6 Working with Web Frameworks

This chapter covers the following topics

- topics
- as
- bullets
- here

additional things to mention

- prod env
- using frameworks for microservices
- ref. input vaslidation in sec
- ref hardening in sec.

maybe add recipe - creating a plugin (e.g middleware) for custom functionality? or rather recommend against? something about interacting with a model...

Introduction

Node core supplies a strong set of well balanced primitives that allow us to create all manner of systems, for service-based architectures, to realtime data server, to robotics there's just enough in the Node core for purpose built libraries to arise from the Node community and ecosystem.

Building web site infrastructure is a very common use case for Node,and several high profile web frameworks have grown to become staple choices for creating web applications.

In this chapter we're going to explore the popular frameworks, and look at common tasks such as implementing server logging, sessions, authentication and validation.

Creating an Express Web App

Express has long been the most popular choice of web framework, which is unsurprising since it was the first Node web framework of a high enough quality for mass consumption whilst also drawing from familiar paradigms presented in the

Sinatra web framework for Ruby on Rails.

In this recipe we'll look at how to put together an Express web application.

Getting Ready

Let's create a folder called `app`, initialize it as a package, and install `express`:

```
$ mkdir app
$ cd app
$ npm install --save express
```

How to do it

Let's start by creating a few files:

```
$ touch index.js
$ mkdir routes public
$ touch routes/index.js
$ touch public/styles.css
```

Now let's open the `index.js` file in our favorite editor, and prepare to write some code.

At the top of the file we'll load the following dependencies:

```
const {join} = require('path')
const express = require('express')
const index = require('./routes/index')
```

We'll write the `routes/index.js` file shortly, but for now let's continue writing the `index.js` file. Next we'll instantiate an Express object, which we'll call `app` while also setting up some configuration:

```
const app = express()
const dev = process.env.NODE_ENV !== 'production'
const port = process.env.PORT || 3000
```

Next we'll register some Express middleware, like so:

```
if (dev) {
  app.use(express.static(join(__dirname, 'public')))
}
```

And mount our `index` route at the `/` path:

```
app.use('/', index)
```

We'll finish off the `index.js` file by telling the Express application to listen on the `port` which we defined earlier.

```
app.listen(port, () => {
  console.log(`Server listening on port ${port}`)
})
```

Our `index.js` file is requiring `./routes/index`, so let's write the `routes/index.js` file:

```
const {Router} = require('express')
const router = Router()

router.get('/', function (req, res, next) {
  const title = 'Express'
  res.send(`
    <html>
      <head>
        <title> ${title} </title>
        <link rel="stylesheet" href="styles.css">
      </head>
      <body>
        <h1> ${title} </h1>
        <p> Welcome to ${title} </p>
      </body>
    </html>
  `)
  next()
})

module.exports = router
```

Now for a little bit of style. Let's complete the picture with a very simply CSS file in `public/styles.css`

```
body {
  padding: 50px;
  font: 14px "Lucida Grande", Helvetica, Arial, sans-serif;
}
```

We should be able to run our server with:

```
$ node index.js
```

If we access our server at http://localhost:3000 in a browser, we should see something like the following image:

How it works

Express is a framework built on top of Node's core `http` (and `https` when relevant) module.

The core `http` module 

> See **Chapter 5 Wielding Web Protocols** for more on the Node's core `http` module.

Express decorates the `req` (`http.IncomingMessage`) and `res` (`http.ServerResponse`) objects which are passed to the `http.createServer` request handler function.

To explain this using code, at a very basic level Express essentially performs the following internally:

```
const http = require('http')
http.createServer((req, res) => {
  /* add extra methods and properties to req and res  */
}))
```

When we call the `express` function, it returns an instance which we called `app` which represents our Express server.

The `app.use` function allows us to register "middleware" which at a fundamental level is a function that is called from the same `http.createServer` request

handling function.

Again, for a pseudo-code explanation:

```
const http = require('http')
http.createServer((req, res) => {
  /* call the middleware registered with app.use */
  /* wait for each piece of middleware to finish
     before calling the next (wait for the next cb) */
}))
```

Each piece of middleware may call methods on `req` and `res`, and extend the objects with additional methods or properties.

