

Dynamic Glossary

```
<script type="module">
  // The URL path to the default glossary page in your instance.
  // You should only need to change the "my-book" and "main-glossary"
  // parts, keep the other parts and the general format the same.
  const defaultGlossaryPage = '/books/glossar/page/glossar';

  // Get a normalised URL path, check if it's a glossary page, and page the page content
  const urlPath = window.location.pathname.replace(/^.*?\books\/, '/books/');
  const isGlossaryPage = urlPath.endsWith('/page/glossar') ||
urlPath.endsWith(defaultGlossaryPage);
  const pageContentEl = document.querySelector('.page-content');

  if (isGlossaryPage && pageContentEl) {
    // Force re-index when viewing glossary pages
    addTermMapToStorage(urlPath, domToTermMap(pageContentEl));
  } else if (pageContentEl) {
    // Get glossaries and highlight when viewing non-glossary pages
    document.addEventListener('DOMContentLoaded', () => highlightTermsOnPage());
  }

  /**
   * Highlight glossary terms on the current page that's being viewed.
   * In this, we get our combined glossary, then walk each text node to then check
   * each word of the text against the glossary. Where exists, we split the text
   * and insert a new glossary term span element in its place.
   */
  async function highlightTermsOnPage() {
    const glossary = await getMergedGlossariesForPage(urlPath);
    const trimRegex = /^[.?:"',;]|[\.?:"',;]$/g;
    const treeWalker = document.createTreeWalker(pageContentEl, NodeFilter.SHOW_TEXT);
    while (treeWalker.nextNode()) {
      const node = treeWalker.currentNode;
      const words = node.textContent.split(' ');
      const parent = node.parentNode;
      let parsedWords = [];
```

```

    let firstChange = true;
    for (const word of words) {
        const normalisedWord = word.toLowerCase().replace(trimRegex, '');
        const glossaryVal = glossary[normalisedWord];
        if (glossaryVal) {
            const preText = parsedWords.join(' ');
            const preTextNode = new Text((firstChange ? '' : ' ') + preText + ' ');
            parent.insertBefore(preTextNode, node)
            const termEl = createGlossaryNode(word, glossaryVal);
            parent.insertBefore(termEl, node);
            const toReplace = parsedWords.length ? preText + ' ' + word : word;
            node.textContent = node.textContent.replace(toReplace, '');
            parsedWords = [];
            firstChange = false;
            continue;
        }

        parsedWords.push(word);
    }
}

/**
 * Create the element for a glossary term.
 * @param {string} term
 * @param {string} description
 * @returns {Element}
 */
function createGlossaryNode(term, description) {
    const termEl = document.createElement('span');
    termEl.setAttribute('data-term', description.trim());
    termEl.setAttribute('class', 'glossary-term');
    termEl.textContent = term;
    return termEl;
}

/**
 * Get a merged glossary object for a given page.
 * Combines the terms for a same-book & global glossary.
 * @param {string} pagePath

```

```

    * @returns {Promise<Object<string, string>>}
    */
    async function getMergedGlossariesForPage(pagePath) {
        const [defaultGlossary, bookGlossary] = await Promise.all([
            getGlossaryFromPath(defaultGlossaryPage),
            getBookGlossary(pagePath),
        ]);

        return Object.assign({}, defaultGlossary, bookGlossary);
    }

    /**
     * Get the glossary for the book of page found at the given path.
     * @param {string} pagePath
     * @returns {Promise<Object<string, string>>}
     */
    async function getBookGlossary(pagePath) {
        const bookPath = pagePath.split('/page/')[0];
        const glossaryPath = bookPath + '/page/glossary';
        return await getGlossaryFromPath(glossaryPath);
    }

    /**
     * Get/build a glossary from the given page path.
     * Will fetch it from the localStorage cache first if existing.
     * Otherwise, will attempt the load it by fetching the page.
     * @param path
     * @returns {Promise<{}|any>}
     */
    async function getGlossaryFromPath(path) {
        const key = 'bsglossary:' + path;
        const storageVal = window.localStorage.getItem(key);
        if (storageVal) {
            return JSON.parse(storageVal);
        }

        let resp = null;
        try {
            resp = await window.$http.get(path);
        } catch (err) {

```

```

    }

    let map = {};
    if (resp && resp.status === 200 && typeof resp.data === 'string') {
        const doc = (new DOMParser).parseFromString(resp.data, 'text/html');
        const contentEl = doc.querySelector('.page-content');
        if (contentEl) {
            map = domToTermMap(contentEl);
        }
    }

    addTermMapToStorage(path, map);
    return map;
}

/**
 * Store a term map in storage for the given path.
 * @param {string} urlPath
 * @param {Object<string, string>} map
 */
function addTermMapToStorage(urlPath, map) {
    window.localStorage.setItem('bsglossary:' + urlPath, JSON.stringify(map));
}

/**
 * Convert the text of the given DOM into a map of definitions by term.
 * @param {string} text
 * @return {Object<string, string>}
 */
function domToTermMap(dom) {
    const textEls = Array.from(dom.querySelectorAll('p,h1,h2,h3,h4,h5,h6,blockquote'));
    const text = textEls.map(el => el.textContent).join('\n');
    const map = {};
    const lines = text.split('\n');
    for (const line of lines) {
        const split = line.trim().split(':');
        if (split.length > 1) {
            map[split[0].trim().toLowerCase()] = split.slice(1).join(':');
        }
    }
}

```

```
        return map;
    }
</script>
```

```
<style>
  /**
   * These are the styles for the glossary terms and definition popups.
   * To keep things simple, the popups are not elements themselves, but
   * pseudo ":after" elements on the terms, which gain their text via
   * the "data-term" attribute on the term element.
   */
  .page-content .glossary-term {
    text-decoration: underline;
    text-decoration-style: dashed;
    text-decoration-color: var(--color-link);
    text-decoration-thickness: 1px;
    position: relative;
    cursor: help;
  }
  .page-content .glossary-term:hover:after {
    display: block;
  }

  .page-content .glossary-term:after {
    position: absolute;
    content: attr(data-term);
    background-color: #FFF;
    width: 200px;
    box-shadow: 0 1px 6px 0 rgba(0, 0, 0, 0.15);
    padding: 0.5rem 1rem;
    font-size: 12px;
    border-radius: 3px;
    z-index: 9999;
    top: 2em;
    inset-inline-start: 0;
    display: none;
  }
  .dark-mode .page-content .glossary-term:after {
    background-color: #000;
  }
}
```

```
.page-content table td, .page-content table th {  
    position: relative;  
    overflow: visible;  
}  
</style>
```

Created 2025-10-06 15:52:23 CEST by Sascha Jelinek
Updated 2025-10-06 15:53:01 CEST by Sascha Jelinek