Web Bundles
draft-yasskin-wpack-bundled-exchanges-04

Abstract

Web bundles provide a way to bundle up groups of HTTP responses, with the request URLs and content negotiation that produced them, to transmit or store together. They can include multiple top-level resources with one identified as the default by a primaryUrl metadata, provide random access to their component exchanges, and efficiently store 8-bit resources.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the WPACK Working Group mailing list (wpack@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/wpack/.

Source for this draft and an issue tracker can be found at https://github.com/WICG/webpackage.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 8 September 2021.
Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the
document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal
license-info) in effect on the date of publication of this document.
Please review these documents carefully, as they describe your rights
and restrictions with respect to this document. Code Components
extracted from this document must include Simplified BSD License text
as described in Section 4.e of the Trust Legal Provisions and are
provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction ............................................... 3
   1.1. Terminology and Conventions ......................... 3
2. Semantics ................................................. 3
   2.1. Operations ............................................ 3
   2.2. Naming a representation .............................. 3
3. Expected performance ...................................... 4
   3.1. Random access ........................................ 4
   3.2. Streaming ............................................. 4
4. Format .................................................... 4
   4.1. Top-level structure .................................. 4
      4.1.1. Trailing length .................................. 6
      4.1.2. Draft version numbers ........................... 6
   4.2. Bundle sections ..................................... 6
      4.2.1. The index section ............................... 7
      4.2.2. The manifest section ............................ 9
      4.2.3. The critical section ............................ 9
   4.3. Responses ............................................ 9
   4.4. Serving constraints .................................. 10
5. Security Considerations ................................... 10
   5.1. Version skew ........................................ 10
   5.2. Content sniffing ..................................... 11
6. IANA considerations ....................................... 12
   6.1. Internet Media Type Registration ................. 12
   6.2. Web Bundle Section Name Registry ............... 13
7. References ............................................... 14
   7.1. Normative References ............................... 14
   7.2. Informative References ............................. 15
Appendix A. Change Log ................................... 16
Appendix B. Acknowledgements ........................... 17
Author’s Address .......................................... 17
1. Introduction

To satisfy the use cases in [I-D.yasskin-wpack-use-cases], this document proposes a new bundling format to group HTTP resources. The format is structured as an initial table of "sections" within the bundle followed by the content of those sections.

1.1. Terminology and Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This specification uses the conventions and terminology defined in the Infra Standard ([INFRA]).

2. Semantics

A bundle is logically a set of HTTP representations (Section 7 of [I-D.ietf-httpbis-semantics]), themselves represented by HTTP response messages (Section 2.1 of [I-D.ietf-httpbis-semantics]). The bundle can include an optional URL identifying the primary resource within the bundle and can include other optional metadata. Particular applications can require that the primary URL and/or other metadata is present.

While the order of the representations is not semantically meaningful, it can significantly affect performance when the bundle is loaded from a network stream.

2.1. Operations

Bundle parsers support two primary operations:

1. They can load the bundle’s metadata given a prefix of the bundle.

2. They can find a representation within the bundle given that representation's URL (Section 2.2) and the content-negotiation information that would appear in an HTTP request’s headers.

2.2. Naming a representation

Representations within a bundle are named by their "Content-Location" (Section 7.8 of [I-D.ietf-httpbis-semantics]), which holds a URL. This is also known as the representation’s URL.
Multiple representations within a bundle can have the same URL, in which case they are distinguished by the content negotiation information contained in their "Variants" and "Variant-Key" headers ([I-D.ietf-httpbis-variants]).

This identifying information for each representation is stored in an index (Section 4.2.1) rather than in that representation’s HTTP response message.

3. Expected performance

Bundles can be used in two different situations: they can be loaded from storage that provides O(1) access to any byte within the bundle, or they can be sent across a stream that provides bytes incrementally. An implementation MAY prefer either or both situations and SHOULD provide the following performance characteristics in its preferred situations:

3.1. Random access

To load a resource when seeing a bundle for the first time, the implementation reads O(size of the metadata and resource index) before starting to return bytes of the resource.

TODO: Is big-O notation the right way to express expectations here?

3.2. Streaming

When sending a bundle over a stream, the implementation will need to wait until it has the sizes of all contained resources before starting to send the resource index.

When reading a bundle from a stream, the implementation starts returning bytes of a resource after receiving O(1) bytes of that resource, which comes after the O(# of resources) bytes of the index.

4. Format

4.1. Top-level structure

A bundle is a CBOR array ([CBORbis]) with the following CDDL ([CDDL]) schema:
webbundle = [
  magic: h'F0 9F 8C 90 F0 9F 93 A6',
  version: bytes .size 4,
  primary-url: whatwg-url,
  section-lengths: bytes .cbor section-lengths,
  sections: [* any ],
  length: bytes .size 8, ; Big-endian number of bytes in the bundle.
]
whatwg-url = tstr

When serialized, the bundle MUST satisfy the core deterministic encoding requirements from Section 4.2.1 of [CBORbis]. This format does not use floating point values or tags, so this specification does not add any deterministic encoding rules for them. If an item doesn't follow these requirements, or a byte-sequence being decoded as a CBOR item contains extra bytes, the parser MUST signal an error instead of any data it can extract from that item.

High-level fields in the bundle format are designed to provide their length-in-bytes before the field starts so that a recipient trying to stream a bundle from the network can always wait for a known number of bytes instead of needing to implement a streaming CBOR parser.

The "magic" number is "" (U+1F310 U+1F4E6) encoded in UTF-8. With the CBOR initial bytes for the array and bytestring, this makes the format identifiable by looking for "8? 48" (in base 16) followed by that UTF-8 encoding. Parsers MUST only check the initial nibble of the initial "8?" byte in order to accommodate any future version's change in the number of array elements (up to 15).

The "version" bytestring MUST be "31 00 00 00" in base 16 (an ASCII "1" followed by 3 0s) for this version of bundles. If the recipient doesn't support the version in this field, it MUST either ignore the bundle or fetch and use the content of the "primary-url" field instead.

