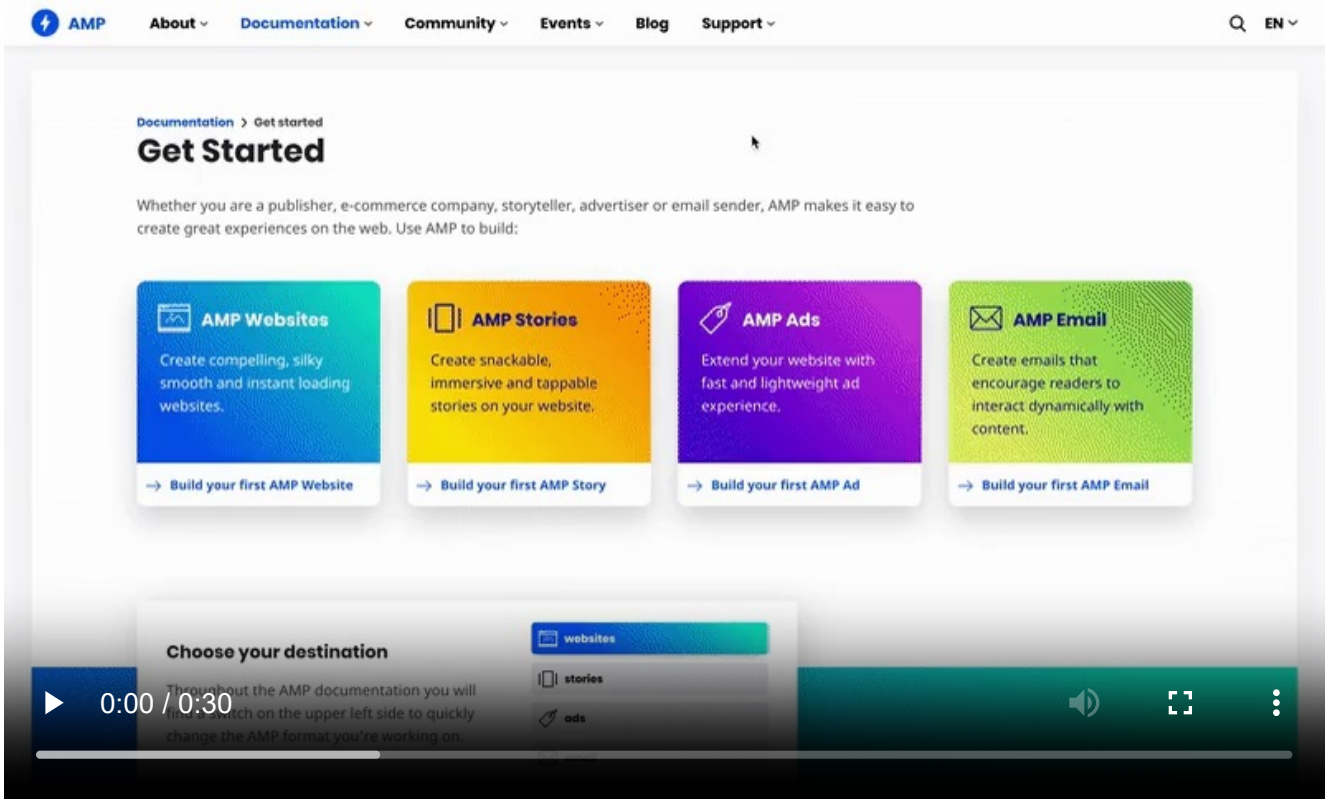
February 14, 2020

In search of the amp.dev search

Websites

Editor's Note: the following guest post was written by Thorsten Harders and Sebil Satici, Developers at Jung von Matt.

Amp.dev offers a lot of resources that make it easier to get started and build with AMP: case studies, details on how each component works, guides with best practices, step-by-step tutorials and plenty of executable code examples. In fact, it offers so many resources that we needed a way for developers to quickly discover and reach all that existing content. We wanted to do so by making all of our content searchable directly on the site. This article walks through the steps we took to implement search on amp.dev.



TL;DR: The new amp.dev search is built using **<amp-lightbox>**, **<amp-list>**, **<amp-autocomplete>**, **<amp-mustache>** and of course **<amp-bind>** to glue them all together. It uses a custom API that is backed by **Google custom search** and it uses **Service Worker functionalities** to cache the latest search query and results.

