The Use of Registries
draft-wilde-registries-03

Abstract

Registries for Internet and Web protocols fulfill a wide range of tasks, ranging from low-level networking aspects such as packet type identifiers, all the way up to application-level protocols and standards. This document summarizes some of the reasons of why, when, and how to use registries. It serves as an informative reference for specification writers considering whether to create and manage a registry, allowing them to better understand some of the issues associated with certain design and operational decisions.

Note to Readers

Please discuss this draft on the ART mailing list (<https://www.ietf.org/mailman/listinfo/art>).

Online access to all versions and files is available on GitHub (<https://github.com/dret/I-D/tree/master/registries>).

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 January 31, 2020.
Copyright Notice

Copyright (c) 2019 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
Provisions Relating to IETF Documents
(https://trustee.ietf.org/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
2. Examples ............................................... 4
   2.1. TCP/UDP Port Numbers ............................... 4
   2.2. Language Tags ...................................... 5
   2.3. Web Linking ........................................ 5
   2.4. Domain Name System (DNS) .......................... 6
   2.5. non-IANA/DNS example ............................ 6
3. Why use Registries ....................................... 6
   3.1. Openness and Extensibility ......................... 7
   3.2. Limited Namespaces ................................ 7
   3.3. Design/Usage Review ............................... 8
   3.4. Identifier Design .................................. 8
   3.5. Identifier Lifecycle ............................... 9
   3.6. Documentation Requirements ....................... 9
   3.7. Centralized Lookup ............................... 10
4. When to use Registries ................................... 10
5. Barrier to Entry Issues .................................. 11
   5.1. Non-Semantic/Reserved Entries ...................... 11
   5.2. Entry Levels ..................................... 12
   5.3. Separation by Value Syntax ....................... 12
6. How to use Registries .................................... 13
   6.1. Registry Operations ............................... 14
   6.2. Registry Creation .................................. 15
   6.3. Registry Interaction .............................. 15
   6.4. Implementation Support ........................... 16
   6.5. Registry Stability ................................ 16
   6.6. Registry History .................................. 16
   6.7. Registry Access .................................. 17
   6.8. Runtime vs. Design-Time ......................... 17
7. IANA Considerations ..................................... 18
8. References .............................................. 18
1. Introduction

Specifications for technologies and standards in computer networking have long used the concept of a "Registry" as a place where well-known information is available and managed. In most cases, the main reason to use a registry is to create a specification that has stable parts (the specification itself), as well as some parts of it that are supposed to evolve while the specification itself remains stable and unchanged.

In essence, a registry is a pattern of how to separate those two aspects of a specification, allowing the specification to remain stable, while the parts of it managed in the registry can evolve over time by updating the registry contents. For specification writers, this has proven to be a useful and successful pattern. The "Protocol Registries" maintained by the "Internet Assigned Numbers Authority (IANA)" have steadily increased in number. At the time of writing (early 2016), the IANA Protocol Registry [IANA-Protocol-Registry] contains 1903 individual registries. This number indicates that registries as a "protocol specification pattern" are quite popular and successful.

Deciding whether a specification should use a registry is not an easy task. It involves identifying those parts that should be kept stable (in the specification itself), and those that should be managed in one or more registries for ongoing management and evolution. Even after identifying this split, it is necessary to define how exactly the registry part(s) should be managed, involving questions such as submission procedures, review processes, revocation/change management, and access to the registry contents for the worldwide developer community.

This document is intended to provide an overview to specification developers in terms of why, when, and how to use registries. It is not meant to provide definitive guidance, but mostly intended as a reference to consider the different ways in which the general "registry pattern" can be used, and what the possible side-effects of some of these solutions may be.

This document also has to tackle a difficult question: Registries as a pattern can be managed and hosted in a large variety of ways. On the other hand, the IANA Protocol Registry mentioned above provides a simple and hosted place for specifications to run their registries. As all existing platforms, the IANA Protocol Registry comes with
certain constraints in terms of how a registry can be defined, managed, and run. This document acknowledges the existence and popularity of the IANA Protocol Registry, but also tries to be open enough so that if specification writer choose to do so for whatever reason, they should also be able to use registries other than one in the IANA Protocol Registry.

