Stream: Internet Engineering Task Force (IETF)
RFC: 9553
Category: Standards Track
Published: March 2024
ISSN: 2070-1721
Authors: R. Stepanek *Fastmail*
M. Loffredo *IIT-CNR*

RFC 9553
JSContact: A JSON Representation of Contact Data

Abstract

This specification defines a data model and JavaScript Object Notation (JSON) representation of contact card information that can be used for data storage and exchange in address book or directory applications. It aims to be an alternative to the vCard data format and to be unambiguous, extendable, and simple to process. In contrast to the JSON-based jCard format, it is not a direct mapping from the vCard data model and expands semantics where appropriate.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9553.

Copyright Notice

Table of Contents

1. Introduction

This document defines a data model for contact card data normally used in address book or directory applications and services. It aims to be an alternative to the vCard data format [RFC6350].

The key design considerations for this data model are as follows:

- The data model and set of attributes should be mostly compatible with the model defined for the vCard data format [RFC6350] and extensions [RFC6473] [RFC6474] [RFC6715] [RFC6869] [RFC8605]. The specification should add new attributes or value types where appropriate. Not all existing vCard definitions need an equivalent in JSContact, especially if the vCard definition is considered to be obsolete or otherwise inappropriate. Conversion between the data formats need not fully preserve semantic meaning.
- The attributes of the card data represented must be described as simple key-value pairs, reducing complexity of their representation.
- The data model should avoid all ambiguities and make it difficult to make mistakes during implementation.

- Extensions, such as new properties and components, **MUST NOT** lead to a required update of this document.

The representation of this data model is defined in the Internet JSON (I-JSON) format [RFC7493], which is a strict subset of the JSON data interchange format [RFC8259]. Using JSON is mostly a pragmatic choice: its widespread use makes JSContact easier to adopt, and the availability of production-ready JSON implementations eliminates a whole category of parser-related interoperability issues.

1.1. Motivation and Relation to vCard, jCard, and xCard

The vCard data format [RFC6350] is an interchange format for contacts data between address book service providers and vendors. However, this format has gone through multiple specification iterations with only a subset of its deprecated version 3 [RFC2426] being widely in use. Consequently, products and services use an internal contact data model that is richer than what they expose when serializing that information to vCard. In addition, service providers often use a proprietary JSON representation of contact data in their APIs.

JSContact provides a standard JSON-based data model and representation of contact data as an alternative to proprietary formats.

At the time of writing this document, several missing features in vCard were brought to the attention of the authors such as social media contacts, gender pronouns, and others. This highlights how vCard is not perceived as an evolving format and, consequently, hasn't been updated for about ten years. JSContact addresses these unmet demands and defines new vCard properties and parameters to allow interchanging them in both formats.

The xCard [RFC6351] and jCard [RFC7095] specifications define alternative representations for vCard data in XML and JSON formats, respectively. Both explicitly aim to not change the underlying data model. Accordingly, they are regarded as equal to vCard in the context of this document.

1.2. 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.

The ABNF definitions in this document use the notations of [RFC5234]. ABNF rules not defined in this document are defined in either [RFC5234] (such as the ABNF for CRLF, WSP, DQUOTE, VCHAR, ALPHA, and DIGIT) or [RFC6350].

1.3. Data Type Notations

This section introduces the notations and terminology used to define data types in JSContact.

The underlying format for JSContact is JSON, so its data types also build on JSON values. The terms "object" and "array" as well as the four primitive types ("strings", "numbers", "booleans", and "null") are to be interpreted as described in Section 1 of [RFC8259]. All JSContact data **MUST** be valid according to the constraints given in I-JSON [RFC7493]. Unless otherwise noted, all member names in JSON objects and all string values are case-sensitive. Within the context of JSON objects, the term "key" is synonymous with "member name" as defined in Section 1 of [RFC8259].

1.3.1. Objects and Properties

JSContact defines data types for contact information such as addresses or names. This information typically consists of multiple related elements; for example, a personal name and surname together form a name. These related elements are organized in JSContact objects. A JSContact object is a JSON object that has the following:

1. A unique type name registered in the IANA "JSContact Types" registry (Section 3.6).
2. One or more object members for which the name and allowed value types are specified. Such members are called "properties".
3. One property named `@type` with a string value that matches the type name of the JSContact object. In general, this property does not need to be set explicitly as outlined in Section 1.3.4.

The following sections specify how to define JSContact object types. Sections 1.7 and 1.8 then define the exact requirements for property names.

The next paragraph illustrates how a JSContact object is defined.

A Foo object has the following properties:

`qux`: Number (mandatory). Defines the qux-ishness of this contact. The value **MUST** be an integer greater than 0 and less than 10.

Here, a JSContact object type named `Foo` is defined. In addition to its `@type` property, it has a property named `qux` for which values **MUST** be valid according to the definition of the `Number` type. The property has one attribute, `mandatory`, which specifies that the property **MUST** be present for an instance of this JSContact object to be valid. Finally, a free-text description describes the semantics and further restrictions.

1.3.2. Type Signatures

Type signatures are given for all JSON values and JSContact definitions in this document. The following conventions are used:

`String`: The JSON string type.

`Number`: The JSON number type.

`Boolean`: The JSON boolean type.

`A[B]`: A JSON object where all keys are of type `A` and all values are of type `B`.

`A[]`: A JSON array of values of type `A`.

`A|B`: The value is either of type `A` or of type `B`.

`* *`: The type is undefined (the value could be any type, although permitted values may be constrained by the context of this value).

Section 1.4 defines common data types, including signed or unsigned integers and dates.

1.3.3. Property Attributes

Object properties may also have a set of attributes defined along with the type signature. These have the following meanings:

`mandatory`: The property **MUST** be set for an instance of this object to be valid.

`optional`: The property can, but need not, be set for an instance of this object to be valid.

`default`: This is followed by a JSON value. That value will be used for this property if it is omitted.

`defaultType`: This is followed by the name of a JSContact object type. A property value of JSContact object type is expected to be of this named type, in case it omits the `@type` property.

1.3.4. The `@type` Property

`@type`: String. Specifies the type of the object. It **MUST** match the type name of the JSContact object of which the JSON object is an instance of.

The purpose of the `@type` property is to help implementations identify which JSContact object type a given JSON object represents. Implementations **MUST** validate that JSON objects with this property conform to the specification of the JSContact object type of that name.

In many cases, the `@type` property value is implied by where its object occurs in JSContact data. Assuming that both `A` and `B` are JSContact object types:

- An object that is set as the value for a property with type signature `A` **MAY** have the `@type` property set. If the `@type` property is not set, then its value is implied to be `A` by the property definition.
- An object that is set as the value for a property with type signature `A|B (defaultType: A)` **MAY** have the `@type` property set if it is an instance of `A`. It **MUST** have the `@type` property set if it is an instance of `B`. If, instead, the `defaultType` attribute is not defined, then the `@type` property **MUST** also be set for `A`.
- An object that is not the value of a property, such as the root of JSON data (directly or as a member of an array), **MUST** have the `@type` property set.

1.4. Common Data Types

In addition to the standard JSON data types, a couple of additional data types are common to the definitions of JSContact objects and properties.

1.4.1. Id

Where `Id` is given as a data type, it means a `String` of at least 1 and a maximum of 255 octets in size, and it **MUST** only contain characters from the `URL and Filename Safe` base64url alphabet, as defined in Section 5 of [RFC4648], excluding the pad character (`=`). This means the allowed characters are the ASCII alphanumeric characters (`A-Za-z0-9`), hyphen (`-`), and underscore (`_`).

In many places in JSContact, a JSON map is used where the map keys are of type Id and the map values are all the same type of object. This construction represents an unordered set of objects, with the added advantage that each entry has a name (the corresponding map key). This allows for more concise patching of objects and, when applicable, for the objects in question to be referenced from other objects within the JSContact object. The map keys **MUST** be preserved across multiple versions of the JSContact object.

Unless otherwise specified for a particular property, there are no uniqueness constraints on an Id value (other than, of course, the requirement that you cannot have two values with the same key within a single JSON map). For example, two Card (Section 2) objects might use the same Ids in their respective `photos` properties. Or within the same Card, the same Id could appear in the `emails` and `phones` properties. These situations do not imply any semantic connections among the objects.

1.4.2. Int and UnsignedInt

Where `Int` is given as a data type, it means an integer in the range $-2^{53}+1 <= value <= 2^{53}-1$, which is the safe range for integers stored in a floating-point double, represented as a JSON `Number`.

Where `UnsignedInt` is given as a data type, it means an integer in the range $0 <= value <= 2^{53}-1$ represented as a JSON `Number`.

1.4.3. PatchObject

A PatchObject is of type `String[*]` and represents an unordered set of patches on a JSON object. Each key is a path represented in a subset of the JSON Pointer format [RFC6901]. The paths have an implicit leading `/`, so each key is prefixed with `/` before applying the JSON Pointer evaluation algorithm.

A patch within a PatchObject is only valid if all the following conditions apply:

1. The pointer **MAY** reference inside an array, but if the last reference token in the pointer is an array index, then the patch value **MUST NOT** be null. The pointer **MUST NOT** use "-" as an array index in any of its reference tokens (i.e., you **MUST NOT** insert/delete from an array, but

you **MAY** replace the contents of its existing members. To add or remove members, one needs to replace the complete array value).

2. All reference tokens prior to the last (i.e., the value after the final slash) **MUST** already exist as values in the object being patched. If the last reference token is an array index, then a member at this index **MUST** already exist in the referenced array.
3. There **MUST NOT** be two patches in the PatchObject where the pointer of one is the prefix of the pointer of the other, e.g., `addresses/1/city` and `addresses`.
4. The value for the patch **MUST** be valid for the property being set (of the correct type and obeying any other applicable restrictions), or if null, the property **MUST** be optional.

The value associated with each pointer determines how to apply that patch:

- If null, remove the property from the patched object. If the key is not present in the parent, this is a no-op.
- If non-null, set the value given as the value for this property (this may be a replacement or addition to the object being patched).

A PatchObject does not define its own `@type` (Section 1.3.4) property. Instead, an `@type` property in a patch **MUST** be handled as any other patched property value.

Implementations **MUST** reject a PatchObject in its entirety if any of its patches are invalid. Implementations **MUST NOT** apply partial patches.

1.4.4. Resource

The `Resource` data type defines a resource associated with the entity represented by the Card, identified by a URI [RFC3986]. Later in this document, several property definitions refer to the Resource data type as the basis for their property-specific value types. The Resource data type defines the properties that are common to all of them. Property definitions making use of Resource **MAY** define additional properties for their value types.

