JavaScript Framework for Themes

> 💡 A lightweight, maintainable framework for managing JavaScript interactions in PrestaShop themes (Hummingbird) using semantic `data-*` attributes.

🧭 Introduction

The Hummingbird theme currently lacks standardized JavaScript conventions. This framework introduces a simple and consistent approach to managing component logic, dynamic content updates, and UI interactions using `data-*` attributes and a centralized selector map.

This new JavaScript framework is not just a theme enhancement — it also has important implications for the PrestaShop core.

To fully adopt this architecture, parts of the PrestaShop core will need to:

- Output consistent `data-ps-*` attributes in dynamic content (e.g., cart updates, modals, quickview)
- Avoid relying on hardcoded `.js-*` classes like `.js-cart-summary` or inline event handlers

- Expose structured data through `data-ps-data` instead of attributes like `data-product`

This will ensure that both the core and the theme can operate in a declarative, modular, and maintainable way — resulting in a cleaner development experience for themes and front-end modules.

▼ ⚠️ PrestaShop Template Dependencies

Some parts of the PrestaShop core rely on refreshing specific `.tpl` files using Ajax. These files not only need to exist, but also must contain certain expected DOM structures or selectors.

For instance, if the core expects `#cart-summary-product-list` inside `cart-detailed.tpl`, and this element is removed or renamed, features like cart refresh will break without error.

To safely adopt the Hummingbird framework:

- Either the PrestaShop core must evolve to use `data-ps-*` selectors instead of `.js-*` or hardcoded IDs

Therefore, some `.tpl` files must remain structurally intact, unless you also modify the related core JavaScript that depends on them.

🎯 Goals

- Easy and consistent JS element targeting
- Clear separation between components, actions, targets, and states
- Support for dynamic DOM updates (e.g., Ajax or quickview)
- Prevent conflicts with Bootstrap or other JS libraries
- Seamless integration with PrestaShop modules and the theme system
- Clearly separate styles from Javascript

✅ Why This Framework Matters

🔄 Declarative DOM Mapping

Using `data-ps-*` attributes makes the relationship between HTML and JavaScript explicit. This declarative mapping:

- Makes the DOM self-documenting

- Reduces reliance on class-based selectors
- Encourages separation of behavior and style

♻️ Reusability and Maintainability

Each component is initialized via `data-ps-component`, making it reusable and modular. Logic is encapsulated per element, enabling:

- Easier debugging and isolation
- Cleaner separation of concerns
- Independent testing and reuse

⚡ Resilience to DOM Changes

Since PrestaShop often updates sections of the DOM dynamically (via Ajax), a MutationObserver tied to `data-ps-observe` ensures:

- Components rebind automatically
- Logic stays intact without manual reinitialization

🧭 Context-Aware Behavior

Using `data-ps-context` on the `<body>` allows your JS to:

- Execute only on specific page types
- Avoid side effects across unrelated views

🧠 Simplified Event Delegation

`data-ps-action` supports centralized event handling via delegation. This is especially useful for:

- Pages with dynamic content (e.g., product list, cart)
- Reducing the number of event listeners
- Making your logic future-proof and Ajax-friendly

🧩 Centralized Selectors

A shared `selectors.js` map:

- Promotes consistency across components and modules
- Avoids hardcoded classes like `.row`, `.btn`, or `.col`

- Makes it easier to audit and refactor logic

> 🧠 This framework is designed to make your PrestaShop front-end more predictable, robust, and scalable — both in Hummingbird and in future themes.

🏷️ Naming Convention Overview

We define a set of `data-ps-*` attributes to organize behaviors:

Attribute	Purpose	Typical Use Case
`data-ps-context`	Identify the current page or context scope	Conditional logic based on page context
`data-ps-template`	Reference a template name to use it in JS	Identify reusable templates for rendering blocks
`data-ps-component`	Initialize JS components	Component behavior like custom components (quickview...)
`data-ps-state`	Represent UI states (e.g., loading)	Show loading, active, or error states
`data-ps-action`	Trigger specific JS actions	Buttons, form submissions, JS logic
`data-ps-target`	Target areas for content updates	Inject/refresh content like cart summaries or product infos
`data-ps-ref`	Reference DOM element for JS logic or data	Target non-component elements for logic/data
`data-ps-data`	Store or expose data for JS usage	Embed structured data like product infos
`data-ps-observe`	Watch elements for DOM mutations	Refresh/rebind components on partial updates
`data-ps-test`	Reference element for end-to-end or unit testing	A way to select an element for testing purposes (❗ **as a last resort**)

1️⃣ `data-ps-component`

Attach JavaScript behavior or initialize a component.

```
<div data-ps-component="voucher-accordion"></div>
```

Use Cases:

- Run JS logic on page load
- Set up event listeners or lifecycle behavior
- Re-initialize after Ajax updates