2. Examples

The following list of examples is intended to be illustrative of some of the existing registries, what kind of identifiers they are using, and how they are managed. This list is not intended to highlight these registries in any special way other than explaining some of the specifics of the managed namespaces. As mentioned above, the list of IANA-managed registries is long (around 2000 individual registries), and the examples listed here have been selected somewhat randomly just to illustrate certain points.

In order to highlight the fact that IANA-managed registries are only one way of how to establish and run a registry, Section 2.4 briefly talks about DNS as an example for the same basic function (providing a managed namespace of identifiers), but one that is based on different constraints and thus results in a different operational model and implementation.

2.1. TCP/UDP Port Numbers

The registry for TCP/UDP port numbers [RFC6335] is one of the oldest well-known registries on the Internet. Because of its core importance for how the Internet functions, it has been around for a long time, there is a long history about managing and running it, and thus the most up-to-date document about it is relatively new (RFC 6335 [RFC6335] from August 2011).

The namespace of ports is limited. Port numbers in TCP and UDP are 16-bit numbers, yielding a namespace of 65,536 port numbers. The port numbers are subdivided into three ranges of system ports (0-1023), user ports (1024-49151), and dynamic ports (49152-65535). RFC 6335 only treats system and user port numbers as assignable, whereas dynamic port numbers cannot be assigned at all.

The TCP/UDP port number registry is a good example for a limited and popular namespace, and thus managing this registry follows a disciplined review process. RFC 6335 [RFC6335] defines specific guidance about assignment, de-assignment, reuse, revocation, and transfer of numbers. This level of detail may not be required for all registries, but it is a good demonstration of what may be necessary in case of constrained and popular namespaces.
2.2. Language Tags

The ability to identify human languages is important for many scenarios and applications on the Internet, and thus RFC 5646 [RFC5646] defines how to do this. The specification uses a registry to manage the actual language identifiers, because this list is constantly evolving and thus better separated from the specification defining the language tag format itself.

Apart from the primary language subtag (identifying for example the English language with the identifier "en"), the language tag format supports additional subtags for extended languages, scripts, regions, variants, extensions, and private use. The primary language subtag uses 2- and 3-letter identifiers that are taken from ISO 639 [ISO.639.1988]. There is a provision for longer identifiers to exist (and be directly registered with IANA), but the goal is to manage actual registration through ISO 639, and use the namespace and identifiers established by this standard.

While the namespace of the primary language subtags is rather restricted (2- and 3-letter identifiers), IANA’s registry itself does not need to be directly concerned with its use and management, as this is handled by ISO through their ISO 639 process.

Without going into the details of how the other subtag namespaces are defined and registered, it should suffice to mention that one of the main goals of RFC 5646 is to ensure that the language registry of ISO 639 (as well as some others, such as the script registry of ISO 15924) and language tags as used on the Internet stay in sync. Thus, the management of the namespaces created for language tags by RFC 5646 [RFC5646] is mostly delegated to ISO, instead of being managed by IANA itself.

Nevertheless, RFC 5646 [RFC5646] does make provisions about the stability of entries in the various namespaces, so that the meaning of language tags remains stable over time. This includes provisions that existing values are not going to be changed, and that even values withdrawn by ISO remain valid and will simply be marked as "deprecated" in the respective IANA registry.

2.3. Web Linking

Web links can be typed by link relation types, and RFC 8288 [RFC8288] defines a model for how this can be done, and a registry for well-known values. One interesting aspect of the model is that the value space is divided: On the one hand there are well-known and registered values identified by strings, and on the other hand it is possible to use URIs, in which case no registration is required. This means that
the namespace for the link relation type registry is that of strings, meaning that it is not highly constrained.

With a rather large namespace, it is possible to accommodate a larger set of entries. However, it is still required that additions to the registry are done by following a process that requires describing the requested entries, and referring to a document that contains their definition and some context.