The `@type` property value **MUST NOT** be `Resource`; instead, it **MUST** be the name of a concrete resource type (see Section 2.6). A Resource object has the following properties.

@type: `String`. Specifies the type of this resource object. The allowed value is defined in later sections of this document for each concrete resource type (Section 2.6).

kind: `String` (optional). The kind of the resource. The allowed values are defined in the property definition that makes use of the Resource type. Some property definitions may change this property from being optional to mandatory.

uri: `String` (mandatory). The resource value. This **MUST** be a *URI* as defined in Section 3 of [RFC3986].

mediaType: `String` (optional). Used for URI resource values. Provides the media type [RFC2046] of the resource identified by the URI.

contexts: `String[Boolean]` (optional). The contexts in which to use this resource. Also see Section 1.5.1.

pref: `UnsignedInt` (optional). The preference of the resource in relation to other resources. Also see Section 1.5.4.

label: `String` (optional). A custom label for the value. Also see Section 1.5.3.

1.4.5. UTCDateTime

`UTCDateTime` is a string in `date-time` format [RFC3339], with further restrictions that any letters **MUST** be in uppercase and the time offset **MUST** be the character `Z`. Fractional second values **MUST NOT** be included unless they are non-zero, and they **MUST NOT** have trailing zeros to ensure there is only a single representation for each date-time.

For example, `2010-10-10T10:10:10.003Z` is conformant, but `2010-10-10T10:10:10.000Z` is invalid; the correct encoding is `2010-10-10T10:10:10Z`.

1.5. Common Properties

Most of the properties in this document are specific to a single JSContact object type. Such properties are defined along with the respective object type. The properties in this section are common to multiple data types and are defined here to avoid repetition. Note that these properties **MUST** only be set for a JSContact object if they are explicitly mentioned as allowable for this object type.

1.5.1. contexts

Type: `String[Boolean]`

This property associates contact information with one or more contexts in which it should be used. For example, someone might have distinct phone numbers for work and private contexts and may set the desired context on the respective phone number in the `phones` (Section 2.3.3) property.

This section defines common contexts. Additional contexts may be defined in the properties or data types that make use of this property. The enumerated (Section 1.7.4) common context values are:

- `private`: the contact information that may be used in a private context.
- `work`: the contact information that may be used in a professional context.

1.5.2. extra

`extra` is a reserved property name. Implementations **MUST NOT** set this property in a JSContact object. Any JSContact object including a property with this name **MUST** be considered invalid.

The purpose of this reserved property name is to provide implementors with a name that is certain to never occur as a property name in a JSContact object. Implementations might want to map unknown or vendor-specific properties to a variable with this name, but this is implementation-specific.

1.5.3. label

Type: `String`

This property allows associating contact data with user-defined labels. Such labels may be set for phone numbers, email addresses, and resources. Typically, these labels are displayed along with their associated contact data in graphical user interfaces. Note that succinct labels are best for proper display on small graphical interfaces and screens.

1.5.4. pref

Type: `UnsignedInt`

This property allows defining a preference order for contact information. For example, a person may have two email addresses and prefer to be contacted with one of them.

Its value **MUST** be in the range of 1 to 100. Lower values correspond to a higher level of preference, with 1 being most preferred. If no preference is set, then the contact information **MUST** be interpreted as being least preferred.

Note that the preference is only defined in relation to contact information of the same type. For example, the preference orders within emails and phone numbers are independent of each other.

1.5.5. phonetic

The phonetic property defines how to pronounce a value in the language indicated in the Card language (Section 2.1.5) property or the language tag of its localizations (Section 2.7.1). Exemplary uses of this property are defining how to pronounce Japanese names and romanizing Mandarin or Cantonese name and address components. The properties are defined as follows:

phonetic: `String`. Contains the phonetic representation of a value. Any script language subtag in the Card language (Section 2.1.5) property **MUST** be ignored for use with the `phonetic` property. If this property is set, then at least one of the `phoneticScript` or `phoneticSystem` properties that relate to this value **MUST** be set.

phoneticScript: `String`. The script used in the value of the related `phonetic` property. This **MUST** be a valid script subtag as defined in Section 2.2.3 of [RFC5646].

phoneticSystem: `String`. The phonetic system used in the related value of the `phonetic` property. The enumerated (Section 1.7.4) values are:

- `ipa`: denotes the International Phonetic Alphabet [IPA].
- `jyut`: denotes the Cantonese romanization system "Jyutping".

- `piny`: denotes the Standard Mandarin romanization system "Hanyu Pinyin".

The relation between the `phoneticSystem`, `phoneticScript`, and `phonetic` properties is type-specific. This specification defines this relation in the Name (Section 2.2.1) and Address (Section 2.5.1) object types, respectively.

The following example illustrates the `phonetic` property for a name (Section 2.2.1):

```
"name": {
  "components": [{
    "kind": "given",
    "value": "John",
    "phonetic": "/ˈdʒɑːn/"
  }, {
    "kind": "surname",
    "value": "Smith",
    "phonetic": "/smɪθ/"
  }],
  "phoneticSystem": "ipa"
}
```

Figure 1: Example of a `phonetic` Property for the Name "John Smith" as Pronounced in the USA

1.6. Internationalization

JSContact aims to be used for international contacts and address book data. Notably, text values such as names and addresses are likely to cover a wide range of languages and cultures. This section describes internationalization for free-form text values as well as Uniform Resource Identifiers (URIs).

1.6.1. Free-Form Text

Properties having free-form text values **MAY** contain any valid sequence of Unicode characters encoded as a JSON string. Such values can contain unidirectional left-to-right and right-to-left text, as well as bidirectional text using Unicode Directional Formatting Characters as described in Section 2 of [UBiDi]. Implementations setting bidirectional text **MUST** make sure that each property value complies with the requirements of the Unicode Bidirectional Algorithm. Implementations **MUST NOT** assume that text values of adjacent properties are processed or displayed as a combined string; for example, the values of a given name component and a surname component may or may not be rendered together.

1.6.2. URIs

Several properties require their string value to be a URI as defined in [RFC3986]. Implementations **MUST** make sure to use proper percent-encoding for URIs that cannot be represented using unreserved URI characters. Section 3.1 of [RFC3987] defines how to convert Internationalized Resource Identifiers to URIs. JSContact makes no recommendation on how to display URIs, but the WHATWG URL Living Standard (see "Internationalization and special characters" (Section 4.8.3) of [WHATWG-URL]) provides guidance for URLs found in the context of a web browser.

1.7. Validating JSContact

This specification distinguishes between three kinds of properties regarding validation: IANA-registered properties and unknown properties, which are defined in this section, and vendor-specific properties, which are defined in Section 1.8.1. A JSContact object is invalid if any of its properties are invalid.

This document defines whether each property is mandatory or optional. A mandatory property **MUST** be present for a JSContact object to be valid. An optional property does not need to be present. The values of both required and optional properties **MUST** adhere to the data type and definition of that property.

1.7.1. Case-Sensitivity

All property names, object type names, and enumerated values are case-sensitive, unless explicitly stated otherwise in their definitions. Implementations **MUST** handle a JSContact object as invalid if a type name, property name, or enumerated value only differs in case from one defined for any JSContact version known to that implementation. This applies regardless of what JSContact version the Card object defines in its `version` (Section 2.1.2) property. Section 1.7.3 defines how to handle unknown properties.

1.7.2. IANA-Registered Properties

An IANA-registered property is any property that has been registered according to the IANA property registry rules as outlined in Section 3. All properties defined in this specification, including their object value types and enumerated values, are registered at IANA.

Implementations **MUST** validate IANA-registered properties in JSContact data, unless they are unknown to the implementation (Section 1.7.3). They **MUST** reject invalid IANA-registered properties. A property is invalid if its name matches the name of an IANA-registered property but the value violates its definition according to the JSContact specification version defined in the Card `version` (Section 2.1.2) property.

IANA-registered property names **MUST NOT** contain US-ASCII control characters (U+0000 to U+001F, U+007F), the `COLON` (U+003A), or the `QUOTATION MARK` (U+0022). They **MUST** only contain US-ASCII alphanumeric characters that match the ALPHA and DIGIT rules defined in Appendix B.1 of [RFC5234] or the `COMMERCIAL AT` (U+0040) character. IANA-registered property names **MUST** be notated in lower camel case.

1.7.3. Unknown Properties

Implementations may encounter JSContact data where a property name is unknown to that implementation but the name adheres to the syntactic restrictions of IANA-registered property names. Implementations **MUST** make sure that such a name does not violate the case-sensitivity rules defined in Section 1.7.1. If the property name is valid, then implementations **MUST NOT** treat such properties as invalid. Instead, they **MUST** preserve them in the JSContact object.

Implementations that create or update JSContact data **MUST** only set IANA-registered properties or vendor-specific properties. Preserving properties that are unknown to the implementation is to allow applications and services to interoperate without data loss, even if not all of them implement the same set of JSContact extensions.

1.7.4. Enumerated Values

Several properties in this document restrict their allowed values to a list of String values. These values are case-sensitive. If not noted otherwise for a specific property, the initial list of values for such properties is registered at IANA in the "JSContact Enum Values" registry (Section 3.7). Implementations **MUST** only set IANA-registered or vendor-specific (Section 1.8.2) values for such properties.

1.8. Vendor-Specific Extensions

Vendors may extend properties and values for experimentation or to store contacts data that is only useful for a single service or application. Such extensions are not meant for interoperation. If, instead, interoperation is desired, vendors are strongly encouraged to define and register new properties, types, and values at IANA as defined in Section 3. Section 1.7.2 defines the naming conventions for IANA-registered elements.

1.8.1. Vendor-Specific Properties

Vendor-specific property names **MUST** start with a vendor-specific prefix followed by a name, as produced by the `v-extension` ABNF below. The prefix and name together form the property name. The vendor-specific prefix **MUST** be a domain name under control of the service or application that sets the property, but it need not resolve in the Domain Name System [RFC1034] [RFC1035]. The prefix `ietf.org` and its subdomain names are reserved for IETF specifications. The name **MUST NOT** contain the `TILDE` (U+007E) and `SOLIDUS` (U+002F) characters, as these require special escaping when encoding a JSON Pointer [RFC6901] for that property.

Vendor-specific properties **MAY** be set in any JSContact object. Implementations **MUST** preserve vendor-specific properties in JSContact data, irrespective if they know their use. They **MUST NOT** reject the property value as invalid, unless they are in control of the vendor-specific property as outlined in the above paragraph.

