret', function (req, res, next) {
  console.log('Accessing the secret section ...');
  next(); // pass control to the next handler
});
```

Route paths

Route paths, in combination with a request method, define the endpoints at which requests can be made to. They can be strings, string patterns, or regular expressions.

> Express uses path-to-regexp for matching the route paths; see its documentation for all the possibilities in defining route paths. Express Route Tester is a handy tool for testing basic Express routes, although it does not support pattern matching.

> Query strings are not a part of the route path.

Examples of route paths based on strings:

```
// will match request to the root
app.get('/', function (req, res) {
  res.send('root');
});

// will match requests to /about
app.get('/about', function (req, res) {
  res.send('about');
});

// will match request to /random.text
app.get('/random.text', function (req, res) {
  res.send('random.text');
});
```

Examples of route paths based on string patterns:

```
// will match acd and abcd
app.get('/ab?cd', function(req, res) {
  res.send('ab?cd');
});

// will match abcd, abbcd, abbbcd, and so on
app.get('/ab+cd', function(req, res) {
  res.send('ab+cd');
```

```
});

// will match abcd, abxcd, abRABDOMcd, ab123cd, and so on
app.get('/ab*cd', function(req, res) {
  res.send('ab*cd');
});

// will match /abe and /abcde
app.get('/ab(cd)?e', function(req, res) {
 res.send('ab(cd)?e');
});
```

The characters ?, +, *, and () are subsets of their Regular Expression counterparts. The hyphen (-) and the dot (.) are interpreted literally by string-based paths.

Examples of route paths based on regular expressions:

```
// will match anything with an a in the route name:
app.get(/a/, function(req, res) {
  res.send('/a/');
});

// will match butterfly, dragonfly; but not butterflyman, dragonfly man, and
app.get(/.*fly$/, function(req, res) {
  res.send('/.*fly$/');
});
```

Route handlers

You can provide multiple callback functions that behave just likemiddleware to handle a request. The only exception is that these callbacks may invoke `next('route')` to bypass the remaining route callback(s). You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there's no reason to proceed with the current route.

Route handlers can come in the form of a function, an array of functions, or various combinations of both, as shown the following examples.

A route can be handled using a single callback function:

```
app.get('/example/a', function (req, res) {
  res.send('Hello from A!');
});
```

A route can be handled using a more than one callback function (make sure to specify the `next` object):

```
app.get('/example/b', function (req, res, next) {
```

```
  console.log('response will be sent by the next function ...');
  next();
}, function (req, res) {
  res.send('Hello from B!');
});
```

A route can be handled using an array of callback functions:

```
var cb0 = function (req, res, next) {
  console.log('CB0');
  next();
}

var cb1 = function (req, res, next) {
  console.log('CB1');
  next();
}

var cb2 = function (req, res) {
  res.send('Hello from C!');
}

app.get('/example/c', [cb0, cb1, cb2]);
```

A route can be handled using a combination of array of functions and independent functions:

```
var cb0 = function (req, res, next) {
  console.log('CB0');
  next();
}

var cb1 = function (req, res, next) {
  console.log('CB1');
  next();
}

app.get('/example/d', [cb0, cb1], function (req, res, next) {
  console.log('response will be sent by the next function ...');
  next();
}, function (req, res) {
  res.send('Hello from D!');
});
```

Response methods

The methods on the response object (`res`) in the following table can send a response to the client and

terminate the request response cycle. If none of them is called from a route handler, the client request will be left hanging.

Method	Description
res.download()	Prompt a file to be downloaded.
res.end()	End the response process.
res.json()	Send a JSON response.
res.jsonp()	Send a JSON response with JSONP support.
res.redirect()	Redirect a request.
res.render()	Render a view template.
res.send()	Send a response of various types.
res.sendFile	Send a file as an octet stream.
res.sendStatus()	Set the response status code and send its string representation as the response body.

app.route()

Chainable route handlers for a route path can be created using `app.route()`. Since the path is specified at a single location, it helps to create modular routes and reduce redundancy and typos. For more information on routes, see Router() documentation.

Here is an example of chained route handlers defined using `app.route()`.

```
app.route('/book')
  .get(function(req, res) {
    res.send('Get a random book');
  })
  .post(function(req, res) {
    res.send('Add a book');
  })
  .put(function(req, res) {
    res.send('Update the book');
  });
```

express.Router

The `express.Router` class can be used to create modular mountable route handlers.
A `Router` instance is a complete middleware and routing system; for this reason it is often referred to as a “mini-app”.

The following example creates a router as a module, loads a middleware in it, defines some routes, and mounts it on a path on the main app.

Create a router file named `birds.js` in the app directory, with the following content:

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

```
var router = express.Router();

// middleware specific to this router
router.use(function timeLog(req, res, next) {
  console.log('Time: ', Date.now());
  next();
});
// define the home page route
router.get('/', function(req, res) {
  res.send('Birds home page');
});
// define the about route
router.get('/about', function(req, res) {
  res.send('About birds');
});

module.exports = router;
```

Then, load the router module in the app:

```
var birds = require('./birds');
...
app.use('/birds', birds);
```

The app will now be able to handle requests to `/birds` and `/birds/about`, along with calling the `timeLog` middleware specific to the route.

Express application generator

Use the application generator tool, `express`, to quickly create an application skeleton.

Install it with the following command.