2️⃣ `data-ps-action`

Trigger a behavior—typically used on buttons, links, or inputs.

```
<button data-ps-action="remove-voucher">Remove</button>
<a href="#" data-ps-action="add-to-cart">Buy</a>
```

Use Cases:

- Delegate events efficiently
- Map actions to JS functions
- Avoid inline `onclick` attributes

3️⃣ `data-ps-target`

Mark a DOM element as a target for injecting or updating content.

```
<div data-ps-target="cart-summary"></div>
```

Use Cases:

- Ajax or JS-driven content updates
- Template injection
- Scoped DOM updates

4️⃣ `data-ps-state`

Track or toggle UI states (e.g., `open`, `active`, `loading`).

```
<div data-ps-state="loading"></div>
```

Use Cases:

- CSS class toggling via JS
- UI state handling
- Accessibility improvements

5️⃣ `data-ps-observe`

Watch for DOM changes using `MutationObserver`.

```
<div data-ps-observe data-ps-component="cart-line"></div>
```

Use Cases:

- Detect with `MutationObserver`
- Rebind component behavior dynamically
- Handle dynamic DOM updates triggered by Ajax or JavaScript

6️⃣ `data-ps-data`

Expose embedded JSON or inline data for use in JavaScript logic.

```
<div data-ps-data="{'id':'123','name':'Product A','pric
e':'19.99'}"></div>
```

Use Cases:

- Pass structured data from server-rendered HTML to JS
- Useful for product pages, faceted filters, or module context

Best Practice:

- Keep values minimal and scoped (avoid large full-object dumps)

7️⃣ `data-ps-context`

Declare the current page or scope for context-aware JavaScript logic.

```
<body data-ps-context="product"></body>
```

Use Cases:

- Load or limit scripts/components to specific page types

- Add global guards or routing conditions in JS
- Makes behavior logic cleaner and less dependent on template class names

8️⃣ `data-ps-template`

Declare a reference to a template name used for rendering or replacement by JS logic.

```
<template data-ps-template="cart-line"></template>
```

Use Cases:

- Declarative marker for partials or DOM fragments
- Helps standardize dynamic UI generation

9️⃣ `data-ps-ref`

Declare a DOM reference for use in JavaScript logic (when not a full component).

```
<div data-ps-ref="product-pricing" data-ps-data="{'price':
'19.99'}"></div>
```

Use Cases:

- Reference non-component DOM nodes (e.g., pricing, meta info)
- Hook logic to semantic UI elements without a full component wrapper
- Safer and cleaner than using classes or ID

🔍 Centralized Selector Map

Using a centralized selector map is strongly recommended — but it must align with the `data-ps-*` convention.

```
// Example
export const productPage = {
  productCombinations: '[data-ps-target="product-combinatio
ns"]',
  addToCart: '[data-ps-action="add-to-cart"]',
```

```
  productZoom: '[data-ps-component="product-zoom"]',
};
```

Usage Example:

```
import { productPage } from './selectors';
document.querySelectorAll(productPage.productZoom).forEach
(initComponent);
```

Benefits:

- Enforces `data-ps-*` consistency across all modules
- Centralized and versionable logic for future maintainability
- Easier to track usage of behavior-driven elements

> 🚨 Avoid using styling/structural classes like `.row`, `.col`, `.btn` in JS. Always prefer semantic selectors via `data-ps-*` and central map references.

🔁 Dynamic Observation with `MutationObserver`

PrestaShop frequently injects HTML dynamically via Ajax (e.g., cart refresh, quickview, filters). If your JS only runs once on `DOMContentLoaded`, it will miss these changes.

`data-ps-observe` + `MutationObserver` solves this:

- Automatically rebind components when the DOM is updated
- No need to manually re-init logic after every Ajax call
- Keeps behavior stable in dynamic environments

This ensures your components are **always alive**, even if PrestaShop reloads parts of the page.

⚡ Event Delegation for Actions

The Hummingbird framework encourages the use of event delegation for dynamic behaviors. Instead of binding event listeners directly to every

interactive element, listeners can be registered at the `document` or `block` level. This:

- Reduces the number of active listeners in memory
- Ensures compatibility with dynamically injected elements
- Makes behavior handling more centralized and consistent
- Is especially effective in PrestaShop, where parts of the page (e.g., cart, quickview, filters) are often refreshed dynamically

Example:

```
document.addEventListener('click', (e) => {
  const el = e.target.closest('[data-ps-action]');
  if (!el) return;

  const action = el.dataset.psAction;

  switch (action) {
    case 'add-to-cart': {
      addToCart(el);
      break;
    }
    case 'remove-from-cart': {
      removeFromCart(el);
      break;
    }
    default:
      console.warn(`Unhandled action: ${action}`);
  }
});
```