The ABNF rule `v-extension` formally defines valid vendor-specific property names. Note that the vendor prefix allows for more values than Internationalized Domain Names (IDNs) [RFC8499]; therefore, JSContact implementations can simply validate property names without implementing the full set of rules that apply to domain names.

```
v-extension = v-prefix ":" v-name

v-prefix = v-label *("." v-label)

v-label = alnum-int / alnum-int *(alnum-int / "-") alnum-int

alnum-int = ALPHA / DIGIT / NON-ASCII
  ; see RFC 6350, Section 3.3

v-name = 1*(WSP / "!" / %x23-2e / %x30-7d / NON-ASCII)
  ; any characters except CTLs, DQUOTE, SOLIDUS, and TILDE
```

Figure 2: ABNF Rules for Vendor-Specific Property Names

The value of vendor-specific properties can be any valid JSON value, and naming restrictions do not apply to such values. Specifically, if the property value is a JSON object, then the keys of such objects need not be named as vendor-specific properties, as illustrated in Figure 3:

```
"example.com:foo": "bar",
"example.com:foo2": {
  "bar": "baz"
}
```

Figure 3: Examples of Vendor-Specific Properties

1.8.2. Vendor-Specific Values

Some JSContact IANA-registered properties allow their values to be vendor-specific. One such example is the `kind` (Section 2.1.4) property, which enumerates its standard values but also allows for arbitrary vendor-specific values. Such vendor-specific values **MUST** be valid `v-extension` values as defined in Section 1.8.1. The example in Figure 4 illustrates this:

```
"kind": "example.com:baz"
```

Figure 4: Example of a Vendor-Specific Value

Vendors are strongly encouraged to specify a new standard value once a vendor-specific one turns out to also be useful for other systems.

1.9. Versioning

Every instance of a JSContact Card (Section 2) indicates which JSContact version its IANA-registered properties and values are based on. The version is indicated both in the `version` (Section 2.1.2) property within the Card and in the version (Section 3.1) parameter of the JSContact MIME content type. All IANA-registered elements indicate the version at which they were introduced or obsoleted.

1.9.1. Version Format and Requirements

A JSContact version consists of a numeric major and minor version, separated by the `FULL STOP` character (U+002E). Later versions are numerically higher than former versions, with the major version being more significant than the minor version. A version value is produced by the following ABNF:

```
jsversion = 1*DIGIT "." 1*DIGIT
```

Figure 5

Differing major version values indicate substantial differences in JSContact semantics and format. Implementations **MUST** be prepared for property definitions and other JSContact elements that differ in a backwards-incompatible manner.

Differing minor version values indicate additions that enrich JSContact data but do not introduce backwards-incompatible changes. Typically, these are new property enum values or properties with a narrow semantic scope. A new minor version **MUST NOT** require implementations to change their processing of JSContact data. Changing the major version number resets the minor version number to zero.

1.9.2. Current Version

This specification registers JSContact version value `1.0` (Table 1).

2. Card

This section defines the JSContact object type Card. A Card stores contact information, typically that of a person, organization, or company.

Its media type is defined in Section 3.1.

Figure 6 shows a basic Card for the person "John Doe". As the object is the topmost object in the JSON data, it has the `@type` property set according to the rules defined in Section 1.3.4.

```
{
  "@type": "Card",
  "version": "1.0",
  "uid": "22B2C7DF-9120-4969-8460-05956FE6B065",
  "kind": "individual",
  "name": {
    "components": [
      { "kind": "given", "value": "John" },
      { "kind": "surname", "value": "Doe" }
    ],
    "isOrdered": true
  }
}
```

Figure 6: Example of a Basic `Card`

2.1. Metadata Properties

This section defines properties about this instance of a Card such as its unique identifier, its creation date, and how it relates to other Cards and other metadata information.

2.1.1. @type

Type: `String` (mandatory)

This **MUST** be `Card`, if set.

2.1.2. version

Type: `String` (mandatory)

This specifies the JSContact version used to define the Card. The value **MUST** be one of the IANA-registered JSContact Enum Values for the `version` property. Also see Section 1.9.2.

```
"version": "1.0"
```

Figure 7: `version` Example

2.1.3. created

Type: `UTCDateTime` (optional)

The date and time when the Card was created.

```
"created": "2022-09-30T14:35:10Z"
```

Figure 8: `created` Example

2.1.4. kind

Type: `String` (optional; default: `individual`)

The kind of the entity the Card represents.

The enumerated (Section 1.7.4) values are:

- `individual`: a single person
- `group`: a group of people or entities
- `org`: an organization
- `location`: a named location
- `device`: a device such as an appliance, a computer, or a network element
- `application`: a software application

```
"kind": "individual"
```

Figure 9: `kind` Example

2.1.5. language

Type: `String` (optional)

This is the language tag, as defined in [RFC5646], that best describes the language used for text in the Card, optionally including additional information such as the script. Note that values **MAY** be localized in the `localizations` (Section 2.7.1) property.

```
"language": "de-AT"
```

Figure 10: `language` Example

2.1.6. members

Type: `String[Boolean]` (optional)

This identifies the set of Cards that are members of this group Card. Each key in the set is the `uid` property value of the member, and each boolean value **MUST** be `true`. If this property is set, then the value of the `kind` property **MUST** be `group`.

The opposite is not true. A group Card will usually contain the `members` property to specify the members of the group, but it is not required to. A group Card without the `members` property can be considered an abstract grouping or one whose members are known empirically (e.g., "IETF Participants").

```
"kind": "group",
"name": {
  "full": "The Doe family"
},
"uid": "urn:uuid:ab4310aa-fa43-11e9-8f0b-362b9e155667",
"members": {
  "urn:uuid:03a0e51f-d1aa-4385-8a53-e29025acd8af": true,
  "urn:uuid:b8767877-b4a1-4c70-9acc-505d3819e519": true
}
```

Figure 11: `members` Example

2.1.7. prodId

Type: `String` (optional)

The identifier for the product that created the Card. If set, the value **MUST** be at least one character long.

```
"prodId": "ACME Contacts App version 1.23.5"
```

Figure 12: `prodId` Example

2.1.8. relatedTo

Type: `String[Relation]` (optional)

This relates the object to other Cards. It is represented as a map, where each key is the `uid` of the related Card, and the value defines the relation. The Relation object has the following properties:

@type: `String`. This **MUST** be `Relation`, if set.

relation: `String[Boolean]` (optional; default: empty Object). Describes how the linked object is related to the linking object. The relation is defined as a set of relation types. The key in the set defines the relation type; the value for each key in the set **MUST** be `true`. The relationship between the two objects is undefined if the set is empty.

The initial list of enumerated (Section 1.7.4) relation types matches the IANA-registered TYPE [IANA-vCard] parameter values of the vCard RELATED property (Section 6.6.6 of [RFC6350]):

- `acquaintance`
- `agent`
- `child`
- `co-resident`
- `co-worker`
- `colleague`
- `contact`

- `crush`
- `date`
- `emergency`
- `friend`
- `kin`
- `me`
- `met`
- `muse`
- `neighbor`
- `parent`
- `sibling`
- `spouse`
- `sweetheart`

```
"relatedTo": {
  "urn:uuid:f81d4fae-7dec-11d0-a765-00a0c91e6bf6": {
    "relation": {
      "friend": true
    }
  },
  "8cacdfb7d1ffdb59@example.com": {
    "relation": {}
  }
}
```

Figure 13: `relatedTo` Example

2.1.9. uid

Type: `String` (mandatory)

An identifier that is used to associate the object as the same across different systems, address books, and views. The value **SHOULD** be a URN [RFC8141], but for compatibility with [RFC6350], it **MAY** also be a URI [RFC3986] or free-text value. The value of the URN **SHOULD** be in the `uuid` namespace [RFC4122]. As of this writing, a revision [UUID] of the Universally Unique Identifier (UUID) Standards Track document [RFC4122] is in progress and will likely introduce new UUID versions and best practices to generate global unique identifiers. Implementors **SHOULD** follow any recommendations described there. Until then, implementations **SHOULD** generate identifiers using the random or pseudorandom UUID version described in Section 4.4 of [RFC4122].

```
"uid": "urn:uuid:f81d4fae-7dec-11d0-a765-00a0c91e6bf6"
```

Figure 14: `uid` Example

2.1.10. updated

Type: `UTCDateTime` (optional)

The date and time when the data in the Card was last modified.

```
"updated": "2021-10-31T22:27:10Z"
```

Figure 15: `updated` Example

2.2. Name and Organization Properties

This section defines properties that name the entity represented by the Card and its related organizations and roles. It also describes how to refer to the entity represented by the Card in spoken or written language.

2.2.1. name

Type: `Name` (optional)

The name of the entity represented by the Card. This can be any type of name, e.g., it can, but need not, be the legal name of a person.

2.2.1.1. Name Object

A Name object has the following properties:

@type: `String`. This **MUST** be `Name`, if set.

components: `NameComponent[]` (optional). The components (Section 2.2.1.2) making up this name. This property **MUST** be set if the `full` property is not set; otherwise, it **SHOULD** be set. The component list **MUST** have at least one entry having a different `kind` than `separator`.

Name components **SHOULD** be ordered such that when their values are joined as a String, a valid full name of the entity is produced. If so, implementations **MUST** set the `isOrdered` property value to `true`.

If the name components are ordered, then the `defaultSeparator` property and name components of kind `separator` give guidance on what characters to insert between components, but implementations are free to choose any others. When lacking a separator, inserting a single space character in between the name component values is a good choice.

If, instead, the name components follow no particular order, then the `isOrdered` property value **MUST** be `false`, the `components` property **MUST NOT** contain a NameComponent of kind `separator`, and the `defaultSeparator` property **MUST NOT** be set.

Figure 16 shows an example for the name "Vincent van Gogh". Note how a single name component value may consist of multiple words.

```
"name": {
  "components": [
    { "kind": "given", "value": "Vincent" },
    { "kind": "surname", "value": "van Gogh" }
  ],
  "isOrdered": true
}
```

Figure 16: Example of a Surname with Two Words

Figure 17 illustrates a name with a second surname such as a Spanish name. Additional examples are shown in Figures 19 and 39.

```
"name": {
  "components": [
    { "kind": "given", "value": "Diego" },
    { "kind": "surname", "value": "Rivera" },
    { "kind": "surname2", "value": "Barrientos" }
  ],
  "isOrdered": true
}
```

Figure 17: Example of a Second Surname

isOrdered: `Boolean` (optional; default: `false`). Indicates if the name component sequence in the `components` property is ordered.

defaultSeparator: `String` (optional). The default separator to insert between name component values when concatenating all name component values to a single String. Also see the definition of the `separator` kind for the NameComponent (Section 2.2.1.2) object. This property **MUST NOT** be set if the Name `isOrdered` property value is `false` or if the `components` property is not set.

full: `String` (optional). The full name representation of the Name. This property **MUST** be set if the `components` property is not set.

```
"full": "Mr. John Q. Public, Esq."
```

Figure 18: `full` Example

sortAs: `String[String]` (optional). Defines how the name lexicographically sorts in relation to other names when compared by a name component type. The key in the map defines the name component type. The value for that key defines the verbatim string to compare when sorting by the name component type. Absence of a key indicates that the name component

type **SHOULD NOT** be considered during sort. Sorting by that missing name component type, or if the `sortAs` property is not set, is implementation-specific. This property **MUST NOT** be set if the `components` property is not set.