```
$ npm install express-generator -g
```

Display the command options with the `-h` option:

```
$ express -h

  Usage: express [options] [dir]

  Options:

    -h, --help          output usage information
    -V, --version       output the version number
    -e, --ejs           add ejs engine support (defaults to jade)
        --hbs           add handlebars engine support
    -H, --hogan         add hogan.js engine support
    -c, --css <engine>  add stylesheet <engine> support (less|stylus|compass
        --git           add .gitignore
    -f, --force         force on non-empty directory
```

For example, the following creates an Express app named ***myapp*** in the current working directory.

```
$ express myapp

   create : myapp
   create : myapp/package.json
   create : myapp/app.js
   create : myapp/public
   create : myapp/public/javascripts
   create : myapp/public/images
   create : myapp/routes
   create : myapp/routes/index.js
   create : myapp/routes/users.js
   create : myapp/public/stylesheets
   create : myapp/public/stylesheets/style.css
   create : myapp/views
   create : myapp/views/index.jade
   create : myapp/views/layout.jade
   create : myapp/views/error.jade
   create : myapp/bin
```

```
   create : myapp/bin/www
```

Then install dependencies:

```
$ cd myapp
$ npm install
```

Run the app (on MacOS or Linux):

```
$ DEBUG=myapp npm start
```

On Windows, use this command:

```
> set DEBUG=myapp & npm start
```

Then load `http://localhost:3000/` in your browser to access the app.

The generated app directory structure looks like the following.

```
.
├── app.js
├── bin
│   └── www
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.jade
    ├── index.jade
    └── layout.jade

7 directories, 9 files
```

Serving static files in Express

Serving files, such as images, CSS, JavaScript and other static files is accomplished with the help of a built-in middleware in Express - `express.static`.

Pass the name of the directory, which is to be marked as the location of static assets, to the `express.static` middleware to start serving the files directly. For example, if you keep your images, CSS, and JavaScript files in a directory named `public`, you can do this:

```
app.use(express.static('public'));
```

Now, you will be able to load the files under the `public` directory:

```
http://localhost:3000/images/kitten.jpg
http://localhost:3000/css/style.css
http://localhost:3000/js/app.js
http://localhost:3000/images/bg.png
http://localhost:3000/hello.html
```

> The files are looked up relative to the static directory, therefore, the name of the static directory is not a part of the URL.

If you want to use multiple directories as static assets directories, you can call the `express.static` middleware multiple times:

```
app.use(express.static('public'));
app.use(express.static('files'));
```

The files will be looked up in the order the static directories were set using the `express.static` middleware.

If you want to create a “virtual” (since the path does not actually exist in the file system) path prefix for the files served by `express.static`, you canspecify a mount path for the static directory, as shown below:

```
app.use('/static', express.static('public'));
```

Now, you will be able to load the files under the `public` directory, from the path prefix “/static”.

```
http://localhost:3000/static/images/kitten.jpg
http://localhost:3000/static/css/style.css
http://localhost:3000/static/js/app.js
http://localhost:3000/static/images/bg.png
http://localhost:3000/static/hello.html
```

Error handling

Define error-handling middleware like other middleware, except with four arguments instead of three, specifically with the signature `(err, req, res, next)`. For example:

```
app.use(function(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

You define error-handling middleware last, after other `app.use()` and routes calls; for example:

```
var bodyParser = require('body-parser');
var methodOverride = require('method-override');

app.use(bodyParser());
app.use(methodOverride());
app.use(function(err, req, res, next) {
  // logic
});
```

Responses from within the middleware are completely arbitrary. You may wish to respond with an HTML error page, a simple message, a JSON string, or anything else you prefer.

For organizational (and higher-level framework) purposes, you may define several error-handling middleware, much like you would with regular middleware. For example suppose you wanted to define an error-handler for requests made via XHR, and those without, you might do:

```
var bodyParser = require('body-parser');
var methodOverride = require('method-override');

app.use(bodyParser());
app.use(methodOverride());
app.use(logErrors);
app.use(clientErrorHandler);
app.use(errorHandler);
```

Where the more generic `logErrors` may write request and error information to stderr, loggly, or similar services:

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

```
  next(err);
}
```

Where `clientErrorHandler` is defined as the following (note that the error is explicitly passed along to the next):

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

The following `errorHandler` "catch-all" implementation may be defined as:

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

If you pass anything to the `next()` function (except the string `'route'`) Express will regard the current request as having errored out and will skip any remaining non-error handling routing/middleware functions. If you want to handle that error in some way you'll have to create an error-handling route as described in the next section.

If you have a route handler with multiple callback functions you can use the 'route' parameter to skip to the next route handler. For example:

```
app.get('/a_route_behind_paywall',
  function checkIfPaidSubscriber(req, res, next) {
    if(!req.user.hasPaid) {

      // continue handling this request
      next('route');
    }
  }, function getPaidContent(req, res, next) {
    PaidContent.find(function(err, doc) {
      if(err) return next(err);
      res.json(doc);
    });
  });
```

In this example, the `getPaidContent` handler will be skipped but any remaining handlers in `app` for `/a_route_behind_paywall` would continue to be executed.

> Calls to `next()` and `next(err)` indicate that the current handler is complete and in what state. `next(err`

`)` will skip all remaining handlers in the chain except for those that are set up to handle errors as described above.