In addition, RFC 8288 [RFC8288] does define some constraints for how registered link relation types have to be defined. A submission process and reviews by designated experts are used to make sure that these constraints are met when new entries are submitted for inclusion in the registry.

2.4. Domain Name System (DNS)

The Domain Name System (DNS) specified in RFCs 1034 [RFC1034] and 1035 [RFC1035] is a distributed infrastructure for hosting a variety of entry types. In the context of the DNS, the entry types are defined by various Resource Record (RR) types. However, since the DNS is used for specific purposes, it is not a generic key/value store. Instead, the assumption is that the key always is a DNS name, whereas the value then is determined by the RR(s) available for that domain.

It could be argued that the DNS is not a typical registry because its design reflects the fact that entries are updated rather frequently, that entries therefore are not stable, and that registry access is a runtime issue Section 6.8. Because of these differences, it may be a bit of a stretch to call the DNS a "registry", but it is a well-known and well-established key/value lookup mechanism for IETF specifications, and therefore deserves mention in this document.

2.5. non-IANA/DNS example

(( Still looking for an example of a registry that’s established/used by an IETF spec and that is neither managed by IANA, nor part of DNS Section 2.4. If you have a suggestion, please let me know. ))

3. Why use Registries

Establishing and using a registry can be done for a number of reasons. The following sections list some of these reasons, and in many cases, registries are used for at least some of the reasons described here.
The list of reasons why to establish a registry may seem long, and in some cases it may feel as if these reasons are valid reasons for managing established long-term values, but that having a registry seems like a potential barrier to entry for short-term experimentation and other not-quite-permanent scenarios. Section 5 discusses this issue and how specifications can deal with it.

3.1. Openness and Extensibility

Registries separate a specification into a stable part that is represented by the specification itself, and a dynamic part that is represented by one or more registries that are established by the specification. This pattern allows a specification to remain stable, while still having well-defined parts that are allowed to evolve over time.

In order for this pattern to work well, the specification should clearly state what implementations should do when encountering unknown values in those locations where allowable values are managed in a registry. The two most popular processing models are to either silently ignore such a value and continue as if the value was not present at all, or to raise an error and notify higher layers of the fact that something unknown was encountered.

Depending on the way values are managed, it is also possible to distinguish between values that are supposed to be registered, and those that are not supposed to be registered and have to be considered unregistered extensions. The link relation types described in Section 2.3 use such an approach, defining that a link relation is either a string and supposed to be a registered value, or a URI in which case it is not supposed to be a registered value. This strategy works when it is possible to clearly separate the namespace of the place where values are expected into ones that are considered to be registered, and those that are not. This can be done lexically, or by having some kind of flag that indicates whether a value is supposed to be well-known, or an unregistered extension.

3.2. Limited Namespaces

Historically, registries started managing the limited namespace of identifier fields in protocol packets or other low-level mechanisms such as the port numbers described in Section 2.1, often limited to a small number of bits or bytes. Carefully managing this limited set of available identifiers was important, as was a way to allow new values to be added without having to update the protocol specification itself.
The higher the level is on which registries are used, the more likely it is that namespaces at least on the technical level are not overly constrained. For example, the link relation types described in Section 2.3 are using strings as identifiers without imposing a length limitation, meaning that the set of possible identifiers is virtually inexhaustible. However, even in this case, the set of helpful and meaningful identifiers (i.e., names that are human-readable and partly self-describing) is limited, and thus even in this case, the realistically useful namespace is much more limited that the theoretical one.

3.3. Design/Usage Review

Registries are established in the context of a given specification, and provide a mechanism to make this specification extensible by allowing the registry to evolve over time. However, the context of the specification often has a clear design rationale for why a registry is established for a certain set of values. Any value added or changed in the registry should fit into this context, and having a registry provides an opportunity to have design and usage reviews before the registry gets updated.

For design and usage reviews to work well, the most crucial aspect is that the context of the registry is well-defined, and states clearly what kind of expectations the design and usage review will be checking. Often this review process is implemented using a mailing list and designated experts, so that registration requests as well as results of the design and usage review are done openly and transparently.

