Rendering pages server-side with Express (and Pug)

> This article was originally published at https://gist.github.com/joepie91/c0069ab0e0da40cc7b54b8c2203befe1.

Terminology

- **View:** Also called a "template", a file that contains markup (like HTML) and optionally additional instructions on how to generate snippets of HTML, such as text interpolation, loops, conditionals, includes, and so on.
- **View engine:** Also called a "template library" or "templater", ie. a library that implements view functionality, and potentially also a custom language for specifying it (like Pug does).
- **HTML templater:** A template library that's designed specifically for generating HTML. It understands document structure and thus can provide useful advanced tools like mixins, as well as more secure output escaping (since it can determine the right escaping approach from the context in which a value is used), but it also means that the templater is not useful for anything other than HTML.
- **String-based templater:** A template library that implements templating logic, but that has no understanding of the content it is generating - it simply concatenates together strings, potentially multiple copies of those strings with different values being used in them. These templaters offer a more limited feature set, but are more widely usable.
- **Text interpolation / String interpolation:** The insertion of variable values into a string of some kind. Typical examples include ES6 template strings, or this example in Pug: `Hello #{user.username}!`
- **Locals:** The variables that are passed into a template, to be used in rendering that template. These are generally specified every time you wish to render a template.

Pug is an example of a HTML templater. Nunjucks is an example of a string-based templater. React could technically be considered a HTML templater, although it's not really designed to be used primarily server-side.

View engine setup

Assuming you'll be using Pug, this is simply a matter of installing Pug...

```
npm install --save pug
```

... and then configuring Express to use it:

```
let app = express();

app.set("view engine", "pug");

/* ... rest of the application goes here ... */
```

You won't need to `require()` Pug anywhere, Express will do this internally.

You'll likely want to explicitly set the directory where your templates will be stored, as well:

```
let app = express();

app.set("view engine", "pug");
app.set("views", path.join(__dirname, "views"));

/* ... rest of the application goes here ... */
```

This will make Express look for your templates in the "views" directory, relative to the file in which you specified the above line.

Rendering a page

homepage.pug:

```
html
    body
        h1 Hello World!
        p Nothing to see here.
```

app.js:

```
router.get("/", (req, res) => {
    res.render("homepage");
});
```

Express will automatically add an extension to the file. That means that - with our Express configuration - the `"homepage"` template name in the above example will point at `views/homepage.pug`.

Rendering a page with locals

homepage.pug:

```
html
    body
        h1 Hello World!
        p Hi there, #{user.username}!
```

app.js:

```
router.get("/", (req, res) => {
    res.render("homepage", {
        user: req.user
    });
});
```

In this example, the `#{user.username}` bit is an example of string interpolation. The "locals" are just an object containing values that the template can use. Since every expression in Pug is written in JavaScript, you can pass *any* kind of valid JS value into the locals, including functions (that you can call from the template).

For example, we could do the following as well - although **there's no good reason to do this**, so this is for illustratory purposes only:

homepage.pug:

```
html
    body
        h1 Hello World!
        p Hi there, #{getUsername()}!
```

app.js:

```
router.get("/", (req, res) => {
    res.render("homepage", {
        getUsername: function() {
            return req.user;
        }
    });
});
```

Using conditionals

homepage.pug:

```
html
    body
        h1 Hello World!

        if user != null
            p Hi there, #{user.username}!
        else
            p Hi there, unknown person!
```

app.js:

```
router.get("/", (req, res) => {
    res.render("homepage", {
        user: req.user
    });
});
```

Again, the expression in the conditional is just a JS expression. All defined locals are accessible and usable as before.

Using loops

homepage.pug:

```
html
    body
        h1 Hello World!

        if user != null
            p Hi there, #{user.username}!
        else
            p Hi there, unknown person!

        p Have some vegetables:

        ul
            for vegetable in vegetables
                li= vegetable
```

app.js:

```
router.get("/", (req, res) => {
    res.render("homepage", {
        user: req.user,
        vegetables: [
            "carrot",
            "potato",
            "beet"
        ]
    });
});
```

Note that this...

```
li= vegetable
```

... is just shorthand for this:

```
li #{vegetable}
```

By default, the contents of a tag are assumed to be a string, optionally with interpolation in one or more places. By suffixing the tag name with `=`, you indicate that the contents of that tag should be a *JavaScript expression* instead.

