Workgroup: HTTPAPI
Internet-Draft:
draft-ietf-httpapi-rfc7807bis-04
Obsoletes: 7807 (if approved)
Published: 5 September 2022
Intended Status: Standards Track
Expires: 9 March 2023
Authors: M. Nottingham E. Wilde S. Dalal

Problem Details for HTTP APIs

Abstract

This document defines a "problem detail" to carry machine-readable details of errors in HTTP response content and/or fields to avoid the need to define new error response formats for HTTP APIs.

This document obsoletes RF7807.

Discussion Venues

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

Source for this draft and an issue tracker can be found at https://github.com/ietf-wg-httpapi/rfc7807bis.

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 9 March 2023.

Copyright Notice

Table of Contents

1. Introduction

HTTP status codes (Section 15 of [HTTP]) cannot always convey enough information about errors to be helpful. While humans using Web browsers can often understand an HTML [HTML5] response content, non-human consumers of HTTP APIs have difficulty doing so.

To address that shortcoming, this specification defines simple JSON [JSON] and XML [XML] document formats and a HTTP field to describe the specifics of problem(s) encountered -- "problem details".

For example, consider a response indicating that the client's account doesn't have enough credit. The API's designer might decide to use the 403 Forbidden status code to inform HTTP-generic software (such as client libraries, caches, and proxies) of the response's general semantics. API-specific problem details (such as the why the server refused the request and the applicable account balance) can be carried in the response content, so that the client can act upon them appropriately (for example, triggering a transfer of more credit into the account).

This specification identifies the specific "problem type" (e.g., "out of credit") with a URI [URI]. HTTP APIs can use URIs under their control to identify problems specific to them, or can reuse existing ones to facilitate interoperability and leverage common semantics (see Section 5.2).

Problem details can contain other information, such as a URI identifying the problem's specific occurrence (effectively giving an identifier to the concept "The time Joe didn't have enough credit last Thursday"), which can be useful for support or forensic purposes.

The data model for problem details is a JSON [JSON] object; when serialized as a JSON document, it uses the "application/problem+json" media type. Appendix B defines an equivalent XML format, which uses the "application/problem+xml" media type.

Note that problem details are (naturally) not the only way to convey the details of a problem in HTTP. If the response is still a representation of a resource, for example, it's often preferable to describe the relevant details in that application's format. Likewise, defined HTTP status codes cover many situations with no need to convey extra detail.

This specification's aim is to define common error formats for applications that need one so that they aren't required to define their own, or worse, tempted to redefine the semantics of existing HTTP status codes. Even if an application chooses not to use it to convey errors, reviewing its design can help guide the design decisions faced when conveying errors in an existing format.

See Appendix D for a list of changes from RFC7807.

2. Notational 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 document uses the following terminology from [STRUCTURED-FIELDS] to specify syntax and parsing: Dictionary, String, and Integer.

3. The Problem Details JSON Object

The canonical model for problem details is a JSON [JSON] object. When serialized in a JSON document, that format is identified with the "application/problem+json" media type.

For example:

```
POST /purchase HTTP/1.1
Host: store.example.com
Content-Type: application/json
Accept: application/json, application/problem+json

{
  "item": 123456,
  "quantity": 2
}
```

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+json
Content-Language: en

{
 "type": "https://example.com/probs/out-of-credit",
 "title": "You do not have enough credit.",
 "detail": "Your current balance is 30, but that costs 50.",
 "instance": "/account/12345/msgs/abc",
 "balance": 30,
 "accounts": ["/account/12345",
              "/account/67890"]
}
```

Here, the out-of-credit problem (identified by its type) indicates the reason for the 403 in "title", identifies the specific problem occurrence with "instance", gives occurrence-specific details in "detail", and adds two extensions: "balance" conveys the account's balance, and "accounts" lists links where the account can be topped up.

When designed to accommodate it, problem-specific extensions can convey more than one instance of the same problem type. For example:

```
POST /details HTTP/1.1
Host: account.example.com
Accept: application/json

{
  "age": 42.3,
  "profile": {
    "color": "yellow"
  }
}
```

```
HTTP/1.1 400 Bad Request
Content-Type: application/problem+json
Content-Language: en

