ecma INTERNATIONAL

Standard ECMA-426

1st Edition, December 2024

Source map format specification

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001

is the registered trademark of Ecma International

Contents

Introduction

This Ecma Standard defines the Source map format, used for mapping transpiled source code back to the original sources.

The source map format has the following goals:

- Support source-level debugging allowing bidirectional mapping
- Support server-side stack trace deobfuscation

The canonical URL for the latest published source map standard is located at https://426.ecma-international.org/. The document at https://tc39.es/ecma426/ is the most accurate and up-to-date draft source map specification. It contains the content of the most recently published snapshot plus any modifications that will be included in the next snapshot.

If you want to get involved you will find more information at the specification repository.

The original source map format (v1) was created by Joseph Schorr for use by Closure Inspector to enable source-level debugging of optimized JavaScript code (although the format itself is language agnostic). However, as the size of the projects using source maps expanded, the verbosity of the format started to become a problem. The v2 format was created by trading some simplicity and flexibility to reduce the overall size of the source map. Even with the changes made with the v2 version of the format, the source map file size was limiting its usefulness. The v3 format is based on suggestions made by Pavel Podivilov (Google).

The source map format does not have version numbers anymore, and it is instead hard-coded to always be "3".

In 2023-2024, the source map format was developed into a more precise Ecma standard, with significant contributions from many people. Further iteration on the source map format is expected to come from TC39-TG4.

Asumu Takikawa, Nicolò Ribaudo, Jon Kuperman
ECMA-426, 1st edition, Project Editors

This Ecma Standard was developed by Technical Committee 39 and was adopted by the General Assembly of December 2024.

Source map format specification

1 Scope

This Standard defines the source map format, used by different types of developer tools to improve the debugging experience of code compiled to JavaScript, WebAssembly, and CSS.

2 Conformance

A conforming source map document is a JSON document that conforms to the structure detailed in this specification.

A conforming source map generator should generate documents which are conforming source map documents, and can be decoded by the algorithms in this specification without reporting any errors (even those which are specified as optional).

A conforming source map consumer should implement the algorithms specified in this specification for retrieving (where applicable) and decoding source map documents. A conforming consumer is permitted to ignore errors or report them without terminating where the specification indicates that an algorithm may optionally report an error.

3 References

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

3.1 Normative references

ECMA-262, *ECMAScript® language specification*, https://262.ecma-international.org/

ECMA-404, *The JSON data interchange syntax*, https://ecma-international.org/publications-and-standards/standards/ecma-404/.

3.2 Informative references

IETF RFC 4648, *The Base16, Base32, and Base64 Data Encodings*, https://www.ietf.org/rfc/rfc4648.txt

WebAssembly Names binary format, https://www.w3.org/TR/wasm-core-2/#names%E2%91%A2. Living Standard.

4 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

4.1
generated code
code generated by the compiler or transpiler

4.2
original source
code which has not been passed through the compiler

4.3
Base64 VLQ
a base64 value, where the most significant bit (the 6th bit) is used as the continuation bit and the "digits" are encoded into the string least significant first, and where the least significant bit of the first digit is used as the sign bit

NOTE The values that can be represented by the VLQ Base64 encoded are limited to 32-bit quantities until some use case for larger values is presented. This means that values exceeding 32-bits are invalid and implementations may reject them. The sign bit is counted towards the limit, but the continuation bits are not.

Example 1 The string `"iB"` represents a Base64 VLQ with two digits. The first digit `"i"` encodes the bit pattern **`0x100010`**, which has a continuation bit of **`1`** (the VLQ continues), a sign bit of **`0`** (non-negative), and the value bits **`0x0001`**. The second digit **`B`** encodes the bit pattern **`0x000001`**, which has a continuation bit of **`0`**, no sign bit, and value bits **`0x00001`**. The decoding of this VLQ string is the number 17.

Example 2 The string `"V"` represents a Base64 VLQ with one digit. The digit `"V"` encodes the bit pattern **`0x010101`**, which has a continuation bit of **`0`** (no continuation), a sign bit of **`1`** (negative), and the value bits **`0x1010`**. The decoding of this VLQ string is the number -10.

4.4
source mapping URL
URL referencing the location of a source map from the generated code

4.5
column
zero-based indexed offset within a line of the generated code, computed as UTF-16 code units for JavaScript and CSS source maps, and as byte indexes in the binary content (represented as a single line) for WebAssembly source maps.