The "primary-url" field identifies both a fallback when the recipient doesn't understand the bundle and a default resource inside the bundle to use when the recipient doesn't have more specific instructions. This field MAY be an empty string, although protocols using bundles MAY themselves forbid that empty value.
The "section-lengths" and "sections" arrays contain the actual content of the bundle and are defined in Section 4.2. The "section-lengths" array is embedded in a byte string to facilitate reading it from a network. This byte string MUST be less than 8192 (8*1024) bytes long, and parsers MUST NOT load any data from a "section-lengths" item longer than this.

The bundle ends with an 8-byte integer holding the length of the whole bundle.

4.1.1. Trailing length

A bundle ends with an 8-byte CBOR byte string holding a big-endian integer that represents the byte-length of the whole bundle.

```
+------------+-----+----+----+----+----+----+----+----+----+----+
| first byte | ... | 48 | 00 | 00 | 00 | 00 | BC | 61 | 4E |
+------------+-----+----+----+----+----+----+----+----+----+----+
```

\[ \text{0xBC614E-10=12345668 omitted bytes} \]

Figure 1: Example trailing bytes

Recipients loading the bundle in a random-access context SHOULD start by reading the last 8 bytes and seeking backwards by that many bytes to find the start of the bundle, instead of assuming that the start of the file is also the start of the bundle. This allows the bundle to be appended to another format such as a generic self-extracting executable.

4.1.2. Draft version numbers

This section is to be removed before publishing as an RFC.

Implementations of drafts of this specification MUST NOT use a "version" string of "31 00 00 00" (base 16). They MUST instead define an implementation-specific 4-byte string starting with "62" ("b") to identify which draft is implemented.

4.2. Bundle sections

A bundle’s content is in a series of sections, which can be accessed randomly using the information in the "section-lengths" CBOR item:

```
section-lengths = [ (section-name: tstr, length: uint) ],
```

Yasskin  Expires 8 September 2021  [Page 6]
This field lists the named sections in the bundle in the order they appear, with each section name followed by the length in bytes of the corresponding CBOR item in the "sections" array. This allows a random-access parser (Section 3) to jump directly to the section it needs. This specification defines the following sections:

* "index" (Section 4.2.1)
* "manifest" (Section 4.2.2)
* "critical" (Section 4.2.3)
* "responses" (Section 4.3)

Future specifications can register new section names as described in Section 6.2, in order to extend the format without incrementing its version number.

The "responses" section MUST appear after the other three sections defined here, and parsers MUST NOT load any data if that is not the case.

The "sections" array contains the sections’ content. The length of this array MUST be exactly half the length of the "section-lengths" array, and parsers MUST NOT load any data if that is not the case.

The bundle MUST contain the "index" and "responses" sections. All other sections are optional.

4.2.1. The index section

index = {* whatwg-url => [ variants-value, +location-in-responses ] }
variants-value = bstr
location-in-responses = (offset: uint, length: uint)

The "index" section defines the set of HTTP representations in the bundle and identifies their locations in the "responses" section. It consists of a CBOR map whose keys are the URLs of the representations in the bundle (Section 2.2). The value of an index entry is an array whose first item is a "Variants" header field value ([I-D.ietf-httpbis-variants]) or the empty string. This is followed by a sequence of offset/length pairs, one for each representation of this resource. The offset is relative to the start of the "responses" section, with an offset of 0 referring to the head of the CBOR "responses" array itself. The length is the length in bytes of the "response" CBOR item holding this representation (Section 4.3).
If the first item in the value of an index entry is empty, it MUST be followed by exactly one offset/length pair. This means there is a single representation for this resource, with no content negotiation.

Otherwise, the first item MUST be followed by one offset/length pair for each of the possible combinations of available-values within the "Variants" value (the first item of the array) in lexicographic (row-major) order.

For example, given a "Variants" value of "accept-encoding=(gzip br), accept-language=(en fr ja)", the list of offset/length pairs will correspond to the "Variant-Key"s:

* (gzip en)
* (gzip fr)
* (gzip ja)
* (br en)
* (br fr)
* (br ja)

The order of variant-axes is important. If the "Variants" value were "accept-language=(en fr ja), accept-encoding=(gzip br)" instead, the "location-in-responses" pairs would instead correspond to:

* (en gzip)
* (en br)
* (fr gzip)
* (fr br)
* (ja gzip)
* (ja br)

If the wrong number of offset/length pairs is present in a resource’s array, the entire index MUST fail to parse.

A combination of available-values that is omitted from the bundle MUST be signaled by setting its offset and length to 0.
4.2.2. The manifest section

manifest = whatwg-url

The "manifest" section records a single URL identifying the manifest of the bundle. The URL MUST refer to a resource with representations contained in the bundle itself.

The bundle can contain multiple representations at this URL, and the client is expected to content-negotiate for the best one. For example, a client might select the one matching an "accept" header of "application/manifest+json" ([appmanifest]) and an "accept-language" header of "es-419".

Many bundles have a choice between identifying their manifest in this section or in their primary resource, especially if that resource is an HTML file. Identifying the manifest in this section can help recipients apply fields in the manifest sooner, for example to show a splash screen before parsing the primary resource.

4.2.3. The critical section

critical = [*tstr]

The "critical" section consists of the names of sections of the bundle that the client needs to understand in order to load the bundle correctly. Other sections are assumed to be optional.

If the client has not implemented a section named by one of the items in this list, the client MUST fail to parse the bundle as a whole.

4.3. Responses

responses = [*response]
response = [headers: bstr .cbor headers, payload: bstr]
headers = {* bstr => bstr}

The "responses" section holds the HTTP responses that represent the HTTP representations in the bundle. It consists of a CBOR array of responses, each of which is pointed to by one or more entries in the "index" section (Section 4.2.1).

