The Importance of DX in API Design

https://bit.ly/dx-in-api-design

Richard Lindsey @Velveeta

Developer Experience

Developers are to DX what users are to UX. They are your consumers, and when your interface is lacking in some way, they're the ones that experience the pain of that.

Just like any internet consumer, developers are real people, and have their own motivations to which you can appeal.

Webpack

Pros

- Highly configurable.
- Very powerful as a developer tool.
- Huge option set, plugin support, custom transforms, etc.

Cons

- A nightmare of an API.
- Too much power makes for a user-base of mostly power-users.
- Slow adoption due to learning curve.

jQuery

Pros

- Idiomatic, intuitive API.
- Chainable functions.
- Plugin support, which led to old school components.

Cons

- Spaghetti code; pages of function chain calls.
- Sometimes, *too* flexible can be a bad, too.

Why is DX Important?

Products live and die by their feature sets and adoption rates of their users. APIs aren't any different, from a developer's perspective.

Why is DX Important?

If your product's API is:

- Difficult to consume
- Difficult to comprehend
- Difficult to remember

There will be fallout. As much as possible, you want to reduce the friction to easily adopt and to easily adapt to changes as they inevitably occur over the lifetime of your product.

Why is DX Important?

Get to know your users:

UX designers often conduct user surveys, interviews, study NPS feedback, etc. The way our customers use products isn't always the way we intend when we publish them, and can help inform ongoing product evolution.

Why is DX Important?

API designers should similarly learn how their consumers are actually using their products, which can help inform where unused cruft can be deprecated or obsoleted, which helps reduce API surface area. It can also help authors determine where functionality can be combined, separated, or otherwise augmented to be more in line with the habits of their actual users.

Why is DX Important?

The React team realized over time that a lot of their user base were using lifecycle functions like componentWillMount and componentWillUpdate to sync local state variables with incoming prop values, so they provided a single function, getDerivedStateFromProps, to handle that use case as part of their deprecation strategy of the componentWill* functions in version 16.

API Design Best Practices

"Make the change easy, then make the change."
-- Joshua Semar

API Design Best Practices

- Not all of your users are going to be power-users.
- Not all of your users are going to be novices.
- A good API should naturally encourage code that reads idiomatically. It should be inherently expressive.
- A good API should, by its very nature, be as declarative as possible, and should mask complexities and implementation details where it makes sense. Give the developer what they need to do their job, don't make them worry about how it gets done.

API Design Best Practices

Array sort methods don't typically give you the option to select your particular sorting algorithm. They simply give you a function to call when you want your array sorted. Internally, the language may opt to use an insertion, merge, or any other kind of sorting algorithm, or a mix based on its own internal rules, but as a consuming developer, those aren't necessary details, unless they're really trying to squeeze performance, in which case they'll probably implement their own custom sort function.

API Design Best Practices

Whenever possible, begin with low-level functions, and write higher-level functions that consume them. This gives your consumers a variety of granularity from which to choose.

```
import get from 'lodash.get';
import set from 'lodash.set';

export default ({
  prop: (prop, value) => {
    if (value === undefined) {
      return get(this, prop);
    }

    set(this, prop, value);
  },

  data: (id, value) => this.prop(`dataset${id === undefined ? '' : `.${id}`}`, value),

  html: (html) => this.prop('innerHTML', html),
});
```

COPY

API Design Best Practices

Whenever multiple functions share similar or overlapping functionality, see if you can consolidate them to reduce the overall API surface and internalize complexities.

```javascript
// MooTools
const myNode = $('.some-identifier'); // returns 1 node
const myNodes = $$('.some-identifier'); // returns N nodes

// jQuery
const myNode = $('.some-identifier').first(); // returns 1 node
const myNode = $('.some-identifier:first'); // returns 1 node
const myNode = $('.some-identifier:eq(0)'); // returns 1 node
const myNodes = $('.some-identifier') // returns N nodes
```

API Design Best Practices

Whenever possible, stick to a single paradigm for the API input/output

```javascript
const myJankyApi = {
  someFunction = () => true,
  someOtherFunction = () => new Promise(resolve => {
    setTimeout(() => {
      resolve(true);
    }, 1000);
  })
};

const myAwesomeApi = {
  someFunction = () => Promise.resolve(true),
  someOtherFunction = () => new Promise(resolve => {
    setTimeout(() => {
      resolve(true);
    }, 1000);
  }),
};
```