Each key in the map **MUST** be a valid name component type value as defined for the `kind` property of the NameComponent object (see below). For each key in the map, there **MUST** exist at least one NameComponent object that has the type in the `components` property of the name.

Figure 19 illustrates the use of sortAs. The property value indicates that the middle name followed by both surnames should be used when sorting the name by surname. The absence of `middle` indicates that the middle name on its own should be disregarded during sort. Even though the name only contains one name component for the given name, the sortAs property still explicitly defines how to sort by the given name; otherwise, sorting by it would be undefined.

phoneticScript: `String` (optional). The script used in the value of the NameComponent `phonetic` property. See Section 1.5.5 for more information and Figure 20 for an example.

phoneticSystem: `String` (optional). The phonetic system used in the NameComponent `phonetic` property. See Section 1.5.5 for more information and Figure 20 for an example.

```
"name": {
  "components": [
    { "kind": "given", "value": "Robert" },
    { "kind": "given2", "value": "Pau" },
    { "kind": "surname", "value": "Shou Chang" }
  ],
  "sortAs": {
    "surname": "Pau Shou Chang",
    "given": "Robert"
  },
  "isOrdered": true
}
```

Figure 19: `sortAs` Example

```
{
  "@type": "Card",
  "language": "zh-Hant",
  "name": {
    "components": [
      { "kind": "surname", "value": "孫" },
      { "kind": "given", "value": "中山" },
      { "kind": "given2", "value": "文" },
      { "kind": "given2", "value": "逸仙" }
    ]
  },
  "localizations": {
    "yue": {
      "name/phoneticSystem": "jyut",
      "name/phoneticScript": "Latn",
      "name/components/0/phonetic": "syun1",
      "name/components/1/phonetic": "zung1saan1",
      "name/components/2/phonetic": "man4",
      "name/components/3/phonetic": "jat6sin1"
    }
  }
}
```

Figure 20: `phonetic` and `localizations` Example

2.2.1.2. NameComponent

A NameComponent object has the following properties:

@type: `String`. This **MUST** be `NameComponent`, if set.

value: `String` (mandatory). The value of the name component. This can be composed of one or multiple words such as "Poe" or "van Gogh".

kind: `String` (mandatory). The kind of the name component. The enumerated (Section 1.7.4) values are:

- `title`: an honorific title or prefix, e.g., "Mr.", "Ms.", or "Dr.".
- `given`: a given name, also known as "first name" or "personal name".
- `given2`: a name that appears between the given and surname such as a middle name or patronymic name.
- `surname`: a surname, also known as "last name" or "family name".
- `surname2`: a secondary surname (used in some cultures), also known as "maternal surname".
- `credential`: a credential, also known as "accreditation qualifier" or "honorific suffix", e.g., "B.A.", "Esq.".
- `generation`: a generation marker or qualifier, e.g., "Jr." or "III".

- `separator`: a formatting separator between two ordered name non-separator components. The `value` property of the component includes the verbatim separator, for example, a hyphen character or even an empty string. This value has higher precedence than the `defaultSeparator` property of the Name. Implementations **MUST NOT** insert two consecutive separator components; instead, they **SHOULD** insert a single separator component with the combined value. This component kind **MUST NOT** be set if the Name `isOrdered` property value is `false`.

phonetic: `String` (optional). Defines how to pronounce the name component. If this property is set, then at least one of the Name object properties, `phoneticSystem` or `phoneticScript`, **MUST** be set. Also see Section 1.5.5.

2.2.1.3. nicknames

Type: `Id[Nickname]` (optional)

The nicknames of the entity represented by the Card. A Nickname object has the following properties:

@type: `String`. This **MUST** be `Nickname`, if set.

name: `String` (mandatory). The nickname.

contexts: `String[Boolean]` (optional). The contexts in which to use the nickname. Also see Section 1.5.1.

pref: `UnsignedInt` (optional). The preference of the nickname in relation to other nicknames. Also see Section 1.5.4.

```
"nicknames": {
  "k391": {
    "name": "Johnny"
  }
}
```

Figure 21: `nicknames` Example

2.2.2. organizations

Type: `Id[Organization]` (optional)

The company or organization names and units associated with the Card. An Organization object has the following properties, of which at least one of the `name` and `units` properties **MUST** be set:

@type: `String`. This **MUST** be `Organization`, if set.

name: `String` (optional). The name of the organization.

units: `OrgUnit[]` (optional). A list of organizational units, ordered as descending by hierarchy (e.g., a geographic or functional division sorts before a department within that division). If set, the list **MUST** contain at least one entry.

sortAs: `String` (optional). Defines how the organization name lexicographically sorts in relation to other organizations when compared by the name. The value defines the verbatim string value to compare. In absence of this property, the `name` property value **MAY** be used for comparison.

contexts: `String[Boolean]` (optional). The contexts in which association with the organization apply. For example, membership in a choir may only apply in a private context. Also see Section 1.5.1.

An OrgUnit object has the following properties:

@type: `String`. This **MUST** be `OrgUnit`, if set.

name: `String` (mandatory). The name of the organizational unit.

sortAs: `String` (optional). Defines how the organization unit name lexicographically sorts in relation to other organizational units of the same level when compared by the name. The level is defined by the array index of the organizational unit in the `units` property of the Organization object. The property value defines the verbatim string value to compare. In absence of this property, the `name` property value **MAY** be used for comparison.

```
"organizations": {
  "o1": {
    "name": "ABC, Inc.",
    "units": [
      { "name": "North American Division" },
      { "name": "Marketing" }
    ],
    "sortAs": "ABC"
  }
}
```

Figure 22: `organizations` Example

2.2.3. speakToAs

Type: `SpeakToAs` (optional)

Provides information on how to address, speak to, or refer to the entity that is represented by the Card. A SpeakToAs object has the following properties, of which at least one of the `grammaticalGender` and `pronouns` properties **MUST** be set:

@type: `String`. This **MUST** be `SpeakToAs`, if set.

grammaticalGender: `String` (optional). Defines which grammatical gender to use in salutations and other grammatical constructs. For example, the German language distinguishes by grammatical gender in salutations such as "Sehr geehrte" (feminine) and "Sehr geehrter" (masculine). The enumerated (Section 1.7.4) values are:

- `animate`
- `common`
- `feminine`
- `inanimate`
- `masculine`
- `neuter`

Note that the grammatical gender does not allow inferring the gender identities or assigned sex of the contact.

pronouns: `Id[Pronouns]` (optional). Defines the pronouns that the contact chooses to use for themselves.

A Pronouns object has the following properties:

@type: `String`. This **MUST** be `Pronouns`, if set.

pronouns: `String` (mandatory). Defines the pronouns. Any value or form is allowed. Examples in English include `she/her` and `they/them/theirs`. The value **MAY** be overridden in the `localizations` (Section 2.7.1) property.

contexts: `String[Boolean]` (optional). The contexts in which to use the pronouns. Also see Section 1.5.1.

pref: `UnsignedInt` (optional). The preference of the pronouns in relation to other pronouns in the same context. Also see Section 1.5.4.

```
"speakToAs": {
  "grammaticalGender": "neuter",
  "pronouns": {
    "k19": {
      "pronouns": "they/them",
      "pref": 2
    },
    "k32": {
      "pronouns": "xe/xir",
      "pref": 1
    }
  }
}
```

Figure 23: `speakToAs` Example

2.2.4. titles

Type: `Id[Title]` (optional)

The job titles or functional positions of the entity represented by the Card. A Title object has the following properties:

@type: `String`. This **MUST** be `Title`, if set.

name: `String` (mandatory). The title or role name of the entity represented by the Card.

kind: `String` (optional; default: `title`). Describes the organizational or situational kind of the title. Some organizations and individuals distinguish between *titles* as organizational positions and *roles* as more temporary assignments such as in project management.

The enumerated (Section 1.7.4) values are:

- `title`
- `role`

organizationId: `Id` (optional). The identifier of the organization in which this title is held.

```
"titles": {
  "le9": {
    "kind": "title",
    "name": "Research Scientist"
  },
  "k2": {
    "kind": "role",
    "name": "Project Leader",
    "organizationId": "o2"
  }
},
"organizations": {
  "o2": {
    "name": "ABC, Inc."
  }
}
```

Figure 24: `titles` Example

2.3. Contact Properties

This section defines how properties contact the entity represented by the Card.

2.3.1. emails

Type: `Id[EmailAddress]` (optional)

The email addresses in which to contact the entity represented by the Card. An EmailAddress object has the following properties:

@type: `String`. This **MUST** be `EmailAddress`, if set.

address: `String` (mandatory). The email address. This **MUST** be an *addr-spec* value as defined in Section 3.4.1 of [RFC5322].

contexts: `String[Boolean]` (optional). The contexts in which to use this email address. Also see Section 1.5.1.

pref: `UnsignedInt` (optional). The preference of the email address in relation to other email addresses. Also see Section 1.5.4.

label: `String` (optional). A custom label for the value. Also see Section 1.5.3.

```
"emails": {
  "e1": {
    "contexts": {
      "work": true
    },
    "address": "jqpublic@xyz.example.com"
  },
  "e2": {
    "address": "jane_doe@example.com",
    "pref": 1
  }
}
```

Figure 25: `emails` Example

2.3.2. onlineServices

Type: `Id[OnlineService]` (optional)

The online services that are associated with the entity represented by the Card. This can be messaging services, social media profiles, and other. An OnlineService object has the following properties, of which at least the `uri` or `user` property **MUST** be set:

@type: `String`. This **MUST** be `OnlineService`, if set.

service: `String` (optional). The name of the online service or protocol. The name **MAY** be capitalized the same as on the service's website, app, or publishing material, but names **MUST** be considered equal if they match case-insensitively. Examples are `GitHub`, `Kakao`, and `Mastodon`.

uri: `String` (optional). Identifies the entity represented by the Card at the online service. This **MUST** be a *URI* as defined in Section 3 of [RFC3986].

user: `String` (optional). Names the entity represented by the Card at the online service. Any free-text value is allowed. The `service` property **SHOULD** be set.

contexts: `String[Boolean]` (optional). The contexts in which to use the service. Also see Section 1.5.1.

pref: `UnsignedInt` (optional). The preference of the service in relation to other services. Also see Section 1.5.4.

label: `String` (optional). A custom label for the value. Also see Section 1.5.3.

```
"onlineServices": {
  "x1": {
    "uri": "xmpp:alice@example.com"
  },
  "x2": {
    "service": "Mastodon",
    "user": "@alice@example2.com",
    "uri": "https://example2.com/@alice"
  }
}
```

Figure 26: `onlineServices` Example

2.3.3. phones

Type: `Id[Phone]` (optional)

The phone numbers in which to contact the entity represented by the Card. A Phone object has the following properties:

@type: `String`. This **MUST** be `Phone`, if set.

number: `String` (mandatory). The phone number as either a URI or free text. Typical URI schemes are `tel` [RFC3966] or `sip` [RFC3261], but any URI scheme is allowed.

features: `String[Boolean]` (optional). The set of contact features that the phone number may be used for. The set is represented as an object, with each key being a method type. The boolean value **MUST** be `true`. The enumerated (Section 1.7.4) method type values are:

- `mobile`: the number for a mobile phone.
- `voice`: the number for calling by voice.
- `text`: the number that supports text messages (SMS).
- `video`: the number that supports video conferencing.
- `main-number`: the main phone number such as the number of the front desk at a company, as opposed to a direct-dial number of an individual employee.
- `textphone`: the number is for a device for people with hearing or speech difficulties.
- `fax`: the number for sending faxes.