NOTE NOTE: That means that "A" (LATIN CAPITAL LETTER A) measures as 1 code unit, and "🔥" (FIRE) measures as 2 code units. Source maps for other content types may diverge from this.

5 Source map format

A source map is a JSON document containing a top-level JSON object with the following structure:

```
{
  "version" : 3,
  "file": "out.js",
  "sourceRoot": "",
  "sources": ["foo.js", "bar.js"],
  "sourcesContent": [null, null],
  "names": ["src", "maps", "are", "fun"],
  "mappings": "A,AAAB;;ABCDE"
  "ignoreList": [0]
}
```

- ***`version`*** is the version field which shall always be the number **`3`** as an integer. The source map may be rejected if the field has any other value.
- ***`file`*** is an optional name of the generated code that this source map is associated with. It's not specified if this can be a URL, relative path name, or just a base name. Source map generators may choose the appropriate

interpretation for their contexts of use.

- `sourceRoot` is an optional source root string, used for relocating source files on a server or removing repeated values in the sources entry. This value is prepended to the individual entries in the sources field.
- `sources` is a list of original sources used by the mappings entry. Each entry is either a string that is a (potentially relative) URL or `null` if the source name is not known.
- `sourcesContent` is an optional list of source content (i.e., the Original Source) strings, used when the source cannot be hosted. The contents are listed in the same order as the sources. Entries may be `null` if some original sources should be retrieved by name.
- `names` is an optional list of symbol names which may be used by the mappings entry.
- `mappings` is a string with the encoded mapping data (see § 5.1 Mappings structure).
- `ignoreList` is an optional list of indices of files that should be considered third party code, such as framework code or bundler-generated code. This allows developer tools to avoid code that developers likely don't want to see or step through, without requiring developers to configure this beforehand. It refers to the sources array and lists the indices of all the known third-party sources in the source map. Some browsers may also use the deprecated `x_google_ignoreList` field if `ignoreList` is not present.

A ***decoded source map*** is a struct with the following fields:

file
A string or null.

sources
A list of decoded sources.

mappings
A list of decoded mappings.

A ***decoded source*** is a struct with the following fields:

URL
A URL or null.

content
A string or null.

ignored
A boolean.

To ***decode a source map from a JSON string*** *str* given a URL *baseURL*, run the following steps:

1. Let *jsonMap* be the result of parsing a JSON string to an Infra value *str*.
2. If *jsonMap* is not a map, report an error and abort these steps.
3. Decode a source map given *jsonMap* and *baseURL*, and return its result if any.

To ***decode a source map*** given a string-keyed map *jsonMap* and a URL *baseURL*, run the following steps:

1. If *jsonMap*[`"version"`] does not exist or *jsonMap*[`"version"`] is not 3, optionally report an error.
2. If *jsonMap*[`"mappings"`] does not exist or *jsonMap*[`"mappings"`], is not a string, throw an error.
3. If *jsonMap*[`"sources"`] does not exist or *jsonMap*[`"sources"`], is not a list, throw an error.
4. Let *sourceMap* be a new decoded source map.
5. Set *sourceMap*'s file to optionally get a string `"file"` from *jsonMap*.
6. Set *sourceMap*'s sources to the result of decoding source map sources given *baseURL* with:
   - sourceRoot set to optionally get a string `"sourceRoot"` from *jsonMap*;
   - sources set to optionally get a list of optional strings `"sources"` from *jsonMap*;
   - sourcesContent set to optionally get a list of optional strings `"sourcesContent"` from *jsonMap*;
   - ignoredSources set to optionally get a list of array indexes `"ignoreList"` from *jsonMap*.
7. Set *sourceMap*'s mappings to the result of decoding source map mappings with:

   - mappings set to *jsonMap*[`"mappings"`];
   - names set to optionally get a list of strings `"names"` from *jsonMap*;
   - sources set to *sourceMap*'s sources.
8. Return *sourceMap*.

To ***optionally get a string*** *key* from a string-keyed map *jsonMap*, run the following steps:

1. If *jsonMap*[*key*] does not exist, return null.
2. If *jsonMap*[*key*] is not a string, optionally report an error and return null.
3. Return *jsonMap*[*key*].

To ***optionally get a list of strings*** *key* from a string-keyed map *jsonMap*, run the following steps:

1. If *jsonMap*[*key*] does not exist, return a new empty list.
2. If *jsonMap*[*key*] is not a list, optionally report an error and return a new empty list.
3. Let *list* be a new empty list.
4. For each *jsonItem* of *jsonMap*[*key*]:
   a. If *jsonItem* is a string, append it to *list*.
   b. Else, optionally report an error and append `""` to *list*.
5. Return *list*.