The `express.static` method comes bundled with Express. It returns a middleware function which is passed into `app.use`. This function will attempt to locate a file based on supplied configuration (in our case, we set the root directory to the `public` folder) for given route. Then it will create a write stream from the file and stream it to the request object (`req`). If it can't find a file, or there's some other error, it will pass an error object to the middleware `next` callback, to allow middleware further down the middleware stack to handle the error.

We only use the static middleware in development mode (based on the value of the `dev` reference, which is assigned based on whether the `NODE_ENV` environment variable is set to "production"). This assumes a production scenario where a reverse proxy (such as nginx or apache) (or, even better a CDN) handles static file serving. Whilst Node has come a long way in recent years, Node's strength remains in generating dynamic content - it still doesn't usually make sense to use it for static assets in production.

The order of middleware is significant. For instance, if we register static file handling middleware before route handling middleware in the case of name collision (where a route could apply to a file or a dynamic route), the file handling middleware will take precedence. However, if the route handling middleware is first, the dynamic route takes will serve the request first instead.

The `app.use` function can accept a string as the first argument, which determines a "mount point" for a piece of middleware. This means instead of the middleware applying to all incoming requests it will only be called when there is a route match.

Route handlers are essentially the same mounted middleware, but are constructed with Express' `Router` utility for cleaner encapsulation. In our `routes/index.js`

file we create a router object which we called `router`. Router objects have methods which correspond to the HTTP verbs (such as GET, PUT, POST, PATCH, DELETE) in relevant specification (rfc7231).

Most commonly we would use `GET` and `POST` for web facing applications. We use `router.get` to register a route (`/`), with and supply a route handling function (which is technically also middleware).

In our route handler, we pass `res.send` a string of HTML content to respond to the client.

The `res.send` method is added by Express, it's the equivalent of `res.end` but with additional features such as content type detection.

We export the `router` instance from the `routes/index.js`, then load it into the `index.js` file and pass it to `app.use` (as the second argument, after a mount point string argument (`/`)).

The `router` instance is itself, middleware. It's a function that accepts `req`, `res` and `next` arguments. When called, it checks its internal state based on any routers registered (via `get` etcetera), and responds accordingly.

The function we pass to `router.get` can also take a `next` callback function. We ignored the `next` callback function in our case (we didn't define it in the route handling functions parameters), because this route handler is a terminal point - there is nothing else to be done after sending the content. However, in other scenarios there may be cause to use the `next` callback and even pass it an error to propagate request handling the next piece of middleware (or route middleware, since a route registering method (like `get`) can be passed multiple subsequent route handling functions).

At the end of `index.js` we call `app.listen` and pass it a callback function. This will in turn call the `listen` method on the core `http` server instance which Express has created internally, and pass our supplied callback to it. Our callback simply logs that the server is now listening on the given port.

What About SSL

> While Express can work with HTTPS, we recommend that the general approach should be to terminate SSL at the load balancer (or reverse proxy) for optimal efficiency.

There's more

Let's explore some more of the functionality offered by Express.

Production

Our Express server defines a `dev` reference, based on the value of the `NODE_ENV` environment variable. This is a standard convention in Node. In fact Express will behave differently when `NODE_ENV` is set to production - for instance views will be cached in memory.

We can check out production mode with

```
$ NODE_ENV=production node index.js
```

We should notice this removes styling from our app. This is because we only serve static assets in development mode, and the `<link>` tags in our views will be generating 404 errors in attempts to fetch the `public/styles.css` file.

Route Parameters and POST requests

> ### CAUTION! 
>
> This example is for demonstration purposes only! Never place user input directly into HTML output in production without sanitizing it first. Otherwise, we make ourselves vulnerable to XSS attacks. See **Chapter 8 Dealing with Security** for details

Let's copy our `app` folder to `params-postable-app`, and then install the `body-parser` middleware module:

```
$ cp -fr app params-postable-app
$ cd params-postable-app
$ npm install --save body-parser
```

In the `index.js` file, we'll load the middleware and use it.

