Inclusion Starts With Docs

A talk about docs, Elixir, and user empathy
by Pete Gamache.

Presented at the first EMPEX LA conference,
10 November 2018.

Hello!

What is this?

- A short guide to getting the most out of Elixir's built-in docs system, ExDoc

- A love letter to the authors of great docs

Some Questions

- Do you want to have to fully understand someone else's code in order to use it?

Some Questions

- Do you trust the code of developers who could not or would not describe it in plain English?

Some Questions

- Do you want to restrict your audience to "geniuses"?

Some Questions

- **Do you want other people to fix and improve your code, at no cost to you??**

Name some great docs

Elixir Has Great Docs

- They look good
- Modules explain their purpose
- Functions explain their semantics
- Sample code everywhere

GenServer behaviour

A behaviour module for implementing the server of a client-server relation.

A GenServer is a process like any other Elixir process and it can be used to keep state, execute code asynchronously and so on. The advantage of using a generic server process (GenServer) implemented using this module is that it will have a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into a supervision tree.

Example

The GenServer behaviour abstracts the common client-server interaction. Developers are only required to implement the callbacks and functionality they are interested in.

Let's start with a code example and then explore the available callbacks. Imagine we want a GenServer that works like a stack, allowing us to push and pop items:

```
defmodule Stack do
  use GenServer

  # Callbacks

  def handle_call(:pop, _from, [h | t]) do
    {:reply, h, t}
  end

  def handle_cast({:push, item}, state) do
    {:noreply, [item | state]}
  end
end
```

```
$ iex
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:
10] [hipe] [kernel-poll:false]

Interactive Elixir (1.6.0) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> h GenServer
```

GenServer

A behaviour module for implementing the server of a client-server relation.

A GenServer is a process like any other Elixir process and it can be used to keep state, execute code asynchronously and so on. The advantage of using a generic server process (GenServer) implemented using this module is that it will have a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into a supervision tree.

Example

The GenServer behaviour abstracts the common client-server interaction. Developers are only required to implement the callbacks and functionality they are interested in.

Let's start with a code example and then explore the available callbacks. Imagine we want a GenServer that works like a stack, allowing us to push and pop items:

ExDoc Cheat Sheet

- Write your docs in Markdown, inline with your source code
- Indent sample code four spaces from the rest
- Add **ex_doc** to your project dependencies in **mix.exs**
- **mix docs** generates HTML docs in **doc/**
- **mix hex.publish docs** publishes docs for a Hex module to hexdocs.pm

Write Docs in Markdown

```
defmodule StripJs do
  @moduledoc ~s"""
  StripJs is an Elixir module for stripping executable JavaScript from
  blocks of HTML and CSS.

  It handles:

* `<script>...</script>` and `<script src="..."></script>` tags
* Event handler attributes such as `onclick="..."`
* `javascript:...` URLs in HTML and CSS
* CSS `expression(...)` directives
* HTML entity attacks (like `&lt;script&gt;`)


## Installation

Add `strip_js` to your application's `mix.exs`:

    def application do
      [applications: [:strip_js]]
    end
```

StripJs

StripJs is an Elixir module for stripping executable JavaScript from blocks of HTML and CSS.

It handles:

- `<script>...</script>` and `<script src="..."></script>` tags
- Event handler attributes such as `onclick="..."`
- `javascript:...` URLs in HTML and CSS
- CSS `expression(...)` directives
- HTML entity attacks (like `&lt;script&gt;`)

Installation

Add `strip_js` to your application's `mix.exs`:

```
def application do
  [applications: [:strip_js]]
end

def deps do
  [{:strip_js, "~> 0.9.0"}]
end
```

Usage

`clean_html/2` removes all JS vectors from an HTML string:

```
iex> html = "<button onclick=\"alert('pwnt')\">Hi!</button>"
iex> StripJs.clean_html(html)
```

Usage

`clean_html/2` removes all JS vectors from an HTML string:

```
iex> html = "<button onclick=\"alert('pwnt')\">Hi!</button>"
iex> StripJs.clean_html(html)
"<button>Hi!</button>"
```

`clean_css/2` removes all JS vectors from a CSS string:

```
iex> css = "body { background-image: url('javascript:alert()'); }"
iex> StripJs.clean_css(css)
"body { background-image: url('removed_by_strip_js:alert()'); }"
```

StripJs relies on the Floki HTML parser library, which is built using Mochiweb. StripJs provides a `clean_html_tree/1` function to strip JS from `Floki.parse/1` - and `:mochiweb_html.parse/1` - style HTML parse trees.

Bugs and Limitations

The brokenness of invalid HTML may be amplified by `clean_html/2`.

In uncommon cases, innocent CSS which very closely resembles JS-injection techniques may be mangled by `clean_css/2`.

StripJs may not block 100% of executable JavaScript, though it gets quite close. If you believe there are JS injection methods not covered by this library, please submit an issue with a test case!

