

Clientseitige Suche: Flexsearch - full-text search library for browser and node.js

Thomas Klampfl

thomas.klampfl@oeaw.ac.at

28.11.2025

Repository: <https://github.com/nextapps-de/flexsearch>

Lizenz: Apache 2.0

Version: 0.8.2

Dokumentation: Startseite und <https://github.com/nextapps-de/flexsearch/tree/master/doc>

Beispiele: <https://github.com/nextapps-de/flexsearch/tree/master/example>

<https://www.nextapps.de/> - Software-Unternehmen in Deutschland, Berlin

Konfiguration:

- `js/flexsearch.bundle.module.min.js`
- `js/search.js`
- `json/interventions.json`
- `search.html`

Für lokale Tests, z.B.: `python -m http.server` im Verzeichnis mit der HTML-Datei, um einen lokalen Webserver zu starten.

search.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Search with Flexsearch</title>
    <script type="module" src="./js/search.js"></script>
  </head>
  <body>
    <h1>Search with Flexsearch</h1>
  </body>
</html>
```

search.js

```
import FlexSearch from "./flexsearch.bundle.module.min.js";
const index = new FlexSearch.Index();
index.add(0, "Wem ist die Königsherrschaft Gottes gleich, und wem soll ...");
index.add(1, "Sie ist einem Senfkorn gleich, das ein Mensch nahm ...");
```

```
const result = index.search("Senfkorn");  
console.log(result);
```

Ausgabe: [1]

In der *html* Datei (*search.html*) wird die *JavaScript* Datei *search.js* eingebunden. In dieser findet sich das *import* Statement für das Laden von *flexsearch*. Es wird ein neuer Index erzeugt, und mit zwei Textzeilen befüllt. Die Angabe einer eindeutigen *Id* ist obligatorisch. Mit der *search* Methode des Indexobjekts wird eine Suche durchgeführt, deren Ergebnis ausgegeben wird: Im zweiten Satz findet sich der Suchbegriff.

Flexsearch läuft im Browser oder in Node.js. Es werden verschiedene Datenbanken unterstützt: *IndexedDB* (in modernen Browsern implementiert), *Redis*, *SQLite*, *Postgres*, *MongoDB* und *Clickhouse*. Größere Arbeitsaufgaben können auf *worker* ausgelagert werden, um Einfügeoperationen in den Index, oder Abfragen parallel auszuführen.

Es werden primär sechs Klassen zur Verfügung gestellt: *Index*, *Document*, *Worker*, *Encoder*, *Resolver*, *IndexedDB*.

Index – Worker / WorkerIndex – Document

Index ist eine Datenstruktur zum Speichern von Paaren von *Ids* und Inhalten. Auf dem *Index*-Objekt werden Suchabfragen durchgeführt. *Worker* und *WorkerIndex* haben die gleiche Aufgabe wie ein *Index*, nur laufen sie asynchron im Hintergrund als eigener Thread. *Document* ist ein *Index* mit mehreren Feldern, der komplexe JSON-Dokumente speichern kann.

Die Methoden des *Index*-Objekts sind unter anderem:

- `index.add(id, string)` – Fügt Inhalt in den Index ein.
- `index.update(id, string)` – Ändert den Inhalt einer bestimmten Position im Index.
- `index.remove(id)` – Entfernt einen Eintrag aus dem Index.
- `index.contain(id)` – Überprüft ob ein spezifischer Inhalt im Index enthalten ist.
- `index.clear()` – Löscht alle Elemente aus dem Index.
- `index.search(string, limit, options)` – Führt eine Suche auf dem Index aus.

Die Methoden des *Document*-Objekts sind analog:

- `document.add(id, document)` – Fügt ein Dokument in den Dokument-Index ein.
- `document.update(id, document)` – Ändert den Inhalt an einer bestimmten Position im Index.
- `document.remove(id)` – Entfernt einen Eintrag aus dem Dokument-Index.
- `document.contain(id)` – Überprüft, ob ein bestimmter Inhalt im Index enthalten ist
- `document.clear()` – Löscht alle Elemente aus dem Dokument-Index.
- `document.search(string, limit, options)` – Führt eine Suche durch.

Beispiel:

Die Daten für das folgende Beispiel stammen aus dem Projekt „Glossen zu Priscian“ (<https://priscian-glosses.acdh-dev.oeaw.ac.at/>). In mittelalterlichen Handschriften finden sich Anmerkungen zum Text der lateinischen Grammatik des Priscian, die im gegebenen Projekt transkribiert und analysiert werden. Die JSON-Daten, die aus der Edition mit XSLT transformiert werden, sehen folgendermaßen aus:

```

id
id_of_intervention
type_of_intervention
lemma
    witness
    text_of_lemma
reading
    witness
    hand
    place
    analysis
        analysis_category_id
        analysis_category
    reading_normalized
    reading_critical
link

```

Der Eintrag `id` ist eine eindeutige Nummer innerhalb der Interventionen. In der Edition trägt jede Intervention eine Id; dies ist das Feld `id_of_intervention`. `type_of_intervention` gibt an, welche Art von Änderung vorliegt: *gloss* | *emendation* | *rubrication* | *text variation* | *reference sign*. Es folgt das Lemma, zu dem es eine Intervention gibt (`lemma`). Sowie der Text des Eingriffes (`reading`). Am Ende steht der Link in die Edition (`link`).

Die fünfte Intervention sieht beispielsweise so aus:

```

{
  "id" : 5,
  "id_of_intervention" : "gloss-1-3-2",
  "type_of_intervention" : "gloss",
  "lemma" : {
    "witness" : "Edition of Hertz",
    "text_of_lemma" : "celebrasse" },
  "reading" : {
    "witness" : "Codex 50 Weiss.",
    "hand" : "Otfrid",
    "place" : "interlinear above",
    "analysis" : [
      {
        "analysis_category_id" : "212",
        "analysis_category" : "Lexical gloss
        providing a synonym"}],
    "reading_normalized" : "diffamasse",
    "reading_critical" : "<diffamasse">"},
  "link" : "edition.html#gloss-1-3-2"
}

```

search.js

```

import FlexSearch from "../flexsearch.bundle.module.min.js";
document.addEventListener("DOMContentLoaded", (event) => {
  getJSONData()
    .then((interventions) => {
      const documentOfInterventions = new FlexSearch.Document({
        document: {
          id: "id",
          index: ["type_of_intervention", "id_of_intervention",
            "lemma:text_of_lemma", "reading:reading_normalized"]

```

```

    });
interventions.interventions.forEach((intervention) =>
    documentOfInterventions.add(intervention));
let results =
    documentOfInterventions.search("diffamasse",{field:
        ["id_of_intervention","lemma:text_of_lemma",
        "reading:reading_normalized"]});
    console.log(results);
});

async function getJSONData() {
    const url = "../json/interventions.json";
    try {
        const response = await fetch(url);
        if (!response.ok) {
            throw new Error(`Response status: ${response.status}`);
        }

        const result = await response.json();
        return(result);
    } catch (error) {
        console.error(error.message);
    }
}
});

```

Am Ende der Datei findet sich eine Funktion, die die JSON-Daten vom Server lädt. Das zurückgegebene Objekt ist ein *Promise*. Zunächst wird ein neuer Dokument-Index erzeugt, und angegeben, welche JSON-Datenfelder (`type_of_intervention`, `id_of_intervention`, `lemma:text_of_lemma`, `reading:reading_normalized`) indiziert werden sollen. Zu beachten ist, dass JSON-Felder mit Doppelpunkt adressiert werden. Danach wird der Index befüllt (`forEach`), und eine Abfrage durchgeführt, in der angegeben wird, welche Felder durchsucht werden sollen (`id_of_intervention`, `lemma:text_of_lemma`, `reading:reading_normalized`). Das Ergebnis sieht so aus:

```
[ { "field": "reading:reading_normalized", "result": [ 5 ] } ]
```

Es wird ein Array von Treffern zurückgegeben, wobei jeder Treffer ein Objekt darstellt, das das Feld angibt, in dem der Treffer aufgetreten ist, und die Id des Indexeintrags.

Da die Daten im Client zur Verfügung stehen, können diese direkt für die Darstellung der Treffer verwendet werden, z.B. mit `replaceAll(pattern, replacement)` für die Hervorhebung. Es ist aber auch möglich, das Suchergebnis um Daten anzureichern: Hierfür muss in der Konfiguration des Indexes angegeben werden, welche Felder gespeichert werden sollen (`store`), sowie in der Suche ein Parameter (`enrich`) gesetzt sein.