At the top of `index.js` file we'll require the body parser middleware like so:

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

Then we'll use it, just above the `port` assignment we'll add:

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

> ### Use extended: false 
>
> We set `extended` to `false` because the `qs` module which provides the parsing functionality for `bodyParse.urlencoded` has options which could (without explicit validation) allow for a Denial of Service attack. See the **Anticipating Malicious** in **Chapter 8 Dealing with Security** for details.

Now in `routes/index.js` we'll alter our original GET route handler to the following:

```
router.get('/:name?', function (req, res) {
  const title = 'Express'
  const name = req.params.name
  res.send(`
    <html>
      <head>
        <title> ${title} </title>
        <link rel="stylesheet" href="styles.css">
      </head>
      <body>
        <h1> ${title} </h1>
        <p> Welcome to ${title}${name ? `, ${name}.` : ''} </p>
        <form method=POST action=data>
        Name: <input name=name> <input type=submit>
        </form>
      </body>
    </html>
  `)
})
```

We're using Express' placeholder syntax here to define a route parameter called `name`. The question mark in the route string indicates that the parameter is optional (which means the original functionality for the `/` route is unaltered). If the `name` parameter is present, we add it into our HTML content.

We've also added a form which will perform a POST request to the `/data` route. By default it will be of type `application/x-www-form-urlencoded` which is why we use the `urlencoded` method on the `body-parser` middleware.

Now to the bottom of `routes/index.js` we'll add a POST route handler:

```
router.post('/data', function (req, res) {
  res.redirect(`/${req.body.name}`)
})
```

Now if we start our server:

```
$ node index.js
```

Then load navigate our browser to http://localhost:3000 we should be able to supply a name to the input box, press the submit button and subsequently see our name in the URL bar and on the page.

> **CAUTION!** 
>
> This example is for demonstration purposes only! Never place user input directly into HTML output in production without sanitizing it first. Otherwise, we make ourselves vulnerable to XSS attacks. See **Chapter 8 Dealing with Security** for details

Creating Middleware

Middleware (functions which are passed to `app.use`) is a fundamental concept in Express (and other web frameworks).

If we need some custom functionality (for instance, business logic related), we can create our middleware.

Let's copy the `app` folder from our main recipe to the `custom-middleware-app` and create a middleware folder with an `answer.js` file:

```
$ cp -fr app custom-middleware-app
$ cd custom-middleware-app
$ mkdir middleware
$ touch middleware/answer.js
```

Now we'll place the following code in `middleware/answer.js` :

```
module.exports = answer
```

```
function answer () {
  return (req, res, next) => {
    res.setHeader('X-Answer', 42)
    next()
  }
}
```

Finally we need to modify the `index.js` file in two places. First at the top, we add our answer middleware to the dependency loading section:

```
const {join} = require('path')
const express = require('express')
const index = require('./routes/index')
const answer = require('./middleware/answer')
```

Then we can place our `answer` middleware at the top of the middleware section, just underneath the `port` assignment:

```
app.use(answer())
```

Now if we start our server:

```
$ node index.js
```

And hit the server with `curl -I` to make a `HEAD` request and view headers:

```
$ curl -I http://localhost:3000
```

We should see output similar to:

```
HTTP/1.1 200 OK
X-Powered-By: Express
X-Answer: 42
Content-Type: text/html; charset=utf-8
Content-Length: 226
ETag: W/"e2-olBsieaMz1W9hKepvcsDX9In8pw"
Date: Thu, 13 Apr 2017 19:40:01 GMT
Connection: keep-alive
```

With our `X-Answer` present.

Middleware isn't just for setting custom headers, there's a vast range of possibilities, parsing the body of a request and session handling to implementing custom protocols on top of HTTP.

See also

- *Creating a Web Server* in **Chapter 5 Wielding Web Protocols**
- *Anticipating Malicious Input* in **Chapter 8 Dealing with Security**
- *Guarding Against Cross Site Scripting (XSS)* in **Chapter 8 Dealing with Security**
- *Adding a View Layer* in this Chapter
- *Implementing Authentication* in this Chapter

Creating a Hapi Web App

Hapi is a fairly recent addition to the "enterprise" web framework offerings. The Hapi web framework has a reputation for stability, but tends to perform slower (for instance, see https://raygun.com/blog/node-performance/) whilst also requiring more boilerplate than alternatives. With a contrasting philosophy and approach to Express (and other "middleware" centric frameworks like Koa and Restify) Hapi may be better suited to certain scenarios and preferences.