That expression may just be a variable name as well, but it doesn't *have* to be - any JS expression is valid. For example, this is completely okay:

```
li= "foo" + "bar"
```

And this is completely valid as well, *as long as the randomVegetable method is defined in the locals*:

```
li= randomVegetable()
```

Request-wide locals

Sometimes, you want to make a variable available in *every* `res.render` for a request, no matter what route or middleware the page is being rendered from. A typical example is the user object for the current user. This can be accomplished by setting it as a property on the `res.locals` object.

homepage.pug:

```
html
    body
        h1 Hello World!
```

```
        if user != null
            p Hi there, #{user.username}!
        else
            p Hi there, unknown person!

        p Have some vegetables:

        ul
            for vegetable in vegetables
                li= vegetable
```

app.js:

```
app.use((req, res, next) => {
    res.locals.user = req.user;
    next();
});

/* ... more code goes here ... */

router.get("/", (req, res) => {
    res.render("homepage", {
        vegetables: [
            "carrot",
            "potato",
            "beet"
        ]
    });
});
```

Application-wide locals

Sometimes, a value even needs to be *application-wide* - a typical example would be the site name for a self-hosted application, or other application configuration that doesn't change for each request. This works similarly to `res.locals`, only now you set it on `app.locals`.

homepage.pug:

```
html
    body
        h1 Hello World, this is #{siteName}!

        if user != null
```

```
            p Hi there, #{user.username}!
        else
            p Hi there, unknown person!

        p Have some vegetables:

        ul
            for vegetable in vegetables
                li= vegetable
```

app.js:

```
app.locals.siteName = "Vegetable World";

/* ... more code goes here ... */

app.use((req, res, next) => {
    res.locals.user = req.user;
    next();
});

/* ... more code goes here ... */

router.get("/", (req, res) => {
    res.render("homepage", {
        vegetables: [
            "carrot",
            "potato",
            "beet"
        ]
    });
});
```

The order of specificity is as follows: `app.locals` are overwritten by `res.locals` of the same name, and `res.locals` are overwritten by `res.render` locals of the same name.

In other words: if we did something like this...

```
router.get("/", (req, res) => {
    res.render("homepage", {
        siteName: "Totally Not Vegetable World",
        vegetables: [
            "carrot",
```

```
            "potato",
            "beet"
        ]
    });
});
```

... then the homepage would show "Totally Not Vegetable World" as the website name, while every *other* page on the site still shows "Vegetable World".

Rendering a page after asynchronous operations

homepage.pug:

```
html
    body
        h1 Hello World, this is #{siteName}!

        if user != null
            p Hi there, #{user.username}!
        else
            p Hi there, unknown person!

        p Have some vegetables:

        ul
            for vegetable in vegetables
                li= vegetable
```

app.js:

```
app.locals.siteName = "Vegetable World";

/* ... more code goes here ... */

app.use((req, res, next) => {
    res.locals.user = req.user;
    next();
});

/* ... more code goes here ... */
```

```
router.get("/", (req, res) => {
    return Promise.try(() => {
        return db("vegetables").limit(3);
    }).map((row) => {
        return row.name;
    }).then((vegetables) => {
        res.render("homepage", {
            vegetables: vegetables
        });
    });
});
```

Basically the same as when you use `res.send`, only now you're using `res.render`.