{
 "type": "https://example.net/validation-error",
 "title": "Your request is not valid.",
 "errors": [
             {
               "detail": "must be a positive integer",
               "pointer": "#/age"
             },
             {
               "detail": "must be 'green', 'red' or 'blue'",
               "pointer": "#/profile/color"
             }
          ]
  }
```

The fictional problem type here defines the "errors" extension, an array that describes the details of each validation error. Each member is an object containing "detail" to describe the issue, and "pointer" to locate the problem within the request's content using a JSON Pointer [JSON-POINTER].

When an API encounters multiple problems that do not share the same type, it is RECOMMENDED that the most relevant or urgent problem be represented in the response. While it is possible to create generic "batch" problem types that convey multiple, disparate types, they do not map well into HTTP semantics.

Note also that the API has responded with the application/problem+json type, even though the client did not list it in Accept, as is allowed by HTTP (see Section 12.5.1 of [HTTP]).

3.1. Members of a Problem Details Object

Problem detail objects can have the following members. If a member's value type does not match the specified type, the member MUST be ignored -- i.e., processing will continue as if the member had not been present.

3.1.1. "type"

The "type" member is a JSON string containing a URI reference [URI] that identifies the problem type. Consumers MUST use the "type" URI (after resolution, if necessary) as the problem type's primary identifier.

When this member is not present, its value is assumed to be "about:blank".

If the type URI is a locator (e.g., those with a "http" or "https" scheme), dereferencing it SHOULD provide human-readable documentation for the problem type (e.g., using HTML [HTML5]). However, consumers SHOULD NOT automatically dereference the type URI, unless they do so when providing information to developers (e.g., when a debugging tool is in use).

When "type" contains a relative URI, it is resolved relative to the document's base URI, as per [URI], Section 5. However, using relative URIs can cause confusion, and they might not be handled correctly by all implementations.

For example, if the two resources "https://api.example.org/foo/bar/123" and "https://api.example.org/widget/456" both respond with a "type" equal to the relative URI reference "example-problem", when resolved they will identify different resources ("https://api.example.org/foo/bar/example-problem" and "https://api.example.org/widget/example-problem" respectively). As a result, it is RECOMMENDED that absolute URIs be used in "type" when possible, and that when relative URIs are used, they include the full path (e.g., "/types/123").

The type URI can also be a non-resolvable URI. For example, the tag URI scheme [TAG] can be used to uniquely identify problem types:

```
tag:mnot@mnot.net,2021-09-17:OutOfLuck
```

Non-resolvable URIs ought not be used when there is some future possibility that it might become desirable to do so. For example, if an API designer used the URI above and later adopted a tool that resolves type URIs to discover information about the error, taking advantage of that capability would require switching to a resolvable

URI, creating a new identity for the problem type and thus introducing a breaking change.

3.1.2. "status"

The "status" member is a JSON number indicating the HTTP status code ([HTTP], Section 15) generated by the origin server for this occurrence of the problem.

The "status" member, if present, is only advisory; it conveys the HTTP status code used for the convenience of the consumer. Generators MUST use the same status code in the actual HTTP response, to assure that generic HTTP software that does not understand this format still behaves correctly. See Section 6 for further caveats regarding its use.

Consumers can use the status member to determine what the original status code used by the generator was when it has been changed (e.g., by an intermediary or cache), and when a message's content is persisted without HTTP information. Generic HTTP software will still use the HTTP status code.

3.1.3. "title"

The "title" member is a JSON string containing a short, human-readable summary of the problem type.

It SHOULD NOT change from occurrence to occurrence of the problem, except for localization (e.g., using proactive content negotiation; see [HTTP], Section 12.1).

The "title" string is advisory, and is included only for users who both are not aware of and cannot discover the semantics of the type URI (e.g., during offline log analysis).

3.1.4. "detail"

The "detail" member is a JSON string containing a human-readable explanation specific to this occurrence of the problem.

The "detail" string, if present, ought to focus on helping the client correct the problem, rather than giving debugging information.

Consumers SHOULD NOT parse the "detail" member for information; extensions are more suitable and less error-prone ways to obtain such information.

3.1.5. "instance"

The "instance" member is a JSON string containing a URI reference that identifies the specific occurrence of the problem.

When the "instance" URI is dereferenceable, the problem details object can be fetched from it. It might also return information about the problem occurrence in other formats through use of proactive content negotiation (see [HTTP], Section 12.5.1).

When the "instance" URI is not dereferenceable, it serves as a unique identifier for the problem occurrence that may be of significance to the server, but is opaque to the client.

When "instance" contains a relative URI, it is resolved relative to the document's base URI, as per [URI], Section 5. However, using relative URIs can cause confusion, and they might not be handled correctly by all implementations.

For example, if the two resources "https://api.example.org/foo/bar/123" and "https://api.example.org/widget/456" both respond with an "instance" equal to the relative URI reference "example-instance", when resolved they will identify different resources ("https://api.example.org/foo/bar/example-instance" and "https://api.example.org/widget/example-instance" respectively). As a result, it is RECOMMENDED that absolute URIs be used in "instance" when possible, and that when relative URIs are used, they include the full path (e.g., "/instances/123").

3.2. Extension Members

Problem type definitions MAY extend the problem details object with additional members that are specific to that problem type.

For example, our "out of credit" problem above defines two such extensions -- "balance" and "accounts" to convey additional, problem-specific information.

Similarly, the "validation error" example defines a "errors" extension that contains a list of individual error occurrences found, with details and a pointer to the location of each.

Clients consuming problem details MUST ignore any such extensions that they don't recognize; this allows problem types to evolve and include additional information in the future.

Future updates to this specification might define additional members that are available to all problem types, distinguished by a name starting with "*". To avoid conflicts, extension member names SHOULD NOT start with the "*" character.

When creating extensions, problem type authors should choose their names carefully. To be used in the XML format (see Appendix B), they will need to conform to the Name rule in Section 2.3 of [XML]. To be used in the HTTP field (see Section 4), they will need to conform to the Dictionary key syntax defined in Section 3.2 of [STRUCTURED-FIELDS].

Problem type authors that wish their extensions to be usable in the Problem HTTP field (see Section 4) will also need to define the Structured Type(s) that their values are mapped to.

4. The Problem HTTP Field

Some problems might best be conveyed in a HTTP header or trailer field, rather than in the message content. For example, when a problem does not prevent a successful response from being generated, or when the problem's details are useful to software that does not inspect the response content.

The Problem HTTP field allows a limited expression of a problem object in HTTP headers or trailers. It is a Dictionary Structured Field (Section 3.2 of [STRUCTURED-FIELDS]) that can contain the following keys, whose semantics and related requirements are inherited from problem objects:

type: the type value (see Section 3.1.1), as a String

status: the status value (see Section 3.1.2), as an Integer

title: The title value (see Section 3.1.3), as a String

detail: The detail value (see Section 3.1.4), as a String

instance: The instance value (see Section 3.1.5), as a String

The title and detail values MUST NOT be serialized in the Problem field if they contain characters that are not allowed by String; see Section 3.3.3 of [STRUCTURED-FIELDS]. Practically, this has the effect of limiting them to ASCII strings.

An extension member (see Section 3.2) MAY occur in the Problem field if its name is compatible with the syntax of Dictionary keys (see Section 3.2 of [STRUCTURED-FIELDS]) and if the defining problem type specifies a Structured Type to serialize the value into.

For example:

```
HTTP/1.1 200 OK
Content-Type: application/json
Problem: type="https://example.net/problems/almost-out",
   title="you're almost out of credit", credit_left=20
```

5. Defining New Problem Types

When an HTTP API needs to define a response that indicates an error condition, it might be appropriate to do so by defining a new problem type.

Before doing so, it's important to understand what they are good for, and what's better left to other mechanisms.

Problem details are not a debugging tool for the underlying implementation; rather, they are a way to expose greater detail about the HTTP interface itself. Designers of new problem types need to carefully take into account the Security Considerations (Section 6), in particular, the risk of exposing attack vectors by exposing implementation internals through error messages.

Likewise, truly generic problems -- i.e., conditions that might apply to any resource on the Web -- are usually better expressed as plain status codes. For example, a "write access disallowed" problem is probably unnecessary, since a 403 Forbidden status code in response to a PUT request is self-explanatory.

Finally, an application might have a more appropriate way to carry an error in a format that it already defines. Problem details are intended to avoid the necessity of establishing new "fault" or "error" document formats, not to replace existing domain-specific formats.

That said, it is possible to add support for problem details to existing HTTP APIs using HTTP content negotiation (e.g., using the Accept request header to indicate a preference for this format; see [HTTP], Section 12.5.1).

New problem type definitions MUST document:

1. a type URI (typically, with the "http" or "https" scheme),

2. a title that appropriately describes it (think short), and

3. the HTTP status code for it to be used with.

Problem type definitions MAY specify the use of the Retry-After response header ([HTTP], Section 10.2.3) in appropriate circumstances.

A problem's type URI SHOULD resolve to HTML [HTML5] documentation that explains how to resolve the problem.

A problem type definition MAY specify additional members on the problem details object. For example, an extension might use typed links [WEB-LINKING] to another resource that machines can use to resolve the problem.

If such additional members are defined, their names SHOULD start with a letter (ALPHA, as per [ABNF], Appendix B.1) and SHOULD comprise characters from ALPHA, DIGIT ([ABNF], Appendix B.1), and "_" (so that it can be serialized in formats other than JSON), and they SHOULD be three characters or longer.

5.1. Example

For example, if you are publishing an HTTP API to your online shopping cart, you might need to indicate that the user is out of credit (our example from above), and therefore cannot make the purchase.

If you already have an application-specific format that can accommodate this information, it's probably best to do that. However, if you don't, you might use one of the problem details formats -- JSON if your API is JSON-based, or XML if it uses that format.

To do so, you might look in the registry (Section 5.2) for an already-defined type URI that suits your purposes. If one is available, you can reuse that URI.

If one isn't available, you could mint and document a new type URI (which ought to be under your control and stable over time), an appropriate title and the HTTP status code that it will be used with, along with what it means and how it should be handled.

5.2. Registered Problem Types

This specification defines the HTTP Problem Type registry for common, widely-used problem type URIs, to promote reuse.

The policy for this registry is Specification Required, per [RFC8126], Section 4.5.

When evaluating requests, the Expert(s) should consider community feedback, how well-defined the problem type is, and this

specification's requirements. Vendor-specific, application-specific, and deployment-specific values are not registrable. Specification documents should be published in a stable, freely available manner (ideally located with a URL), but need not be standards.

Registrations MAY use the prefix "https://iana.org/assignments/http-problem-types#" for the type URI.

Registration requests should use the following template:

- Type URI: [a URI for the problem type]

- Title: [a short description of the problem type]

- Recommended HTTP status code: [what status code is most appropriate to use with the type]

- Reference: [to a specification defining the type]

See the registry at https://iana.org/assignments/http-problem-types for details on where to send registration requests.

5.2.1. about:blank

This specification registers one Problem Type, "about:blank".

- Type URI: about:blank

- Title: See HTTP Status Code

- Recommended HTTP status code: N/A

- Reference: [this document]

The "about:blank" URI [ABOUT], when used as a problem type, indicates that the problem has no additional semantics beyond that of the HTTP status code.