In this recipe, we'll create a simple Hapi web application.

Getting Ready

Let's create a folder called `app`, initialize it as a package, and install `hapi` and `inert`:

```
$ mkdir app
$ cd app
$ npm install --save hapi inert
```

How to do it

Let's start by creating a few files:

```
$ touch index.js
$ mkdir routes public
```

```
$ touch routes/index.js
$ touch routes/dev-static.js
$ touch public/styles.css
```

We'll begin by populating the `index.js` file.

At the top of `index.js` let's require some dependencies:

```
const hapi = require('hapi')
const inert = require('inert')
const routes = {
  index: require('./routes/index'),
  devStatic: require('./routes/dev-static')
}
```

Now we'll instantiate a Hapi server, and set up `dev` and `port` constants:

```
const dev = process.env.NODE_ENV !== 'production'
const port = process.env.PORT || 3000

const server = new hapi.Server()
```

Next we'll supply Hapi server connection configuration:

```
server.connection({
  host: 'localhost',
  port: port
})
```

We're only going to use `inert` (a static file handling Hapi plugin) in development mode, so let's conditionally register the `inert` plugin, like so:

```
if (dev) server.register(inert, start)
else start()
```

We'll finish off `index.js` by supplying the `start` function we just referenced:

```
function start (err) {
  if (err) throw err

  routes.index(server)
```

```
  if (dev) routes.devStatic(server)

  server.start((err) => {
    if (err) throw err
    console.log(`Server listening on port ${port}`)
  })
}
```

This invokes our route handlers, and calls `server.start`.

Our `index.js` file is relying on two other files, `routes/index.js` and `routes/devStatic.js`.

Let's write the `routes/index.js` file:

```
module.exports = index

function index (server) {
  server.route({
    method: 'GET',
    path: '/',
    handler: function (request, reply) {
      const title = 'Hapi'
      reply(`
        <html>
          <head>
            <title> ${title} </title>
            <link rel="stylesheet" href="styles.css">
          </head>
          <body>
            <h1> ${title} </h1>
            <p> Welcome to ${title} </p>
          </body>
        </html>
      `)
    }
  })
}
```

And now the `routes/dev-static.js` file:

```
module.exports = devStatic

function devStatic (server) {
  server.route({
    method: 'GET',
    path: '/{param*}',
```

```
    handler: {
      directory: {
        path: 'public'
      }
    }
  })
}
```

Finally we need to supply the `public/styles.css` file:

```
body {
  padding: 50px;
  font: 14px "Lucida Grande", Helvetica, Arial, sans-serif;
}
```

Now we can start our server:

```
$ node index.js
```

If we navigate to http://localhost:3000 in our browser, we should see something like the following:

How it works

After we create a `hapi.Server` instance (which we named `server`) we call the `connection` method. This will register the settings we pass in a list of connections.

When we later call `server.start` Hapi creates an `http` (or `https` if a `tls` object is supplied with `key` and `cert` buffer values). Unlike Express or Koa, Hapi allows for multiple connections, which in turn will create multiple core `http` server instances (with Express or Koa we would simply instantiate multiple instances of Express/Koa and reuse any routes/middleware between them as required).

In our case we call `server.connection` once, as a result, upon calling `server.start`, a single `http` server is created which listens to port `3000` (unless otherwise set in the `PORT` environment variable, according to how we've defined the `port` constant).

We use the separate `inert` Hapi plugin to serve static files in development mode. The `server.register` function can take a single plugin, or an array of plugins. We

currently only have one plugin (and only when `dev` is `true`), so we pass the `inert` plugin to `server.register` and supply the `start` function as the second argument. The second argument to `server.register` is a callback, which is triggered once plugins are loaded. If our server was in production mode (that is, if the `NODE_ENV` environment variable was set to production), then we simply call `start` directly as we have no other plugins to register.

We have two routes files, `routes/index.js` and `routes/dev-static.js`. These files simply export a function that takes the `server` object which we created in `index.js`.