The length of the "headers" byte string in a response MUST be less than 524288 (512*1024) bytes, and recipients MUST fail to load a response with longer headers.
When receiving a bundle in a stream, the recipient MAY process the headers before the payload has been received and MAY start processing the beginning of the payload before the end of the payload has been received.

The keys of the headers map MUST consist of lowercase ASCII as described in Section 8.1.2 of [RFC7540]. Response pseudo-headers (Section 8.1.2.4 of [RFC7540]) are included in this headers map.

Each response’s headers MUST include a ":status" pseudo-header with exactly 3 ASCII decimal digits and MUST NOT include any other pseudo-headers.

If a response’s payload is not empty, its headers MUST include a "Content-Type" header (Section 7.4 of [I-D.ietf-httpbis-semantics]). The client MUST interpret the following payload as this specified media type instead of trying to sniff a media type from the bytes of the payload, for example by appending an artificial "X-Content-Type-Options: nosniff" header field ([FETCH]) to downstream protocols.

4.4. Serving constraints

When served over HTTP, a response containing an "application/webbundle" payload MUST include at least the following response header fields, to reduce content sniffing vulnerabilities (Section 5.2):

* Content-Type: application/webbundle
* X-Content-Type-Options: nosniff

5. Security Considerations

5.1. Version skew

Bundles currently have no mechanism for ensuring that any signed exchanges they contain constitute a consistent version of those resources. Even if a website never has a security vulnerability when resources are fetched at a single time, an attacker might be able to combine a set of resources pulled from different versions of the website to build a vulnerable site. While the vulnerable site could have occurred by chance on a client’s machine due to normal HTTP caching, bundling allows an attacker to guarantee that it happens. Future work in this specification might allow a bundle to constrain its resources to come from a consistent version.
5.2. Content sniffing

While modern browsers tend to trust the "Content-Type" header sent with a resource, especially when accompanied by "X-Content-Type-Options: nosniff", plugins will sometimes search for executable content buried inside a resource and execute it in the context of the origin that served the resource, leading to XSS vulnerabilities. For example, some PDF reader plugins look for "%PDF" anywhere in the first 1kB and execute the code that follows it.

The "application/webbundle" format defined above includes URLs and request headers early in the format, which an attacker could use to cause these plugins to sniff a bad content type.

To avoid vulnerabilities, in addition to the response header requirements in Section 4.4, servers are advised to only serve an "application/webbundle" resource from a domain if it would also be safe for that domain to serve the bundle's content directly, and to follow at least one of the following strategies:

1. Only serve bundles from dedicated domains that don’t have access to sensitive cookies or user storage.

2. Generate bundles "offline", that is, in response to a trusted author submitting content or existing signatures reaching a certain age, rather than in response to untrusted-reader queries.

3. Do all of:

   1. If the bundle’s contained URLs (e.g. in the manifest and index) are derived from the request for the bundle, percent-encode (https://url.spec.whatwg.org/#percent-encode) ([URL]) any bytes that are greater than 0x7E or are not URL code points (https://url.spec.whatwg.org/#url-code-points) ([URL]) in these URLs. It is particularly important to make sure no unescaped nulls (0x00) or angle brackets (0x3C and 0x3E) appear.

   2. Similarly, if the request headers for any contained resource are based on the headers sent while requesting the bundle, only include request header field names *and values* that appear in a static allowlist. Keep the set of allowed request header fields smaller than 24 elements to prevent attackers from controlling a whole CBOR length byte.

   3. Restrict the number of items a request can direct the server to include in a bundle to less than 12, again to prevent attackers from controlling a whole CBOR length byte.
4. Do not reflect request header fields into the set of response headers.

If the server serves responses that are written by a potential attacker but then escaped, the "application/webbundle" format allows the attacker to use the length of the response to control a few bytes before the start of the response. Any existing mechanisms that prevent polyglot documents probably keep working in the face of this new attack, but we don’t have a guarantee of that.

To encourage servers to include the "X-Content-Type-Options: nosniff" header field, clients SHOULD reject bundles served without it.

6. IANA considerations

6.1. Internet Media Type Registration

IANA is requested to register the MIME media type ([IANA.media-types]) for web bundles, application/webbundle, as follows:

* Type name: application
* Subtype name: webbundle
* Required parameters:
  - v: A string denoting the version of the file format. ([RFC5234] ABNF: "version = 1*(DIGIT/%%61-7A)") The version defined in this specification is "1".

  Note: RFC EDITOR PLEASE DELETE THIS NOTE; Implementations of drafts of this specification MUST NOT use simple integers to describe their versions, and MUST instead define implementation-specific strings to identify which draft is implemented.

* Optional parameters: N/A
* Encoding considerations: binary
* Security considerations: See Section 5 of this document.
* Interoperability considerations: N/A
* Published specification: This document
Applications that use this media type: None yet, but it is expected that web browsers will use this format.

Fragment identifier considerations: N/A

Additional information:
- Deprecated alias names for this type: N/A
- Magic number(s): 86 48 F0 9F 8C 90 F0 9F 93 A6
- File extension(s): .wbn
- Macintosh file type code(s): N/A

Person & email address to contact for further information: See the Author’s Address section of this specification.

Intended usage: COMMON

Restrictions on usage: N/A

Author: See the Author’s Address section of this specification.

Change controller: The IESG iesg@ietf.org (mailto:iesg@ietf.org)

Provisional registration? Yes.

6.2. Web Bundle Section Name Registry

IANA is directed to create a new registry with the following attributes:

Name: Web Bundle Section Names

Review Process: Specification Required

Initial Assignments:
Table 1

Requirements on new assignments:

Section Names MUST be encoded in UTF-8.

A section’s specification MAY say that, if it is present, another section is not processed.

7. References

7.1. Normative References

[appmanifest]


7.2. Informative References

[IANA.media-types]
IANA, "Media Types",
<http://www.iana.org/assignments/media-types>.