The Default Error Handler

Express comes with an in-built error handler, which takes care of any errors that might be encountered in the app. This default error-handling middleware is added at the end of the middleware stack.

If you pass an error to `next()` and you do not handle it in an error handler, it will be handled by the built-in error handler - the error will be written to the client with the stack trace. The stack trace is not included in the production environment.

> Set the environment variable `NODE_ENV` to "production", to run the app in production mode.

If you call `next()` with an error after you have started writing the response, for instance if you encounter an error while streaming the response to the client, Express' default error handler will close the connection and make the request be considered failed.

So when you add a custom error handler you will want to delegate to the default error handling mechanisms in express, when the headers have already been sent to the client.

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

Using middleware

Express is a routing and middleware web framework with minimal functionality of its own: An Express application is essentially a series of middleware calls.

Middleware is a function with access to the request object (`req`), theresponse object (`res`), and the next middleware in the application's request-response cycle, commonly denoted by a variable named `next`.

Middleware can:

- Execute any code.
- Make changes to the request and the response objects.
- End the request-response cycle.
- Call the next middleware in the stack.

If the current middleware does not end the request-response cycle, it must call `next()` to pass control to the next middleware, otherwise the request will be left hanging.

An Express application can use the following kinds of middleware:

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

You can load application-level and router-level middleware with an optional mount path. Also, you can load a series of middleware functions together, creating a sub-stack of the middleware system at a mount point.

Application-level middleware

Bind application-level middleware to an instance of the app object with `app.use()` and `app.METHOD()`, where `METHOD` is is the HTTP method of the request that it handles, such as GET, PUT, POST, and so on, in lowercase. For example:

```
var app = express();

// a middleware with no mount path; gets executed for every request to the a
app.use(function (req, res, next) {
  console.log('Time:', Date.now());
  next();
});

// a middleware mounted on /user/:id; will be executed for any type of HTTP
app.use('/user/:id', function (req, res, next) {
  console.log('Request Type:', req.method);
  next();
});

// a route and its handler function (middleware system) which handles GET re
```

```
app.get('/user/:id', function (req, res, next) {
  res.send('USER');
});
```

Here is an example of loading a series of middleware at a mount point with a mount path:

```
// a middleware sub-stack which prints request info for any type of HTTP requ
app.use('/user/:id', function(req, res, next) {
  console.log('Request URL:', req.originalUrl);
  next();
}, function (req, res, next) {
  console.log('Request Type:', req.method);
  next();
});
```

Route handlers enable you to define multiple routes for a path. The example below defines two routes for GET requests to `/user/:id`. The second router will not cause any problems, however it will never get called, because the first route ends the request-response cycle.

```
// a middleware sub-stack which handles GET requests to /user/:id
app.get('/user/:id', function (req, res, next) {
  console.log('ID:', req.params.id);
  next();
}, function (req, res, next) {
  res.send('User Info');
});

// handler for /user/:id which prints the user id
app.get('/user/:id', function (req, res, next) {
  res.end(req.params.id);
});
```

To skip the rest of the middleware from a router middleware stack, call `next('route')` to pass control to the next route. **NOTE**: `next('route')` will work only in middleware loaded using `app.METHOD()` or `router.METHOD()`.

```
// a middleware sub-stack which handles GET requests to /user/:id
app.get('/user/:id', function (req, res, next) {
  // if user id is 0, skip to the next route
  if (req.params.id == 0) next('route');
  // else pass the control to the next middleware in this stack
  else next(); //
}, function (req, res, next) {
  // render a regular page
  res.render('regular');
```

```
});

// handler for /user/:id which renders a special page
app.get('/user/:id', function (req, res, next) {
  res.render('special');
});
```

Router-level middleware

Router-level middleware works just like application-level middleware except it is bound to an instance of `express.Router()`.

```
var router = express.Router();
```

Load router-level middleware using `router.use()` and `router.METHOD()`.

The following example code replicates the middleware system shown above for application-level middleware using router-level middleware:

```
var app = express();
var router = express.Router();

// a middleware with no mount path, gets executed for every request to the r
router.use(function (req, res, next) {
  console.log('Time:', Date.now());
  next();
});

// a middleware sub-stack shows request info for any type of HTTP request to
router.use('/user/:id', function(req, res, next) {
  console.log('Request URL:', req.originalUrl);
  next();
}, function (req, res, next) {
  console.log('Request Type:', req.method);
  next();
});

// a middleware sub-stack which handles GET requests to /user/:id
router.get('/user/:id', function (req, res, next) {
  // if user id is 0, skip to the next router
  if (req.params.id == 0) next('route');
  // else pass the control to the next middleware in this stack
  else next(); //
}, function (req, res, next) {
  // render a regular page
  res.render('regular');
});
```

```
// handler for /user/:id which renders a special page
router.get('/user/:id', function (req, res, next) {
  console.log(req.params.id);
  res.render('special');
});

// mount the router on the app
app.use('/', router);
```

Error-handling middleware

> Error-handling middleware always takes ***four*** arguments. You must provide four arguments to identify it as an error-handling middleware. Even if you don't need to use the `next` object, you must specify it to maintain the signature, otherwise it will be interpreted as regular middleware and fail to handle errors.