In both `routes/index.js` and `routes/dev-static.js` we call `server.route` to register a new route with Hapi.

The `server.route` method takes an object which describes the route. We supply an object with `method`, `path` and `handler` properties in both cases.

> ### Route options 
>
> In addition to the required three properties (`method`, `path`, `handler`), another possible key on the settings object passed to `server.route` is the `config` property. This allows for a vast amount of behavioral tweaks both for internal Hapi and for additional plugins. See https://hapijs.com/api#route-options for more information.

In `routes/index.js` we set the `method` to `GET`, the `path` to `/` (because it's our index route) and the `handler` to a function which accepts `request` and `reply` arguments.

The `request` and `reply` parameters whilst analogous to the parameters passed to the `http.createServer` request handler function (often callef `req` and `res`) are quite distinct. Unlike Express which *decorates* `req` and `res`, Hapi creates separate abstractions (`request` and `reply`) which interface with `req` and `res` internally.

In the `handler` function, we call `reply` as a function, passing it our HTML content.

In `routes/dev-static.js` the `path` property is using route parametization with segment globbing to allow match any route. In our case the use of `param` in `/{param*}` is irrelevant. It could be named anything at all, this is just a necessity to get the required functionality. The asterisk following `param` will cause any number

of route segments (parts of the route separated by `/`) to match. Instead of a function, the `handler` in our `routes/dev-static.js` file is an object with `directory` set to an object containing a `path` property which points to our `public` folder. This is as route configuration settings by the `inert` plugin.

Our `start` function checks for any error (rethrowing if there is one) and passes the `server` object to the the `routes.index` and `routes.devStatic` functions, then calls `server.start` which causes Hapi to create the `http` server and bind to the host and port supplied to `server.connection` earlier on. The `server.start` method also takes a callback function, which is called once all servers have been bound to their respective hosts and ports. The callback we supply checks and rethrows an error, and logs out confirmation message that the server is now up.

There's more

Let's explore some more of Hapi's functionality.

Creating a plugin

Let's copy the `app` folder from our main recipe to the `custom-plugin-app` and create a `plugins` folder with an `answer.js` file:

```
$ cp -fr app custom-plugin-app
$ cd custom-plugin-app
$ mkdir plugins
$ touch plugins/answer.js
```

We'll make the contents of `plugins/answer.js` look like so:

```
module.exports = answer

function answer (server, options, next) {
  server.ext('onPreResponse', (request, reply) => {
    request.response.header('X-Answer', 42)
    reply.continue()
  })
  next()
}

answer.attributes = {name: 'answer'}
```

The `next` callback is supplied to allow for any asynchronous activity. We call it to

let Hapi know we've finished setting up the plugin. Under the hood Hapi would call the `server.register` callback once all the plugins had called their respective `next` callback functions.

> ### Events and Extensions 
>
> There are a variety of server events (which we can listen to with `server.on`) and "extensions". Extensions are very similar to server events, except we use `server.ext` to listen to them and must call `reply.continue()` when we're ready to proceed. See https://hapijs.com/api#request-lifecycle as a starting point to learn more.

We use the `onPreResponse` extension (which is very much like an event) to add our custom header. The `onPreResponse` extension is the *only* place we can register headers (the `onRequest` extension is too early and the `response` event is too late).

We'll add the answer plugin near the top of the `index.js` file like so:

```
const answer = require('./plugins/answer')
```

Then at the bottom of `index.js` we'll modify the boot up code to the following:

```
const plugins = dev ? [answer, inert] : [answer]
server.register(plugins, start)

function start (err) {
  if (err) throw err

  routes.index(server)

  if (dev) routes.devStatic(server)

  server.start((err) => {
    if (err) throw err
    console.log(`Server listening on port ${port}`)
  })
}
```

Label Selecting

Each Hapi connection can be labelled with one or more identifiers, which can in turn be used to conditionally register plugins and define routes or perform other

connection specific tasks.