[INFRA] WHATWG, "Infra", March 2021,
<https://infra.spec.whatwg.org/>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
Requirement Levels", BCP 14, RFC 2119,
DOI 10.17487/RFC2119, March 1997,

Specifications: ABNF", STD 68, RFC 5234,
DOI 10.17487/RFC5234, January 2008,

Transfer Protocol Version 2 (HTTP/2)", RFC 7540,
DOI 10.17487/RFC7540, May 2015,

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC
2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174,


Appendix A. Change Log

This section is to be removed before publishing as an RFC.

draft-04
* Rewrite to be more declarative and less algorithmic.
* Make a bundle represent a set of HTTP Representations, with the Content-Location replacing what was the request URL, and the Variants information, as before, driving content negotiation.
* Make the primary URL optional.
* Remove the signatures section.
* Update Variants examples for the latest Variants draft.
* Removed the distinction between "metadata" and non-metadata sections.

draft-03
* Make the manifest optional.
* Update the reference to draft-yasskin-wpack-use-cases.
* Retitle to "web bundles".

draft-02
* Fix the initial bytes of the format.
* Allow empty responses to omit their content type.
* Provisionally register application/webbundle.

draft-01
* Include only section lengths in the section index, requiring sections to be listed in order.
* Have the "index" section map URLs to sets of responses negotiated using the Variants system ([I-D.ietf-httpbis-variants]).

* Require the "manifest" to be embedded into the bundle.

* Add a content sniffing security consideration.

* Add a version string to the format and its mime type.

* Add a fallback URL in a fixed location in the format, and use that fallback URL as the primary URL of the bundle.

* Add a "signatures" section to let authorities (like domain-trusted X.509 certificates) vouch for subsets of a bundle.

* Use the CBORbis "deterministic encoding" requirements instead of "canonicalization" requirements.

Appendix B. Acknowledgements

Thanks to the Chrome loading team, especially Kinuko Yasuda and Kouhei Ueno for making the format work well when streamed.

Author’s Address

Jeffrey Yasskin
Google

Email: jyasskin@chromium.org
Abstract

This document lists use cases for signing and/or bundling collections of web pages, and extracts a set of requirements from them.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the WPACK Working Group mailing list (wpack@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/wpack/.

Source for this draft and an issue tracker can be found at https://github.com/WICG/webpackage.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 15 October 2021.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.
# Table of Contents

1. Introduction .................................................. 3
2. Use cases ....................................................... 4
   2.1. Essential .................................................. 4
       2.1.1. Offline installation ................................. 4
       2.1.2. Offline browsing .................................... 6
       2.1.3. Save and share a web page .......................... 6
       2.1.4. Privacy-preserving prefetch ........................ 7
   2.2. Nice-to-have ............................................... 7
       2.2.1. Packaged Web Publications .......................... 8
       2.2.2. Avoiding Censorship ................................ 9
       2.2.3. Third-party security review ......................... 9
       2.2.4. Building packages from multiple libraries ......... 10
       2.2.5. Cross-CDN Serving .................................. 10
       2.2.6. Pre-installed applications .......................... 11
       2.2.7. Protecting Users from a Compromised Frontend .... 12
       2.2.8. Installation from a self-extracting executable ... 13
       2.2.9. Packages in version control .......................... 13
       2.2.10. Subresource bundling ................................ 13
       2.2.11. Archival .............................................. 14
3. Requirements .................................................... 15
   3.1. Essential .................................................. 15
       3.1.1. Indexed by URL ....................................... 15
       3.1.2. Request headers ...................................... 15
       3.1.3. Response headers .................................... 15
       3.1.4. Signing as an origin ................................ 15
       3.1.5. Random access ...................................... 16
       3.1.6. Resources from multiple origins in a package ..... 16
       3.1.7. Cryptographic agility ................................ 16
       3.1.8. Unsigned content .................................... 16
       3.1.9. Certificate revocation ............................... 16
       3.1.10. Downgrade prevention ................................. 16
       3.1.11. Metadata ............................................. 17
       3.1.12. Implementations are hard to get wrong ............. 17
   3.2. Nice to have ............................................... 17
       3.2.1. Streamed loading ..................................... 17
       3.2.2. Signing without origin trust ......................... 17
       3.2.3. Additional signatures ................................. 17
1. Introduction

People would like to use content offline and in other situations where there isn’t a direct connection to the server where the content originates. However, it’s difficult to distribute and verify the authenticity of applications and content without a connection to the network. The W3C has addressed running applications offline with Service Workers ([ServiceWorkers]), but not the problem of distribution.

Previous attempts at packaging web resources (e.g. Resource Packages (https://www.mnot.net/blog/2010/02/18/resource_packages) and the W3C TAG’s packaging proposal (https://w3ctag.github.io/packaging-on-the-web/)) were motivated by speeding up the download of resources from a single server, which is probably better achieved through other mechanisms like HTTP/2 PUSH, possibly augmented with a simple manifest of URLs a page plans to use (https://lists.w3.org/Archives/Public/public-web-perf/2015Jan/0038.html). This attempt is instead motivated by avoiding a connection to the origin server at all. It may still be useful for the earlier use cases, so they’re still listed, but they’re not primary.
2. Use cases

These use cases are in rough descending priority order. If use cases have conflicting requirements, the design should enable more important use cases.

2.1. Essential

2.1.1. Offline installation

Alex can download a file containing a website (a PWA (https://developers.google.com/web/progressive-web-apps/checklist)) including a Service Worker from origin "O", and transmit it to their peer Bailey, and then Bailey can install the Service Worker with a proof that it came from "O". This saves Bailey the bandwidth costs of transferring the website.

There are roughly two ways to accomplish this:

1. Package just the Service Worker Javascript and any other Javascript that it importScripts() (https://w3c.github.io/ServiceWorker/#importscripts), with their URLs and enough metadata to synthesize a navigator.serviceWorker.register(scriptURL, options) call (https://w3c.github.io/ServiceWorker/#navigator-service-worker-register), along with an uninterpreted but signature-checked blob of data that the Service Worker can interpret to fill in its caches.