Define error-handling middleware like other middleware, except with four arguments instead of three, specifically with the signature `(err, req, res, next)`):

```
app.use(function(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

For details about error-handling middleware, see Error handling.

Built-in middleware

As of 4.x, Express no longer depends on Connect. Except for `express.static`, all of the middleware previously included with Express' are now in separate modules. Please view the list of middleware.

express.static(root, [options])

Express' only built-in middleware is `express.static`. It is based on serve-static, and is responsible for serving the static assets of an Express application.

The `root` argument specifies the root directory from which to serve static assets.

The optional `options` object can have the following properties.

Property	Description	Type	Default
`dotfiles`	Option for serving dotfiles. Possible values are "allow", "deny", and "ignore"	String	"ignore"
`etag`	Enable or disable etag generation	Boolean	`true`
`extensions`	Sets file extension fallbacks.	Array	`[]`
`index`	Sends directory index file.	Mixed	"index.html"

	Set `false` to disable directory indexing.		
`lastModified`	Set the `Last-Modified` header to the last modified date of the file on the OS. Possible values are `true` or `false`.	Boolean	`true`
`maxAge`	Set the max-age property of the Cache-Control header in milliseconds or a string in ms format	Number	0
`redirect`	Redirect to trailing "/" when the pathname is a directory.	Boolean	`true`
`setHeaders`	Function for setting HTTP headers to serve with the file.	Function	

Here is an example of using the `express.static` middleware with an elaborate options object.

```
var options = {
  dotfiles: 'ignore',
  etag: false,
  extensions: ['htm', 'html'],
  index: false,
  maxAge: '1d',
  redirect: false,
  setHeaders: function (res, path, stat) {
    res.set('x-timestamp', Date.now());
  }
}

app.use(express.static('public', options));
```

You can have more than one static directory per app.

```
app.use(express.static('public'));
app.use(express.static('uploads'));
app.use(express.static('files'));
```

For more details about `serve-static` and its options, see the serve-staticdocumentation.

Third-party middleware

Use third-party middleware to add functionality to Express apps.

Install the Node module for the required functionality and load it in your app at the application level or at the router level.

The following example illustrates installing and loading cookie-parsing middleware `cookie-parser`.

```
$ npm install cookie-parser
```

```
var express = require('express');
var app = express();
var cookieParser = require('cookie-parser');

// load the cookie parsing middleware
app.use(cookieParser());
```

See Third-party middleware for a partial list of third-party middleware commonly used with Express.

Debugging Express

Express uses the debug module internally to log information about route matches, middleware in use, application mode, and the flow of the request-response cycle.

> `debug` is like an augmented version of `console.log`. But unlike `console.log`, you don't have to comment out `debug` logs in production code. It is turned off by default and can be conditionally turned on with the use an environment variable named `DEBUG`.

To see all the internal logs used in Express, simply set the `DEBUG`environment variable to `express:*` when launching your app.

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

On Windows, use the corresponding command.

```
> set DEBUG=express:* & node index.js
```

Running this on the default app generated by the express generator would print the following.

```
$ DEBUG=express:* node ./bin/www
  express:router:route new / +0ms
  express:router:layer new / +1ms
  express:router:route get / +1ms
  express:router:layer new / +0ms
  express:router:route new / +1ms
  express:router:layer new / +0ms
  express:router:route get / +0ms
  express:router:layer new / +0ms
  express:application compile etag weak +1ms
  express:application compile query parser extended +0ms
  express:application compile trust proxy false +0ms
  express:application booting in development mode +1ms
  express:router use / query +0ms
  express:router:layer new / +0ms
  express:router use / expressInit +0ms
  express:router:layer new / +0ms
  express:router use / favicon +1ms
  express:router:layer new / +0ms
  express:router use / logger +0ms
  express:router:layer new / +0ms
  express:router use / jsonParser +0ms
  express:router:layer new / +1ms
  express:router use / urlencodedParser +0ms
  express:router:layer new / +0ms
```

```
  express:router use / cookieParser +0ms
  express:router:layer new / +0ms
  express:router use / stylus +90ms
  express:router:layer new / +0ms
  express:router use / serveStatic +0ms
  express:router:layer new / +0ms
  express:router use / router +0ms
  express:router:layer new / +1ms
  express:router use /users router +0ms
  express:router:layer new /users +0ms
  express:router use / <anonymous> +0ms
  express:router:layer new / +0ms
  express:router use / <anonymous> +0ms
  express:router:layer new / +0ms
  express:router use / <anonymous> +0ms
  express:router:layer new / +0ms
```

Now, when a request is made to the app, you will see the logs specified in the Express code.

```
  express:router dispatching GET / +4h
  express:router query  : / +2ms
  express:router expressInit  : / +0ms
  express:router favicon  : / +0ms
  express:router logger  : / +1ms
  express:router jsonParser  : / +0ms
  express:router urlencodedParser  : / +1ms
  express:router cookieParser  : / +0ms
  express:router stylus  : / +0ms
  express:router serveStatic  : / +2ms
  express:router router  : / +2ms
  express:router dispatching GET / +1ms
  express:view lookup "index.jade" +338ms
  express:view stat "/projects/example/views/index.jade" +0ms
  express:view render "/projects/example/views/index.jade" +1ms
```

To see the logs only from the router implementation set the value of `DEBUG`to `express:router`. Likewise, to see logs only from the application implementation set the value of `DEBUG` to `express:application`, and so on.

express-generated app

The app generated by the `express` command also uses the `debug` module and its debug namespace is scoped to the name of the application.

If you generated the app with