Built with AMP components

We had three goals when building the amp.dev search functionality:

1. The search should be accessible in its own layer from all pages.
2. The search should be optimized for finding AMP components, as these are core to the AMP developer experience.
3. The search should be implemented using valid AMP functionalities.

Hide and show the search layer (with amp-lightbox)

Looking around the web you'll see many different types of search functions. Some are just plain input fields while others expand and collapse with fancy animations. Some trigger a page-load to paginated result pages, others asynchronously display results. Some auto-suggest search terms, others don't. For amp.dev we wanted to combine the best of those approaches to come up with a solution that manages to stay out of the user's way as much as possible while at the same time being quickly accessible. Therefore, we decided to encapsulate the whole search in a fullscreen layer to:

- suggest interesting articles to users (e.g new guides or components that have been published) no matter what page the user is on
- avoid distracting the user with other elements on the page
- quickly display the search results inline without having to load an extra results page

To accomplish our needs we have decided to continue with <amp-lightbox>. It makes it easy for us to hide and show the search layer while offering us helpful actions and events we could use for the integration in our AMP frontend.

Adding seamless interaction

From a user experience perspective, it was particularly important to us that the user achieves a seamless transition between entering the search query and seeing the result. When the user opens the search layer, the input field is automatically focused and the user can start typing right away. When the search layer is closed, we focus the search toggle again, so that keyboard users can continue in the same position as before.

We implemented this feature using the <amp-lightbox> open and close events in combination with the global focus action:

```
<amp-lightbox layout="nodisplay"
      on="lightboxOpen:searchInput.focus;
          lightboxClose:searchTriggerOpen.focus"
      scrollable>
```

Listing the search results (with amp-list and amp-mustache)

For displaying the search results on the page, we use the <amp-list> component as it has paging and infinite scroll already built-in – exactly what you need when listing search results. The actual search is implemented server-side and can be accessed via an API endpoint `/search/do` which returns a JSON object similar to this:

```
{
  "result": {
    "pageCount": 10,
    "pages": [
      {
        "title": "Some title",
        "description": "Description",
        "url": "http://amp.dev/some/link"
      },
      ...
    ]
  },
  "nextUrl": "/search/do?q=amp&page=2"
}
```

We use amp-bind to update the full search URL in our `<amp-list>`, by binding the `[src]` attribute to the input query. To enable infinite scrolling, we set the load-more attribute and for a cleaner reload experience when triggering subsequent searches, we set the reset-on-refresh attribute. In the `<amp-mustache>` template we use the data from the result object to render the list dynamically. Here is the code:

```
<amp-list id="searchList"
          src="/search/initial-items"
          [src]="query ? '/search/initial-items : '/search/do?q=' +
encodeURIComponent(query)"
          binding="no"
          items="."
          height="80vh"
          layout="fixed-height"
          load-more="auto"
          load-more-bookmark="nextUrl"
          reset-on-refresh
          single-item>
  <template type="amp-mustache">
    <div class="search-result-list">
      {{#result.pages}}
```

```
      <a href="{{url}}">
        <h4>{{title}}</h4>
        <p>{{description}}</p>
      </a>
      {{/result.pages}}
    </div>
  </template>
</amp-list>
```

Suggest what matters (with amp-autocomplete)

According to our analytics data, developers most often visit amp.dev to access AMP component documentation and samples. That's why we wanted to make these as easy to discover as possible. With `<amp-autocomplete>`, AMP offers an out-of-the-box solution for implementing auto-suggestions. The goal is to auto-suggest all available AMP components. The static datasource is provided by the `/search/autosuggest` endpoint and the autocomplete generates suggestions client-side using the `filter="substring"` designation.

```
<amp-autocomplete filter="substring"
                  min-characters="1"
                  on="select:AMP.setState({    query: event.value })"
                  submit-on-enter="false"
                  src="/search/autosuggest"
>
    <input placeholder="What are you looking for?">
</amp-autocomplete>
```

Executing the search (with amp-form)

Users can trigger the search by selecting one of the auto-suggested options or

submitting the form. To execute the actual search request we are using the state `query` as a URL parameter.

```
<amp-list [src]="'/search/do?q=' + encodeURIComponent(query) + '&locale=en'"
```

