Module 6: Best Practices; More Frontend Topics; More Backend Topics;

Edgardo Molina, PhD | Lead Instructor

CUNY Tech Prep 2016-2017

Outline

- Best Practices
- More Frontend Topics
- More Backend Topics

Best Practices

- `.gitignore`
- `node_modules` (and other build files)
- credential management
- **DRY** Principle
- Modules

.gitignore

Your git repository should NOT track certain files/information:

- OS/IDE/environment specific [`.DS_Store`, `.vagrant`, `.idea`, ...]
- Build files [`.exe`, `.o`, `.jar`, ...]
- Packages [`node_modules`]
- Credentials [`passwords`, `tokens`, `keys`, ...]

WHY?

.gitignore

Example from ctp-microblog:

```
.DS_Store
.vagrant
node_modules
.eslintrc.*
*.box
```

Add other files/directories you wish git to ignore during development.

Other example `.gitignore` files:

`https://github.com/github/gitignore`

node_modules

The `npm` command manages project packages within the `node_modules` directory.

DO NOT add node_modules to git repository

DO NOT add or modify code in node_modules

What if we want to modify a package?

node_modules

The `npm` command manages project packages within the `node_modules` directory.

DO NOT add node_modules to git repository

DO NOT add or modify code in node_modules

> *What if we want to modify a package?*
>
> *Fork and Patch the package on github, and link to your fork in* `package.json`
>
> *Example Tutorial:* `http://debuggable.com/posts/how-to-fork-patch-npm-modules:4e2eb9f3-e584-44be-b1a9-3db7cbdd56cb`

Credential management

Even the smallest web applications will need multiple credentials, such as:

- Database username, passwords, hostnames, etc
- Secret keys for sessions
- 3rd Party Services API tokens and keys

These should never be exposed (or saved) in a git repository

Always provide these via an outside channel (provisioning tools or manually)

DRY Principle

DRY: Don’t Repeat Yourself

Good software reduces repitition of code or information

In JavaScript code we can achieve this by using modules

In Views we can do this by seperating templates into reusable components

We should apply this principle at all levels of our application: the build system, tests, configuration, application code, and documentation.

Modules

Modules should be small, testable, and portable.

A good module performs a specific task and does not do more.

It exposes a well documented API, making the module replaceable.

Larger functionality can be achieved by the composition of smaller modules.

Modules are key to following the dry principle.

Modules

In JavaScript:

- Each `js` file is treated as a module that *exports* properties and functions
- Other `js` files can import another file via a require statement
  - `var mymodule = require('./mycodefile')`
- A directory can be treated as a module if it contains an `index.js` file
  - `var mymodule = require('./myfolder')`
  - where the file `myfolder/index.js` exists.
  - `index.js` can in turn require all other files in the directory
  - See `blog/controllers/index.js` for an example

More Frontend Topics

Topics

- Static Files
- Partials
- HTML Escaping
- Flash Messages

Static files

Our apps consist of `js` files that are executed to generate output.

Some files we serve clients don't need to be executed, we use each *as is*

Images, videos, downloads (pdfs, zips, etc), and some CSS files.

Express allows us to designate directories that should just be served as they are. We call these static files.

In Express: `app.use(express.static('./public'));`

Partials

Views are used to dynamically generate HTML output.

HTML that should wrap all pages is placed in **layout** view files.

HTML for specific pages make up the majority of **views**.

Some HTML components should be available to multiple views, but are not necessarily used by all views.

Partials

To follow the DRY principle, we extract reusable view components into **partials**.

Partials are available for use from all views, and can be inserted via a `{{> path/partialName}}` tag

For an example see: `https://github.com/ericf/express-handlebars/tree/master/examples/advanced/views`

View Helpers

Views should not contain complex logic!

But sometimes we do need to perform parsing or complicated formatting (think dates)

That logic doesn't quite belong in the controller. For these cases we can expose reusable `js` functions to the views as **helpers**.

See: `https://github.com/ericf/express-handlebars/blob/master/examples/advanced/lib/helpers.js`

HTML escaping

Our views are tasked with generating HTML output sent to users.