- `pager`: the number for a pager or beeper.

contexts: `String[Boolean]` (optional). The contexts in which to use the number. Also see Section 1.5.1.

pref: `UnsignedInt` (optional). The preference of the number in relation to other numbers. Also see Section 1.5.4.

label: `String` (optional). A custom label for the value. Also see Section 1.5.3.

```
"phones": {
  "tel0": {
    "contexts": {
      "private": true
    },
    "features": {
      "voice": true
    },
    "number": "tel:+1-555-555-5555;ext=5555",
    "pref": 1
  },
  "tel3": {
    "contexts": {
      "work": true
    },
    "number": "tel:+1-201-555-0123"
  }
}
```

Figure 27: phones Example

2.3.4. preferredLanguages

Type : `Id[LanguagePref]` (optional)

Defines the preferred languages for contacting the entity associated with the Card.

A LanguagePref object has the following properties:

@type: `String`. This **MUST** be `LanguagePref`, if set.

language: `String` (mandatory). The preferred language. This **MUST** be a language tag as defined in [RFC5646].

contexts: `String[Boolean]` (optional). Defines the contexts in which to use the language. Also see Section 1.5.1.

pref: `UnsignedInt` (optional). Defines the preference of the language in relation to other languages of the same contexts. Also see Section 1.5.4.

```
"preferredLanguages": {
  "l1": {
    "language": "en",
    "contexts": {
      "work": true
    },
    "pref": 1
  },
  "l2": {
    "language": "fr",
    "contexts": {
      "work": true
    },
    "pref": 2
  },
  "l3": {
    "language": "fr",
    "contexts": {
      "private": true
    }
  }
}
```

Figure 28: `preferredLanguages` Example

2.4. Calendaring and Scheduling Properties

This section defines properties for scheduling calendar events with the entity represented by the Card.

2.4.1. calendars

Type: `Id[Calendar]` (optional)

These are resources for calendaring such as using calendars to look up free-busy information for the entity represented by the Card. A Calendar object has all properties of the Resource (Section 1.4.4) data type, with the following additional definitions:

- The `@type` property value **MUST** be `Calendar`, if set.
- The `kind` property is mandatory. Its enumerated (Section 1.7.4) values are:
  - `calendar`: The resource is a calendar that contains entries such as calendar events or tasks.
  - `freeBusy`: The resource allows for free-busy lookups, for example, to schedule group events.

```
"calendars": {
  "calA": {
    "kind": "calendar",
    "uri": "webcal://calendar.example.com/calA.ics"
  },
  "project-a": {
    "kind": "freeBusy",
    "uri": "https://calendar.example.com/busy/project-a"
  }
}
```

Figure 29: `calendars` Example

2.4.2. schedulingAddresses

Type: `Id[SchedulingAddress]` (optional)

The scheduling addresses by which the entity may receive calendar scheduling invitations. A SchedulingAddress object has the following properties:

@type: `String`. This **MUST** be `SchedulingAddress`, if set.

uri: `String` (mandatory). The address to use for calendar scheduling with the contact. This **MUST** be a URI as defined in Section 3 of [RFC3986].

contexts: `String[Boolean]` (optional). The contexts in which to use the scheduling address. Also see Section 1.5.1.

pref: `UnsignedInt` (optional). The preference of the scheduling address in relation to other scheduling addresses. Also see Section 1.5.4.

label: `String` (optional). A custom label for the scheduling address. Also see Section 1.5.3.

```
"schedulingAddresses": {
  "sched1": {
    "uri": "mailto:janedoe@example.com"
  }
}
```

Figure 30: `schedulingAddresses` Example

2.5. Address and Location Properties

This section defines properties for postal addresses and geographical locations associated with the entity represented by the Card.

2.5.1. addresses

Type: `Id[Address]` (optional)

A map of address identifiers to Address objects, containing physical locations.

2.5.1.1. Address Object

An Address object has the following properties:

@type: `String`. This **MUST** be `Address`, if set.

components: `AddressComponent[]` (optional). The components (Section 2.5.1.2) that make up the address. This property **MUST** be set if the `full` property is not set; otherwise, it **SHOULD** be set. The component list **MUST** have at least one entry that has a `kind` other than `separator`.

Address components **SHOULD** be ordered such that when their values are joined as a String, a valid full address is produced. If so, implementations **MUST** set the `isOrdered` property value to `true`.

If the address components are ordered, then the `defaultSeparator` property and address components of kind `separator` give guidance on what characters to insert between components, but implementations are free to choose any others. When lacking a separator, inserting a single space character in between address component values is a good choice.

If, instead, the address components follow no particular order, then the `isOrdered` property value **MUST** be `false`, the `components` property **MUST NOT** contain an AddressComponent of kind `separator`, and the `defaultSeparator` property **MUST NOT** be set.

isOrdered: `Boolean` (optional; default: `false`). Indicates if the address component sequence in the `components` property is ordered.

countryCode: `String` (optional). The Alpha-2 country code [ISO.3166-1].

coordinates: `String` (optional). A "geo:" URI [RFC5870] for the address.

timeZone: `String` (optional). Identifies the time zone the address is located in. This **MUST** be a time zone name registered in IANA's Time Zone Database [IANA-TZ].

contexts: `String[Boolean]` (optional). The contexts of the address information. The boolean value **MUST** be `true`. In addition to the common contexts (Section 1.5.1), allowed key values are:

- `billing`: an address to be used for billing.
- `delivery`: an address to be used for delivering physical items.

full: `String` (optional). The full address, including street, region, or country. The purpose of this property is to define an address, even if the individual address components are not known. If the `street` property is set, the `full` property **SHOULD NOT** be set.

defaultSeparator: `String` (optional). The default separator to insert between address component values when concatenating all address component values to a single String. Also see the definition of the `separator` kind for the AddressComponent (Section 2.5.1.2) object. This property **MUST NOT** be set if the Address `isOrdered` property value is `false` or if the `components` property is not set.

pref: `UnsignedInt` (optional). The preference of the address in relation to other addresses. Also see Section 1.5.4.

phoneticScript: `String` (optional). The script used in the value of the AddressComponent `phonetic` property. Also see Section 1.5.5.

phoneticSystem: `String` (optional). The phonetic system used in the AddressComponent `phonetic` property. Also see Section 1.5.5.

The following example illustrates the use of the `address` property. Additional examples are shown in Section 2.5.1.3.

```
"addresses": {
  "k23": {
    "contexts": {
      "work": true
    },
    "components": [
      { "kind": "number", "value": "54321" },
      { "kind": "separator", "value": " " },
      { "kind": "name", "value": "Oak St" },
      { "kind": "locality", "value": "Reston" },
      { "kind": "region", "value": "VA" },
      { "kind": "separator", "value": " " },
      { "kind": "postcode", "value": "20190" },
      { "kind": "country", "value": "USA" }
    ],
    "countryCode": "US",
    "defaultSeparator": ", ",
    "isOrdered": true
  }
}
```

Figure 31: Example of the Address "54321 Oak St, Reston, VA 20190, USA"

2.5.1.2. AddressComponent Object

An AddressComponent object has the following properties:

@type: `String`. This **MUST** be `AddressComponent`, if set.

value: `String` (mandatory). The value of the address component.

kind: `String` (mandatory). The kind of the address component. The enumerated (Section 1.7.4) values are:

- `room`: the room, suite number, or identifier.
- `apartment`: the extension designation such as the apartment number, unit, or box number.
- `floor`: the floor or level the address is located on.
- `building`: the building, tower, or condominium the address is located in.
- `number`: the street number, e.g., "123". This value is not restricted to numeric values and can include any value such as number ranges ("112-10"), grid style ("39.2 RD"), alphanumerics ("N6W23001"), or fractionals ("123 1/2").
- `name`: the street name.
- `block`: the block name or number.
- `subdistrict`: the subdistrict, ward, or other subunit of a district.
- `district`: the district name.
- `locality`: the municipality, city, town, village, post town, or other locality.
- `region`: the administrative area such as province, state, prefecture, county, or canton.
- `postcode`: the postal code, post code, ZIP code, or other short code associated with the address by the relevant country's postal system.
- `country`: the country name.
- `direction`: the cardinal direction or quadrant, e.g., "north".
- `landmark`: the publicly known prominent feature that can substitute the street name and number, e.g., "White House" or "Taj Mahal".
- `postOfficeBox`: the post office box number or identifier.
- `separator`: a formatting separator between two ordered address non-separator components. The `value` property of the component includes the verbatim separator, for example, a hyphen character or even an empty string. This value has higher precedence than the `defaultSeparator` property of the Address. Implementations **MUST NOT** insert two consecutive separator components; instead, they **SHOULD** insert a single separator component with the combined value. This component kind **MUST NOT** be set if the Address `isOrdered` property value is `false`.

phonetic: `String` (optional). Defines how to pronounce the name component. If this property is set, then at least one of the Address object `phoneticSystem` or `phoneticScript` properties **MUST** be set. Also see Section 1.5.5.