3.4. Identifier Design

Depending on the namespace, managing the registry namespace may follow certain guidelines. For numeric values, there may be certain number ranges that are supposed to be used in certain ways. For string values, there may be some convention or best practices on how to mint identifiers so that the namespace contains values that are following these principles.

Note that this is different from the design and usage review Section 3.3. Whereas the design and usage review is about testing whether the meaning associated with a new value follow the constraints defined in the context that established the registry, the identifier design simply checks for how the registered values are chosen. It thus is a lower bar than a design and usage review, but still requires a review process that allows to propose new values, and provides some feedback about whether these values follow the guidelines or not.
3.5. Identifier Lifecycle

The main reason for registries to exist in contrast to just including a set of predefined values in the underlying specification itself is the ability for these values to change over time. However, registries only make sense if there is some sense of stability to their contents, so that looking at existing registry values at some point in time can be assumed to be a reasonable snapshot for some amount of time.

Usually, registry entries are added at a modest pace, and an implementation not supporting the latest additions shouldn't fail, but simply implement some default behavior when encountering unsupported values. This pattern ensures that the namespace can evolve separately from the landscape of implementations.

However, adding identifiers is the easiest aspects of registry updates. Things get more complicated when it comes to updating and removing entries. The reason why these things are more complicated is that implementations depending on an identifier having certain semantics will behave incorrectly when the registry has been updated for this identifier with either a change in semantics, or a withdrawal of the entry.

For this reason, it often makes sense to include rules in the management of the registry about if/how entries can be updated, or removed. One popular approach is disallow updates with breaking changes, and to allow withdrawal but keep the identifier and mark it as "deprecated". This way it can be ensured that no incompatible entry will be created by somebody using an identifier that was previously used and removed.

The exact way how this process is defined depends on the context and purpose of the registry, and also on the namespace size. Tightly constrained namespaces mean that values probably should be managed more carefully, so that the registry does not run out of values. Also, while impossible to predict reliably, it is also important to look at the possible lifetime of implementations (that will use snapshots of the registry at some point in time), and on the long-term effects of having outdated registry snapshots in implementations.

3.6. Documentation Requirements

Registering a value means that people encountering this value should be able to learn about what it represents. This means that there should be documentation associated with it that can be used to learn more about the value’s meaning. Many registries at least require a
short explanation to be submitted with a registration request, so that the registry itself can list those texts as helpful explanations.

Going further, many registries also require links to more detailed specifications, so that people looking for complete explanations of the meaning of registered values can follow those links and will find specifications or at least explanations. The exact requirement on what such a link must refer to is something that the specification creating the registry has to define. One popular requirement is that it must be publicly available information, so that anybody looking for it can openly access it.

3.7. Centralized Lookup

With a registry containing all current values (and possibly listing changed/deprecated ones as well) along with some registration metadata, they provide valuable information for anybody looking for information about registered values in the registry namespace. All IANA protocol registries [IANA-Protocol-Registry] are openly accessible on the Web, allowing everybody to lookup the current state of all these registries.

Even though centralized lookup is an important aspect of openness and extensibility Section 3.1, the usual usage model of these lookup facilities is to use them at design-time rather than at runtime Section 6.8. This means that the central lookup facilities are meant to be used by developers, and not by the implementations created by those developers. For the latter model a much more scalable infrastructure would be required, and thus it is important to consider the fact if the namespace managed by a registry fits this model of being useful for developer lookup at design-time, and for value lookup at runtime.

4. When to use Registries

Based on the examples given in Section 2 and the possible reasons described in Section 3, the next question is how for designers to decide when they should establish one or more registries to complement a specification. All the issues describes in Section 3 are reasonable motivations, and in many cases it is more than just one of them.

For developers using a specification, it is helpful if the specification clearly describes which reasons were most important when deciding to establish one or more registries. This is even more true for developers who are looking to update the registry, because
they should be aware of the reasons that were considered when the registry was created.

For every registry that is established, it is helpful if a specification explains the following general aspects:

What were the main design rationales behind establishing the registry? The reasons described in Section 3 may be a good starting point to pick from.