2. Package the resources so that the Service Worker can fetch() them to populate its cache.

Associated requirements for just the Service Worker:

* Indexed by URL: The "register()" and "importScripts()" calls have semantics that depend on the URL.

* Signing as an origin: To prove that the file came from "O".

* Signing uses existing TLS certificates: So "O" doesn’t have to spend lots of money buying a specialized certificate.

* Cryptographic agility: Today’s algorithms will eventually be obsolete and will need to be replaced.

* Certificate revocation: "O"’s certificate might be compromised or mis-issued, and the attacker shouldn’t then get an infinite ability to mint packages.
* Downgrade prevention: "O"’s site might have an XSS vulnerability, and attackers with an old signed package shouldn’t be able to take advantage of the XSS forever.

* Metadata: Just enough to generate the "register()" call, which is less than a full W3C Application Manifest.

Additional associated requirements for packaged resources:

* Indexed by URL: Resources on the web are addressed by URL.

* Request headers: If Bailey’s running a different browser from Alex or has a different language configured, the "accept*" headers are important for selecting which resource to use at each URL.

* Response headers: The meaning of a resource is heavily influenced by its HTTP response headers.

* Resources from multiple origins in a package: So the site can be built from multiple components (Section 2.2.4).

* Metadata: The browser needs to know which resource within a package file to treat as its Service Worker and/or initial HTML page.

2.1.1.1. Online use

Bailey may have an internet connection through which they can, in real time, fetch updates to the package they received from Alex.

2.1.1.2. Fully offline use

Or Bailey may not have any internet connection a significant fraction of the time, either because they have no internet at all, because they turn off internet except when intentionally downloading content, or because they use up their plan partway through each month.

Associated requirements beyond Offline installation:

* Packaged validity information: Even without a direct internet connection, Bailey should be able to check that their package is still valid.
2.1.2. Offline browsing

Alex can download a file containing a large website (e.g. Wikipedia) from its origin, save it to transferrable storage (e.g. an SD card), and hand it to their peer Bailey. Then Bailey can browse the website with a proof that it came from "O". Bailey may not have the storage space to copy the website before browsing it.

This use case is harder for publishers to support if we specialize Section 2.1.1 for Service Workers since it requires the publisher to adopt Service Workers before they can sign their site.

Associated requirements beyond Offline installation:

* Random access: To avoid needing a long linear scan before using the content.
* Compress stored packages: So that more content can fit on the same storage device.

2.1.3. Save and share a web page

Casey is viewing a web page and wants to save it either for offline use or to show it to their friend Dakota. Since Casey isn’t the web page’s publisher, they don’t have the private key needed to sign the page. Browsers currently allow their users to save pages, but each browser uses a different format (MHTML, Web Archive, or files in a directory), so Dakota and Casey would need to be using the same browser. Casey could also take a screenshot, at the cost of losing links and accessibility.

Associated requirements:

* Unsigned content: A client can’t sign content as another origin.
* Resources from multiple origins in a package: General web pages include resources from multiple origins.
* Indexed by URL: Resources on the web are addressed by URL.
* Response headers: The meaning of a resource is heavily influenced by its HTTP response headers.
2.1.4. Privacy-preserving prefetch

Lots of websites link to other websites. Many of these source sites would like the targets of these links to load quickly. The source could use "<link rel="prefetch">" to prefetch the target of a link, but if the user doesn’t actually click that link, that leaks the fact that the user saw a page that linked to the target. This can be true even if the prefetch is made without browser credentials because of mechanisms like TLS session IDs.

Because clients have limited data budgets to prefetch link targets, this use case is probably limited to sites that can accurately predict which link their users are most likely to click. For example, search engines can predict that their users will click one of the first couple results, and news aggregation sites like Reddit or Slashdot can hope that users will read the article if they’ve navigated to its discussion.

Two search engines have built systems to do this with today’s technology: Google’s AMP (https://www.ampproject.org/) and Baidu’s MIP (https://www.mipengine.org/) formats and caches allow them to prefetch search results while preserving privacy, at the cost of showing the wrong URLs for the results once the user has clicked. A good solution to this problem would show the right URLs but still avoid a request to the publishing origin until after the user clicks.

Associated requirements:

* Signing as an origin: To prove the content came from the original origin.
* Streamed loading: If the user clicks before the target page is fully transferred, the browser should be able to start loading early parts before the source site finishes sending the whole page.
* Compress transfers
* Subsetting and reordering: If a prefetched page includes subresources, its publisher might want to provide and sign both WebP and PNG versions of an image, but the source site should be able to transfer only best one for each client.

2.2. Nice-to-have
2.2.1. Packaged Web Publications

The W3C’s Publishing Working Group (https://www.w3.org/publishing/groups/publ-wg/), merged from the International Digital Publishing Forum (IDPF) and in charge of EPUB maintenance, wants to be able to create publications on the web and then let them be copied to different servers or to other users via arbitrary protocols. See their Packaged Web Publications use cases (https://www.w3.org/TR/pwp-ucr/#pwp) for more details.

Associated requirements:

* Indexed by URL: Resources on the web are addressed by URL.

* Signing as an origin: So that readers can be sure their copy is authentic and so that copying the package preserves the URLs of the content inside it.

* Downgrade prevention: An early version of a publication might contain incorrect content, and a publisher should be able to update that without worrying that an attacker can still show the old content to users.

* Metadata: A publication can have copyright and licensing concerns; a title, author, and cover image; an ISBN or DOI name; etc.; which should be included when that publication is packaged.

Other requirements are similar to those from Offline installation:

* Random access: To avoid needing a long linear scan before using the content.

* Compress stored packages: So that more content can fit on the same storage device.

* Request headers: If different users’ browsers have different capabilities or preferences, the "accept" headers are important for selecting which resource to use at each URL.

* Response headers: The meaning of a resource is heavily influenced by its HTTP response headers.

* Signing uses existing TLS certificates: So a publisher doesn’t have to spend lots of money buying a specialized certificate.