```
$ express sample-app
```

You can enable the debug statements with the following command

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

You can specify more than one debug namespace by assigning a comma separated list of names, as shown below.

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

Database integration

Adding database connectivity capability to Express apps is just a matter of loading an appropriate Node.js driver for the database in your app. This document briefly explains how to add and use some of the most popular Node modules for database systems in your Express app:

- Cassandra
- CouchDB
- LevelDB
- MySQL
- MongoDB
- Neo4j
- PostgreSQL
- Redis
- SQLite
- ElasticSearch

> These database drivers are among many that are available. For other options, search on the npm site.

Cassandra

Module: cassandra-driver
Installation

```
$ npm install cassandra-driver
```

Example

```
var cassandra = require('cassandra-driver');
var client = new cassandra.Client({ contactPoints: ['localhost']});

client.execute('select key from system.local', function(err, result) {
  if (err) throw err;
  console.log(result.rows[0]);
});
```

CouchDB

Module: nano
Installation

```
$ npm install nano
```

Example

```
var nano = require('nano')('http://localhost:5984');
nano.db.create('books');
var books = nano.db.use('books');

//Insert a book document in the books database
books.insert({name: 'The Art of war'}, null, function(err, body) {
  if (!err){
    console.log(body);
  }
});

//Get a list of all books
books.list(function(err, body){
  console.log(body.rows);
}
```

LevelDB

Module: levelup
Installation

```
$ npm install level levelup leveldown
```

Example

```
var levelup = require('levelup');
var db = levelup('./mydb');

db.put('name', 'LevelUP', function (err) {

  if (err) return console.log('Ooops!', err);
  db.get('name', function (err, value) {
    if (err) return console.log('Ooops!', err);
    console.log('name=' + value)
  });

});
```

MySQL

Module: mysql
Installation

```
$ npm install mysql
```

Example

```
var mysql      = require('mysql');
var connection = mysql.createConnection({
  host     : 'localhost',
  user     : 'dbuser',
  password : 's3kreee7'
});

connection.connect();

connection.query('SELECT 1 + 1 AS solution', function(err, rows, fields) {
  if (err) throw err;
  console.log('The solution is: ', rows[0].solution);
});

connection.end();
```

MongoDB

Module: mongoskin
Installation

```
$ npm install mongoskin
```

Example

```
var db = require('mongoskin').db('localhost:27017/animals');

db.collection('mamals').find().toArray(function(err, result) {
  if (err) throw err;
  console.log(result);
});
```

If you want a object model driver for MongoDB, checkout Mongoose.

Neo4j

Module: apoc
Installation

```
$ npm install apoc
```

Example

```
var apoc = require('apoc');

apoc.query('match (n) return n').exec().then(
```

```
  function (response) {
    console.log(response);
  },
  function (fail) {
    console.log(fail);
  }
);
```

PostgreSQL

Module: pg
Installation

```
$ npm install pg
```

Example

```
var pg = require('pg');
var conString = "postgres://username:password@localhost/database";

pg.connect(conString, function(err, client, done) {

  if (err) {
    return console.error('error fetching client from pool', err);
  }
  client.query('SELECT $1::int AS number', ['1'], function(err, result) {
    done();
    if (err) {
      return console.error('error running query', err);
    }
    console.log(result.rows[0].number);
  });

});
```

Redis

Module: redis
Installation

```
$ npm install redis
```

Example

```
var client = require('redis').createClient();
```

```
client.on('error', function (err) {
  console.log('Error ' + err);
});

client.set('string key', 'string val', redis.print);
client.hset('hash key', 'hashtest 1', 'some value', redis.print);
client.hset(['hash key', 'hashtest 2', 'some other value'], redis.print);

client.hkeys('hash key', function (err, replies) {

  console.log(replies.length + ' replies:');
  replies.forEach(function (reply, i) {
    console.log('    ' + i + ': ' + reply);
  });

  client.quit();

});
```

SQLite

Module: sqlite3
Installation

```
$ npm install sqlite3
```

Example

```
var sqlite3 = require('sqlite3').verbose();
var db = new sqlite3.Database(':memory:');

db.serialize(function() {

  db.run('CREATE TABLE lorem (info TEXT)');
  var stmt = db.prepare('INSERT INTO lorem VALUES (?)');

  for (var i = 0; i < 10; i++) {
    stmt.run('Ipsum ' + i);
  }

  stmt.finalize();

  db.each('SELECT rowid AS id, info FROM lorem', function(err, row) {
    console.log(row.id + ': ' + row.info);
  });
});
```

```
db.close();
```

ElasticSearch

Module: elasticsearch
Installation

```
$ npm install elasticsearch
```

Example

```
var elasticsearch = require('elasticsearch');
var client = elasticsearch.Client({
  host: 'localhost:9200'
});