2.5.1.3. Address Examples

Examples of addresses are shown below; also see Figure 31.

```
"addresses": {
  "k25": {
    "components": [
      { "kind": "number", "value": "46" },
      { "kind": "name", "value": "1 Sukhumvit 51 Alley" },
      { "kind": "subdistrict", "value": "Khlong Tan Nuea" },
      { "kind": "district", "value": " Watthana" },
      { "kind": "locality", "value": "Bangkok" },
      { "kind": "country", "value": "Thailand" },
      { "kind": "postcode", "value": "10110" }
    ],
    "defaultSeparator": ", ",
    "isOrdered": true
  }
}
```

Figure 32: Example of the Address "46, 1 Sukhumvit 51 Alley, Khlong Tan Nuea, Watthana, Bangkok 10110, Thailand"

The following example illustrates the use of an address in Tokyo and its localization (Section 2.7.1) in Japanese.

```
"addresses": {
   "k26": {
     "components": [
       { "kind": "block", "value": "2-7" },
       { "kind": "separator", "value": "-" },
       { "kind": "number", "value": "2" },
       { "kind": "separator", "value": " " },
       { "kind": "district", "value": "Marunouchi" },
       { "kind": "locality", "value": "Chiyoda-ku" },
       { "kind": "region", "value": "Tokyo" },
       { "kind": "separator", "value": " " },
       { "kind": "postcode", "value": "100-8994" }
     ],
     "defaultSeparator": ", ",
     "full": "2-7-2 Marunouchi, Chiyoda-ku, Tokyo 100-8994",
     "isOrdered": true
   }
},
"localizations": {
  "jp": {
    "addresses/k26": {
      "components": [
        { "kind": "region", "value": "東京都" },
        { "kind": "locality", "value": "千代田区" },
        { "kind": "district", "value": "丸ノ内" },
        { "kind": "block", "value": "2-7" },
        { "kind": "separator", "value": "-" },
        { "kind": "number", "value": "2" },
        { "kind": "postcode", "value": "〒100-8994" }
      ],
      "defaultSeparator": "",
      "full": "〒100-8994東京都千代田区丸ノ内2-7-2",
      "isOrdered": true
    }
  }
}
```

Figure 33: Example of an Address in Tokyo and Its Localization in Japanese

2.6. Resource Properties

This section defines properties for digital resources associated with the entity represented by the Card.

2.6.1. cryptoKeys

Type: `Id[CryptoKey]` (optional)

These are cryptographic resources such as public keys and certificates associated with the entity represented by the Card. A CryptoKey object has all properties of the Resource (Section 1.4.4) data type, with the following additional definition:

- The `@type` property value **MUST** be `CryptoKey`, if set.

The following example shows how to refer to an external cryptographic resource.

```
"cryptoKeys": {
  "mykey1": {
    "uri": "https://www.example.com/keys/jdoe.cer"
  }
}
```

Figure 34: Example of `cryptoKeys` with External Data

The following example shows how to embed key data in the CryptoKey. The key data is depicted in multiple lines only for demonstration purposes.

```
"cryptoKeys": {
  "mykey2": {
    "uri": "data:application/pgp-keys;base64,LS0tLS1CRUdJTiBSU0EgUFVC
            TElDIEtFWS0tLS0tCk1JSUJDZ0tDQVFFQSt4R1ovd2N6OXVnRnBQMDdOc
            3BvNlUxN2wwWWhGaUZweHhVNHBUazNMaWZ6OVIzenNJc3UKRVJ3dGE3K2
            ZXSWZ4T28yMDhldHQvamhza2lWb2RTRXQzUUJHaDRYQmlweVdvcEt3Wjk
            zSEhhRFZaQUFMaS8yQQoreFRCdFdkRW83WEdVdWpLRHZDMi9hWkt1a2Zq
            cE9pVUk4QWhMQWZqbWxjRC9VWjFRUGgwbUhzZ2xSTkNtcEN3Cm13U1hBO
            VZObWh6K1BpQitEbWw0V1duS1cvVkhvMnVqVFh4cTcrZWZNVTRIMmZueT
            NTZTNLWU9zRlBGR1oxVE4KUVNZbEZ1U2hXckhQdGlMbVVkUG9QNkNWMm1
            NTDF0aytsN0RJSXFYclFoTFVLREFDZU01cm9NeDBrTGhVV0I4UAorMHVq
            MUNObE5ONEpSWmxDN3hGZnFpTWJGUlU5WjRONll3SURBUUFCCi0tLS0tR
            U5EIFJTQSBQVUJMSUMgS0VZLS0tLS0K"
  }
}
```

Figure 35: Example of `cryptoKeys` with Embedded Data

2.6.2. directories

Type: `Id[Directory]` (optional)

These are directory service resources such as entries in a directory or organizational directories for lookup. A Directory object has all properties of the Resource (Section 1.4.4) data type, with the following additional definitions:

- The `@type` property value **MUST** be `Directory`, if set.
- The `kind` property is mandatory. Its enumerated (Section 1.7.4) values are:
  - `directory`: the resource is a directory service that the entity represented by the Card is a part of. This typically is an organizational directory that also contains associated entities, e.g., co-workers and management in a company directory.
  - `entry`: the resource is a directory entry of the entity represented by the Card. In contrast to the `directory` type, this is the specific URI for the entity *within* a directory.

In addition, the Directory object has the following property:

listAs: `UnsignedInt` (optional). Defines the position of the directory resource in the list of all Directory objects having the same `kind` in the Card. If set, the `listAs` value **MUST** be higher than zero. Multiple directory resources **MAY** have the same `listAs` property value or none. Sorting such entries is implementation-specific.

```
"directories": {
  "dir1": {
    "kind": "entry",
    "uri": "https://dir.example.com/addrbook/jdoe/Jean%20Dupont.vcf"
  },
  "dir2": {
    "kind": "directory",
    "uri": "ldap://ldap.example/o=Example%20Tech,ou=Engineering",
    "pref": 1
  }
```

Figure 36: `directories` Example

2.6.3. links

Type: `Id[Link]` (optional)

These are links to resources that do not fit any of the other use-case-specific resource properties. A Link object has all properties of the Resource (Section 1.4.4) data type, with the following additional definitions:

- The `@type` property value **MUST** be `Link`, if set.
- The `kind` property is optional. Its enumerated (Section 1.7.4) values are:
  - `contact`: the resource is a URI by which the entity represented by the Card may be contacted; this includes web forms or other media that require user interaction.

```
"links": {
  "link3": {
    "kind": "contact",
    "uri": "mailto:contact@example.com",
    "pref": 1
  }
}
```

Figure 37: `links` Example

2.6.4. media

Type: `Id[Media]` (optional)

These are media resources such as photographs, avatars, or sounds that are associated with the entity represented by the Card. A Media object has all properties of the Resource (Section 1.4.4) data type, with the following additional definitions:

- The `@type` property value **MUST** be `Media`, if set.
- The `kind` property is mandatory. Its enumerated (Section 1.7.4) values are:
  - `photo`: the resource is a photograph or avatar.
  - `sound`: the resource is audio media, e.g., to specify the proper pronunciation of the name property contents.
  - `logo`: the resource is a graphic image or logo associated with the entity represented by the Card.

```
"media": {
  "res45": {
    "kind": "sound",
    "uri": "CID:JOHNQ.part8.19960229T080000.xyzMail@example.com"
  },
  "res47": {
    "kind": "logo",
    "uri": "https://www.example.com/pub/logos/abccorp.jpg"
  },
  "res1": {
    "kind": "photo",
    "uri": "..."
  }
}
```

Figure 38: `media` Example

2.7. Multilingual Properties

This section defines properties for localizing the content of the Card in human languages.

2.7.1. localizations

Type: String[PatchObject] (optional)

This localizes property values to languages (other than the main language) in the Card. Localizations provide language-specific alternatives for existing property values and **SHOULD NOT** add new properties.

The keys in the localizations property object are language tags [RFC5646]. The values are patch objects that localize the Card in the respective language tag. The paths in the PatchObject are relative to the Card that includes the `localizations` property. A patch **MUST NOT** target the `localizations` property.

Conceptually, a Card is localized as follows:

- Determine the language tag in which the Card should be localized.

- If the localizations property includes a key for that language, obtain the PatchObject value. If there is no such key, stop.
- Create a copy of the Card, but do not copy the localizations property.
- Apply all patches in the PatchObject to the copy of the Card.
- Optionally, set the `language` property in the copy of the Card.
- Use the patched copy of the Card as the localized variant of the original Card.

A patch in the PatchObject may contain any value type. Its value **MUST** be a valid value according to the definition of the patched property.

Figure 39 localizes the `name` property by completely replacing its contents in Ukrainian language with Cyrillic script.

```
{
  "name": {
    "components": [
      { "kind": "title", "value": "Mr." },
      { "kind": "given", "value": "Ivan" },
      { "kind": "given2", "value": "Petrovich" },
      { "kind": "surname", "value": "Vasiliev" }
    ]
  },
  "localizations": {
    "uk-Cyrl": {
      "name": {
        "components": [
          { "kind": "title", "value": "г-н" },
          { "kind": "given", "value": "Иван" },
          { "kind": "given2", "value": "Петрович" },
          { "kind": "surname", "value": "Васильев" }
        ]
      }
    }
  }
}
```

Figure 39: Example of Localizing a Top-Level Property

Figure 40 localizes the title name by patching *inside* the `titles` property. All properties, except the `name` property in the Title object, are left as is.

```
"name": {
  "full": "Gabriel García Márquez"
},
"titles": {
  "t1": {
    "kind": "title",
    "name": "novelist"
  }
},
"localizations": {
  "es": {
    "titles/t1/name": "autor"
  }
}
```

Figure 40: Example of Localizing a Nested Property

2.8. Additional Properties

This section defines properties for which none of the previous sections are appropriate.

2.8.1. anniversaries

Type: Id[Anniversary] (optional)

These are memorable dates and events for the entity represented by the Card. An Anniversary object has the following properties:

@type: `String`. This **MUST** be `Anniversary`, if set.

kind: `String` (mandatory). Specifies the kind of anniversary. The enumerated (Section 1.7.4) values are:

- `birth`: a birthday anniversary
- `death`: a deathday anniversary
- `wedding`: a wedding day anniversary

date: `PartialDate|Timestamp` (mandatory; defaultType: `PartialDate`). The date of the anniversary in the Gregorian calendar. This **MUST** be either a whole or partial calendar date or a complete UTC timestamp (see the definition of the Timestamp and PartialDate object types below).

place: Address (optional). An address associated with this anniversary, e.g., the place of birth or death.