API Design Best Practices

Futureproof your inputs and/or make them extensible where it makes sense.

```
const sum = (a, b) => a + b;
const betterSum = (...args) => args.reduce((acc, arg) => acc + arg, 0);

const originalFn = (param1, param2, param3, param4 = true) => {
  // do some stuff
};

// Uh oh, new requirements for someOtherParam!
const newFn = (param1, param2, param3, someOtherParam = 'default', param4 = true) => {
  // do some stuff
};

const betterNewFn = ({ param1, param2, param3, param4 = true, someOtherParam = 'default' }) => {
  // do some stuff
};

// Now our consumers don't need to modify their call signatures until they opt-in
```

API Design Best Practices

Don't be afraid to add guardrails to protect your own systems from careless consumers, and to protect users from themselves where it might make sense. Rate-limiting can help ensure that your servers aren't performing expensive operations past the point of entry, but won't help ensure that your servers aren't being hammered by pointless requests that are going to be dropped. Request batching and throttling within the client application can help with the latter.

API Design Best Practices

```javascript
import SomeExpensiveService from './service';
const clientRateMap = new Map();
const MAX_PER_SECOND = 20;

setInterval(() => {
  clientRateMap.clear();
}, 1000);

const someEndPoint = (req, res) => {
  if (!clientRateMap.has(req.clientId)) {
    clientRateMap.set(req.clientId, 0);
  }

  const numRequests = clientRateMap.get(req.clientId);

  if (numRequests === MAX_PER_SECOND) {
    res.status(429).json({ error: `You have exceeded your plan's rate limits.` })
    return;
  }

  clientRateMap.set(req.clientId, numRequests + 1);
  res.status(200).json(SomeExpensiveService.someOperation());
};
```

API Design Best Practices

If a user is doing something within a client application that might lead to overly-aggressive DOM manipulations, consider providing a Promise-based API that batches incoming calls and defers the DOM output to the next animation frame. This will help to enhance browser performance, with a negligent impact on the application behavior.

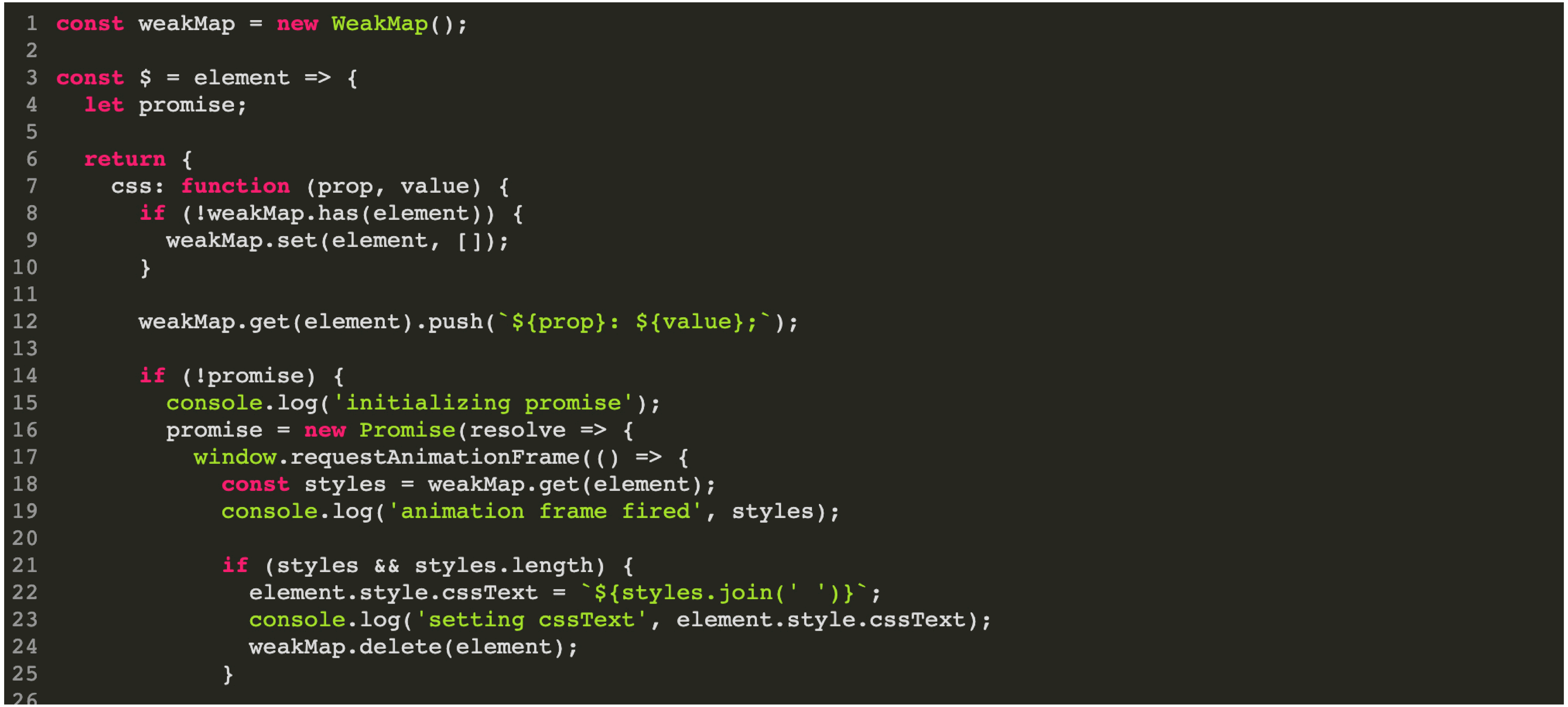
Part of the allure of the React library is that all of the logic for flushing changes to the DOM is encapsulated within the library itself, removed from the developer's purview entirely. All they need to do is write their code to recognize that given a specific state, the UI should look and function a specific way, without worrying about the timing of flushing to the DOM, or any ongoing performance optimizations of future releases.