To ***optionally get a list of optional strings*** *key* from a string-keyed map *jsonMap*, run the following steps:

1. If *jsonMap*[*key*] does not exist, return a new empty list.
2. If *jsonMap*[*key*] is not a list, optionally report an error and return a new empty list.
3. Let *list* be a new empty list.
4. For each *jsonItem* of *jsonMap*[*key*]:
   a. If *jsonItem* is a string, append it to *list*.
   b. Else,
      i. If *jsonItem* is not null, optionally report an error.
      ii. Append null to *list*.
5. Return *list*.

To ***optionally get a list of array indexes*** *key* from a string-keyed map *jsonMap*, run the following steps:

1. If *jsonMap*[*key*] does not exist, return a new empty list.
2. If *jsonMap*[*key*] is not a list, optionally report an error and return a new empty list.
3. Let *list* be a new empty list.
4. For each *jsonItem* of *jsonMap*[*key*]:
   a. If *jsonItem* is a non-negative integer number, append it to *list*.
   b. Else,
      i. If *jsonItem* is not null, optionally report an error.
      ii. Append null to *list*.
5. Return *list*.

To ***optionally report an error***, implementations can choose to:

- Do nothing.
- Report an error to the user, and continue processing.
- Throw an error to abort the running algorithm. (*Infra* §3.6 Control flow)

5.1 Mappings structure

The mappings data is broken down as follows:

- each group representing a line in the generated file is separated by a semicolon (`;`)
- each segment is separated by a comma (`,`)
- each segment is made up of 1, 4, or 5 variable length fields.

The fields in each segment are:

1. The zero-based starting column of the line in the generated code that the segment represents. If this is the first field of the first segment, or the first segment following a new generated line (`;`), then this field holds the whole Base64 VLQ. Otherwise, this field contains a Base64 VLQ that is relative to the previous occurrence of this field. *Note that this is different from the subsequent fields below because the previous value is reset after every generated line.*
2. If present, the zero-based index into the sources list. This field contains a Base64 VLQ relative to the previous occurrence of this field, unless it is the first occurrence of this field, in which case the whole value is represented.
3. If present, the zero-based starting line in the original source. This field contains a Base64 VLQ relative to the previous occurrence of this field, unless it is the first occurrence of this field, in which case the whole value is represented. Shall be present if there is a source field.
4. If present, the zero-based starting column of the line in the original source. This field contains a Base64 VLQ relative to the previous occurrence of this field, unless it is the first occurrence of this field, in which case the whole value is represented. Shall be present if there is a source field.
5. If present, the zero-based index into the names list associated with this segment. This field contains a Base64 VLQ relative to the previous occurrence of this field, unless it is the first occurrence of this field, in which case the whole value is represented.

NOTE The purpose of this encoding is to reduce the source map size. VLQ encoding reduced source maps by 50% relative to the v2 format in tests performed using Google Calendar.

NOTE Segments with one field are intended to represent generated code that is unmapped because there is no corresponding original source code, such as code that is generated by a compiler. Segments with four fields represent mapped code where a corresponding name does not exist. Segments with five fields represent mapped code that also has a mapped name.

NOTE Using file offsets was considered but rejected in favor of using line/column data to avoid becoming misaligned with the original due to platform-specific line endings.

A ***decoded mapping*** is a struct with the following fields:

generatedLine
A non-negative integer.

generatedColumn
A non-negative integer.

originalSource
A decoded source or null.

originalLine
A non-negative integer or null.

originalColumn
A non-negative integer or null.

name
A string or null.

To ***decode source map mappings*** given a string ***mappings***, a list of strings ***names***, and a list of decoded sources ***sources***, run the following steps:

1. Validate base64 VLQ groupings with *mappings*.
2. Let *decodedMappings* be a new empty list.
3. Let *groups* be the result of strictly splitting *mappings* on `;`.