client.search({
  index: 'books',
  type: 'book',
  body: {
    query: {
      multi_match: {
        query: 'express js',
        fields: ['title', 'description']
      }
    }
  }
}).then(function(response) {
  var hits = response.hits.hits;
}, function(error) {
  console.trace(error.message);
});
```

Moving to Express 4

Overview

Express 4 is a breaking change from Express 3. That means an existing Express 3 app will not work if you update the Express version in its dependencies.

This article covers:

- Changes in Express 4.
- An example of migrating an Express 3 app to Express 4.
- Upgrading to the Express 4 app generator.

Changes in Express 4

The main changes in Express 4 are:

- Changes to Express core and middleware system: The dependency on Connect and built-in middleware were removed, so you must add middleware yourself.
- Changes to the routing system.
- Various other changes.

See also:

- New features in 4.x.
- Migrating from 3.x to 4.x.

Changes to Express core and middleware system

Express 4 no longer depends on Connect, and removes all built-in middleware from its core, except `express.static`. This means Express is now an independent routing and middleware web framework, and Express versioning and releases are not affected by middleware updates.

With the built-in middleware gone, you must explicitly add all the middleware required to run your app. Simply follow these steps:

1. Install the module: `npm install --save <module-name>`
2. In your app, require the module: `require('module-name')`
3. Use the module according to its documentation: `app.use( ... )`

The following table lists Express 3 middleware and their counterparts in Express 4.

Express 3	Express 4
`express.bodyParser`	body-parser + multer
`express.compress`	compression
`express.cookieSession`	cookie-session

`express.cookieParser`	cookie-parser
`express.logger`	morgan
`express.session`	express-session
`express.favicon`	serve-favicon
`express.responseTime`	response-time
`express.errorHandler`	errorhandler
`express.methodOverride`	method-override
`express.timeout`	connect-timeout
`express.vhost`	vhost
`express.csrf`	csurf
`express.directory`	serve-index
`express.static`	serve-static

Here is the complete list of Express 4 middleware.

In most cases, you can simply replace the old version 3 middleware with its Express 4 counterpart. For details, see the module documentation in GitHub.

`app.use` accepts parameters

In version 4 you can now load middleware on a path with a variable parameter and read the parameter value from the route handler. For example:

```
app.use('/book/:id', function(req, res, next) {
  console.log('ID:', req.params.id);
  next();
});
```

The routing system

Apps now implicitly load routing middleware, so you no longer have to worry about the order in which middleware is loaded with respect to the `router` middleware.

The way you define routes is unchanged, but the routing system has two new features to help organize your routes:

- A new method, `app.route()`, to create chainable route handlers for a route path.
- A new class, `express.Router`, to create modular mountable route handlers.

`app.route()` method

The new `app.route()` method enables you to create chainable route handlers for a route path. Since the path is specified in a single location, it helps to create modular routes and reduce redundancy and typos. For more information on routes, see `Router()` documentation.

Here is an example of chained route handlers defined using `app.route()`.

```
app.route('/book')
  .get(function(req, res) {
    res.send('Get a random book');
  })
  .post(function(req, res) {
    res.send('Add a book');
  })
  .put(function(req, res) {
    res.send('Update the book');
  });
```

`express.Router` class

The other feature to help organize routes is a new class, `express.Router`, that you can use to create modular mountable route handlers. A `Router`instance is a complete middleware and routing system; for this reason it is often referred to as a “mini-app”.

The following example creates a router as a module, loads a middleware in it, defines some routes, and mounts it on a path on the main app.

Create a router file named `birds.js` in the app directory, with the following content:

```
var express = require('express');
var router = express.Router();

// middleware specific to this router
router.use(function timeLog(req, res, next) {
  console.log('Time: ', Date.now());
  next();
});
// define the home page route
router.get('/', function(req, res) {
  res.send('Birds home page');
});
// define the about route
router.get('/about', function(req, res) {
  res.send('About birds');
});

module.exports = router;
```

Then, load the router module in the app:

```
var birds = require('./birds');
...
```

```
app.use('/birds', birds);
```

The app will now be able to handle requests to `/birds` and `/birds/about`, along with calling the `timeLog` middleware specific to the route.

Other changes

The following table lists other small but important changes in Express 4.

Object	Description
Node	Express 4 requires Node 0.10.x or later and has dropped support for Node 0.8.x.
`http.createServer()`	The `http` module is no longer needed, unless you need to directly work with it (socket.io/SPDY/HTTPS). The app can be started using `app.listen()`.
`app.configure()`	`app.configure()` has been removed. Use `process.env.NODE_ENV` or `app.get('env')` to detect the environment and configure the app accordingly.
`json spaces`	The `json spaces` application property is disabled by default in Express 4.
`req.accepted()`	Use `req.accepts()`, `req.acceptsEncodings()`, `req.acceptsCharsets()`, and `req.acceptsLanguages()`.
`res.location()`	No longer resolves relative URLs.
`req.params`	Was an array; now an object.
`res.locals`	Was a function; now an object.
`res.headerSent`	Changed to `res.headersSent`.
`app.route`	Now available as `app.mountpath`.
`res.on('header')`	Removed.
`res.charset`	Removed.

`res.setHeader('Set-Cookie', val)`	Functionality is now limited to setting the basic cookie value. Use `res.cookie()` for added functionality.

Example app migration

Here is an example of migrating an Express 3 application to Express 4. The files of interest are `app.js` and `package.json`.

Version 3 app

`app.js`

Consider an Express v.3 application with the following `app.js` file:

```
var express = require('express');
var routes = require('./routes');
var user = require('./routes/user');
var http = require('http');
var path = require('path');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.methodOverride());
app.use(express.session({ secret: 'your secret here' }));
app.use(express.bodyParser());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get('/', routes.index);
app.get('/users', user.list);

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

`package.json`

The accompanying version 3 `package.json` file might look something like this:

