

Du web moderne sans framework JavaScript, est-ce possible ?

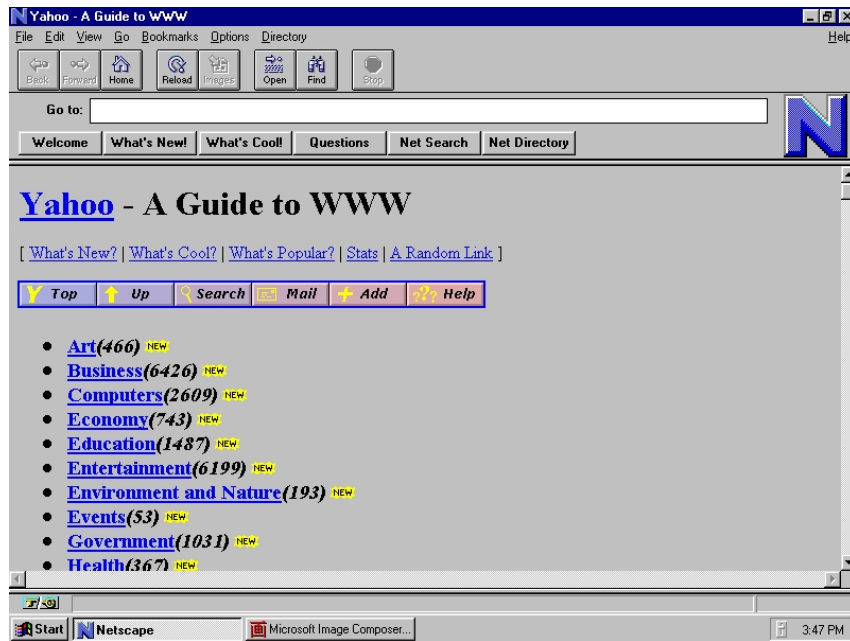
Sonny LION





Web 1.0 : L'âge statique (1989-2004)

- Basé sur HTTP, URL et HTML
- Expérience utilisateur passive : pages statiques, peu d'interactions (formulaires, liens)
- Limitations techniques : bande passante réduite, ordinateurs et navigateurs limités





Web 2.0 : L'âge interactif (après 2004)

- Nouvelles possibilités grâce à l'ajout de CSS, JavaScript, (Flash - 🦴)
- Amélioration de la bande passante et de la puissance des appareils
- Expérience interactive : blogs, réseaux sociaux, médias interactifs





Le web d'aujourd'hui

- Usage mobile prédominant
- Évolution des technologies web : d'HTML5, CSS3 à WebAssembly, WebGPU, etc.
- Limite entre applications web et applications natives de plus en plus floue
- Progressive Web Apps (PWA - utilisation hors ligne, notifications)
- Solutions de packaging d'applications web (Electron, Tauri)





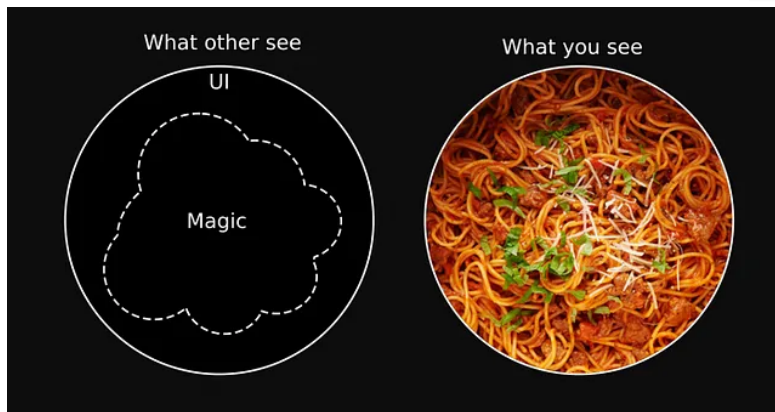
jQuery : Le couteau suisse du web (depuis 2006)

- Simplification de la manipulation du DOM (avec le fameux `$()`)
- Compatibilité cross-navigateurs
- Adoption massive par les développeurs web
 - En 2024, jQuery est toujours utilisé dans environ 75% des sites web (source : W3Techs)

Mais ...