4. Let *generatedLine* be `0`.
5. Let *originalLine* be `0`.
6. Let *originalColumn* be `0`.
7. Let *sourceIndex* be `0`.
8. Let *nameIndex* be `0`.
9. While *generatedLine* is less than *groups*'s size:
    a. If *groups*[*generatedLine*] is not the empty string, then:
        i. Let *segments* be the result of strictly splitting *groups*[*generatedLine*] on `,`.
        ii. Let *generatedColumn* be `0`.
        iii. For each *segment* in *segments*:
            1. Let *position* be a position variable for *segment*, initially pointing at *segment*'s start.
            2. Decode a base64 VLQ from *segment* given *position* and let *relativeGeneratedColumn* be the result.
            3. If *relativeGeneratedColumn* is null,optionally report an error and continue with the next iteration.
            4. Increase *generatedColumn* by *relativeGeneratedColumn*. If the result is negative, optionally report an error and continue with the next iteration.
            5. Let *decodedMapping* be a new decoded mapping whose generatedLine is *generatedLine*, generatedColumn is *generatedColumn*, originalSource is null, originalLine is null, originalColumn is null, and name is null.
            6. Append *decodedMapping* to *decodedMappings*.
            7. Decode a base64 VLQ from *segment* given *position* and let *relativeSourceIndex* be the result.
            8. Decode a base64 VLQ from *segment* given *position* and let *relativeOriginalLine* be the result.
            9. Decode a base64 VLQ from *segment* given *position* and let *relativeOriginalColumn* be the result.
            10. If *relativeOriginalColumn* is null, then:
                a. If *relativeSourceIndex* is not null, optionally report an error.
                b. Continue with the next iteration.
            11. Increase *sourceIndex* by *relativeSourceIndex*.
            12. Increase *originalLine* by *relativeOriginalLine*.
            13. Increase *originalColumn* by *relativeOriginalColumn*.
            14. If any of *sourceIndex*, *originalLine*, or *originalColumn* are less than `0`, or if *sourceIndex* is greater than or equal to *sources*'s size, optionally report an error.
            15. Else,
                a. Set *decodedMapping*'s originalSource to *sources*[*sourceIndex*].
                b. Set *decodedMapping*'s originalLine to *originalLine*.
                c. Set *decodedMapping*'s originalColumn to *originalColumn*.
            16. Decode a base64 VLQ from *segment* given *position* and let *relativeNameIndex* be the result.
            17. If *relativeNameIndex* is not null, then:
                a. Increase *nameIndex* by *relativeNameIndex*.
                b. If *nameIndex* is negative or greater than *names*'s size, optionally report an error.
                c. Else, set *decodedMapping*'s name to *names*[*nameIndex*].
            18. If *position* does not point to the end of *segment*, optionally report an error.
    b. Increase *generatedLine* by 1.
10. Return *decodedMappings*.

To ***validate base64 VLQ groupings*** from a string *groupings*, run the following steps:

1. If *groupings* is not an ASCII string, throw an error.
2. If *groupings* contains any code unit other than:
    - U+002C (,) or U+003B (;);
    - U+0030 (0) to U+0039 (9);
    - U+0041 (A) to U+005A (Z);
    - U+0061 (a) to U+007A (z);
    - U+002B (+), U+002F (/)

    NOTE These are the valid base64 characters (excluding the padding character `=`), together with `,` and `;`.

3. then throw an error.

To ***decode a base64 VLQ*** from a string *segment* given a position variable *position*, run the following steps:

1. If *position* points to the end of *segment*, return null.
2. Let *first* be a byte whose the value is the number corresponding to *segment*'s *position*th code unit, according to the base64 encoding.

   NOTE The two most significant bits of *first* are `0`.

3. Let *sign* be 1 if *first* & `0x01` is `0x00`, and -1 otherwise.
4. Let *value* be (*first* >> 1) & `0x0F`, as a number.
5. Let *nextShift* be 16.
6. Let *currentByte* be *first*.
7. While *currentByte* & `0x20` is `0x20`:
   a. Advance *position* by 1.
   b. If *position* points to the end of *segment*, throw an error.
   c. Set *currentByte* to the byte whose the value is the number corresponding to *segment*'s *position*th code unit, according to the base64 encoding.
   d. Let *chunk* be *currentByte* & `0x1F`, as a number.
   e. Add *chunk* * *nextShift* to *value*.
   f. If *value* is greater than or equal to 2^{31}, throw an error.
   g. Multiply *nextShift* by 32.
8. Advance *position* by 1.
9. If *value* is `0` and *sign* is -1, return -2147483648.

   NOTE -2147483648 is the smallest 32-bit signed integer.

10. Return *value* * *sign*.

NOTE In addition to returning the decoded value, this algorithm updates the position variable in the calling algorithm.