* Cryptographic agility: Today’s algorithms will eventually be obsolete and will need to be replaced.
Certificate revocation: The publisher’s certificate might be compromised or mis-issued, and an attacker shouldn’t then get an infinite ability to mint packages.

### 2.2.2. Avoiding Censorship

Some users want to retrieve resources that their governments or network providers don’t want them to see. Right now, it’s straightforward for someone in a privileged network position to block access to particular hosts, but TLS makes it difficult to block access to particular resources on those hosts.

Today it’s straightforward to retrieve blocked content from a third party, but there’s no guarantee that the third-party has sent the user an accurate representation of the content: the user has to trust the third party.

With signed web packages, the user can re-gain assurance that the content is authentic, while still bypassing the censorship. Packages don’t do anything to help discover this content.

Systems that make censorship more difficult can also make legitimate content filtering more difficult. Because the client that processes a web package always knows the true URL, this forces content filtering to happen on the client instead of on the network.

Associated requirements:

* Indexed by URL: So the user can see that they’re getting the content they expected.

* Signing as an origin: So that readers can be sure their copy is authentic and so that copying the package preserves the URLs of the content inside it.

### 2.2.3. Third-party security review

Some users may want to grant certain permissions only to applications that have been reviewed for security by a trusted third party. These third parties could provide guarantees similar to those provided by the iOS, Android, or Chrome OS app stores, which might allow browsers to offer more powerful capabilities than have been deemed safe for un audited websites.

Binary transparency for websites is similar: like with Certificate Transparency [RFC6962], the transparency logs would sign the content of the package to provide assurance that experts had a chance to audit the exact package a client received.
Associated requirements:

* Additional signatures

2.2.4. Building packages from multiple libraries

Large programs are built from smaller components. In the case of the web, components can be included either as Javascript files or as "<iframe>"d subresources. In the first case, the packager could copy the JS files to their own origin; but in the second, it may be important for the "<iframe>"d resources to be able to make same-origin (https://html.spec.whatwg.org/multipage/origin.html#same-origin) requests back to their own origin, for example to implement federated sign-in.

Associated requirements:

* Resources from multiple origins in a package: Each component may come from its own origin.

* Deduplication of diamond dependencies: If we have dependencies A->B->D and A->C->D, it’s important that a request for a D resource resolves to a single resource that both B and C can handle.

2.2.4.1. Shared libraries

In ecosystems like Electron (https://electron.atom.io/) and Node (https://nodejs.org/en/), many packages may share some common dependencies. The cost of downloading each package can be greatly reduced if the package can merely point at other dependencies to download instead of including them all inline.

Associated requirements:

* External dependencies

2.2.5. Cross-CDN Serving

When a web page has subresources from a different origin, retrieval of those subresources can be optimized if they’re transferred over the same connection as the main resource. If both origins are distributed by the same CDN, in-progress mechanisms like [I-D.ietf-httpbis-http2-secondary-certs] allow the server to use a single connection to send both resources, but if the resource and subresource don’t share a CDN or don’t use a CDN at all, existing mechanisms don’t help.
If the subresource is signed by its publisher, the main resource’s server can forward it to the client.

There are some yet-to-be-solved privacy problems if the client and server want to avoid transferring subresources that are already in the client’s cache: naively telling the server that a resource is already present is a privacy leak.

Associated requirements:

* Streamed loading: To get optimal performance, the browser should be able to start loading early parts of a resource before the distributor finishes sending the whole resource.

* Signing as an origin: To prove the content came from the original origin.

* Compress transfers

2.2.6. Pre-installed applications

Device manufacturers would like to ship their devices with some web applications pre-installed and usable even if the application is first used without an internet connection. Thereafter, the application should use the normal Service Worker update mechanism to stay up to date.

One way to accomplish this would be to pre-create a browser profile in the device’s default browser and navigate it to each of the pre-installed apps before recording the device image. However, this means end-users miss the browser’s initial setup flow and possibly that any "unique" cookies the sites set are now shared across everyone who bought the device. It also doesn’t help users who change their default browser.

If multiple browsers supported an unsigned web package format, with an option to trust it as if it were signed if it’s in a particular section of the filesystem that’s as protected as the browser’s executable, and if registering a Service Worker from a page inside a package passed the full package contents to the Service Worker’s "install" event, the device manufacturer could provide web packages for each pre-installed application that would work in the user’s chosen browser.

Associated requirements:

* Service Worker integration: To pass the package into the "install" event and from there get its contents into a "Cache".

Yasskin  Expires 15 October 2021
2.2.7. Protecting Users from a Compromised Frontend

If an attacker gains control over a frontend server, any user who visits that server while they have control can have their web app upgraded to a hostile version. On the other hand, native applications either control their own update process or delegate it to an app store, which allows them to protect users by requiring that updates are signed by a trusted key. This protection isn’t perfect --- it’s a Trust-On-First-Use mechanism that doesn’t protect users who first install the application while the attacker controls the server they get it from, and attackers can bypass it by compromising the app’s build system --- but since both of those risks also apply to web apps, it does make the attack surface for native applications smaller than for web apps.

Not all application developers should choose to require signed updates, since doing so adds the risk of losing the signing key, but having this option gives security-sensitive applications like Dashlane (https://app.dashlane.com/) an incentive to build native apps instead of web apps.

It has been difficult to add a signature requirement for web app upgrades because we haven’t had a way to sign web resources. Web Packaging is expected to provide that, so we’ll be able to consider the best way to do it.

Both HTTP Strict Transport Security (HSTS, [RFC6797]) and HTTP Public Key Pinning (HPKP, [RFC7469]) have established ways to pin assertions about a site’s security for a bounded time after a visit. We could do the same with a web app’s signing key.

Note that HPKP has been turned off in Chromium (https://groups.google.com/a/chromium.org/d/topic/blink-dev/he9tr7p3zX8/discussion) because it was difficult to use and made it too easy to "brick" a website. To reduce the chance of bricking the website, this key pinning design could require an active Service Worker before enforcing the pins. It could also avoid the need for users to take manual action to recover from a lost signing key by allowing a new key to be used if it’s been consistently for a site-chosen amount of time, instead of waiting for the whole pin to expire. However, these mitigations don’t guarantee that browsers would find the tradeoffs more acceptable than they did for HPKP.

