# Frax

Experimental abstraction layer for coding of web UIs

# Concepts

- **Abstracts away much of the complexity of frontend development**
- **Declarative**
- **Has no external dependencies**
- **Exposes a single flexible method to unify data binding and UI rendering**
- **What?** *Yeah, I don't know...*
- **How large is it?** *1 KB (gzipped)*
- **How performant is it?** *No idea!*
- **Is it shitty?** *Isn't everything?*

```
frax("foo", data, fn)
```

# One method to rule them all

# Initialization

**The *frax*-method is used throughout an application, and thus has varying use cases. The very first time we call the method, it is to register our templates (required) and workers (optional) and create the scope for our application.**

The first argument should in this case be an array containing: 1) an object literal of methods that return template strings, and 2) an array of Web Worker instances to be used for any computationally heavy tasks (can be left empty). The second argument contains a callback function which will be invoked when the web page has finished loading. This functions body will contain the rest of our applications logic, and is executed with the parameter *store* that contains application state properties and methods.

```
frax([
  {
    header: data => `<h1>${data.title}</h1><span>By: ${data.author}</span>`,
    main: data => `<article>${data.article}</article>`
  },
  [myWorker1, myWorker2]
], store => {
  // Everything happens in here...
});
```

# Render and data binding

**Next we want to render something based on our template. Using the *frax*-method we perform render and define state at the same time.**

The first argument should in this case be a string which denotes the name of the section being rendered. This name corresponds to the naming in the template object. The second argument contains the data being passed. A *frax*-method with these two parameters defined will render markup in the body-element of the web page.

```
frax([
  {
    header: data => `<h1>${data.title}</h1><span>By: ${data.author}</span>`,
    main: data => `<article>${data.article}</article>`
  },
  [myWorker1, myWorker2]
], store => {
  frax("header", { title: "How to go about pooping your pants", author: "Mikael" });
});
```

The properties set for the *header* namespace is now available by calling the *get*-method on the store.
For example: `store.get("header").title` will now output: `"How to go about pooping your pants"`.

# The store

**The store exposes a *get*-method for reading properties, as well as *set*- and *append*-methods for writing. However, the main way of writing to the store is to pass data via the second *frax*-parameter like we saw in the last example. There are three variants of doing this:**

- Passing some data. This will add (or replace if already written) the specified properties in the *header* namespace on the store:
  ```
  frax("header", { title: "How to go about pooping your pants", author: "Mikael" });
  ```

- Passing the result of a fetch request (as JSON). The data returned will replace everything written in the *header* namespace on the store:
  ```
  frax("header", "https://www.poop-api.com/api/about");
  ```

- Setting a specific property with the result of a fetch request (as JSON). This will add (or replace if already written) the specified properties in the *header* namespace on the store with the data returned from the fetch request:
  ```
  frax("header", "title->https://www.poop-api.com/api/about/title");
  ```

# Nesting

## Got poop?

In order to ensure persistent ordering of our rendered markup as well as timely execution of business logic, we use a nested structure where every new level signifies additional markup to be appended and/or additional logic to be executed. Thusly, the third parameter of the *frax*-method is a callback function that enables us to build our UI as a multilevel tree of *frax*-method calls. Further manipulation to a namespace (as seen below, where we change header title value when, sadly, no poop is retrieved) will result in an isolated re-render of only that particular markup, leaving the overall structure alone.

```
frax("header", { title: "Here's the poop" }, () => {
  frax(
    "myPoopList",
    "https://www.poop.io/api/poop",
    () => {
      if (store.data.myPoops.length === 0) {
        frax("header", { title: "Sorry, no poop." };
      }
    }
  );
});
```

# Events

## Event handling is painful.

In Frax, the event handling for our DOM-elements is represented as special store namespaces called clickableClassNames and changeableClassNames (implementation of more event types are planned). These can be populated with methods that are applied to any DOM-element that has a class name matching the property name of the method. This way we can cleanly assign methods to execute when an events happens, and Frax takes care of attaching and detaching event listeners under the hood.

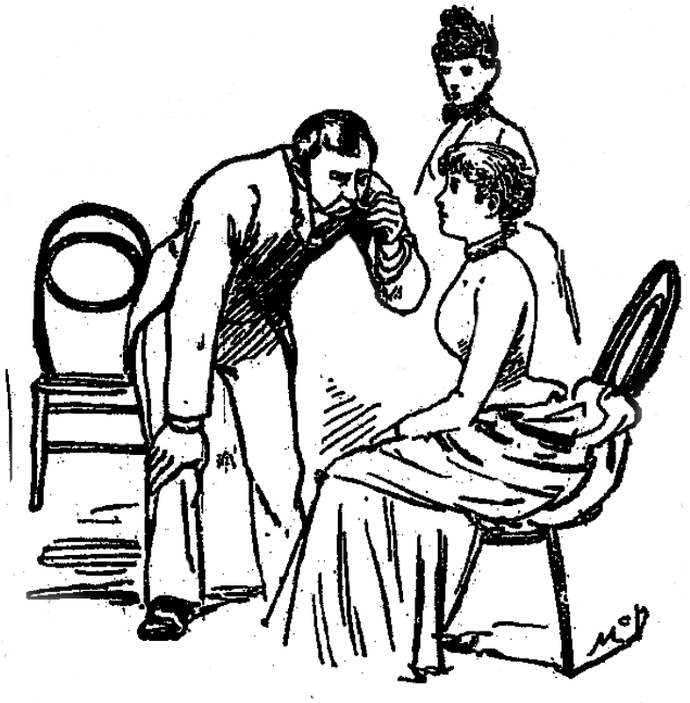
# The pooping of the pants

I’ve probably overlooked a lot of stuff.

# Thanks.

Check it out on Github (or bookmark it and never check it out):

https://github.com/Mikael/frax



AN INSANITY EXPERT AT WORK.