When "about:blank" is used, the title SHOULD be the same as the recommended HTTP status phrase for that code (e.g., "Not Found" for 404, and so on), although it MAY be localized to suit client preferences (expressed with the Accept-Language request header).

Please note that according to how the "type" member is defined (Section 3.1), the "about:blank" URI is the default value for that member. Consequently, any problem details object not carrying an explicit "type" member implicitly uses this URI.

6. Security Considerations

When defining a new problem type, the information included must be carefully vetted. Likewise, when actually generating a problem -- however it is serialized -- the details given must also be scrutinized.

Risks include leaking information that can be exploited to compromise the system, access to the system, or the privacy of users of the system.

Generators providing links to occurrence information are encouraged to avoid making implementation details such as a stack dump available through the HTTP interface, since this can expose sensitive details of the server implementation, its data, and so on.

The "status" member duplicates the information available in the HTTP status code itself, bringing the possibility of disagreement between the two. Their relative precedence is not clear, since a disagreement might indicate that (for example) an intermediary has changed the HTTP status code in transit (e.g., by a proxy or cache). Generic HTTP software (such as proxies, load balancers, firewalls, and virus scanners) are unlikely to know of or respect the status code conveyed in this member.

7. IANA Considerations

Please update the "application/problem+json" and "application/problem+xml" registrations in the "Media Types" registry to refer to this document.

Please create the "HTTP Problem Types Registry" as specified in Section 5.2, and populate it with "about:blank" as per Section 5.2.1.

Please register the following entry into the "Hypertext Transfer Protocol (HTTP) Field Name Registry":

Field Name: Problem

Status: Permanent

Reference: RFC nnnn

8. References

8.1. Normative References

[ABNF] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI

10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/rfc/rfc5234>.

[HTTP] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/rfc/rfc9110>.

[JSON] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/rfc/rfc8259>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[STRUCTURED-FIELDS] Nottingham, M. and P-H. Kamp, "Structured Field Values for HTTP", RFC 8941, DOI 10.17487/RFC8941, February 2021, <https://www.rfc-editor.org/rfc/rfc8941>.

[URI] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/rfc/rfc3986>.

[XML] Maler, E., Ed., Yergeau, F., Ed., Paoli, J., Ed., Sperberg-McQueen, M., Ed., and T. Bray, Ed., "Extensible Markup Language (XML) 1.0 (Fifth Edition)", W3C REC REC-xml-20081126, W3C REC-xml-20081126, 26 November 2008, <https://www.w3.org/TR/2008/REC-xml-20081126/>.

8.2. Informative References

[ABOUT] Moonesamy, S., Ed., "The "about" URI Scheme", RFC 6694, DOI 10.17487/RFC6694, August 2012, <https://www.rfc-editor.org/rfc/rfc6694>.

[HTML5] WHATWG, "HTML - Living Standard", n.d., <https://html.spec.whatwg.org>.

[ISO-19757-2]
International Organization for Standardization, "Information Technology -- Document Schema Definition Languages (DSDL) -- Part 2: Grammar-based Validation -- RELAX NG", ISO/IEC 19757-2, 2003.

[JSON-POINTER] Bryan, P., Ed., Zyp, K., and M. Nottingham, Ed., "JavaScript Object Notation (JSON) Pointer", RFC 6901, DOI 10.17487/RFC6901, April 2013, <https://www.rfc-editor.org/rfc/rfc6901>.

[JSON-SCHEMA] Wright, A., Andrews, H., Hutton, B., and G. Dennis, "JSON Schema: A Media Type for Describing JSON Documents", Work in Progress, Internet-Draft, draft-bhutton-json-schema-01, 10 June 2022, <https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-01>.

[RDFA] Adida, B., Ed., Herman, I., Ed., Birbeck, M., Ed., and S. McCarron, Ed., "RDFa Core 1.1 - Third Edition", W3C REC REC-rdfa-core-20150317, W3C REC-rdfa-core-20150317, 17 March 2015, <https://www.w3.org/TR/2015/REC-rdfa-core-20150317/>.

[TAG] Kindberg, T. and S. Hawke, "The 'tag' URI Scheme", RFC 4151, DOI 10.17487/RFC4151, October 2005, <https://www.rfc-editor.org/rfc/rfc4151>.

[WEB-LINKING] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <https://www.rfc-editor.org/rfc/rfc8288>.

[XSLT] Thompson, H., Ed., Clark, J., Ed., and S. Pieters, Ed., "Associating Style Sheets with XML documents 1.0 (Second Edition)", W3C REC REC-xml-stylesheet-20101028, W3C REC-xml-stylesheet-20101028, 28 October 2010, <https://www.w3.org/TR/2010/REC-xml-stylesheet-20101028/>.

Appendix A. JSON Schema for HTTP Problems

This section presents a non-normative JSON Schema [JSON-SCHEMA] for HTTP Problem Details. If there is any disagreement between it and the text of the specification, the latter prevails.