One can think of a CDN as a potentially-compromised frontend and use this mechanism to limit the damage it can cause. However, this doesn’t make it safe to use a wholly-untrustworthy CDN because of the risk to first-time users.
Associated requirements:

* Signing without origin trust: To let a backend system vouch for the content. This would likely be augmented with origin trust by receiving the signed content over TLS.

* Streamed loading: To get optimal performance, the browser should be able to start loading early parts of a resource before the server finishes sending the whole resource.

2.2.8. Installation from a self-extracting executable

The Node and Electron communities would like to install packages using self-extracting executables. The traditional way to design a self-extracting executable is to concatenate the package to the end of the executable, have the executable look for a length at its own end, and seek backwards from there for the start of the package.

Associated requirements:

* Trailing length

2.2.9. Packages in version control

Once packages are generated, they should be stored in version control. Many popular VC systems auto-detect text files in order to "fix" their line endings. If the first bytes of a package look like text, while later bytes store binary data, VC may break the package.

Associated requirements:

* Binary

2.2.10. Subresource bundling

Text based subresources often benefit from improved compression ratios when bundled together.

At the same time, the current practice of JS and CSS bundling, by compiling everything into a single JS file, also has negative side-effects:

1. Dependent execution - in order to start executing _any_ of the bundled resources, it is required to download, parse and execute _all_ of them.

2. Loss of caching granularity - Modification of _any_ of the resources results in caching invalidation of _all_ of them.
3. Loss of module semantics - ES6 modules must be delivered as independent resources. Therefore, current bundling methods, which deliver them with other resources under a common URL, require transpilation to ES5 and result in loss of ES6 module semantics.

An on-the-fly readable packaging format, that will enable resources to maintain their own URLs while being physically delivered with other resources, can resolve the above downsides while keeping the upsides of improved compression ratios.

To improve cache granularity, the client needs to tell the server which versions of which resources are already cached, which it could do with a Service Worker or perhaps with [I-D.ietf-httpbis-cache-digest].

Associated requirements:
* Indexed by URL
* Streamed loading: To solve downside 1.
* Compress transfers: To keep the upside.
* Response headers: At least the Content-Type is needed to load JS and CSS.
* Unsigned content: Signing same-origin content wastes space.

2.2.11. Archival

Existing formats like WARC ([ISO28500]) do a good job of accurately representing the state of a web server at a particular time, but a browser can’t currently use them to give a person the experience of that website at the time it was archived. It’s not obvious to the author of this draft that a new packaging format is likely to improve on WARC, compared to, for example, implementing support for WARC in browsers, but folks who know about archiving seem interested, e.g.: https://twitter.com/anjacks0n/status/950861384266416134 (https://twitter.com/anjacks0n/status/950861384266416134).

Because of the time scales involved in archival, any signatures from the original host would likely not be trusted anymore by the time the archive is viewed, so implementations would need to sandbox the content instead of running it on the original origin.

Associated requirements:
3. Requirements

3.1. Essential

3.1.1. Indexed by URL

Resources should be keyed by URLs, matching how browsers look resources up over HTTP.

3.1.2. Request headers

Resource keys should include request headers like "accept" and "accept-language", which allows content-negotiated resources to be represented.

This would require an extension to [MHTML], which uses the "content-location" response header to encode the requested URL, but has no way to encode other request headers. MHTML also has no instructions for handling multiple resources with the same "content-location".

This also requires an extension to [ZIP]: we’d need to encode the request headers into ZIP’s filename fields.

3.1.3. Response headers

Resources should include their HTTP response headers, like "content-type", "content-encoding", "expires", "content-security-policy", etc.

This requires an extension to [ZIP]: we’d need something like [JAR]’s "META-INF" directory to hold extra metadata beyond the resource’s body.

3.1.4. Signing as an origin

Resources within a package are provably from an entity with the ability to serve HTTPS requests for those resources’ origin [RFC6454].
Note that previous attempts to sign HTTP messages ([I-D.thomson-http-content-signature], [I-D.burke-content-signature], and [I-D.cavage-http-signatures]) omit a description of how a client should use a signature to prove that a resource comes from a particular origin, and they’re probably not usable for that purpose.

This would require an extension to the [ZIP] format, similar to [JAR]’s signatures.

In any cryptographic system, the specification is responsible to make correct implementations easier to deploy than incorrect implementations (Section 3.1.12).

3.1.5. Random access

When a package is stored on disk, the browser can access arbitrary resources without a linear scan.

[MHTML] would need to be extended with an index of the byte offsets of each contained resource.

3.1.6. Resources from multiple origins in a package

A package from origin "A" can contain resources from origin "B" authenticated at the same level as those from "A".

3.1.7. Cryptographic agility

Obsolete cryptographic algorithms can be replaced.

Planning to upgrade the cryptography also means we should include some way to know when it’s safe to remove old cryptography (Section 3.2.6).

3.1.8. Unsigned content

Alex can create their own package without a CA-signed certificate, and Bailey can view the content of the package.

3.1.9. Certificate revocation

When a package is signed by a revoked certificate, online browsers can detect this reasonably quickly.

3.1.10. Downgrade prevention

Attackers can’t cause a browser to trust an older, vulnerable version of a package after the browser has seen a newer version.
3.1.11. Metadata

Metadata like that found in the W3C’s Application Manifest [W3C.WD-appmanifest-20170828] can help a client know how to load and display a package.

3.1.12. Implementations are hard to get wrong

The design should incorporate aspects that tend to cause incorrect implementations to get noticed quickly, and avoid aspects that are easy to implement incorrectly. For example:

* Explicitly specifying a cryptographic algorithm identifier in [RFC7515] made it easy for implementations to trust that algorithm, which caused vulnerabilities (https://paragonie.com/blog/2017/03/jwt-json-web-tokens-is-bad-standard-that-everyone-should-avoid).

