

Criterion E: Product development

Product directory structure highlights:

extension/ – The browser extension

src/ – JavaScript source of the extension

common.js – miscellaneous utility functions

content.js – content script (runs on the EHR page itself)

ezyvet.js – Utilities for extracting information from the ezyvet site

input.js – Object-oriented forms

menu.js – Controls the drop-down menu

units.js – Unit conversion

html/menu.html – Drop-down menu

ezymock/

ezymock.py – Mock server

static/ – Static files for the mock server

templates/ – Jinja2 templates for dynamic portions of the mock server

har/

harcapture.py – Script for capturing HAR files

hardiff.py – Assorted utilities that I accumulated for analyzing HAR files

This file structure clearly separates the project into 3 components. JavaScript files are separated into individual modules of related code.

Examples of techniques from Criterion D

Technique: Arrays and Iterators

A few examples of array usage:

* An array of pdf urls is collected in content.js and sent as a response to browser.tabs.sendMessage calls from menu.js.

* menu.js stores an array of tools which are exchanged as the displayed element when drop-down menu items are moused over

In several places where arrays would typically be used, I opted to use iterators instead. In common.js, I define an imap generator function which acts like the Array.map builtin in JavaScript, returning a. imap is nice when dealing with non-array iterables like Map objects (hash tables). I could convert the Map object into an array as an intermediary step to mapping, but this seems inelegant. By using ES6 generator functions, I can concisely abstract the concept of mapping on arbitrary iterable streams.

```
export function* imap(iterable, func) {  
  for (var item of iterable) {  
    yield func(item);  
  }  
}
```

Babel.js then uses the regenerator library to convert the generator function into a function compatible with older versions of JavaScript.

Technique: String Handling

In ezyvet.js, I use some regular expressions to extract information from EzyVet. For example, I use `/Weight\(([^\)]*)\)/` to get the weight units. This regular expression matches Weight followed by a parenthesis, followed by an arbitrary number of non-close-parenthesis characters, followed by a close paren. The inner non-escaped parentheses create a match group which selects everything within the surrounding parens.

I believe that EzyVet always uses pounds, but if this is not correct, my client could end up using an incorrect weight value in the automatic calculations and giving an improper dosage to an animal. I made this regular expression very specific to how EzyVet presents records so that if something changes, the weight value becomes null rather than improperly doing something else.

Technique: Object-oriented design

input.js abstracts the concept of “Inputs” – which are essentially form components – using classes. Inputs are abstractly said to have a name and value, as well as the ability to be listened to in the event of changes to their value and a method for adding them to the DOM.

The base Input class is extended by subclasses to create various input types, and are combined together into even more abstract Input types, like a Form which combines many Inputs and uses their values as parameters to a function to compute a value.

Below is an example of an Input type (MeasurementInput) which combines the functionality of two different inputs (NumberInput and SelectInput). The value of the class is computed dynamically with a descriptor property. This shows the flexibility of the Input class hierarchy and its usefulness in easily creating

abstract input types which greatly reduce the complexity of building forms. With a MeasurementInput, the caller need not worry about the value of either of its components or any of the DOM structure that makes it up.

```
export class MeasurementInput extends NumberInput {
  constructor(opts) {
    super(opts);

    this.domain = opts.domain;
    this.unit = opts.unit;

    this.unit_selection = new SelectInput({
      name: `${this.name} units`,
      container: no_container,
      optional: this.optional,
      default: this.unit,
      options: new Map(common.imap(this.domain, ([unit, data]) => {
        return [data.abbr, unit];
      })))
    });

    this.unit_selection.listen(this.tell.bind(this));
    this.unit_selection.add_to(this.container.body);
  }

  get value() {
    return convert(
      super.value, this.domain,
      this.unit_selection.value, this.unit
    );
  }
}
```

Asynchronous programming

I used several asynchronous APIs, such as the sending of XMLHttpRequests and passing of messages between menu.js and content.js.