5.1.1 Mappings for generated JavaScript code

Generated code positions that may have mapping entries are defined in terms of *input elements* as per ECMAScript Lexical Grammar. Mapping entries shall point to either:

1. the first code point of the source text matched by IdentifierName, PrivateIdentifier, Punctuator, DivPunctuator, RightBracePunctuator, NumericLiteral and RegularExpressionLiteral.
2. any code point of the source text matched by Comment, HashbangComment, StringLiteral, Template, TemplateSubstitutionTail, WhiteSpace and LineTerminator.

5.2 Resolving sources

If the sources are not absolute URLs after prepending the sourceRoot, the sources are resolved relative to the SourceMap (like resolving the script `src` attribute in an HTML document).

To ***decode source map sources*** given a URL *baseURL*, a string or null ***sourceRoot***, a list of either strings or nulls ***sources***, a list of either strings or nulls ***sourcesContent***, and a list of numbers ***ignoredSources***, run the following steps:

1. Let *decodedSources* be a new empty list.
2. Let *sourceURLPrefix* be "".
3. If *sourceRoot* is not null, then:
   a. If *sourceRoot* contains the code point U+002F (/), then:

i. Let *index* be the index of the last occurrence of U+002F (/) in *sourceRoot*.
ii. Set *sourceURLPrefix* to the substring of *sourceRoot* from 0 to *index* + 1.

b. Else, set *sourceURLPrefix* to the concatenation of *sourceRoot* and "/".

4. For each *source* of *sources* with index *index*:
   a. Let *decodedSource* be a new decoded source whose URL is null, content is null, and ignored is false.
   b. If *source* is not null:
      i. Set *source* to the concatenation of *sourceURLPrefix* and *source*.
      ii. Let *sourceURL* be the result of URL parsing *source* with *baseURL*.
      iii. If *sourceURL* is failure, optionally report an error.
      iv. Else, set *decodedSource*'s URL to *sourceURL*.
   c. If *index* is in *ignoredSources*, set *decodedSource*'s ignored to true.
   d. If *sourcesContent*'s size is greater than or equal to *index*, set *decodedSource*'s content to *sourcesContent*[*index*].
   e. Append *decodedSource* to *decodedSources*.
5. Return *decodedSources*.

NOTE Implementations that support showing source contents but do not support showing multiple sources with the same URL and different content will arbitrarily choose one of the various contents corresponding to the given URL.

5.3 Extensions

Source map consumers shall ignore any additional unrecognized properties, rather than causing the source map to be rejected, so that additional features can be added to this format without breaking existing users.

6 Index map

To support concatenating generated code and other common post-processing, an alternate representation of a map is supported:

```
{
  "version" : 3,
  "file": "app.js",
  "sections": [
    {
      "offset": {"line": 0, "column": 0},
      "map": {
        "version" : 3,
        "file": "section.js",
        "sources": ["foo.js", "bar.js"],
        "names": ["src", "maps", "are", "fun"],
        "mappings": "AAAA,E;;ABCDE"
      }
    },
    {
      "offset": {"line": 100, "column": 10},
      "map": {
        "version" : 3,
        "file": "another_section.js",
        "sources": ["more.js"],
        "names": ["more", "is", "better"],
        "mappings": "AAAA,E;AACA,C;ABCDE"
      }
    }
  ]
}
```

The index map follows the form of the standard map. Like the regular source map, the file format is JSON with a top-level object. It shares the version and file field from the regular source map, but gains a new sections field.

`sections` is an array of Section objects.

6.1 Section

Section objects have the following fields:

- ***`offset`*** is an object with two fields, **`line`** and **`column`**, that represent the offset into generated code that the referenced source map represents.
- ***`map`*** is an embedded complete source map object. An embedded map does not inherit any values from the containing index map.

The sections shall be sorted by starting position and the represented sections shall not overlap.

7 Retrieving source maps

7.1 Linking generated code to source maps

While the source map format is intended to be language and platform agnostic, it is useful to define how to reference to them for the expected use-case of web server-hosted JavaScript.

There are two possible ways to link source maps to the output. The first requires server support in order to add an HTTP header and the second requires an annotation in the source.

Source maps are linked through URLs as defined in URL; in particular, characters outside the set permitted to appear in URIs shall be percent-encoded and it may be a data URI. Using a data URI along with sourcesContent allows for a completely self-contained source map.