Authorship and License

Moduledocs

- Explain the purpose of a module (or project)
- Sample code illustrates common usage
- Installation instructions
- In lesser languages, we'd use a README for this

TIP! Install Instructions

- Scenario: you add a feature and increase the version number of your project

- Oh no! You forget to change it in the @moduledoc, so your docs are telling people to install an old version

- Solution: variable interpolation in @moduledoc, using **~s** sigil (not **~S**) and **#{MyApp.Mixfile.project[:version]}**

```
* CSS `expression(...)` directives
* HTML entity attacks (like `&lt;script&gt;`)


## Installation

Add `strip_js` to your application's `mix.exs`:

    def application do
      [applications: [:strip_js]]
    end

    def deps do
      [{:strip_js, "~> #{StripJs.Mixfile.project[:version]}"}]
    end

""" <> ~S"""

## Usage

`clean_html/2` removes all JS vectors from an HTML string:

    iex> html = "<button onclick=\"alert('pwnt')\">Hi!</button>"
```

Function Docs

- Provide the intent and specification for each function
- ...So be specific! Esp. input and output types
- Sample code goes deeper than moduledoc
- Use **@doc** just before function definition

```
@doc ~S"""
Removes JS vectors from the given CSS string; i.e., the contents of a
stylesheet or `<style>` tag.

Does not HTML-escape its output.  Care is taken to maintain valid CSS
syntax.

Example:

    iex> css = "tt { background-color: expression('alert()'); }"
    iex> StripJs.clean_css(css)
    "tt { background-color: removed_by_strip_js('alert()'); }"

Warning: this step is performed using regexes, not a parser, so it is
possible for innocent CSS containing either of the strings `javascript:`
or `expression(` to be mangled.
"""
@spec clean_css(String.t, opts) :: String.t
def clean_css(css, _opts \\ []) when is_binary(css) do
  css
  |> String.replace(~r/javascript \s* :/xi, "removed_by_strip_js:")
  |> String.replace(~r/expression \s* \(/xi, "removed_by_strip_js(")
end
```

Doctests

- Problem: sample code in docs can go stale
- Elixir's solution: execute sample code as tests
- Rules are fairly simple: code starts with **iex>**, results don't, no blank lines allowed
- Just put **doctest MyApp.Modulename** in one of your test files

TIP! Doctests

- Always add doctest MyApp.Modulename to your tests, even when you have no doctests
- Harmless with no doctests
- Avoids having unevaluated doctests later
- Also "good" for test coverage. Don't ask me why

Data Types

- **String.starts_with?(string, prefix)** is pretty clear already
- Many functions are not so lucky
- In languages with static typing, this is solved automatically
- We're not using one of those languages, so we need to do a tiny bit of work in order to ensure that a function's input and output data types are obvious
- Elixir and Erlang's solution: Typespecs

```
Example:

    iex> css = "tt { background-color: expression('
    iex> StripJs.clean_css(css)
    "tt { background-color: removed_by_strip_js('al

Warning: this step is performed using regexes, not
possible for innocent CSS containing either of the
or `expression(` to be mangled.
"""
@spec clean_css(String.t, opts) :: String.t
def clean_css(css, _opts \\ []) when is_binary(css)
  css
  |> String.replace(~r/javascript \s* :/xi, "remove
  |> String.replace(~r/expression \s* \(/xi, "remov
end
```

```
@type opts :: Keyword.t  # reserved for future use

@type html_tag :: String.t

@type html_attr :: {String.t, String.t}

@type html_node :: String.t | {html_tag, [html_attr], [html_node]}

@type html_tree :: html_node | [html_node]
```

Dialyzer

- Typespecs aren't only for docs

- Dialyzer is Erlang's "discrepancy analyzer"

- Add **dialyxir** to your **mix.exs** deps to use it in Elixir

- TL;DR you write typespecs, then run **mix dialyzer**, then Erlang tells you if you were lying

EMPEX

Source Code

- Sometimes you just need to RTFS
- Usually not that hard to Google "<projectname> github", plow into lib/, find the module in question, grep for function name
- Come on, are you kidding me? What a PITA
- Elixir's solution: ExDoc supports Github repos natively, via **--source-url** option
- ...And with a Mix task alias, automatically

```elixir
def project do
  [
    app: :strip_js,
    version: "0.9.0",
    description: "Strip JavaScript from HTML and CSS",
    package: package(),
    elixir: "~> 1.2",
    build_embedded: Mix.env == :prod,
    start_permanent: Mix.env == :prod,
    deps: deps(),
    aliases: [
      docs: "docs --source-url https://github.com/appcues/strip_js",
    ],
  ]
end
```

StripJs

StripJs is an Elixir module for stripping executable JavaScript from blocks of HTML and CSS.