A PartialDate object represents a complete or partial calendar date in the Gregorian calendar. It represents a complete date, a year, a month in a year, or a day in a month. It has the following properties, of which at least `year` or `month` and `day` **MUST** be set:

@type: `String`. This **MUST** be `PartialDate`, if set.

year: `UnsignedInt` (optional). The calendar year.

month: `UnsignedInt` (optional). The calendar month, represented as the integers 1 <= month <= 12. If this property is set, then either `year` or `day` **MUST** be set.

day: `UnsignedInt` (optional). The calendar month day, represented as the integers 1 <= day <= 31, depending on the validity within the month and year. If this property is set, then `month` **MUST** be set.

calendarScale: `String` (optional). The calendar system in which this date occurs, in lowercase. This **MUST** be either a calendar system name registered as a Common Locale Data Repository (CLDR) [RFC7529] or a vendor-specific value. The year, month, and day still **MUST** be represented in the Gregorian calendar. Note that the `year` property might be required to convert the date between the Gregorian calendar and the respective calendar system.

A Timestamp object has the following properties:

@type: `String`. This **MUST** be `Timestamp`, if set.

utc: `UTCDateTime` (mandatory). Specifies the point in time in UTC time.

Figure 41 illustrates anniversaries with partial dates and a timestamp. Note how the `@type` property is set for the `Timestamp` object value according to the rules defined in Section 1.3.4.

```
"anniversaries": {
  "k8": {
    "kind": "birth",
    "date": {
      "year": 1953,
      "month": 4,
      "day": 15
    }
  },
  "k9": {
    "kind": "death",
    "date": {
      "@type": "Timestamp",
      "utc": "2019-10-15T23:10:00Z"
    },
    "place": {
      "full": "4445 Tree Street\nNew England, ND 58647\nUSA"
    }
  }
}
```

Figure 41: `anniversaries` Example

2.8.2. keywords

Type: `String[Boolean]` (optional)

A set of free-text keywords, also known as *tags*. The set is represented as an object, with each key being a keyword. The boolean value **MUST** be `true`.

```
"keywords": {
  "internet": true,
  "IETF": true
}
```

Figure 42: `keywords` Example

2.8.3. notes

Type: `Id[Note]` (optional)

Free-text notes that are associated with the Card. A Note object has the following properties:

@type: `String`. This **MUST** be `Note`, if set.

note: `String` (mandatory). The free-text value of this note.

created: `UTCDateTime` (optional). The date and time when this note was created.

author: `Author` (optional). The author of this note.

An Author object has the following properties, of which at least one property other than `@type` **MUST** be set:

@type: `String`. This **MUST** be `Author`, if set.

name: `String` (optional). The name of this author.

uri: `String` (optional). A URI value that identifies the author.

```
"notes": {
  "n1": {
    "note": "Open office hours are 1600 to 1715 EST, Mon-Fri",
    "created": "2022-11-23T15:01:32Z",
    "author": {
      "name": "John"
    }
  }
}
```

Figure 43: `notes` Example

2.8.4. personalInfo

Type: `Id[PersonalInfo]` (optional)

Defines personal information about the entity represented by the Card. A PersonalInfo object has the following properties:

@type: `String`. This **MUST** be `PersonalInfo`, if set.

kind: `String` (mandatory). Specifies the kind of personal information. The enumerated (Section 1.7.4) values are:

- `expertise`: a field of expertise or a credential
- `hobby`: a hobby
- `interest`: an interest

value: `String` (mandatory). The actual information.

level: `String` (optional). Indicates the level of expertise or engagement in hobby or interest. The enumerated (Section 1.7.4) values are:

- `high`
- `medium`
- `low`

listAs: `UnsignedInt` (optional). Defines the position of the personal information in the list of all PersonalInfo objects that have the same `kind` in the Card. If set, the `listAs` value **MUST** be higher than zero. Multiple personal information entries **MAY** have the same `listAs` property value or none. Sorting such entries is implementation-specific.

label: `String` (optional). A custom label. See Section 1.5.3.

```
"personalInfo": {
  "pi2": {
    "kind": "expertise",
    "value": "chemistry",
    "level": "high"
  },
  "pi1": {
    "kind": "hobby",
    "value": "reading",
    "level": "high"
  },
  "pi6": {
    "kind": "interest",
    "value": "r&b music",
    "level": "medium"
  }
}
```

Figure 44: `personalInfo` Example

3. IANA Considerations

3.1. Media Type Registration

This document defines a media type for use with JSContact data formatted in JSON.

Type name: application

Subtype name: jscontact+json

Required parameters: None

Optional parameters: version

> This parameter conveys the version of the JSContact data in the body part. It **MUST NOT** occur more than once. If this parameter is set, then the values of all JSContact `version` (Table 2) properties in the body part **MUST** match the parameter value.

Encoding considerations: This is the same as the encoding considerations of application/json, as specified in Section 11 of [RFC8259].

Security considerations: See Section 4 of RFC 9553.

Interoperability considerations: While JSContact is designed to avoid ambiguities as much as possible, when converting objects from other contact formats to/from JSContact, it is possible that differing representations for the same logical data or ambiguities in interpretation might arise. The semantic equivalence of two JSContact objects may be determined differently by different applications, for example, where URL values differ in case between the two objects.

Published specification: RFC 9553

Applications that use this media type: Applications that currently make use of the text/vCard media type can use this as an alternative.

Fragment identifier considerations: A JSON Pointer fragment identifier may be used, as defined in [RFC6901], Section 6.

Additional information:

> Magic number(s): N/A
> File extensions(s): N/A
> Macintosh file type code(s): N/A

Person & email address to contact for further information: calsify@ietf.org

Intended usage: COMMON

Restrictions on usage: N/A

Author: See the "Authors' Addresses" section of RFC 9553.

Change controller: IETF

3.2. Creation of the JSContact Registry Group

IANA has created the "JSContact" registry group. The new registry definitions in the following sections all belong to that group.

3.3. Registry Policy and Change Procedures

Registry assignments that introduce backwards-incompatible (Section 1.9) changes require the JSContact major version to change; other changes only require a change to the minor version. The registry policy for assignments that require the JSContact major version to change is Standards Action ([RFC8126], Section 4.9). The registry policy for other assignments is Specification Required ([RFC8126], Section 4.6).

The designated expert (DE) decides if a major or minor version change is required and assigns the new version to the "JSContact Version" registry (Section 3.4). Version numbers increment by one, and a major version change resets the minor version to zero. An assignment may apply multiple changes and to more than one registry at once, in which case a single version change is sufficient. If the registry policy is Specification Required, then the DE may decide that it is enough to document the new assignment in the Description item of the respective registry.

A registration **MUST** have an intended usage of `common`, `reserved`, or `obsolete`.

- A `common` usage denotes an item with shared semantics and syntax across systems. Up-to-date systems **MUST** expect such items to occur in JSContact data.
- A `reserved` usage reserves an item in the registry without assigning semantics to avoid name collisions with future extensions or protocol use.
- An `obsolete` usage denotes an item that is no longer expected to be added by up-to-date systems. A new assignment has probably been defined, covering the obsolete item's semantics.

The registration procedure is not a formal standards process but rather an administrative procedure intended to allow community comments and to check whether it is coherent without excessive time delay. It is designed to encourage vendors to document and register new items they add for use cases not covered by the original specification, leading to increased interoperability.

3.3.1. Preliminary Community Review

Notice of a potential new registration **MUST** be sent to the Calext WG mailing list <calsify@ietf.org> for review. This mailing list is appropriate for soliciting community feedback on a proposed registry assignment.

The intent of the public posting to this list is to solicit comments and feedback on the choice of the item name or value, the unambiguity of its description, and a review of any interoperability or security considerations. The submitter may submit a revised registration proposal or abandon the registration completely at any time.

3.3.2. Submit Request to IANA

Registration requests can be sent to IANA <iana@iana.org>.

3.3.3. Designated Expert Review

The primary concern of the DE is preventing name collisions and encouraging the submitter to document security and privacy considerations.

A new type name, property name, or enumerated value **MUST NOT** differ only in case from an already-registered name or value.

For a common-use registration, the DE is expected to confirm that suitable documentation is available to ensure interoperability. The DE should also verify that the new assignment does not conflict with work that is active or already published within the IETF.

The DE will either approve or deny the registration request and publish a notice of the decision to the Calext WG mailing list or its successor, as well as inform IANA. A denial notice must be justified by an explanation, and in the cases where it is possible, concrete suggestions on how the request can be modified to become acceptable should be provided.

3.3.4. Change Procedures

Once a JSContact registry group item has been published by IANA, the Change Controller may request a change to its definition. The same procedure that would be appropriate for the original registration request is used to process a change request.

JSContact registrations do not get deleted; instead, items that are no longer believed appropriate for use are declared obsolete by a change to their "Intended Usage" field; such items will be clearly marked in the IANA registry.

Significant changes to a JSContact registry item's definition should be requested only when there are serious omissions or errors in the published specification, as such changes may cause interoperability issues. When review is required, a change request may be denied if it renders entities that were valid under the previous definition invalid under the new definition.

3.4. Creation of the JSContact Version Registry

IANA has created the "JSContact Version" registry. The purpose of this new registry is to define the allowed value range of JSContact major and minor version numbers.

The registry entries sort numerically in ascending order by the "Major Version" column.

The registry process is outlined in Section 3.3.

3.4.1. JSContact Version Registry Template

Major Version: The numeric value of a JSContact major version number. It **MUST** be a positive integer.

Highest Minor Version: The maximum numeric value of a JSContact minor version for the given major version. It **MUST** be zero or a positive integer. All numbers less than or equal to this value are valid minor version values for the given major version.

3.4.2. Initial Contents of the JSContact Version Registry

The following table lists the initial valid major and minor version number ranges.

Major Version	Highest Minor Version	Reference
1	0	RFC 9553

Table 1: JSContact Version Registry

3.5. Creation of the JSContact Properties Registry

IANA has created the "JSContact Properties" registry. The purpose of this new registry is to allow interoperability of extensions to JSContact objects.

The registry entries sort alphabetically in ascending order by the following columns: "Property Name" first, "Property Context" second, and "Since Version" third. Equal entries sort in any order.

The registry process for a new property is outlined in Section 3.3.

3.5.1. JSContact Properties Registry Template

Property Name: The name of the property. The property name **MUST NOT** already be registered for any of the object types listed in the "Property Context" field of this registration. Other object types **MAY** have already registered a different property with the same name; however, the same name **MUST** only be used when the semantics are analogous.

Property Type: The type of the property, using type signatures, as specified in Section 1.3.2. The property type **MUST** be registered in the "JSContact Types" registry.

Property Context: A comma-separated list of JSContact object types (Section 3.6.2) that contain the property.

Intended Usage: May be "common", "reserved", or "obsolete".

Since Version: The JSContact version on which the property definition is based. The version **MUST** be one of the allowed values of the `version` property in the "JSContact Enum Values" registry (see Table 1).