What are the management policies for the registry? Depending on a variety of factors such as namespace size, expected frequency of updates, level of review before acceptance, required level of documentation, and possibly others, management can be rather lightweight or a carefully managed process.

What is the size of the namespace and the expected rate of how it will be used and possibly exhausted?

Even if it makes sense to establish a registry based on the reasons given in Section 3, and if a specification makes use of this pattern, it is possible that registries are a potential barrier to entry. Section 5 discusses how these issues can be addressed.

5. Barrier to Entry Issues

For some of the reasons described in Section 3, specification authors may decide to establish one or more registries, as described in Section 4. However, it may be the case that having such a registry can make it harder for developers to experiment with implementations (without violating the specification), and/or that there should be a more "lightweight" way for establishing some identifiers than for others.

This section presents some design options for this issue, and discusses how these design options are influenced by the registry's identifier namespace, and the registration policy.

5.1. Non-Semantic/Reserved Entries

One possible approach is to explicitly reserve parts of the available namespace for "non-semantic" purposes. This means that whenever such a value is encountered, it is impossible to conclude based on this value alone which concept it represents. Instead, the interpretation of the value depends on additional information and/or on context, and usually the exact mechanism is outside of the scope of the specification.
One example for this method is the TCP/UDP port example given in Section 2.1. The namespace for port numbers is two bytes and thus numbers from 0 to 65535. However, numbers in the range 49152-65535 are considered to be "dynamic ports", and are reserved for uses that are outside of the scope of registered values.

5.2. Entry Levels

A different pattern to differentiate values is to have different "levels" that values can be registered for. This effectively partitions the registry into different classes, which sometime follow certain naming conventions, sometimes not. In the latter case, when encountering a value, it is necessary to consult the registry to understand which level a value is associated with.

The levels often carry differentiations in a variety of factors, such as how established represented concepts are, how well and/or how open they are documented, how the expected stability of the entry is, and what level of scrutiny is applied during the review process when registering a new value. Specifications have a wide latitude defining those levels and specifying what they represent and how they affect value semantics and registration procedures.

As one example, media types [RFC6838] establish a "standards tree" and as well as other trees ("vendor" and "personal"). Entries in these trees are distinguished by prefix or lack thereof. The general idea is that the non-standard trees contain entries that are not quite at the same level, significance-wise, as the standards tree.

As another example, URI schemes [RFC7595] differentiate registrations into "permanent" and "provisional" schemes. In addition, URI schemes can also be registered as being "historical". A "provisional" registration is intended to be used in cases where URI deployment is likely to happen outside of a private environment, but still controlled by a private party. It is also possible to update a registration from "provisional" to "permanent" once the associated protocol has become stable or more widely deployed.

The classification of registry entries into "permanent" and "provisional" is a pattern being followed by some IETF specifications and registries as well, for example by the registration procedures for message header fields [RFC3864].

5.3. Separation by Value Syntax

While the examples given in Section 5.2 sometimes use ways to differentiate entry levels by name, this is just a convenience so that it is possible to determine a value’s level by value syntax. In
this case, the value syntax is enforced by registration policy, meaning that during the registration process it is ensured that when a registration request for a certain level is made, the registered value conforms to the required syntax. However, in this case, all values still need to be registered.

A different approach is to separate the value range itself by syntax, and only require registration for partitions of that range. For example, link relation types as described in Section 2.3 allow strings and URIs as values. For strings, it is expected that any used values are registered. For URIs, however, no such registration is required or even possible. This means that "private link relation types" can be safely identified by URI values, and since it is possible to make these dereferenceable, it is also possible to make these values self-documenting and/or self-describing by making human- and/or machine-readable information available at that URI.

While this separation by value syntax can be useful, it is important to note that by design, any approach that reflects value classification through value syntax automatically means that values cannot moved between categories. Renaming established and deployed values is an expensive thing to do, which means that the approach of separation by value syntax should be carefully considered, in particular in light of the fact that it makes it impossible for values to be moved between categories. This may be acceptable for some scenarios, but may be undesirable in other. RFC 6648 [RFC6648] discusses this problem starting from prefix-based value syntax approaches (the famous "X-" for extension values), but is not limited to discussing this prefix-based approach alone.