```

const documentOfInterventions = new FlexSearch.Document({
    document: {
        id: "id",
        index: ["type_of_intervention","id_of_intervention",
            "lemma:text_of_lemma","reading:reading_normalized"],
        store: ["lemma:text_of_lemma","reading:reading_normalized"]
    });

let results = documentOfInterventions.search("diffamasse", { field:
    ["id_of_intervention","lemma:text_of_lemma",

```

```
"reading:reading_normalized"], enrich: true});
```

Ergebnis:

```
[
  {
    "field": "reading:reading_normalized",
    "result": [
      {
        "id": 5,
        "doc": {
          "lemma": {
            "text_of_lemma": "celebrasse"
          },
          "reading": {
            "reading_normalized": "diffamasse"
          }
        }
      }
    ]
  }
]
```

Folgende Suche mit zwei Suchbegriffen liefert alle Texte, in denen die beiden Wörter gemeinsam vorkommen, wobei sie nicht unmittelbar aufeinander folgen müssen, also eine gewisse Distanz zwischen den Treffern liegen kann.

```
let results = documentOfInterventions.search({
  query: "inarticulata et",
  field: "reading:reading_normalized",
  enrich: true });
```

Treffer:

```
{ "reading_normalized": "inarticulata |ABBREVIATION ET|
literata" }

{ "reading_normalized": "|REFERENCE SIGN CURSIVE ZETA|
Articulata· |ABBREVIATION ET| literata· Vt
arma· Articulata· |ABBREVIATION ET| inliterata· Vt sibili
hominum· Inarticulata· |ABBREVIATION ET| literata· Vt
coax· Inarticulata· et inliterata· Vt crepitus; "
```

Beispiel mit Tags:

Es können sogenannte Tags gesetzt werden, um die Suche auf bestimmte Rubriken einzuschränken. Im folgenden Beispiel wird der Typ der Intervention als Tag verwendet, um es zu ermöglichen, z.B. nur nach Glossen zu suchen:

Konfiguration des Dokument-Indexes:

```
const documentOfInterventions = new FlexSearch.Document({
  document: {
    id: "id",
    tag: "type_of_intervention",
    index: ["type_of_intervention", "id_of_intervention",
      "lemma:text_of_lemma", "reading:reading_normalized"],
    store: ["type_of_intervention", "lemma:text_of_lemma",
      "reading:reading_normalized"]
  }
});
```

Suchabfrage:

```
let results = documentOfInterventions.search("et",
  { tag: { type_of_intervention: "gloss" } },
  { field: ["id_of_intervention", "lemma:text_of_lemma"],
    enrich: true });
```

Auszug aus den Suchergebnissen:

```
{
  "id": 6,
  "doc": {
    "type_of_intervention": "gloss",
    "lemma": {
      "text_of_lemma": "et"
    },
    "reading": {
      "reading_normalized": "|CONSTRUE MARK TWO  
VERTICAL DOTS, DICOLON|" } } }
```

Filtern von zu indizierenden Daten:

Es kann zu indizierenden Feldern eine Funktion zum Filtern mitgegeben werden. Wenn diese auf *true* auswertet, wird das Dokument in den Index aufgenommen, bei *false* nicht.

Im folgenden Beispiel werden alle Interventionstypen ausgefiltert, die keine Glossen (gloss) sind.

```
const documentOfInterventions = new FlexSearch.Document({
  document: {
    id: "id",
    index: [ {
      field: "type_of_intervention",
      filter: function(data) {
        if (data.type_of_intervention === 'gloss') { return true; }
        else { return false; };
      }
    }, "id_of_intervention", "lemma:text_of_lemma",
      "reading:reading_normalized"],
    store: ["type_of_intervention", "lemma:text_of_lemma",
      "reading:reading_normalized"]
  }
});
```

In dem Funktionsparameter *data* findet sich das ganze zu indizierende Dokument, weshalb auf das einzelne Datenfeld zugegriffen werden muss.

Einfügen von neu erzeugten Datenfeldern in den Index:

Bei einzelnen Feldern kann eine Funktion angegeben werden, die aus den indizierten oder externen Daten Werte für ein Feld generiert.