Code difficile à maintenir quand les applications deviennent complexes

Gestion d'événements et manipulation du DOM dans du JS non structuré == **code spaghetti** !

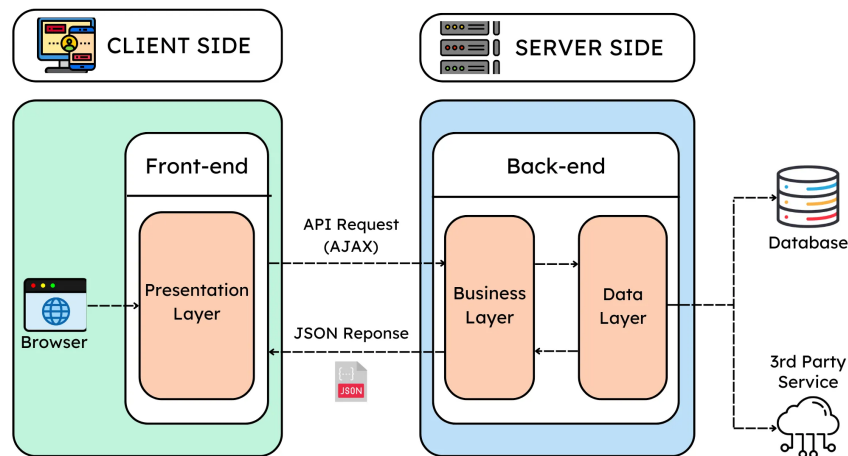




La voie des frameworks JavaScript

Les applications monopages (SPA)

- Expérience utilisateur plus fluide sans rechargement de page
- Architecture basée sur des composants réutilisables
- Système de gabarits ou JSX (JavaScript XML) pour décrire les composants
- Séparation client-serveur claire (échange de données en JSON)





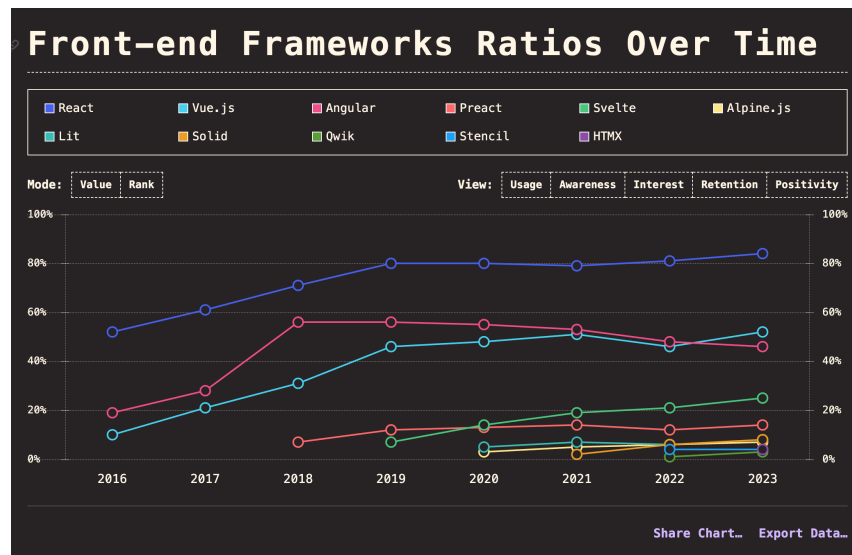
La jungle des frameworks JavaScript

Côté client :

- AngularJS (2010)
- React (2013)
- Vue.js (2014)
- Svelte, Preact, etc.



Et même côté serveur avec NextJS, NuxtJS, etc.



Source : State Of JS 2023



Une voie alternative ?