6. How to use Registries

If a specification does introduce registries as part of how the specification divides static and dynamic parts, then it is interesting to look at how those registries will actually be used. As in the previous sections, the list of topics included here is not necessarily mutually exclusive, and it is likely that for any established registry, there is more than one way how the registry is being used.

For authors of specifications establishing registries, the following list of possible ways how a registry might be used may be a good starting point to consider the design options for the registry, such as how to design the submission and update process, and how to provide access to registry contents.
6.1. Registry Operations

A registry is an abstract idea, mostly consisting of a separation of a stable and fixed specifications, and associated registry-based values that can be looked up and which may change over time. How a registry is operated is a different matter. This is true for both the workflows associated with registry operations, and the technical implementation of those workflows.

For both the workflows, and their technical implementation, there are a large number of issues to consider, and a large number of possible solutions that satisfy those issues. For an organization like the IETF, with a substantial stream of technical specifications, it makes a lot of sense to establish both a common workflow, and a common implementation, so that specifications can rely on this infrastructure, instead of having to re-invent registry operations every time they want to establish a registry.

In the context of IETF specifications, registries are typically maintained by the "Internet Assigned Numbers Authority (IANA)". Since there is a large number of registries, and they should be maintained in a coherent, systematic, and efficient way, there is a set of "Principles for Operation of Internet Assigned Numbers Authority (IANA) Registries" which are described in RFC 7500 [RFC7500]. The key principles are defined as follows:

- Ensure Uniqueness: The same protocol identifier must not be used for more than one purpose.
- Stable: Protocol identifier assignment must be lasting.
- Predictable: The process for making assignments must not include unexpected steps.
- Public: The protocol identifiers must be made available in well-known locations in a manner that makes them freely available to everyone.
- Open: The process that sets the policy for protocol identifier assignment and registration must be open to all interested parties.
- Transparent: The protocol registries and their associated policies should be developed in a transparent manner.
- Accountable: Registry policy development and registry operations need to be accountable to the affected community.
The way RFC 7500 currently defines operational principles leaves some possible issues open. For example, the principle of the identifiers being "public" is described as the identifiers being "freely available to everyone without restrictions." This means that it is sufficient to make the identifiers available in human-readable form, as opposed to the more specific ways in which machine-readable access (as discussed in Section 6.7) would have to be enabled and possibly managed.

While the exact requirements for a registry can be spelled out in the "IANA Considerations" section of a specification establishing a registry (see Section 6.2), there are hard limitations based on the current implementation which is hosted by IANA. It is possible that a different model may be implemented at a later time, but the current model is biased towards email-based workflows and human-readable registry access. If specification authors feel that IANA’s implementation will not fit their needs, then it is (at least theoretically) possible for a specification to define its own registry operations and infrastructure, but that would require a lot of effort for a

6.2. Registry Creation

If a specification creates one or more registries, then RFC 5226 [RFC5226] sets some guidelines for the setup process and the substance of each registry. Without repeating those guidelines here, it should suffice to mention that those revolve around possible status labels for assigned values (private use, experimental, unassigned, reserved), and a number of predefined policies that define how a registry is managed. RFC 5226 also mentions the fact that a registry’s management policy can change, in which case a new specification is required that updates the definition of the registry.

6.3. Registry Interaction

Since registry contents establish a controlled vocabulary, and each registry has some policies around how that vocabulary can be updated, it usually makes sense to have a template or a form that allows applicants to prepare and submit update requests. It is up to the registry to define how detailed this template is, whether applicants are required to use it, and whether submission is implemented based on that template. RFC 5226 [RFC5226] provides guidelines for specification writers how to include any registry interactions in their documents.

Following some structure in the interaction process help with keeping a better record of requested and performed updates of a registry,
which can be helpful when it comes to maintaining and providing a registry history Section 6.6.