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "3.12.0",
    "jade": "*"
  }
}
```

Process

Begin the migration process by installing the required middleware for the Express 4 app and updating Express and Jade to their respective latest version with the following command:

```
$ npm install serve-favicon morgan method-override express-session body-p
```

Make the following changes to `app.js`:

1. The built-in Express middleware `express.favicon`, `express.logger`, `express.methodOverride`, `express.session`, `express.bodyParser` and `express.errorHandler` are no longer available on thee `xpress` object. You must install their alternatives manually and load them in the app.
2. You no longer need to load `app.router`. It is not a valid Express 4 app object, so remove `app.use(app.router);`
3. Make sure the middleware are loaded in the right order - load `errorHandler` after loading the app routes.

Version 4 app

`package.json`

Running the above `npm` command will update `package.json` as follows:

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
```

```
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "body-parser": "^1.5.2",
    "errorhandler": "^1.1.1",
    "express": "^4.8.0",
    "express-session": "^1.7.2",
    "jade": "^1.5.0",
    "method-override": "^2.1.2",
    "morgan": "^1.2.2",
    "multer": "^0.1.3",
    "serve-favicon": "^2.0.1"
  }
}
```

app.js

Then, remove invalid code, load the required middleware, and make other changes as necessary. Then `app.js` will look like this:

```
var http = require('http');
var express = require('express');
var routes = require('./routes');
var user = require('./routes/user');
var path = require('path');

var favicon = require('serve-favicon');
var logger = require('morgan');
var methodOverride = require('method-override');
var session = require('express-session');
var bodyParser = require('body-parser');
var multer = require('multer');
var errorHandler = require('errorhandler');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');
app.use(favicon(__dirname + '/public/favicon.ico'));
app.use(logger('dev'));
app.use(methodOverride());
app.use(session({ resave: true,
                  saveUninitialized: true,
```

```
                  secret: 'uwotm8' }));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(multer());
app.use(express.static(path.join(__dirname, 'public')));

app.get('/', routes.index);
app.get('/users', user.list);

// error handling middleware should be loaded after the loading the route
if ('development' == app.get('env')) {
  app.use(errorHandler());
}

var server = http.createServer(app);
server.listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Unless you need to work directly with the `http` module (socket.io/SPDY/HTTPS), loading it is not required, and the app can be simple started this way:

```
app.listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Run the app

With that, the migration process is complete, and the app is now an Express 4 app. To confirm, start the app with the following command:

```
$ node .
```

Load http://localhost:3000 and see the home page being rendered by Express 4.

Upgrading to the Express 4 app generator

The command-line tool to generate an Express app is still `express`, but to upgrade to the new version, you must uninstall the Express 3 app generator and then install the new `express-generator`.

Installing

If you already have the Express 3 app generator installed on your system, you must uninstall it as follows:

```
$ npm uninstall -g express
```

Depending on how your file and directory privileges are configured, you may need to run this command with `sudo`.

Now install the new generator:

```
$ npm install -g express-generator
```

Depending on how your file and directory privileges are configured, you may need to run this command with `sudo`.

Now the `express` command on your system is updated to the Express 4 generator.

Changes to the app generator

Command options and use largely remain the same, with the following exceptions:

- Removed the `--sessions` option.
- Removed the `--jshtml` option.
- Added the `--hogan` option to support Hogan.js.

Example

Execute the following command to create an Express 4 app:

```
$ express app4
```

If you look at the contents of `app4/app.js`, you will notice that all the middleware (except `express.static`) required for the app are loaded as independent modules and the `router` middleware is no longer explicitly loaded in the app.

You will also notice that the `app.js` file is now a Node module, compared to the standalone app generated by the old generator.

After installing the dependencies, start the app using the following command:

```
$ npm start
```

If you peek at the npm start script in `package.json` file, you will notice that the actual command that starts the app is `node ./bin/www`, which used to be `node app.js` in Express 3.

Since the `app.js` file generated by the Express 4 generator is now a Node module, it can no longer be started independently as an app (unless you modify the code). It has to be to be loaded in a Node file and started via the Node file. The Node file is `./bin/www` in this case.

Neither the `bin` directory nor the extensionless `www` file is mandatory for creating an Express app or starting the app. They are just suggestions by the generator, so feel free to modify them to suit your needs.

To get rid of the `www` directory and keep things the “Express 3 way”, delete the line that says `module.exports = app;` at the end of `app.js`, and paste the following code in its place.

```
app.set('port', process.env.PORT || 3000);
```

```
var server = app.listen(app.get('port'), function() {
  debug('Express server listening on port ' + server.address().port);
});
```

Make sure to load the `debug` module at the top of `app.js` with the following code.

```
var debug = require('debug')('app4');
```

Next, change `"start": "node ./bin/www"` in the `package.json` file to `"start": "node app.js"`.

With that, you just moved the functionality of `./bin/www` back to `app.js`. Not that it is recommended, but the exercise helps to understand how `./bin/www` works, and why `app.js` won't start on its own anymore.