It handles:

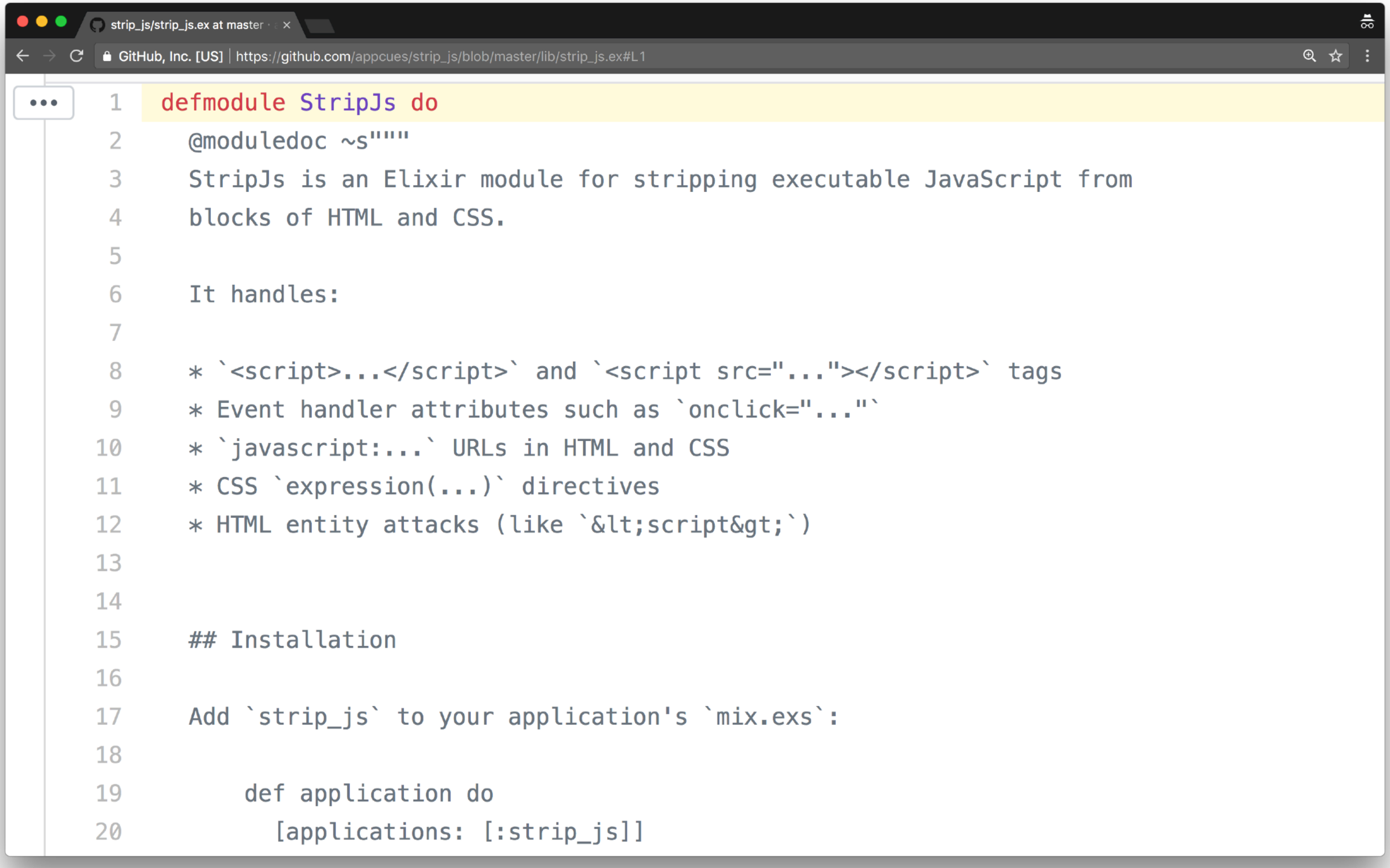
- `<script>...</script>` and `<script src="..."></script>` tags
- Event handler attributes such as `onclick="..."`
- `javascript:...` URLs in HTML and CSS
- CSS `expression(...)` directives
- HTML entity attacks (like `&lt;script&gt;`)

Installation

Add `strip_js` to your application's `mix.exs`:

```
def application do
  [applications: [:strip_js]]
end

def deps do
  [{:strip_js, "~> 0.9.0"}]
end
```


strip_js/strip_js.ex at master
GitHub, Inc. [US] | https://github.com/appcues/strip_js/blob/master/lib/strip_js.ex#L1
1 defmodule StripJs do
2 @moduledoc ~s"""
3 StripJs is an Elixir module for stripping executable JavaScript from
4 blocks of HTML and CSS.
5
6 It handles:
7
8 * `<script>...</script>` and `<script src="..."></script>` tags
9 * Event handler attributes such as `onclick="..."`
10 * `javascript:...` URLs in HTML and CSS
11 * CSS `expression(...)` directives
12 * HTML entity attacks (like `<script>`)
13
14
15 ## Installation
16
17 Add `strip_js` to your application's `mix.exs`:
18
19 def application do
20 [applications: [:strip_js]]

Epilogue

We Have Great Docs

- Module's purpose is explained, with examples
- Functions are explained in depth, with examples
- These examples never go stale
- Dialyzer keeps us honest
- One-click access to source code
- And it all looks sharp

Read All About It!

- https://hexdocs.pm/ex_doc/readme.html
- https://hexdocs.pm/elixir/typespecs.html
- https://github.com/jeremyjh/dialyxir

EMPEX

Thanks!!

Pete Gamache
@gamache
pete@gamache.org