Il existe des alternatives mais elles ne seront pas adaptées à tous les cas d'usage :

- WebAssembly : exécuter du code C, Rust, etc. dans le navigateur
- Dash, Streamlit : surcouche Python pour créer des applications web interactives
- HTMX et Alpine.js : interactivité des pages web sans presque écrire aucune ligne de JavaScript












Focus sur htmx





Htmx en quelques mots

htmx est une bibliothèque JavaScript qui permet de créer des applications web interactives sans écrire de JavaScript.

-  Requêtes AJAX sans JavaScript
-  Extends HTML : Tout élément peut émettre des requêtes HTTP (GET, POST...)
-  Backend-Agnostique : Compatible avec toutes les technologies backend (PHP, Python, Go...)
-  Rendu serveur : Améliore le référencement naturel (SEO), l'accessibilité et les performances
-  WebSockets : Support du temps réel
-  Taille réduite : Seulement **16,9 kB** minifié + gzippé (en comparaison, Vue.js 3.5.7 fait **44,9 kB**)
-  Facile à apprendre : Idéal pour les développeurs non familiers avec JavaScript (et aussi pour ceux qui le détestent...)



Installation

Via un CDN (le plus simple) :

```
<script src="https://unpkg.com/htmx.org@2.0.2"
  integrity="sha384-Y7hw+L/jvKewIIRrkqWYfPcvVxHzVzn5REgzbawhxAuQGwX1XWe70vji+VSeH0ThJ "
  crossorigin="anonymous">
</script>
```

Via un gestionnaire de paquets (npm ou yarn) :

```
npm install htmx.org@2.0.2
```



Comment ça marche ?

HTML standard

```
<a href="/blog">Blog</a>
```

Cette balise dit au navigateur :

"Quand un utilisateur clique sur ce lien, fais une requête HTTP GET vers `/blog` et charge le contenu de la réponse dans la fenêtre du navigateur."

htmx

```
<button hx-post="/clicked"  
  hx-trigger="click"  
  hx-target="#parent-div"  
  hx-swap="outerHTML"  
>  
  Click Me!  
</button>
```

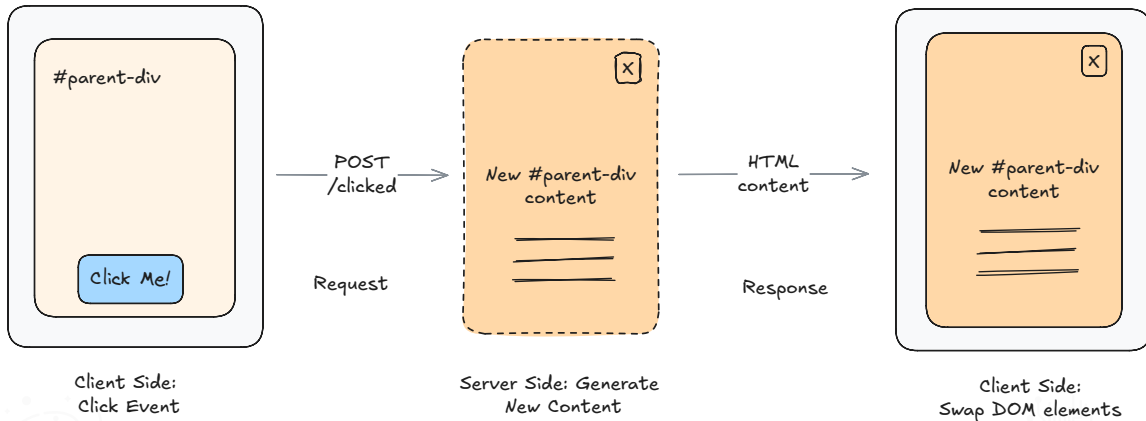
Ce bouton dit à htmx :

"Quand un utilisateur clique sur ce bouton, fais une requête HTTP POST vers `/clicked` et remplace le contenu de l'élément avec l'id `parent-div`."