API Design Best Practices

COPY

```javascript
const $ = element => ({
  css: function (prop, value) {
    // This will cause a repaint/reflow every time it's called
    element.style[prop] = value;

    return this;
  },
});

// Consumers
$(element).css('background-color', 'red').css('margin', '10px').css('padding', '10px');
```

API Design Best Practices



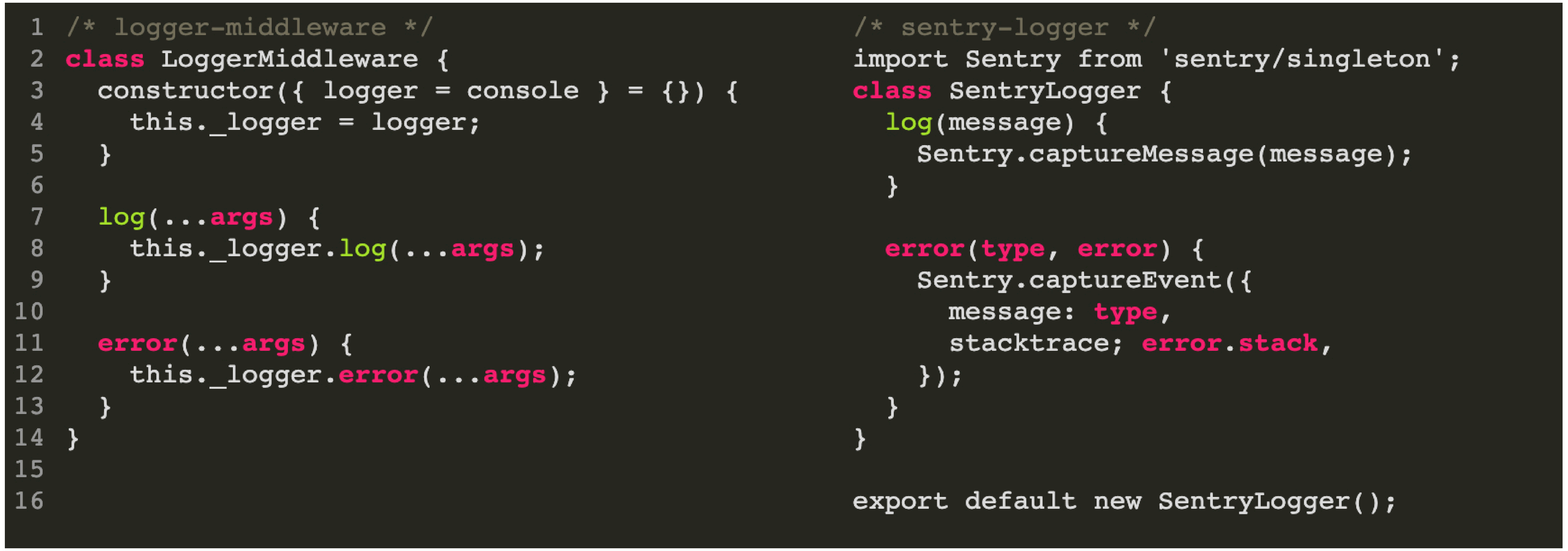
```javascript
const weakMap = new WeakMap();

const $ = element => {
  let promise;

  return {
    css: function (prop, value) {
      if (!weakMap.has(element)) {
        weakMap.set(element, []);
      }

      weakMap.get(element).push(`${prop}: ${value};`);

      if (!promise) {
        console.log('initializing promise');
        promise = new Promise(resolve => {
          window.requestAnimationFrame(() => {
            const styles = weakMap.get(element);
            console.log('animation frame fired', styles);

            if (styles && styles.length) {
              element.style.cssText = `${styles.join(' ')}`;
              console.log('setting cssText', element.style.cssText);
              weakMap.delete(element);
            }