Let's copy the `app` folder from our main recipe to `label-app`:

```
$ node index.js
```

Now we'll alter our `index.js` to the following:

```
const hapi = require('hapi')
const inert = require('inert')
const routes = {
  index: require('./routes/index'),
  devStatic: require('./routes/dev-static')
}

const devPort = process.env.DEV_PORT || 3000
const prodPort = process.env.PORT || 8080

const server = new hapi.Server()

server.connection({
  host: 'localhost',
  port: devPort,
  labels: ['dev', 'staging']
})

server.connection({
  host: '0.0.0.0',
  port: prodPort,
  labels: ['prod']
})

server.register({
  register: inert,
  select: ['dev', 'staging']
}, start)

function start (err) {
  if (err) throw err

  routes.index(server)

  routes.devStatic(server)

  server.start((err) => {
    if (err) throw err
    console.log(`Dev/Staging server listening on port ${devPort}`)
    console.log(`Prod server listening on port ${prodPort}`)
```

```
  })
}
```

We removed the `dev` constant as we're using Hapi labels to handle conditional environment logic. We now have two port constants, `devPort` and `prodPort` and we use them to create two server connections. The first listens on the local loopback interface (`localhost`) as normal, on the `devPort` which defaults to port `3000` . The second listens on the public interface (`0.0.0.0`), on the `prodPort` default to port `8080` .

We add `label` property to each connection, on the first we supply an array of `['dev', 'staging']` and to the second a string containing `prod` . This means we can treat our development connection as a staging connection when it makes sense - for instance in our case we're using `inert` for static file hosting on both development and staging but in production we assume a separate layer in the deployment architecture is handling this.

We've removed the `if` statement checking for `dev` and have instead housed the `inert` plugin in an object as we pass it to `server.register` . Passing `inert` directly or passing as the `register` property of an object are equivalent. However passing it inside an object allows us to supply other configuration. In this case add a `select` property which is set to `['dev', 'staging']` . This means the `inert` plugin will only register on the development connection, but will not be present on the production connection.

In the `start` function we've also removed the `if(dev)` statemenet preceeding our call to `routes.devStatic` . We need to modify `routes/dev-static.js` so that the static route handler is only registered for the development connection.

Let's change `routes/dev-static.js` to the following:

```
module.exports = devStatic

function devStatic (server) {
  server.select(['dev', 'staging']).route({
    method: 'GET',
    path: '/{param*}',
    handler: {
      directory: {
        path: 'public'
      }
    }
  })
```

```
}
```

We've added in call to `server.select`. When we call `route` on the resulting object, the route is only applied to connections that match the supplied labels.

We can confirm that our changes are working by running our server:

```
$ node index.js
```

And using curl to check whether the development server delivers static assets (which it should) and the production server responds with 404:

```
$ curl http://localhost:3000/styles.css
```

This should respond with the contents of `public/styles.css`.

However the following should respond with `{"statusCode":404,"error":"Not Found"}`:

```
$ curl http://localhost:8080/styles.css
```

This approach does uses more ports than necessary in production, may lead to reduced performance and does beg some security questions. However, we could side step these problems while still getting the benefits of labelling (in this specific case) by reintroducing the `dev` constant and only conditionally creating the connections based on whether `dev` is true or false.

For example:

```
const dev = process.env.NODE_ENV !== 'production'

if (dev) server.connection({
  host: 'localhost',
  port: devPort,
  labels: ['dev', 'staging']
})

if (!dev) server.connection({
  host: '0.0.0.0',
  port: prodPort,
  labels: 'prod'
})
```

However this would require a modification to any conditional routing, since there Hapi requires at least one connection before a route can be added. For instance in our case we would have to modify the top of the function exported from `routes/dev-static.js` like so:

```
function devStatic (server) {
  const devServer = server.select(['dev', 'staging'])
  if (!devServer.connections.length) return
  devServer.route({ /* ... etc ... */ })
}
```

See also

- TBD

Creating a Koa Web App

Getting Ready

Let's create a folder called `app`, initialize it as a package, and install `express`:

```
$ mkdir app
$ cd app
$ npm install --save koa koa-router koa-static
```

How to do it

Let's start by creating a few files:

```
$ touch index.js
$ mkdir routes public
$ touch routes/index.js
$ touch public/styles.css
```