Based on the `on` action in the `amp-autocomplete`, the `query` is updated (and `amp-list` rerenders) both when the form submits **and** when autocomplete emits a `select` event.

```
<form action-xhr="/search/echo"
      on="submit:
          AMP.setState({ query: queryInput }),
          searchResult.focus,
          searchList.changeToLayoutContainer"
      method="POST" target="_top">
  <amp-autocomplete filter="substring"
                    min-characters="1"
                    on="select:AMP.setState({ query: event.value })"
                    submit-on-enter="false"
                    src="/search/autosuggest"
  >
    <input id="searchInput"
           placeholder="What are you looking for?"
           on="input-throttled:AMP.setState({ queryInput: event.value })"
    >
    <button disabled [disabled]="!queryInput">Search</button>
  </amp-autocomplete>
</form>
```

When the form is submitted we also focus the search result container to let the keyboard navigation start directly with the first entry. Additionally, we call `changeToLayoutContainer` of the `<amp-list>` to ensure the list's height will change according to the content and can get smaller. Because in our case the result of the form `submit` event is not needed, we simply point to an echo action. Sidenote: this won't be needed in the future as it's soon going to be possible to use àmp-autocomplete` without a form.

Caching previous search results via Service Worker

After we launched the first version of the search we received a related feature request fairly quickly: keep showing the results for the latest search query, even when the user navigates to another page.

To achieve this, we built upon AMP's one-line Service Worker **amp-sw** which offers basic PWA functionalities like caching and offline pages. We extended it to store the

latest search query and the corresponding search results.

When a search is started, we display the previous search query and its results. Otherwise, we will display a list of suggested articles. On page load, we initialize an `amp-state` object from a server endpoint `/search/latest-query` which populates the search input field and the search results:

```
// search.html
<amp-state id="query" src="/search/latest-query"></amp-state>

<amp-list src="/search/initial-items"
          [src]="query ? '/search/initial-items : '/search/do?q=' + encodeURI
...
</amp-list>
```

The trick is: this server-endpoint does not exist. The magic happens in the Service Worker which intercepts the route and creates a new response with the cached search query and search results from the user's last search request and sends it back to the page instead of loading the original response from the network.

To save the latest search query, we grab the `query` parameter from the requested URL with a regular expression and store it in a newly created response object in our cache.

Then the route handler checks if there is an entry in the cache that matches the search request. If there is the results are returned from the cache immediately. Otherwise, the request falls through to the server and then gets cached for the following calls.

```
// serviceworker.js
async function searchDoRequestHandler(url, request) {
  const searchQuery = decodeURIComponent(url.search.match(/q=([^&]+)/)[1]);
  const cache = await caches.open(SEARCH_CACHE_NAME);


  cache.put(SEARCH_LATEST_QUERY_PATH, new Response(`"${searchQuery}"`));

  let response = await cache.match(request);
  if (response) return response;

  response = await fetch(request);
  if (response.status == 200) {
    cache.delete(request, {
      ignoreSearch: true,
    });
    cache.put(request, response.clone());
  }

  return response;
```

```
}
```

This way when the user opens the search layer on another page, they automatically receive their previous search results back and can continue where they left off.

Here you can see how a handler function is registered:

```
// serviceworker.js
self.addEventListener('fetch', (event) => {
  const requestUrl = new URL(event.request.url);
  if (requestUrl.pathname === '/search/do') {
    event.respondWith(searchDoRequestHandler(requestUrl, event.request));
  }
});
```

Intercepting and dynamically changing requests with the help of the Service Worker API is a neat way whenever you want to personalize data used by AMP components that load data from remote endpoints like `<amp-state>` or `<amp-list>` et al. Just like it helped us to enhance the user experience for the search by caching the user's latest search query.

Conclusion

With the implementation of a search function within amp.dev, we accomplished our goal of allowing users to precisely navigate the content of the site in an intuitive and efficient manner. For even better user experience, we are also caching previous search results with Service Worker functionalities.

The cool thing about this is that we integrated the search without a single line of JavaScript (except the Service Worker part). Just by making use of AMP's existing components we could integrate useful features like auto-suggestion and infinite scrolling, which would be quite challenging to implement otherwise!

Written by Thorsten Harders and Sebil Satici, Developers at Jung von Matt

---