```

API Design Best Practices

Inversion of control allows you to provide a more flexible interface
by giving up power over the implementation of a dependency (is-a)
in favor of specifying a necessary interface on a dependency (has-a).

This can still be combined with sane defaults
for your most-common use case(s).

API Design Best Practices



```javascript
/* logger-middleware */
class LoggerMiddleware {
  constructor({ logger = console } = {}) {
    this._logger = logger;
  }

  log(...args) {
    this._logger.log(...args);
  }

  error(...args) {
    this._logger.error(...args);
  }
}
```

```javascript
/* sentry-logger */
import Sentry from 'sentry/singleton';
class SentryLogger {
  log(message) {
    Sentry.captureMessage(message);
  }

  error(type, error) {
    Sentry.captureEvent({
      message: type,
      stacktrace; error.stack,
    });
  }
}

export default new SentryLogger();
```

API Design Best Practices

```
/* console-logger-middleware.js */
export default new LoggerMiddleware();

/* sentry-logger-middleware.js */
import sentryLogger from './sentry-logger';

export default new LoggerMiddleware({ logger: sentryLogger });

// Differentiating by environment
/* logger-middleware.js */
let logger;
if (process.env.NODE_ENV === 'production') {
  logger = require('./sentry-logger-middleware');
} else {
  logger = require('./console-logger-middleware');
}

/* downstream consumer */
this.logger.log('Something went bonkers!');

try {
  doSomething();
} catch (error) {
  this.logger.error('Bailed out early!', error);
}
```

API Design Best Practices

Class/method decorators can help inform consumers of ongoing changes:

```
class ThisClassIsInFlux {
  @deprecate('oldAndBustedFn has been deprecated. Please call newHotnessFn in the future.')
  oldAndBustedFn(param1) {
    /**
     * If you're simply replacing one function with another, and the new one has a
     * similar interface, consider making oldAndBustedFn a proxy to the new one, so
     * that the consumer is already using the new function and simply needs to be
     * alerted to any outstanding calls to the oldAndBustedFn to be replaced. If
     * this isn't the case, consider including a link in the deprecation message,
     * to direct the developer to information on how to migrate their function calls.
     **/
    return this.theNewHotnessFn({ param1 });
  }

  theNewHotnessFn({ param1 }) {
    return whateverThisFnDoes(param1);
  }
}
```