- That output is generated from DB content
- That DB content came from user input

Can I trust the users input!?

HTML escaping

Our views are tasked with generating HTML output sent to users.

- That output is generated from DB content
- That DB content came from user input

Can I trust the users input!?

What if they send me HTML or `js` code that is then stored in my DB, and rendered in my views.

HTML escaping

Most modern template engines automatically escape all strings (unless we say otherwise).

This converts all brackets and semicolons to HTML safe char codes.

Otherwise we would be rendering potentially harmful code to all of our users

See example: `http://handlebarsjs.com/#html-escaping`

Flash Messages

When a user succeeds or fails at submitting form data or an action, the user should receive a notification of the success or failure of that action.

We typically see this is a temporary message at the top of the page.

We call this a **flash message**. This message is set by the controller and rendered by the view only if necessary.

We use the `req.flash` property for this purpose.

See: `https://github.com/medgardo/ctp-microblog/blob/master/blog/controllers/profile.js`

More Backend Topics

Topics

- Middleware
- Cookies
- Sessions
- Authentication
    - Passport
    - BCrypt

Middleware

A **middleware** component is a piece of reusable code that can be run prior to controller/route actions.

Middleware is used for:

- Logging
- Authentication
- Authorization
- Redirection
- and many other purposes.

Middleware

In express, all controller actions are middleware.

We can provide multiple middleware handlers for each route.

Each component decides whether to end or continue by calling `next()`.

See: `https://github.com/medgardo/ctp-microblog/blob/master/blog/controllers/profile.js`

Cookies

Web browsers store a file for each website a user accesses.

The website can send the browser requests to save and manipulate data in this *cookie* file.

Why might this be useful?

Cookies

Users have the freedom to delete cookies.

Users have the freedom to deny all cookies.

This means they are no good for **persistent** data, only for small temporary data.

Why do cookies have a bad reputation?

Cookies

Users have the freedom to delete cookies.

Users have the freedom to deny all cookies.

This means they are no good for **persistent** data, only for small temporary data.

Why do cookies have a bad reputation?

> *Cookies are used for ad and behavior tracking.*
>
> *Many users don't know that cookies can be deleted.*

Sessions

HTTP is a *stateless* protocol.

The server receives multiple connections and does not distinguish one from the other.

This is a problem, How can our application distinguish requests between two different logged in users?

Sessions

HTTP is a *stateless* protocol.

The server receives multiple connections and does not distinguish one from the other.

This is a problem, How can our application distinguish requests between two different logged in users?

We use sessions! which use cookies.

Sessions

The sessions feature in webapps write a unique session ID to the users cookie.

This ID is sent along with every request from the browser.

The application keeps data for each unique session ID to help it distinguish among different clients.

This session data can be stored on the app server in memory, files, or in a database.

Authentication

Authentication is the process of confirming that a user is who they say they are.

In webapps we achieve this via the username and password.

But HTTP is stateless, and we do not want the user to provide their username and password with every request.

How do we know when a user has already logged in?

Authentication

We use *sessions* to store the userID when the user logs in.

And we remove it when they logout.

Any request in between will assume the user is logged in if the userID is present, and logged out if the userID is not present.

How do we check passwords?

Passport

Passport is middleware for authentication.

It allows for the management of multiple authentication strategies (such as OAuth logins and local logins).

It does not actually store passwords or check them. It only provides a uniform interface for authentication

`http://passportjs.org/docs/overview`

Storing and Checking Passwords

NEVER store passwords in plaintext!!!

NEVER come up with your own hashing/salting/peppering scheme!!!

(unless you're a cryptographer)

Password Storage

Stay up to date with best practices

Currently use `bcrypt` or `scrypt`

These are strong *PBKDF*'s (Password Based Key Derivation Functions), sometimes still referred to as *password hashing*.

More Info about Passwords

JavaScript implementation of `bcrypt`:

`https://github.com/shaneGirish/bcrypt-nodejs`

Password Storage Articles:

- `https://adambard.com/blog/3-wrong-ways-to-store-a-password/`
- `https://www.praetorian.com/blog/secure-password-storage-in-go-python-ruby-java-haskell-and`