```
const documentOfInterventions = new FlexSearch.Document({
  document: {
    id: "id",
    index: [{ field: "type_of_intervention" } ],
```

```

    { field: "type_of_intervention_and_id",
      custom: function(data){
        return data.type_of_intervention + ' - ' +
              data.id_of_intervention;
      }}, {
    field: "id_of_intervention" }, {
    field: "lemma:text_of_lemma" }, {
    field: "reading:reading_normalized" }],
  store: [{ field: "type_of_intervention_and_id",
    custom: function(data){
      return data.type_of_intervention + ' - ' +
            data.id_of_intervention;
    }}, "lemma:text_of_lemma", "reading:reading_normalized"]
  });

```

Optionen des Indizierungsprozesses:

Option preset:

- `memory` – Optimiert für geringen Speicherbedarf.
- `performance` – Optimiert für hohe Performance.
- `match` – Optimiert für Trefferfähigkeiten
- `score` – Optimiert für Scoringfähigkeiten, also die Reihenfolge der Ergebnisse.
- `default` – Ein ausbalanziertes Profil.

Option tokenize:

- `strict / exact / default` – Indiziert den ganzen Term, also die Wörter als ganze.
- `tolerant` – Indiziert den ganzen Term, ist aber tolerant gegen Tippfehler, wie vertauschte Buchstaben oder fehlende Buchstaben.
- `forward` – Indiziert in Vorwärtsrichtung und findet so Teile eines Wortes, mit denen Wörter beginnen.
- `reverse / bidirectional` – Indiziert in Vorwärts- und Rückwärtsrichtung.
- `full` – Indiziert jedes Teil eines Wortes, also aufeinanderfolgende Buchstaben.

Option encoder:

- `new Encoder(options)` – Verwendung eines eigenen Encoders.
- `Charset.Exact` – Der Encodingschritt wird übergangen, und der exakte Input genommen.
- `Charset.Default / Charset.Normalize` – Encoding ist case-insensitive, Normalisierung der Buchstaben, z.B. "é" to "e", Dedublizierung von Buchstaben, z.B. "missing" zu "mising".
- `Charset.LatinBalance` – Encoding ist case-insensitive, Normalisierung der Buchstaben, Dedublizierung, grundlegende phonetische Transformation.
- `Charset.LatinAdvanced` - Encoding ist case-insensitive, Normalisierung der Buchstaben, Dedublizierung, fortgeschrittene phonetische Transformation.
- `Charset.LatinExtra` - Encoding ist case-insensitive, Normalisierung der Buchstaben, Dedublizierung, Soundex Transformation.
- `Charset.LatinSoundex` – Volle Soundex Transformation.

Die default Encoderkonfiguration ist folgende:

```
const encoder = new Encoder({
  normalize: true,
  dedupe: true,
  cache: true,
  include: {
    letter: true,
    number: true,
    symbol: false,
    punctuation: false,
    control: false,
    char: "" } });
```

D.h. dass Symbole nicht mitaufgenommen werden, und auch keine Piktuationszeichen sowie Control-Zeichen. Mit folgender Zeile kann man spezifische Sonderzeichen aufnehmen: `char: ["#", "@", "-"]`.

Es können für jeden Schritt des Encoding-Prozesses eigene Funktionen angegeben werden. Im folgenden Beispiel wird zu Kleinbuchstaben normalisiert; ein Ampersand durch „und“ ersetzt, und alle Wörter, die weniger als zwei Zeichen haben, aussortiert. Für weiterführende Details siehe: <https://github.com/nextapps-de/flexsearch/blob/master/doc/encoder.md>

```
const encoder = new Encoder({
  normalize: function(str){
    return str.toLowerCase();
  },
  prepare: function(str){
    return str.replace(/&/g, " and ");
  },
  finalize: function(arr){
    return arr.filter(term => term.length > 2);
  }
});
```

Setzen eines Encoders für einen Index:

```
const index = new Index({
  encoder: encoder
});
```

Folgende Character Sets stehen zur Verfügung: Latin, Chinese, Korean, Japanese (CJK), Hindi, Arabic, Cyrillic, Greek and Coptic, Hebrew.

Resolver – Komplexe Suchabfragen:

Es sind Verknüpfungen mittels `and()`, `or()`, `xor()`, `not()` möglich. Es gibt wiederum eine Reihe von Optionen. Siehe: <https://github.com/nextapps-de/flexsearch/blob/master/doc/resolver.md>

```
const result = new FlexSearch.Resolver({
  index: documentOfInterventions,
  query: "et",
  pluck: "lemma:text_of_lemma"
}).and({
```

```
    index: documentOfInterventions,  
    query: "gloss-1-3-5-1",  
    pluck: "id_of_intervention"  
}).resolve({ enrich: true });  
console.log(result);
```

Diese Abfrage liefert die Glosse mit der Id `gloss-1-3-5-1`, in deren Lemma das Wort „et“ vorkommt.