Bringing Components to Adblock Plus UI

The goal of this shared document is to discuss possible implementation of ABP UI components.

Reasons

Current ABP popup, and in general the whole ABP UI, is made of single, big, pages with entire layout and logic in single files: popup.html, desktop-options.html, mobile-options.html, etc.

Accordingly, we are incapable to reuse similar UI components (checkboxes, toggles, lists) and related logic in different parts of the same page, or cross pages.
This does not play well with the idea of standardizing, and synchronize, similar parts of the UI across our pages and sections.

This also makes it impossible to simplify layout and logic by splitting it in sub modules / components: easier to maintain, easier to test, and also easier to be replaced when/if needed.

Last, but not least, in case we'd like to reuse our UI on Web or other projects, it's basically impossible right now.

Proposal

After successfully putting together few components (switcher, "cards", bubble-ui), all of them with related CSS and JS that does not interfere with each other, I've decided to propose, or at least discuss, currently adopted solution, before I'll start pushing components for code reviews.

In a nutshell

- even if ideal, avoid Custom Elements due relatively poor adoption and worse accessibility by default
- do not trash current layout logic, make i18n possible without too much effort, keep semantics in the HTML where appropriated
- define a class per each component to drive behaviors, not styles and not HTML (as much as possible, where possible)
- keep CSS a part so that one component has a basic HTML definition (documentable), some related CSS to include when needed, and some JS to include, also when needed.

Common needs / principles

- each component might have a unique `id` for the simple reason that most a11y works by ids for labels, popups, and descriptions
- each components is initialized on top of the regular layout, and will enrich the behavior and/or place the i18n string in it as needed

A simple base class that provides common needs would accept a dictionary we can use as we need, accordingly with the component, having at least one property: `target`

```
let counter = 0;
class BaseComponent {
constructor(details) {
  this.id = [this.constructor.name.toLowerCase(), counter++].join('-');
  this.target = typeof details.target === 'string' ?
                document.querySelector(details.target) : details.target;
  }
}
```

Such property can be either a CSS selector or, directly, a DOM node.

Common definition - HTML

Taking the bubble-card component, as example, we can define the layout as such:

```
<div data-component="bubble-card" data-icon="blocked">
    <span class="count">0</span>
    <span class="info">ads blocked</span>
</div>
```

This could even be inside a template, if we want, as long as all the bits we want/need are in place. Above layout can be tested in isolation, providing also some CSS and, eventually, JS.

Common definition - CSS

Each component should be testable in isolation, giving us the ability to start testing parts of our UI and its related logic. Each CSS per each component should work out of the box and be as less obtrusive as possible, like in this case:
https://webreflection.github.io/eyeo/bubble-card/class.css

It is OK to assume some common CSS applied in the outer container such
`box-sizing: border-box;`
or even a default font-family, and it would be silly/redundant to repeat these all over every time.

Last, but not least, each component should be able to render itself in RTL direction and respect as much as possible all accessibility concerns through HTML, CSS, and JS too.

Common definition - JS

As visible in this example file, a component needs to follow a common initialization and do some internal operations to ensure its behavior.

```
class BubbleCard extends BaseComponent {
 constructor(details) {
   // obtain a target and a unique id
   super(details);
   // setup current element
   this.count = details.count || 0;
   // eventually populate the component
   // with the right i18n text then update
   // its content
   this.update();
 }
 update() {
   // populate the counter with a value
   this.target.querySelector('.count').textContent = this.count;
 }
}

// initialize components once DOMContentLoaded happens
document.addEventListener(
 'DOMContentLoaded',
 () => {
   // but every component could be initialized at any time
   const cards = document.querySelectorAll("[data-component=bubble-card]");
   for (const el of cards)
     // define a target and a counter
     new BubbleCard({
       target: el,
       count: (Math.random() * 20) >>> 0
     })
 },
 {once: true}
);
```

As commonly happens in many UI related libraries, components will interact with each other through events and/or properties (we can use MutationObserver per attributes and react/update accordingly).

Nice to have, but not essential.

It can be boring to use directly DOM APIs to create nodes full of aria attributes, when necessary. It is also not immediately obvious to spot, layout speaking, when/if how these properties have been set, neither is as simple/straightforward to update them.

I'd like to simplify every dynamic layout creation through *hyperHTML* which demonstrated already great performance and flexibility with the infinite-list component.

Following the strengths of the library:

- It's fully based on JS and DOM standard. No wheel reinvented, no vDOM needed
- It's blazing fast in DOM updates without trashing layout and preserving RAM
- It's been used in production and it has a little but active/vibrant community
- It weights only 5K and it doesn't require any transformation/toolchain/runtime
- It supports self closing tags which plays particularly well with our layouts where the text is defined by either class or attributes

```
<button i18n="agree-on-activating-aa" onclick=${behavior}/>
```

Differently from JSX and React, it doesn't have vDOM overhead and layout doesn't require transformation but its strings can be highlighted in some IDE too (example: VS Code).

Again, this is not a blocker for components I'm working on right now but it's been used due performance benefits over simplicity for the infinite list.

If you'd like to know more I'll put here links:

- Documentation
- Repository
- Examples and comparison VS most known frameworks (+ live examples)
- HNPWA demo (which does SSR too), which is still the fastest HNPWA of them all (look for Viper-news)

If interested, beside having already light components in core, the next release goal is to bring in hydratation of already available layout too.

Thanks for consideration.