Let's kick off the `index.js` file by loading necessary dependencies:

```
const Koa = require('koa')
```

```
const serve = require('koa-static')
const router = require('koa-router')()
const {join} = require('path')
const index = require('./routes/index')
```

Next we'll create a Koa app and assign `dev` and `port` configuration references:

```
const app = new Koa()
const dev = process.env.NODE_ENV !== 'production'
const port = process.env.PORT || 3000
```

Now we'll register relevant middleware and routes:

```
if (dev) {
  app.use(serve(join(__dirname, 'public')))
}

router.use('/', index.routes(), index.allowedMethods())

app.use(router.routes())
app.use(router.allowedMethods())
```

Finally in `index.js` we'll bind Koa's internal server to our `port` by calling `app.listen`:

```
app.listen(port, () => {
  console.log(`Server listening on port ${port}`)
})
```

Our `index.js` file is relying on `routes/index.js` so let's write it. Our code in `routes/index.js` should look as follows:

```
const router = require('koa-router')()

router.get('/', async function (ctx, next) {
  const title = 'Koa'
  ctx.body = `
    <html>
      <head>
        <title> ${title} </title>
        <link rel="stylesheet" href="styles.css">
      </head>
      <body>
        <h1> ${title} </h1>
```

```
        <p> Welcome to ${title} </p>
      </body>
    </html>
  `
  await next()
})

module.exports = router
```

Finally the `public/styles.css` file:

```
body {
  padding: 50px;
  font: 14px "Lucida Grande", Helvetica, Arial, sans-serif;
}
```

If we start our server with:

```
$ node index.js
```

And access http://localhost:3000 in a browser, we should see something like the following image:

How it works

There's more

Creating Middleware

Let's copy the `app` folder from our main recipe to the `custom-middleware-app` and create a middleware folder with an `answer.js` file:

```
$ cp -fr app custom-middleware-app
$ cd custom-middleware-app
$ mkdir middleware
$ touch middleware/answer.js
```

```
module.exports = answer
```

```
function answer () {
  return async (ctx, next) => {
    ctx.set('X-Answer', 42)
    await next()
  })
}
```

```
const Koa = require('koa')
const serve = require('koa-static')
const router = require('koa-router')()
const {join} = require('path')
const index = require('./routes/index')
const answer = require('./middleware/answer')
```

```
app.use(answer())
```

See also

Adding a view layer

... in there's more maybe discuss using template strings ... and also using frontend frameworks for SSR ... view layer includes css

Getting Ready

For this recipe we're going to copy the express application from our **Creating an Express Web App** recipe (we'll cover view layers for Hapi and Koa in the **There's More** section).

Let's copy the folder called `express-views` and add the `ejs` module:

```
$ cp -fr creating-an-express-web-app/app express-views
$ cd express-views
$ npm install --save ejs
```

How to do it

```
$ mkdir views
```

```
$ touch views/index.ejs
```

index.js alter:

```
app.set('views', join(__dirname, 'views'))
app.set('view engine', 'ejs')
```

views/index.ejs

```
<html>
  <head>
    <title> <%= title %> </title>
    <link rel="stylesheet" href="styles.css">
  </head>
  <body>
    <h1> <%= title %> </h1>
    <p> Welcome to <%= title %> </p>
  </body>
</html>
```

> ### Escaping Inputs 
>
> <%= vs <$- - ref to sec chapter

routes/index.js:

```
const {Router} = require('express')
const router = Router()

router.get('/', function(req, res, next) {
  const title = 'Express'
  res.render('index', {title: 'Express'})
  next()
})

module.exports = router
```

How it works

There's more

Adding a view layer to Koa

```
$ cp -fr creating-a-koa-web-app/app express-views
$ cd koa-views
$ npm install --save koa-views ejs
$ cp -fr ../express-views/views views
```

index.js top

```
const Koa = require('koa')
const serve = require('koa-static')
const views = require('koa-views')
const router = require('koa-router')()
const {join} = require('path')
const index = require('./routes/index')
```

index.js middle

```
app.use(views(join(__dirname, 'views'), {
  extension: 'ejs'
}))
```

```
const router = require('koa-router')()

router.get('/', async function (ctx, next) {
  ctx.state = {
    title: 'Koa'
  }
  await ctx.render('index')
  await next()
})

module.exports = router
```