* [ZIP]’s duplicate specification of filenames makes it easy for implementations to check the signature of one copy but use the other (https://nakedsecurity.sophos.com/2013/07/10/anatomy-of-a-security-hole-googles-android-master-key-debacle-explained/).


3.2. Nice to have

3.2.1. Streamed loading

The browser can load a package as it downloads.

This conflicts with ZIP, since ZIP’s index is at the end.

3.2.2. Signing without origin trust

It’s possible to sign a resource with a key that has some effect on trust other than asserting that the origin’s owner vouches for it. These keys could be expressed as raw public keys or as certificates with other key usages.

3.2.3. Additional signatures

Third-parties can vouch for packages by signing them.
3.2.4. Binary

The format is identified as binary by tools that might try to "fix" line endings.

This conflicts with using an [MHTML]-based format.

3.2.5. Deduplication of diamond dependencies

Nested packages that have multiple dependency routes to the same sub-package, can be transmitted and stored with only one copy of that sub-package.

3.2.6. Old crypto can be removed

The ecosystem can identify when an obsolete cryptographic algorithm is no longer needed and can be removed.

3.2.7. Compress transfers

Transferring a package over the network takes as few bytes as possible. This is an easier problem than Compress stored packages since it doesn’t have to preserve Random access.

3.2.8. Compress stored packages

Storing a package on disk takes as few bytes as possible.

3.2.9. Subsetting and reordering

Resources can be removed from and reordered within a package, without breaking signatures (Section 3.1.4).

3.2.10. Packaged validity information

Certificate revocation and Downgrade prevention information can itself be packaged or included in other packages.

3.2.11. Signing uses existing TLS certificates

A "normal" TLS certificate can be used for signing packages. Avoiding extra requirements like "code signing" certificates makes packaging more accessible to all sites.
3.2.12. External dependencies

Sub-packages can be "external" to the main package, meaning the browser will need to either fetch them separately or already have them. (#35, App Installer Story (https://github.com/WICG/webpackage/issues/35))

3.2.13. Trailing length

The package’s length in bytes appears a fixed offset from the end of the package.

This conflicts with [MHTML].

3.2.14. Time-shifting execution

In some unsigned packages, Javascript time-telling functions should return the timestamp of the package, rather than the true current time.

We should explore if this has security implications.

3.2.15. Service Worker integration

When a web page inside a package registers a Service Worker, that Service Worker’s "install" event should receive a reference to the full package, with a way to copy the package’s contents into a "Cache" object. ([ServiceWorkers])

4. Non-goals

Some features often come along with packaging and signing, and it’s important to explicitly note that they don’t appear in the list of Requirements.

4.1. Store confidential data

Packages are designed to hold public information and to be shared to people with whom the original publisher never has an interactive connection. In that situation, there’s no way to keep the contents confidential: even if they were encrypted, to make the data public, anyone would have to be able to get the decryption key.

It’s possible to maintain something similar to confidentiality for non-public packaged data, but doing so complicates the format design and can give users a false sense of security.
We believe we’ll cause fewer privacy breaches if we omit any mechanism for encrypting data, than if we include something and try to teach people when it’s unsafe to use.

4.2. Generate packages on the fly

See discussion at WICG/webpackage#6 (https://github.com/WICG/webpackage/issues/6#issuecomment-275746125).

4.3. Non-origin identity

A package can be primarily identified as coming from something other than a Web Origin (https://html.spec.whatwg.org/multipage/browsers.html#concept-origin).

4.4. DRM

Special support for blocking access to downloaded content based on licensing. Note that DRM systems can be shipped inside the package even if the packaging format doesn’t specifically support them.

4.5. Ergonomic replacement for HTTP/2 PUSH

HTTP/2 PUSH ([RFC7540], section 8.2) is hard for developers to configure, and an explicit package format might be easier. However, experts in this area believe we should focus on improving PUSH directly instead of routing around it with a bundling format.

Trying to bundle resources in order to speed up page loads has a long history, including Resource Packages (https://www.mnot.net/blog/2010/02/18/resource_packages) from 2010 and the W3C TAG’s packaging proposal (https://w3ctag.github.io/packaging-on-the-web/) from 2015.

However, the HTTPWG is doing a lot of work to let servers optimize the PUSHed data, and packaging would either have to re-do that or accept lower performance. For example:

* [I-D.vkrasnov-h2-compression-dictionaries] should allow individual small resources to be compressed as well as they would be in a bundle.

* [I-D.ietf-httpbis-cache-digest] tells the server which resources it doesn’t need to PUSH.

Associated requirements:
* Streamed loading: If the whole package has to be downloaded before the browser can load a piece, this will definitely be slower than PUSH.

* Compress transfers: Keep up with [I-D.vkrasnov-h2-compression-dictionaries].

* Indexed by URL: Resources on the web are addressed by URL.

* Request headers: PUSH_PROMISE (http://httpwg.org/specs/rfc7540.html#PUSH_PROMISE) ([RFC7540], section 6.6) includes request headers.

* Response headers: PUSHed resources include their response headers.

5. Security Considerations

The security considerations will depend on the solution designed to satisfy the above requirements. See [I-D.yasskin-dispatch-web-packaging] for one possible set of security considerations.

6. IANA Considerations

This document has no actions for IANA.

7. Informative References

[I-D.burke-content-signature]

[I-D.cavage-http-signatures]

[I-D.ietf-httpbis-cache-digest]
[I-D.ietf-httpbis-http2-secondary-certs]

[I-D.thomson-http-content-signature]

[I-D.vkrasnov-h2-compression-dictionaries]

[I-D.yasskin-dispatch-web-packaging]


Appendix A. Acknowledgements

Thanks to Yoav Weiss for the Subresource bundling use case and discussions about content distributors.

Author’s Address

Jeffrey Yasskin
Google

Email: jyasskin@chromium.org