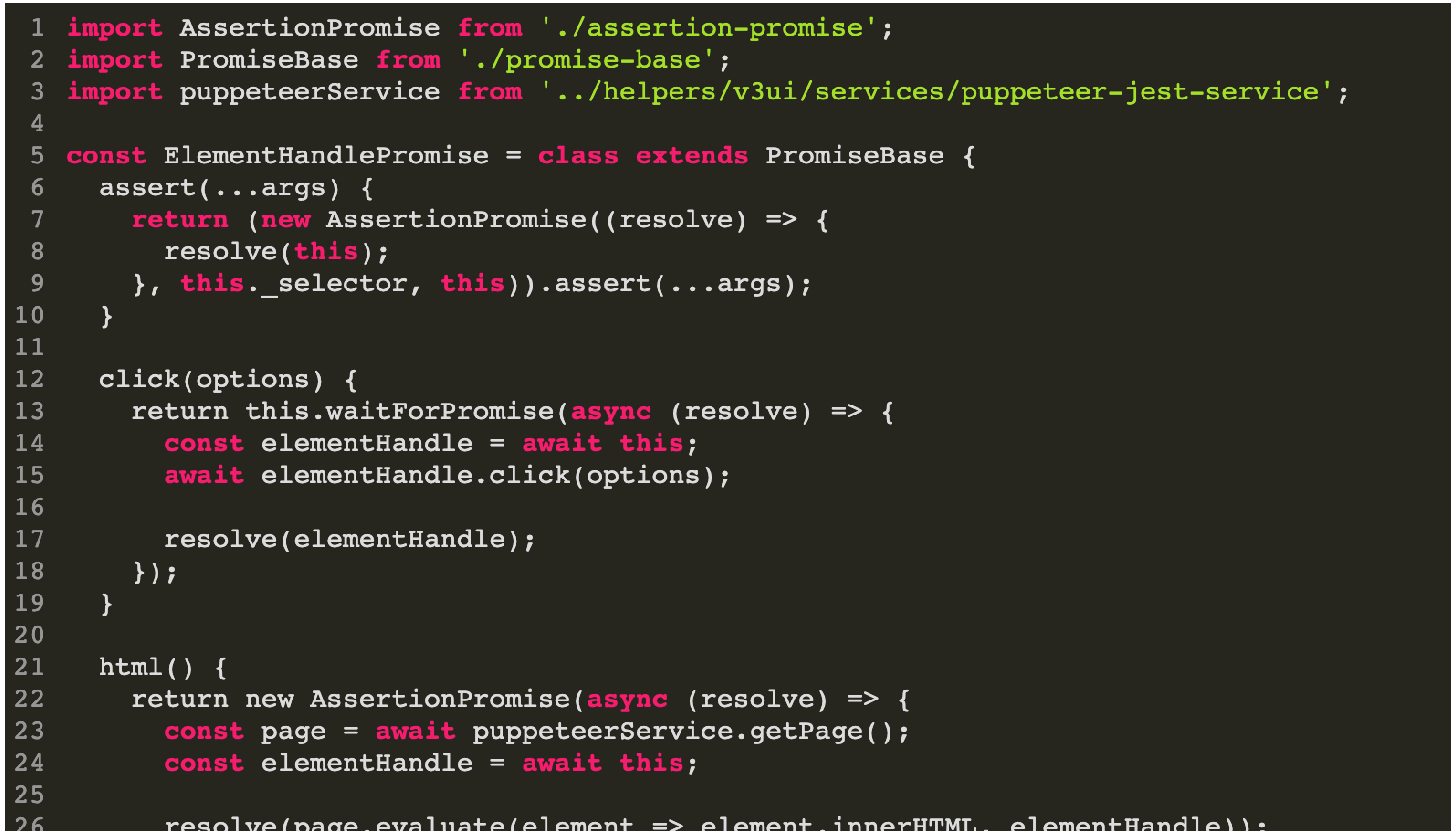
Authoring your own APIs

- When your entire API uses a consistent interface (e.g. Promise-based), composition becomes much simpler, and building higher-level abstractions becomes almost common-sense in some cases.
- Given a low-level API, don't be afraid to build your own custom APIs on top to simplify your own workflows where it makes sense. If you're lucky, you could be the next big library author!

Authoring your own APIs

When authoring APIs, think about your own favorite API experiences, and what you loved most about them. Draw inspiration where it makes sense, even if it feels derivative. In software engineering, people care more about the comfort of familiar and predictable interfaces than they do about original thinking for the sake of being original.

Authoring your own APIs



```
import AssertionPromise from './assertion-promise';
import PromiseBase from './promise-base';
import puppeteerService from '../helpers/v3ui/services/puppeteer-jest-service';

const ElementHandlePromise = class extends PromiseBase {
  assert(...args) {
    return (new AssertionPromise((resolve) => {
      resolve(this);
    }, this._selector, this)).assert(...args);
  }

  click(options) {
    return this.waitForPromise(async (resolve) => {
      const elementHandle = await this;
      await elementHandle.click(options);

      resolve(elementHandle);
    });
  }

  html() {
    return new AssertionPromise(async (resolve) => {
      const page = await puppeteerService.getPage();
      const elementHandle = await this;

      resolve(page.evaluate(element => element.innerHTML, elementHandle));
```

Summation

- Good APIs encourage well-read, declarative code
- Minimize API surface where you can
- Don't be afraid to ***not*** reinvent the wheel
- Consistent interfaces for predictability and composition
- Reduce the risk of footguns anywhere you can
- Unobtrusive messaging can help consumers stay up to date
- Dogfood/Champagne/Your-favorite-metaphor here

fn.