The HTTP **`sourcemap`** header has precedence over a source annotation, and if both are present, the header URL should be used to resolve the source map file.

Regardless of the method used to retrieve the Source mapping URL the same process is used to resolve it, which is as follows:

When the Source mapping URL is not absolute, then it is relative to the generated code's ***source origin***. The source origin is determined by one of the following cases:

- If the generated source is not associated with a script element that has a **`src`** attribute and there exists a **`//# sourceURL`** comment in the generated code, that comment should be used to determine the source origin.

  > NOTE Previously, this was **`//@ sourceURL`**, as with **`//@ sourceMappingURL`**, it is reasonable to accept both but **`//#`** is preferred.

- If the generated code is associated with a script element and the script element has a **`src`** attribute, the **`src`** attribute of the script element will be the source origin.
- If the generated code is associated with a script element and the script element does not have a **`src`** attribute, then the source origin will be the page's origin.
- If the generated code is being evaluated as a string with the **`eval()`** function or via **`new Function()`**, then the source origin will be the page's origin.

7.1.1 Linking through HTTP headers

If a file is served through HTTP(S) with a **`sourcemap`** header, the value of the header is the URL of the linked source map.

```
sourcemap: <url>
```

NOTE Previous revisions of this document recommended a header name of **`x-sourcemap`**. This is now deprecated; **`sourcemap`** is now expected.

7.1.2 Linking through inline annotations

The generated code should include a comment, or the equivalent construct depending on its language or format, named **`sourceMappingURL`** and that contains the URL of the source map. This specification defines how the comment should look like for JavaScript, CSS, and WebAssembly. Other languages should follow a similar convention.

For a given language there can be multiple ways of detecting the **`sourceMappingURL`** comment, to allow for different implementations to choose what is less complex for them. The generated code ***unambiguously links to a source map*** if the result of all the extraction methods is the same.

If a tool consumes one or more source files that unambiguously links to a source map and it produces an output file that links to a source map, it shall do so unambiguously.

Example 3 The following JavaScript code links to a source map, but it does not do so unambiguously:

```
let a = `
  //# sourceMappingURL=foo.js.map
  //`;
```

Extracting a Source map URL from it through parsing gives null, while without parsing gives **`foo.js.map`**.

Issue 1 Having multiple ways to extract a source map URL, that can lead to different results, can have negative security and privacy implications. Implementations that need to detect which source maps are potentially going to be loaded are strongly encouraged to always apply both algorithms, rather than just assuming that they will give the same result.

A fix to this problem is being worked on, and is expected to be included in a future version of the standard. It will likely involve early returning from the below algorithms whenever there is a comment (or comment-like) that contains the characters U+0060 (`), U+0022 ("), or U+0027 ('), or the sequence U+002A U+002F (*/).

7.1.2.1 Extraction methods for JavaScript sources

To ***extract a Source map URL from JavaScript through parsing*** a string *source*, run the following steps:

1. Let *tokens* be the list of tokens obtained by parsing *source* according to ECMA-262.
2. For each *token* in *tokens*, in reverse order:
   a. If *token* is not a single-line comment or a multi-line comment, return null.

   b. Let *comment* be the content of *token*.
   c. If matching a Source map URL in *comment* returns a string, return it.
3. Return null.

To ***extract a Source map URL from JavaScript without parsing*** a string *source*, run the following steps:

1. Let *lines* be the result of strictly splitting *source* on ECMAScript line terminator code points.
2. Let *lastURL* be null.
3. For each *line* in *lines*:
   a. Let *position* be a position variable for *line*, initially pointing at the start of *line*.
   b. While *position* doesn't point past the end of *line*:
      i. Collect a sequence of code points that are ECMAScript white space code points from *line* given *position*.

         NOTE The collected code points are not used, but *position* is still updated.

      ii. If *position* points past the end of *line*, break.
      iii. Let *first* be the code point of *line* at *position*.
      iv. Increment *position* by 1.
      v. If *first* is U+002F (/) and *position* does not point past the end of *line*, then:
         1. Let *second* be the code point of *line* at *position*.
         2. Increment *position* by 1.
         3. If *second* is U+002F (/), then:
            a. Let *comment* be the code point substring from *position* to the end of *line*.
            b. If matching a Source map URL in *comment* returns a string, set *lastURL* to it.
            c. Break.
         4. Else if *second* is U+002A (*), then:
            a. Let *comment* be the empty string.
            b. While *position* + 1 doesn't point past the end of *line*:
               i. Let *c1* be the code point of *line* at *position*.
               ii. Increment *position* by 1.
               iii. Let *c2* be the code point of *line* at *position*.
               iv. If *c1* is U+002A (*) and *c2* is U+002F (/), then:
                  i. If matching a Source map URL in *comment* returns a string, set *lastURL* to it.
                  ii. Increment *position* by 1.
               v. Append *c1* to *comment*.
         5. Else, set *lastURL* to null.
      vi. Else, set *lastURL* to null.