Et en images ...

```
<button hx-post="/clicked"  
  hx-trigger="click"  
  hx-target="#parent-div"  
  hx-swap="outerHTML"  
>  
  Click Me!  
</button>
```





Tout se joue dans les attributs HTML






Attribut	Description
<u>hx-get</u>	effectue un <code>GET</code> vers l'URL spécifiée
<u>hx-on*</u>	gérer des événements avec des scripts inline sur les éléments
<u>hx-push-url</u>	ajoute une URL dans la barre d'adresse pour créer un historique
<u>hx-select</u>	sélectionner le contenu à remplacer dans une réponse
<u>hx-vals</u>	ajouter des valeurs à envoyer avec la requête (format JSON)

La documentation complète est disponible sur le site officiel : htmx.org







Et si on ajoutait de la modularité avec django-components ?

Fonctionnalités

-  Réutilisabilité
-  Encapsulation : HTML, CSS, JS propres à chaque composant
-  Rendu serveur : Améliore SEO et temps de chargement
-  Intégration Django : Utilise les concepts Django natifs
-  Chargement asynchrone : Compatible avec HTMX et Alpine.js

Avantages

-  Moins de duplication de code
-  Meilleure maintenabilité
-  Gestion simplifiée des UIs complexes
-  Collaboration frontend-backend facilitée

La documentation complète est disponible sur le site officiel : [django-components](#)



Votre premier composant Django

Les composants sont des classes Python qui héritent de `django_components.Component` .

Avec des attributs `template` , `css` et `js` pour le contenu HTML, CSS et JavaScript.

```
# In a file called [project root]/components/calendar.py
from django_components import Component, register, types
```

```
@register("calendar")
class Calendar(Component):
    def get_context_data(self, date=None):
        return {
            "date": date,
        }
```

```
template: types.django_html = """
    <div class="calendar-component">
        Today's date is <span>{{ date }}</span>
    </div>
    """
```

```
css: types.css = """
    ...my css code here...
    """
```

```
js: types.js = """
    ...my javascript code here...
    """
```

Le code peut aussi être organisé en plusieurs fichiers pour plus de clarté.



Puis dans votre template Django...

On charge d'abord `component_tags`, puis on utilise les tags `component_*_dependencies` et `component` pour faire le rendu du composant sur la page.

```
{% load component_tags %}
<!DOCTYPE html>
<html>
<head>
  <title>My example calendar</title>
  {% component_css_dependencies %}
</head>
<body>
  {% component "calendar" date="2015-06-19" %}
  {% endcomponent %}
  {% component_js_dependencies %}
</body>
</html>
```

Et le rendu final :

```
<!DOCTYPE html>
<html>
  <head>
    <title>My example calendar</title>
    <link
      href="/static/calendar/style.css"
      type="text/css"
      media="all"
      rel="stylesheet"
    />
  </head>
  <body>
    <div class="calendar-component">
      Today's date is <span>2015-06-19</span>
    </div>
    <script src="/static/calendar/script.js"></script>
  </body>
</html>
```



Maintenant on mixe htmx et django-components...

Côté serveur, on crée une route qui renvoie le composant.

```
from django.urls import path
from components.calendar import Calendar
urlpatterns = [
    path('calendar/', Calendar.as_view()),
]
```

Et côté client, on utilise htmx pour charger le composant.

```
<button hx-get="/calendar/" hx-swap="outerHTML" hx-trigger="click" hx-target="#calendar-container">
    Click to load/open the calendar
</button>
```

Et voilà ! Une page web interactive sans écrire une seule ligne de JavaScript !





Un exemple concret d'utilisation

La mise à jour en masse de contacts dans une application de gestion de contacts.