6.4. Implementation Support

The usual pattern for using registry-based identifiers in implementations is to support some snapshot of the registry, which either can be complete, or just a subset of the registry contents. Implementation support then is based on the semantics associated with registry values at the time of this snapshot. This means that any registry updates changing semantics will affect and possibly break those implementations, unless there is a strong policy to only allow backwards-compatible changes to identifier semantics.

6.5. Registry Stability

Since implementations often use registries based on snapshots Section 6.4, a key issue for registries is the stability of entries. While it is clear that new entries can be added (this after all is the minimal use case of registries: the ability to add identifiers without the need for updating the specification itself), things get more complicated when it comes to updates of existing entries, or removal of exiting entries.

When it comes to updating entries, then often the goal is to avoid breaking changes. This means that entries can only be updated in a way that the updated semantics are backwards-compatible with the old semantics. This way, implementations based on the old semantics can safely use those and will not conflict when encountering data or implementations assuming the updated semantics.

In terms of removal, it also is important to consider whether removed entries should remain registered and blocked for future registrations, so that they cannot be re-used (which essentially would be equivalent to making a breaking change to an existing entry). If such a policy is in-place, then technically speaking there is no actual "removal" of a value rom the registry. Instead, a value can be updated to be deprecated, but it remains in the registry so that it is not re-assigned.

6.6. Registry History

While not strictly necessary for registry usage and management, in terms of openness and transparency it can be helpful to provide a registry history. This way it is possible to recreate all actions that changed the registry, and to reconstruct the state of the registry at any point in time.
Some registries have mailing lists associated with them which can be regarded as some sort of history. However, this is a weak form of history since reconstructing the registry history and its state requires to read all the emails, and infer the resulting registry actions. Having access to the actual actions in machine-readable form can make it much easier to access and recreate registry history.

Since registry usually are highly structured (often tabular models with small number of columns), they would lend themselves well to representing any updates to the contents in a similarly structured way, along with some metadata about the entry update (such as date, applicant, expert, links to email archives, and similar ways to contextualize the entry update).

6.7. Registry Access

Depending on the intended use of a registry, an important question is how developers and/or implementations can access the registry. While the current IANA registries can be accessed via HTTP, they are clearly intended and designed to be used as human-readable HTML pages for developers. Alternative or complementary models could provide API-based access, with documented and stable ways how to provide machine-based access to registry contents.

However, if an API is considered and provided, then an important question is whether it is intended for accessing registry contents only, or providing full-fledged access to all services of the registry, such as updates Section 6.3 or history access Section 6.6. The former kind of access is easier to accomplish, because the sets of provided is smaller, and the requirements for authentication and authorization are probably simpler.

In addition to the question of how the registry API would be designed, a more important question may be how it would be managed. Any Internet-wide registry that provided API access would have to carefully consider the implications of providing such a service, mostly in terms of registry operations. Such as API could easily become overloaded, and then would become a possible point of failure. Also, it could become a point of attack, both in terms of denial or service attacks, or in attempts to use the API to access the registry in unauthorized ways.

6.8. Runtime vs. Design-Time

Most implementations use registry snapshots (complete or partial) for using a registry’s contents Section 6.4. If a registry provides API access, then it would be possible to author implementations that use
registry contents at runtime. However, there are two important concerns about this possibility:

Having access to registry contents may be of little use other than learning about the existence of new identifiers. In most cases, a registry entry alone will not be sufficient to understand the semantics of a new entry encountered at runtime. If that is the case, then all an implementation can do is to verify that it encountered a new entry that is a valid identifier according to the current registry state, but it cannot implement the behavior associated with that new entry.

If the model of the registry allows meaningful implementation behavior by runtime updates, then this can result in this registry becoming overwhelmed by the number of accesses. After all, dynamic implementation behavior then may be preferable over the more traditional snapshot implementation pattern, which then results in the majority of implementations converging to the runtime access model.

Because of the second issue in particular, any registry supporting and intended for runtime access should make sure that provisions are in place to control registry access. This is no different from any other service on the Internet or Web that also needs to have mechanisms in place to protect itself against suffering under too much load.

7. IANA Considerations

This document has no IANA actions.