      NOTE We reset *lastURL* to null whenever we find a non-comment code character.

4. Return *lastURL*.

NOTE The algorithm above has been designed so that the source lines can be iterated in reverse order, returning early after scanning through a line that contains a **`sourceMappingURL`** comment.

NOTE The algorithm above is equivalent to the following JavaScript implementation:

```
const JS_NEWLINE = /^/m;

// This RegExp will always match one of the following:
// - single-line comments
// - "single-line" multi-line comments
// - unclosed multi-line comments
// - just trailing whitespaces
// - a code character
// The loop below differentiates between all these cases.
const JS_COMMENT =
/\s*(?:\/\/(?<single>.*)|\/\*(?<multi>.*?)\*\/|\/\*.*|$|(?<code>[^\/]+))/uym;

const PATTERN = /^[@#]\s*sourceMappingURL=(\S*?)\s*$/;

let lastURL = null;
for (const line of source.split(JS_NEWLINE)) {
  JS_COMMENT.lastIndex = 0;
  while (JS_COMMENT.lastIndex < line.length) {
    let commentMatch = JS_COMMENT.exec(line).groups;
    let comment = commentMatch.single ?? commentMatch.multi;
    if (comment != null) {
      let match = PATTERN.exec(comment);
      if (match !== null) lastURL = match[1];
    } else if (commentMatch.code != null) {
      lastURL = null;
    } else {
      // We found either trailing whitespaces or an unclosed comment.
      // Assert: JS_COMMENT.lastIndex === line.length
    }
  }
}
return lastURL;
```

To ***match a Source map URL in a comment*** *comment* (a string), run the following steps:

1. Let *pattern* be the regular expression `/^[@#]\s*sourceMappingURL=(\S*?)\s*$/`.
2. Let *match* be ! RegExpBuiltInExec(*pattern*, *comment*).
3. If *match* is not null, return *match*[1].
4. Return null.

NOTE The prefix for this annotation was initially `//@` however this conflicts with Internet Explorer's Conditional Compilation and was changed to `//#`.

Source map generators shall only emit `//#` while source map consumers shall accept both `//@` and `//#`.

7.1.2.2 Extraction methods for CSS sources

Extracting source mapping URLs from CSS is similar to JavaScript, with the exception that CSS only supports `/* ... */`-style comments.

7.1.2.3 Extraction methods for WebAssembly binaries

To ***extract a Source map URL from a WebAssembly source*** given a byte sequence *bytes*, run the following steps:

1. Let *module* be module_decode(*bytes*).
2. If *module* is error, return null.
3. For each custom section *customSection* of *module*,
   a. Let *name* be the `name` of *customSection*, decoded as UTF-8.
   b. If *name* is "sourceMappingURL", then:
      i. Let *value* be the `bytes` of *customSection*, decoded as UTF-8.
      ii. If *value* is failure, return null.
      iii. Return *value*.
4. Return null.

Since WebAssembly is not a textual format and it does not support comments, it supports a single unambiguous extraction method. The URL is encoded using WasmNamesBinaryFormat, and it's placed as the content of the custom section. It is invalid for tools that generate WebAssembly code to generate two or more custom sections with the "sourceMappingURL" name.

7.2 Fetching source maps

To fetch a source map given a URL *url*, run the following steps:

1. Let *promise* be a new promise.
2. Let *request* be a new request whose URL is *url*.
3. Fetch *request* with processResponseConsumeBody set to the following steps given response *response* and null, failure, or a byte sequence *bodyBytes*:
   a. If *bodyBytes* is null or failure, reject *promise* with a `TypeError` and abort these steps.
   b. If *url*'s scheme is an HTTP(S) scheme and *bodyBytes* starts with `` `)]}'` ``, then:
      i. While *bodyBytes*'s length is not `0` and *bodyBytes*'s `0`th byte is not an HTTP newline byte:
         1. remove the `0`th byte from *bodyBytes*.