	Name	Email	Status
<input checked="" type="checkbox"/>	Kathy Lang	kathy.lang@gmail.com	Active
<input type="checkbox"/>	Roberto Perez	roberto.perez@hotmail.com	Inactive
<input type="checkbox"/>	Cody House	cody.house@yahoo.com	Active

Activate

Deactivate



Le composant Table

```
@component.register("table_bulk_update")
class TableBulkUpdateComponent(component.Component):
    template = """
        <form id="checked-contacts">
            <table class="table">
                <... headers .../>
                <tbody id="tbody">
                    {% component "tbody_bulk_update" contacts=contacts %}{% endcomponent %}
                </tbody>
            </table>
        </form>
        <div class="mt-4" hx-include="#checked-contacts" hx-target="#tbody">
            <button class="btn-primary"
                hx-post="{% url 'contacts_bulk_update' update='activate' %}">Activate</button>
            <button class="btn-secondary"
                hx-post="{% url 'contacts_bulk_update' update='deactivate' %}">Deactivate</button>
        </div>
    """

    def get_context_data(self, **kwargs):
        return {"contacts": Contact.objects.all().order_by("id")}
```



Le composant TBody

```
from django_components import component

from app.models import Contact

@component.register("tbody_bulk_update")
class TBodyBulkUpdateComponent(component.Component):
    template = """
        {% for contact in contacts %}
        <tr class="tr {% if contact.id in ids %} {{ update }}
            {% endif %}">
            <td class="td">
                <input class="checkbox"
                    type='checkbox'
                    name='ids'
                    value='{ contact.id }'>
            </td>
            <td class="td">{{ contact.first_name }}</td>
            <td class="td">{{ contact.email }}</td>
            <td class="td">{{ contact.status }}</td>
        </tr>
        {% endfor %}
    """

    def get_context_data(self, contacts, **kwargs):
        return {"contacts": contacts}
```

```
def post(self, request, update, *args, **kwargs):
    if update == "activate":
        Contact.objects.filter(
            id__in=request.POST.getlist("ids")
        ).update(
            status="Active"
        )
    elif update == "deactivate":
        # ... same as above but with status="Inactive"
    context = {
        "contacts": Contact.objects.all().order_by("id"),
        "update": update,
        "ids": [
            int(id_)
            for id_ in request.POST.getlist("ids")
        ],
    }
    return self.render_to_response(context)
```



L'URL pour mettre à jour les contacts

```
from django.urls import path

from components.bulk_update.tbody import TBodyBulkUpdateComponent

urlpatterns = [
    path(
        "contacts/<str:update>",
        TBodyBulkUpdateComponent.as_view(),
        name="contacts_bulk_update",
    ),
]
```

Et c'est tout !

Plus d'exemples [ici](#) (source : [django-htmx-components](#))





Retour d'expérience : htmx vs React (+Django)

Points forts de htmx :

- Bibliothèque légère et sans dépendances : simplicité de mise en place et de maintenance
- Moins de code : Pas de gestion d'état client (Redux, MobX), pas de duplication de code client-serveur
- Pas de cycle de vie complexe : Évite les effets de bord
- Facile à tester : Les tests se concentrent côté serveur
- Flexibilité : Adapté à la plupart des cas d'usage (en ajoutant un peu de JS si nécessaire)



Retour d'expérience : htmx vs React (+Django)

Inconvénients (mineurs) :

- Peu de composants prêts à l'emploi, mais intégration possible avec Bootstrap, Tailwind, etc.
- Communauté petite par rapport à React (mais toutes les ressources Django sont utilisables)
- Templates Django moins lisibles que JSX si le code n'est pas bien structuré

Attention : htmx n'est pas une solution miracle et ne conviendra pas à tous les cas d'usage. Pour des applications plus complexes (Google Sheets, Figma), ou des besoins spécifiques (application disponible hors ligne), il sera plus judicieux d'utiliser un framework JavaScript.

Autre témoignage : [DjangoCon 2022 | From React to htmx on a real-world SaaS product: we did it, and it's awesome!](#)





Merci pour votre attention !

Pour aller plus loin :

- htmx.org
- [django-components](https://django-components.com)
- alpinejs.dev - Pages interactives sans JavaScript côté client
- tailwindcss.com - Simplifier la gestion du CSS