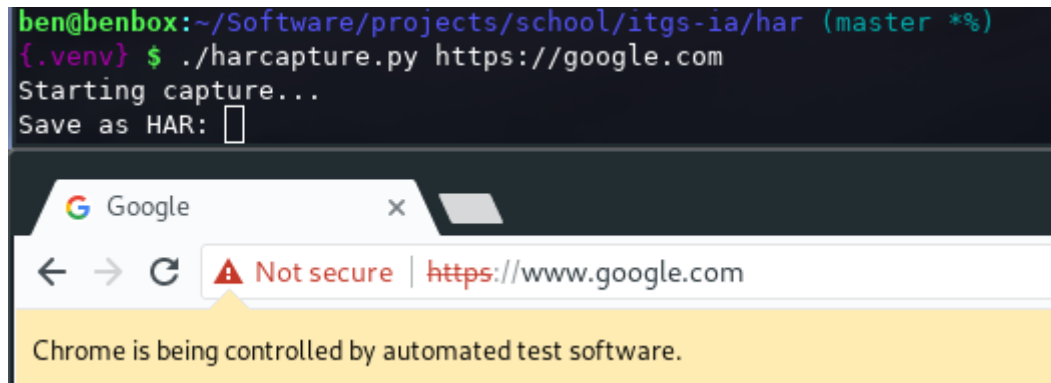
Use of CSS

I pretty much just used bootstrap, so not really much here. I used CSS directly to fix the width of the drop-down menu popup because otherwise it constantly resized which was poor for usability.

Analysis of HTTP requests sent by the browser

Firefox and Chromium both have a feature in their dev tools to save HTTP logs as HAR, but for some inexplicable reason, both browsers failed to save a HAR file properly for EzyVet. Firefox didn't save anything at all, while Chromium truncated the file at seemingly arbitrary points, destroying the JSON structure of the data. I'm not sure why this happened. For other sites, I was able to use the dev tools to save HAR just fine. In any case, I switched to browsermobproxy for

capturing http requests, creating a simple script that attached the proxy to the browser and provided an interface for saving HAR files in `harcapture.py`. I had to use Chromium as the browser because I couldn't get Firefox to accept the phony https certificates which `browsermobproxy` generates. Below is a screenshot of a session with `harcapture.py` on `https://google.com`:



After capturing the HAR files, I had to analyze them. Because I needed to perform arbitrary tasks such as filtering by when query parameters on URLs matched something of interest, I couldn't create a single script for dealing with them. Instead I manipulated the HAR structure in the Python REPL. For convenience, I created `hardiff.py` which has a few tools which I used frequently. Notably, it has an `install` function which writes the responses to all requests that match a certain path to the filesystem with names that include the HTTP method like `./General/SideList-GET.html`. I could then easily write `general` for such files in Flask and edit them to replace content with dynamic data from the mock server.

Use of external resources

As mentioned elsewhere, I used a few software libraries:

- * `pdf.js` (for extracting text from pdfs)
- * `moment.js` (for parsing and manipulating time)
- * `bootstrap` (for styling)
- * `Flask` (for the mock server)
- * `browsermobproxy-python` (for capturing HAR)
- * `selenium-python` (for opening web browsers with the desired proxy configuration)

I used the documentation at <https://developer.mozilla.org/en-US/> (MDN) extensively, but didn't copy any example code. Similarly, I made some use of the Python docs and the Flask docs. To a lesser extent, I used the `moment.js` docs, but I didn't use `moment.js` very heavily at all. `pdf.js` didn't have much in the way of docs, so I learned how to use it from the answers to

<https://stackoverflow.com/q/1554280>, but I did not copy the code, preferring to do things a little differently with ESNext async functions.

I copied the configuration for babeljs in my webpack.config.js from <https://babeljs.io/docs/setup> and the configuration for bootstrap from <https://getbootstrap.com/docs/4.0/getting-started/webpack/>

I used the answer to <https://stackoverflow.com/q/11450158/3154567> to make the Chrome WebDriver accept the browsermobproxy proxy.