Adding a view layer to Hapi

```
$ cp -fr creating-a-hapi-web-app/app express-views
$ cd hapi-views
$ npm install --save vision ejs
$ cp -fr ../express-views/views views
```

index.js top

```
const hapi = require('hapi')
```

```
const inert = require('inert')
const vision = require('vision')
const ejs = require('ejs')
```

index.js bottom:

```
const plugins = dev ? [vision, inert] : [vision]
server.register(plugins, start)

function start (err) {
  if (err) throw err

  server.views({
    engines: { ejs },
    relativeTo: __dirname,
    path: 'views'
  })

  routes.index(server)

  if (dev) routes.devStatic(server)

  server.start((err) => {
    if (err) throw err
    console.log(`Server listening on port ${port}`)
  })
}
```

routes/index.js

```
module.exports = index

function index (server) {
  server.route({
    method: 'GET',
    path: '/',
    handler: function (request, reply) {
      const title = 'Hapi'
      reply.view('index', {title})
    }
  })
}
```

ES2015 Template Strings as Views

-- maybe include tagged escape function

Registering Styling Preprocessor

See Also

Adding Logging

Getting Ready

```
$ cp -fr adding-a-view-layer/express-views express-logging
$ cd express-logging
$ npm install --save pino express-pino-logger
```

How to do it

index.js top

```
const {join} = require('path')
const express = require('express')
const pino = require('pino')()
const logger = require('express-pino-logger')({
  instance: pino
})
const index = require('./routes/index')
```

index.js middle

```
app.use(logger)
```

index.js bottom

```
app.listen(port, () => {
  pino.info(`Server listening on port ${port}`)
})
```

routes/index.js

```
router.get('/', function (req, res, next) {
  const title = 'Express'
```

```
  req.log.info(`rendering index view with ${title}`)
  res.render('index', {title: 'Express'})
  next()
})
```

How it works

> ## Log Processing 
>
> Talk about transports, give e.g. elasticsearch etc

There's more

Using the Winston Logger

Adding Logging to Koa

```
$ cp -fr adding-a-view-layer/koa-views koa-logging
$ cd koa-logging
$ npm install --save pino koa-pino-logger
```

```
const pino = require('pino')()
const logger = require('koa-pino-logger')({
  instance: pino
})
```

```
app.use(logger)
```

```
app.listen(port, () => {
  pino.info(`Server listening on port ${port}`)
})
```

routes/index.js

```
router.get('/', async function (ctx, next) {
  ctx.state = {
    title: 'Koa'
  }
  ctx.log.info(`rendering index view with ${ctx.state.title}`)
```

```
  await ctx.render('index')
  await next()
})
```

Adding Logging to Hapi

```
$ cp -fr adding-a-view-layer/hapi-views hapi-logging
$ cd hapi-logging
$ npm install --save pino hapi-pino
```

```
const pino = require('pino')()
const hapiPino = require('hapi-pino')
```

```
const plugins = dev ? [{
  register: hapiPino,
  options: {instance: pino}
}, vision, inert] : [{
  register: hapiPino,
  options: {instance: pino}
}, vision]
```

```
  server.start((err) => {
    if (err) throw err
    server.log(`Server listening on port ${port}`)
  })
```

routes/index.js

```
module.exports = index

function index (server) {
  server.route({
    method: 'GET',
    path: '/',
    handler: function (request, reply) {
      const title = 'Hapi'
      request.logger.info(`rendering index view with ${title}`)
      reply.view('index', {title})
    }
  })
}
```

Debug Logging with Pino

```
$ npm install -g pino-debug
```

```
$ node -r pino-debug index.js
```

```
$ node -r pino-debug index.js | pino
```

```
$ npm install -g pino-coloda
```

```
$ node -r pino-debug index.js | pino-colada
```

Log Levels

convert the route to trace, enable tracing with options (maybe do in hapi?)

Reusing a Logging Instance

See also

Implementing Authentication

Getting Ready

How to do it

How it works

> #### Session Storage 
>
> explain to not use in mem

There's more

Hapi

Koa

OAuth

express and hapi example

See also