```
# NOTE: '\' line wrapping per RFC 8792
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "A problem object RFC 7807bis",
  "type": "object",
  "properties": {
    "type": {
      "type": "string",
      "format": "uri-reference",
      "description": "A URI reference RFC3986 that identifies the \
problem type."
    },
    "title": {
      "type": "string",
      "description": "A short, human-readable summary of the \
problem type. It SHOULD NOT change from occurrence to occurrence \
of the problem, except for purposes of localization (e.g., using \
proactive content negotiation; see RFC7231, Section 3.4)"
    },
    "status": {
      "type": "integer",
      "description": "The HTTP status code (RFC7231, Section 6) \
generated by the origin server for this occurrence of the problem.",
      "minimum": 100,
      "maximum": 599
    },
    "detail": {
      "type": "string",
      "description": "A human-readable explanation specific to \
this occurrence of the problem."
    },
    "instance": {
      "type": "string",
      "format": "uri-reference",
      "description": "A URI reference that identifies the \
specific occurrence of the problem. It may or may not yield \
further information if dereferenced."
    }
  }
}
```

Appendix B. HTTP Problems and XML

HTTP-based APIs that use XML [XML] can express problem details using the format defined in this appendix.

The RELAX NG schema [ISO-19757-2] for the XML format is:

```
   default namespace ns = "urn:ietf:rfc:7807"

   start = problem

   problem =
     element problem {
       (  element  type           { xsd:anyURI }?
        & element  title          { xsd:string }?
        & element  detail         { xsd:string }?
        & element  status         { xsd:positiveInteger }?
        & element  instance       { xsd:anyURI }? ),
       anyNsElement
     }

   anyNsElement =
     (  element    ns:*  { anyNsElement | text }
      | attribute  *     { text })*
```

Note that this schema is only intended as documentation, and not as a normative schema that captures all constraints of the XML format. It is possible to use other XML schema languages to define a similar set of constraints (depending on the features of the chosen schema language).

The media type for this format is "application/problem+xml".

Extension arrays and objects are serialized into the XML format by considering an element containing a child or children to represent an object, except for elements that contain only child element(s) named 'i', which are considered arrays. For example, the example above appears in XML as follows:

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+xml
Content-Language: en

<?xml version="1.0" encoding="UTF-8"?>
<problem xmlns="urn:ietf:rfc:7807">
  <type>https://example.com/probs/out-of-credit</type>
  <title>You do not have enough credit.</title>
  <detail>Your current balance is 30, but that costs 50.</detail>
  <instance>https://example.net/account/12345/msgs/abc</instance>
  <balance>30</balance>
  <accounts>
    <i>https://example.net/account/12345</i>
    <i>https://example.net/account/67890</i>
  </accounts>
</problem>
```

This format uses an XML namespace, primarily to allow embedding it into other XML-based formats; it does not imply that it can or should be extended with elements or attributes in other namespaces. The RELAX NG schema explicitly only allows elements from the one namespace used in the XML format. Any extension arrays and objects MUST be serialized into XML markup using only that namespace.

When using the XML format, it is possible to embed an XML processing instruction in the XML that instructs clients to transform the XML, using the referenced XSLT code [XSLT]. If this code is transforming the XML into (X)HTML, then it is possible to serve the XML format, and yet have clients capable of performing the transformation display human-friendly (X)HTML that is rendered and displayed at the client. Note that when using this method, it is advisable to use XSLT 1.0 in order to maximize the number of clients capable of executing the XSLT code.

Appendix C. Using Problem Details with Other Formats

In some situations, it can be advantageous to embed problem details in formats other than those described here. For example, an API that uses HTML [HTML5] might want to also use HTML for expressing its problem details.

Problem details can be embedded in other formats either by encapsulating one of the existing serializations (JSON or XML) into that format or by translating the model of a problem detail (as specified in Section 3) into the format's conventions.

For example, in HTML, a problem could be embedded by encapsulating JSON in a script tag:

```
<script type="application/problem+json">
  {
   "type": "https://example.com/probs/out-of-credit",
   "title": "You do not have enough credit.",
   "detail": "Your current balance is 30, but that costs 50.",
   "instance": "/account/12345/msgs/abc",
   "balance": 30,
   "accounts": ["/account/12345",
                "/account/67890"]
  }
</script>
```

or by inventing a mapping into RDFa [RDFA].

This specification does not make specific recommendations regarding embedding problem details in other formats; the appropriate way to embed them depends both upon the format in use and application of that format.

Appendix D. Changes from RFC7807

This revision has made the following changes:

- Section 5.2 introduces a registry of common problem type URIs
- Section 3 clarifies how multiple problems should be treated
- Section 3.2 reserves a prefix for future standards-defined object members
- Section 3.1.1 provides guidance for using type URIs that cannot be dereferenced
- Section 4 allows problem details to be communicate in a HTTP header or trailer field

Acknowledgements

The authors would like to thank Jan Algermissen, Subbu Allamaraju, Mike Amundsen, Roy Fielding, Eran Hammer, Sam Johnston, Mike McCall, Julian Reschke, and James Snell for review of this specification.

Authors' Addresses

Mark Nottingham
Prahran
Australia

Email: mnot@mnot.net
URI: https://www.mnot.net/

Erik Wilde

Email: erik.wilde@dret.net
URI: http://dret.net/netdret/

Sanjay Dalal
United States of America

Email: sanjay.dalal@cal.berkeley.edu
URI: https://github.com/sdatspun2