> NOTE For historic reasons, when delivering source maps over HTTP(S), servers may prepend a line starting with the string `)]}'` to the source map.
>
> ```
> )]}'garbage here
> {"version": 3, ...}
> ```
>
> is interpreted as
>
> ```
> {"version": 3, ...}
> ```

   c. Let *sourceMap* be the result of parsing JSON bytes to a JavaScript value given *bodyBytes*.
   d. If the previous step threw an error, reject *promise* with that error.
   e. Otherwise, resolve *promise* with *sourceMap*.
4. Return *promise*.

8 Conventions

The following conventions should be followed when working with source maps or when generating them.

8.1 Source map naming

Commonly, a source map will have the same name as the generated file but with a `.map` extension. For example, for `page.js` a source map named `page.js.map` would be generated.

8.2 Linking eval'd code to named generated code

There is an existing convention that should be supported for the use of source maps with eval'd code, it has the following form:

```
//# sourceURL=foo.js
```

It is described in EvalSourceURL.

9 Language neutral stack mapping notes

Stack tracing mapping without knowledge of the source language is not covered by this document.

10 Multi-level mapping notes

It is getting more common to have tools generate sources from some DSL (templates) or compile TypeScript -> JavaScript -> minified JavaScript, resulting in multiple translations before the final source map is created. This problem can be handled in one of two ways. The easy but lossy way is to ignore the intermediate steps in the process for the purposes of debugging, the source location information from the translation is either ignored (the intermediate translation is considered the ”Original Source“) or the source location information is carried through (the intermediate translation hidden). The more complete way is to support multiple levels of mapping: if the Original Source also has a source map reference, the user is given the choice of using that as well.

However, It is unclear what a "source map reference" looks like in anything other than JavaScript. More specifically, what a source map reference looks like in a language that doesn't support JavaScript-style single-line comments.

Annex A
(informative)

Index

A.1 Terms defined by this specification

A.2 Terms defined by reference

[ECMA-262] defines the following terms:

- Comment
- DivPunctuator
- ECMAScript Lexical Grammar
- HashbangComment
- IdentifierName
- line terminator code points
- LineTerminator
- multi-line comment
- NumericLiteral
- PrivateIdentifier
- Punctuator
- RegExpBuiltinExec
- RegularExpressionLiteral
- RightBracePunctuator
- single-line comment
- StringLiteral
- Template
- TemplateSubstitutionTail
- tokens
- white space code points
- WhiteSpace

[ENCODING] defines the following terms:

- UTF-8 decode without BOM or fail

[FETCH] defines the following terms:

- fetch
- HTTP newline byte
- HTTP(S) scheme
- processResponseConsumeBody
- request
- response
- URL

[INFRA] defines the following terms:

- append
- ASCII string
- boolean
- break
- byte
- byte sequence
- code point
- code point substring
- code unit
- code unit substring
- collect a sequence of code points

- concatenate
- exist
- for each
- length
- list
- map
- parse a JSON string to an Infra value
- parse JSON bytes to a JavaScript value
- position variable
- size
- starts with
- strictly split
- string
- struct
- value
- while

[URL] defines the following terms:

- scheme
- URL
- URL parser

[WASM] defines the following terms:

- custom section
- module_decode

[WEBIDL] defines the following terms:

- TypeError
- a new promise
- reject
- resolve

Bibliography

[1] Anne van Kesteren. *Encoding Standard*, https://encoding.spec.whatwg.org/. Living Standard.

[2] Anne van Kesteren. *Fetch Standard*, https://fetch.spec.whatwg.org/. Living Standard.

[3] Anne van Kesteren; Domenic Denicola. *Infra Standard*, https://infra.spec.whatwg.org/. Living Standard.

[4] *URL Standard*, https://url.spec.whatwg.org/. Living Standard.

[5] Edgar Chen; Timothy Gu. *Web IDL Standard*, https://webidl.spec.whatwg.org/. Living Standard.

[6] *Give your eval a name with //@ sourceURL*, https://web.archive.org/web/20120814122523/http://blog.getfirebug.com/2009/08/11/give-your-eval-a-name-with-sourceurl/. archive.

[7] *Source map Revision 2 Proposal*, https://docs.google.com/document/d/1xi12LrcqjqIHTtZzrzZKmQ3lbTv9mKrN076UB-j3UZQ/edit?hl=en_US.

[8] *Variable-length quantity*, https://en.wikipedia.org/wiki/Variable-length_quantity. reference article.