Template inheritance in Pug

It would be very impractical if you had to define the *entire* site layout in every individual template - not only that, but the duplication would also result in bugs over time. To solve this problem, Pug (and most other templaters) support *template inheritance*. An example is below.

layout.pug:

```
html
    body
        h1 Hello World, this is #{siteName}!

        if user != null
            p Hi there, #{user.username}!
        else
            p Hi there, unknown person!

        block content
            p This page doesn't have any content yet.
```

homepage.pug:

```
extends layout

block content
    p Have some vegetables:

    ul
        for vegetable in vegetables
            li= vegetable
```

app.js:

```
app.locals.siteName = "Vegetable World";

/* ... more code goes here ... */

app.use((req, res, next) => {
    res.locals.user = req.user;
    next();
});

/* ... more code goes here ... */

router.get("/", (req, res) => {
    return Promise.try(() => {
        return db("vegetables").limit(3);
    }).map((row) => {
        return row.name;
    }).then((vegetables) => {
        res.render("homepage", {
            vegetables: vegetables
        });
    });
});
```

That's basically all there is to it. You define a `block` in the base template - optionally with default content, as we've done here - and then each template that "extends" (inherits from) that base template can *override* such `block`s. Note that you never render `layout.pug` directly - you still render the page layouts themselves, and they just inherit from the base template.

Things of note:

- Overriding a `block` is *optional*. If you don't override a `block`, it will simply contain either the default content from the base template (if any is specified), or no content at all (if not).
- You can have an unlimited number of `block`s with different names - for example, the one in our example is called `content`. You can decide to override any of them from a template, all of them, or none at all. It's up to you.
- You can nest multiple `block`s with different names. This can be useful for more complex layout variations.
- You can have multiple levels of inheritance - any template you are inheriting from can itself inherit from another template. This can be especially useful in combination with nested `block`s, for complex cases.

Static files

You'll probably also want to serve static files on your site, whether they are CSS files, images, downloads, or anything else. By default, Express ships with `express.static`, which does this for you.

All you need to do, is to tell Express where to look for static files. You'll usually want to put `express.static` at the very start of your middleware definitions, so that no time is wasted on eg. initializing sessions when a request for a static file comes in.

```
let app = express();

app.set("view engine", "pug");
app.set("views", path.join(__dirname, "views"));

app.use(express.static(path.join(__dirname, "public")));

/* ... rest of the application goes here ... */
```

Your directory structure might look like this:

```
your-project
|- node_modules ...
|- public
|  |- style.css
|  `- logo.png
|- views
|  |- homepage.pug
|  `- layout.pug
`- app.js
```

In the above example, `express.static` will look in the `public` directory for static files, relative to the `app.js` file. For example, if you tried to access `https://your-project.com/style.css`, it would send the user the contents of `your-project/public/style.css`.

You can optionally also specify a *prefix* for static files, just like for any other Express middleware:

```
let app = express();

app.set("view engine", "pug");
app.set("views", path.join(__dirname, "views"));
```

```
app.use("/static", express.static(path.join(__dirname, "public")));

/* ... rest of the application goes here ... */
```

Now, that same `your-project/public/style.css` can be accessed through `https://your-project.com/static/style.css` instead.

An example of using it in your **layout.pug**:

```
html
    head
        link(rel="stylesheet", href="/static/style.css")
    body
        h1 Hello World, this is #{siteName}!

        if user != null
            p Hi there, #{user.username}!
        else
            p Hi there, unknown person!

        block content
            p This page doesn't have any content yet.
```

The slash at the start of `/static/style.css` is important - it tells the browser to ask for it *relative to the domain*, as opposed to *relative to the page URL*.

An example of URL resolution without a leading slash:

- **Page URL:** `https://your-project.com/some/deeply/nested/page`
- **Stylesheet URL:** `static/style.css`
- **Resulting stylesheet request URL:** `https://your-project.com/some/deeply/nested/static/style.css`

An example of URL resolution *with* the loading slash:

- **Page URL:** `https://your-project.com/some/deeply/nested/page`
- **Stylesheet URL:** `/static/style.css`
- **Resulting stylesheet request URL:** `https://your-project.com/static/style.css`

That's it! You do the same thing to embed images, scripts, link to downloads, and so on.

Revision #1
Created 2024-12-11 01:29:10 UTC by joepie91
Updated 2024-12-11 18:44:19 UTC by joepie91