Command Line

Installation

via npm:

```
$ npm install jade --global
```

Usage

```
$ jade [options] [dir|file ...]
```

Options

```
-h, --help              output usage information
-V, --version           output the version number
-O, --obj <path|str>    JavaScript options object or JSON file co
-o, --out <dir>         output the compiled html to <dir>
-p, --path <path>       filename used to resolve includes
-P, --pretty            compile pretty html output
-c, --client            compile function for client-side runtime.
-n, --name <str>        the name of the compiled template (requi
-D, --no-debug          compile without debugging (smaller functi
-w, --watch             watch files for changes and automatically
-E, --extension <ext>   specify the output file extension
--name-after-file       name the template after the last section
                        (requires --client and overriden by --nam
--doctype <str>         specify the doctype on the command line (
                        is not specified by the template)
```

Examples

Translate jade the templates dir:

```
$ jade templates
```

Create {foo,bar}.html

```
$ jade {foo,bar}.jade
```

Jade over stdio

```
$ jade < my.jade > my.html
```

Jade over stdio

```
$ echo "h1 Jade!" | jade
```

foo, bar dirs rendering to /tmp

```
$ jade foo bar --out /tmp
```

Tutorial

Introduction

> This tutorial is a work in progress. Once it's finished it will become a fully fledged getting started guide.

Welcome to the Jade templating engine. Jade is designed primarily for server side templating in node.js, however it can be used in many other environments. It is only intended to produce XML like documents (HTML, RSS etc.) so don't use it to create plain text/markdown/CSS/whatever documents.

This tutorial will take you through:

The Basics

Jade can be used just as a short hand for HTML. This section covers everything you need to know to do that.

Creating Simple Tags

Jade is whitespace sensitive, so there's no need to close your tags; Jade does that for you. You can also nest tags within other tags just by indenting them.

```
div
  address
  i
  strong
```

```
<div>
  <address></address><i></i><strong></strong>
</div>
```

Putting Text Inside your Tags

By default, the content of a tag is parsed as more jade. There are three ways to put plain text inside a tag.

```
h1 Welcome to Jade
p
  | Text can be included in a number of
  | different ways.
p.
  This way is shortest if you need big
  blocks of text spanning multiple
  lines.
```

```
<h1>Welcome to Jade</h1>
<p>
  Text can be included in a number of
  different ways.
</p>
<p>
  This way is shortest if you need big
  blocks of text spanning multiple
  lines.
</p>
```

Adding Attributes to your Tags

To add attributes you put them in parentheses after the tag name, separated by an optional comma.

```
h1(id="title") Welcome to Jade
button(class="btn", data-action="bea").
  Be Awesome
```

```
<h1 id="title">Welcome to Jade</h1>
<button data-action="bea" class="btn">Be Awesome</button>
```

IDs and Classes

Adding IDs and Classes is super common, so we made it super easy to do by adding a simple short hand. The syntax is just like that of CSS selectors:

```
h1#title Welcome to Jade
button.btn(data-action="bea") Awesome
```

```
<h1 id="title">Welcome to Jade</h1>
<button data-action="bea" class="btn">Awesome</button>
```

JavaScript

Jade is much more than just a short hand for HTML. It also has features that let you build dynamic templates.

Outputing Text

You can output raw text from JavaScript variables. Jade will also helpfully filter the text for you so it's safe from nasty HTML injection attacks.

```
var jade = require('jade');
var fn = jade.compile(jadeTemplate);
var htmlOutput = fn({
  maintainer: {
    name: 'Forbes Lindesay',
    twitter: '@ForbesLindesay',
    blog: 'forbeslindesay.co.uk'
  }
});
```

```
h1
  | Maintainer:
  = ' ' + maintainer.name
table
  tr
    td Twitter
    td= maintainer.twitter
  tr
    td Blog
    td= maintainer.blog
```

```
<h1>Maintainer: Forbes Lindesay</h1>
<table>
  <tr>
```

```
    <td>Twitter</td>
    <td>@ForbesLindesay</td>
  </tr>
  <tr>
    <td>Blog</td>
    <td>forbeslindesay.co.uk</td>
  </tr>
</table>
```

If you *don't* want Jade to filter your output, use != instead of =.

Setting Attributes

Setting attributes to JavaScript values requires no extra work:

```
h1(name=maintainer.name)
  | Maintainer:
  = ' ' + maintainer.name
table
  tr
    td(style='width: '+(100/2)+'%').
      Twitter
    td= maintainer.twitter
  tr
    td(style='width: '+(100/2)+'%').
      Blog
    td= maintainer.blog
```

```
<h1 name="Forbes Lindesay">
  Maintainer: Forbes Lindesay
</h1>
<table>
  <tr>
    <td style="width: 50%">Twitter</td>
    <td>@ForbesLindesay</td>
  </tr>
  <tr>
    <td style="width: 50%">Blog</td>
    <td>forbeslindesay.co.uk</td>
  </tr>
</table>
```

Loops and Conditionals

You can use if statements to decide what to display depending on various factors - maybe a user is logged in or not, or some content exists or not, or a combination of factors.

Jade's if statements are almost exactly like those present in JavaScript, except the parentheses are optional, and you don't need braces!

```
- var user = { name: 'John' }
if user
  div.welcomebox
    // Filtered inline output
    p.
```

```
      Welcome, #{user.name}
else
  div.loginbox
    form(name="login", action="/login", method="post")
      input(type="text", name="user")
      input(type="password", name="pass")
      input(type="submit", value="login")
```

```
<div class="welcomebox">
  <!-- Filtered inline output-->
  <p>Welcome, John</p>
</div>
```

Advanced Templating

Extends & Blocks

Includes

Mixins

Filters

To Be Continued....

The jade source for this file can be viewed (and edited/improved) onGitHub