8. References

[IANA-Protocol-Registry]


Appendix A. W3C Examples

As an example of when registries can be useful, this appendix lists "registries" defined by the World Wide Web Consortium (W3C). The W3C does not currently have a registry model such as IETF’s IANA, and thus W3C specification authors have to come up with their own solutions how to define and manage evolving sets of values.

The following list is not meant to be complete, but it does highlight that a lack of registry capabilities in larger organizations can lead to the "registry problem" being approached and solved in a variety of ways, each of them requiring their own processes and infrastructure, and each of them having their own side-effects.

Over time, various W3C specifications have used managed lists of values. One of these specifications already is a stable specification, it is the "XPointer" specification. XPointer has a scheme registry which is managed as a manually updated Web page, which has a documented registry policy.

Of the W3C drafts under development, one common model is to define the values in a separate document, and then to evolve this document. This has the advantage of decoupling the actual specification from the set of managed values. The downside is that value evolution now is based on the process defined for W3C documents, event though that’s not primarily the intention and the goal.

When following this approach, one set of specifications defines the values in a Working Draft (WD) document. Here are the current W3C specifications following this approach:

- Identifiers for WebRTC’s Statistics API: https://www.w3.org/TR/webrtc-stats/
- UI Events KeyboardEvent code Values: https://www.w3.org/TR/uievents-code/
- UI Events KeyboardEvent key Values: https://www.w3.org/TR/uievents-key/
- Basic Card Payment: https://www.w3.org/TR/payment-method-basic-card/
- Payment Method Identifiers: https://www.w3.org/TR/payment-method-id/
- Timing Entry Names Registry: https://www.w3.org/TR/timing-entrytypes-registry/
A second set of specifications also uses a separately evolved document, but manages this as a Note (NOTE) document. Here are the current W3C specifications following this approach:

- Encrypted Media Extensions Stream Format Registry: 
  [https://www.w3.org/TR/eme-stream-registry/](https://www.w3.org/TR/eme-stream-registry/)

- TTML Media Type Definition and Profile Registry: 
  [https://www.w3.org/TR/ttml-profile-registry/](https://www.w3.org/TR/ttml-profile-registry/)

- Media Source Extensions Byte Stream Format Registry: 
  [https://www.w3.org/TR/mse-byte-stream-format-registry/](https://www.w3.org/TR/mse-byte-stream-format-registry/)

- Trace Context Protocols Registry: [https://www.w3.org/TR/trace-context-protocols-registry/](https://www.w3.org/TR/trace-context-protocols-registry/)

Another approach is to keep the values inside the specification itself. This has the side-effect that any change in the "registered" values requires a change of the complete specification, which usually is one is trying to avoid when identifying a potentially evolving set of values. Here are the current W3C specifications following this approach:

- Performance Timeline Level 2 "entryType": [https://www.w3.org/TR/performance-timeline-2/#dom-performanceentry-entrytype](https://www.w3.org/TR/performance-timeline-2/#dom-performanceentry-entrytype)

- Permissions "Permission Registry": [https://www.w3.org/TR/permissions/#permission-registry](https://www.w3.org/TR/permissions/#permission-registry)

And finally, the last practice seen in W3C specifications is to use a wiki as the managed list of values. Here are the current W3C specifications following this approach:

- Web App Manifest Platform Values (wiki is located on GitHub): 
  [https://github.com/w3c/manifest/wiki/Platforms](https://github.com/w3c/manifest/wiki/Platforms)

As can be seen from this list of practices, in larger organizations it can become increasingly useful to support registries as a model of how to define and manage evolving sets of values. Whether or not this results in an organization creating and managing their own registry infrastructure is one question. But apart from that, at least identifying this as an issue that various teams are facing in the organization, and giving them guidance on how to solve it, can already help to better support teams, and avoid a fragmentation of approaches that all create their own side-effects and technical debt in the long run.
Appendix B. Acknowledgements

Thanks for comments and suggestions provided by John Curran, Doug Ewell, Graham Klyne, and Andrew Malis.

Author’s Address

Erik Wilde

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