Until Version: The JSContact version after which the property was obsoleted; therefore, it **MUST NOT** be used in later versions. The Until Version value either **MUST NOT** be set or **MUST** be one of the allowed values of the `version` property in the "JSContact Enum Values" registry (see Table 1).

Change Controller: This is who may request a change to the entry's definition (`IETF` for RFCs from the IETF stream).

Reference or Description: A brief description or RFC number and section reference where the property is specified (but omitted for "reserved" property names). This must include references to all RFC documents where this property is introduced or updated.

3.5.2. Initial Contents of the JSContact Properties Registry

The following table lists the initial `common` usage entries of the "JSContact Properties" registry. For all properties, the Since Version is `1.0`, the Until Version is not set, the Change Controller is `IETF`, and RFC section references are for RFC 9553.

Property Name	Property Type	Property Context	Ref
@type	String	Address, AddressComponent, Anniversary, Author, Card, Calendar, CryptoKey, Directory, EmailAddress, LanguagePref, Link, Media, Name, NameComponent, Nickname, Note, OnlineService, Organization, OrgUnit, PartialDate, PersonalInfo, Phone, Pronouns, Relation, SchedulingAddress, SpeakToAs, Timestamp, Title	Sections 2.5.1, 2.8.1, 2.1.1, 2.4.1, 2.6.1, 2.1.8, 2.4.2, 2.2.4
address	String	EmailAddress	Section 2.3.1
addresses	Id[Address]	Card	Section 2.5.1
anniversaries	Id[Anniversary]	Card	Section 2.8.1
author	Author	Note	Section 2.8.3
calendars	Id[Calendar]	Card	Section 2.4.1
calendarScale	String	PartialDate	Section 2.8.1

Property Name	Property Type	Property Context	Ref
components	AddressComponent[]	Address	Section 2.5.1
components	NameComponent[]	Name	Section 2.2.1
contexts	String[Boolean]	Address, Calendar, CryptoKey, Directory, EmailAddress, LanguagePref, Link, Media, Nickname, OnlineService, Organization, Phone, Pronouns, SchedulingAddress	Sections 1.4.4, 1.5.1, 2.5.1, 2.4.1, 2.6.1,
coordinates	String	Address	Section 2.5.1
countryCode	String	Address	Section 2.5.1
created	UTCDateTime	Card, Note	Sections 2.1.3, 2.8.3
date	PartialDate | Timestamp	Anniversary	Section 2.8.1
day	UnsignedInt	PartialDate	Section 2.8.1
defaultSeparator	String	Address, Name	Sections 2.5.1, 2.2.1
directories	Id[Directory]	Card	Section 2.6.2
emails	Id[EmailAddress]	Card	Section 2.3.1
features	String[Boolean]	Phone	Section 2.3.3
full	String	Address, Name	Sections 2.5.1, 2.2.1
grammaticalGender	String	SpeakToAs	Section 2.2.3
isOrdered	Boolean	Address, Name	Sections 2.5.1, 2.2.1
keywords	String[Boolean]	Card	Section 2.8.2

Property Name	Property Type	Property Context	Ref
kind	String	AddressComponent, Anniversary, Calendar, Card, CryptoKey, Directory, Link, Media, NameComponent, PersonalInfo, Title	Sections 2.5.1, 2.8.1, 2.4.1, 2.1.4, 2.6.1,
label	String	Calendar, CryptoKey, Directory, EmailAddress, Link, Media, OnlineService, PersonalInfo, Phone, SchedulingAddress	Sections 1.4.4, 1.5.3, 2.4.1, 2.6.1, 2.6.2,
language	String	Card, LanguagePref	Sections 2.1.5, 2.3.4
level	String	PersonalInfo	Section 2.8.4
links	Id[Link]	Card	Section 2.6.3
listAs	UnsignedInt	Directory, PersonalInfo	Sections 2.6.2, 2.8.4
localizations	String[PatchObject]	Card	Section 2.7.1
media	Id[Media]	Card	Section 2.6.4
mediaType	String	Calendar, CryptoKey, Directory, Link, Media	Sections 1.4.4, 2.4.1, 2.6.1, 2.6.2, 2.6.3,
members	String[Boolean]	Card	Section 2.1.6
month	UnsignedInt	PartialDate	Section 2.8.1
name	Name	Card	Section 2.2.1

Property Name	Property Type	Property Context	Ref
name	String	Author, Nickname, Organization, OrgUnit, Title	Sections 2.8.3, 2.2.1.3, 2.2.2, 2.2.4
nicknames	Id[Nickname]	Card	Section 2.2.1.3
note	String	Note	Section 2.8.3
notes	Id[Note]	Card	Section 2.8.3
number	String	Phone	Section 2.3.3
onlineServices	Id[OnlineService]	Card	Section 2.3.2
organizationId	String	Title	Section 2.2.4
organizations	Id[Organization]	Card	Section 2.2.2
personalInfo	Id[PersonalInfo]	Card	Section 2.8.4
phones	Id[Phone]	Card	Section 2.3.3
phonetic	String	AddressComponent, NameComponent	Sections 2.5.1.2, 2.2.1.2
phoneticScript	String	Address, Name	Sections 2.2.1, 2.5.1
phoneticSystem	String	Address, Name	Sections 2.2.1, 2.5.1
place	Address	Anniversary	Section 2.8.1
pref	UnsignedInt	Address, Calendar, CryptoKey, Directory, EmailAddress, LanguagePref, Link, Media, Nickname, OnlineService, Phone, Pronouns, SchedulingAddress	Sections 1.4.4, 1.5.4, 2.5.1, 2.4.1, 2.6.1,
preferredLanguages	String[LanguagePref]	Card	Section 2.3.4
prodId	String	Card	Section 2.1.7
pronouns	Id[Pronouns]	SpeakToAs	Section 2.2.3

Property Name	Property Type	Property Context	Ref
relatedTo	String[Relation]	Card	Section 2.1.8
relation	String[Boolean]	Relation	Section 2.1.8
schedulingAddresses	Id[SchedulingAddress]	Card	Section 2.4.2
service	String	OnlineService	Section 2.3.2
sortAs	String[String]	Name	Section 2.2.1
sortAs	String	Organization, OrgUnit	Section 2.2.2
speakToAs	SpeakToAs	Card	Section 2.2.3
timeZone	String	Address	Section 2.5.1
titles	Id[Title]	Card	Section 2.2.4
uid	String	Card	Section 2.1.9
units	OrgUnit[]	Organization	Section 2.2.2
updated	UTCDateTime	Card	Section 2.1.10
uri	String	Author, Calendar, CryptoKey, Directory, Link, Media, OnlineService, SchedulingAddress	Sections 1.4.4, 2.8.3, 2.4.1, 2.6.1, 2.6.2,
user	String	OnlineService	Section 2.3.2
utc	UTCDateTime	Timestamp	Section 2.8.1
value	String	AddressComponent, NameComponent, PersonalInfo	Sections 2.5.1, 2.2.1, 2.8.4
version	String	Card	Section 2.1.2
year	UnsignedInt	PartialDate	Section 2.8.1

Table 2: JSContact Properties with "common" Usage

The following table lists the initial `reserved` usage entries of the "JSContact Properties" registry. For this property, the Change Controller is `IETF`, and the RFC section reference is for RFC 9553.

Property Name	Property Type	Property Context	Reference/ Description	Intended Usage
extra	not applicable	not applicable	Section 1.5.2	reserved

Table 3: JSContact Properties with "reserved" Usage

3.6. Creation of the JSContact Types Registry

IANA has created the "JSContact Types" registry. The purpose of this new registry is to avoid name collisions for JSContact type names and provide a complete reference for all data types used for JSContact property values.

The registry entries sort alphabetically in ascending order by the "Type Name" column. Equal entries sort in any order.

The registry process for a new type is outlined in Section 3.3.

3.6.1. JSContact Types Registry Template

Type Name: The name of the type.

Intended Usage: May be "common", "reserved", or "obsolete".

Since Version: The JSContact version on which this type definition is based. The version **MUST** be one of the allowed values of the `version` property in the "JSContact Enum Values" registry (see Table 1).

Until Version: The JSContact version after which the type definition was obsoleted; therefore, it **MUST NOT** be used in later versions. The Until Version value either **MUST NOT** be set or **MUST** be one of the allowed values of the `version` property in the "JSContact Enum Values" registry (see Table 1).

Change Controller: This is who may request a change to the entry's definition (`IETF` for RFCs from the IETF stream).

Reference or Description: A brief description or RFC number and section reference where the Type is specified (may be omitted for "reserved" type names).

3.6.2. Initial Contents of the JSContact Types Registry

The following table lists the initial `common` usage entries in the "JSContact Types" registry. For all of these types, the Since Version is `1.0`, the Until Version is not set, the Change Controller is `IETF`, and RFC section references are for RFC 9553.

Type Name	Reference or Description
Address	Section 2.5.1

Type Name	Reference or Description
AddressComponent	Section 2.5.1
Anniversary	Section 2.8.1
Author	Section 2.8.3
Boolean	Section 1.3.2
Calendar	Section 2.4.1
Card	Section 2
CryptoKey	Section 2.6.1
Directory	Section 2.6.2
EmailAddress	Section 2.3.1
Id	Section 1.4.1
Int	Section 1.4.2
LanguagePref	Section 2.3.4
Link	Section 2.6.3
Media	Section 2.6.4
Name	Section 2.2.1
NameComponent	Section 2.2.1
Nickname	Section 2.2.1.3
Note	Section 2.8.3
Number	Section 1.3.2
OnlineService	Section 2.3.2
Organization	Section 2.2.2
OrgUnit	Section 2.2.2
PartialDate	Section 2.8.1
PatchObject	Section 1.4.3

Type Name	Reference or Description
PersonalInfo	Section 2.8.4
Phone	Section 2.3.3
Pronouns	Section 2.2.3
Relation	Section 2.1.8
SchedulingAddress	Section 2.4.2
SpeakToAs	Section 2.2.3
String	Section 1.3.2
Timestamp	Section 2.8.1
Title	Section 2.2.4
UnsignedInt	Section 1.4.2
UTCDateTime	Section 1.4.5

Table 4: JSContact Types with "common" Usage

The following table lists the initial `reserved` usage entry of the "JSContact Types" registry. For this type, the version is `1.0`, the Change Controller is `IETF`, and the RFC section reference is for RFC 9553.

Type Name	Reference or Description
Resource	Section 1.4.4

Table 5: JSContact Types with "reserved" Usage

3.7. Creation of the JSContact Enum Values Registry

IANA has created the "JSContact Enum Values" registry. The purpose of the new registry is to allow interoperable extension of semantics for JSContact properties with enumerable values. Each such property will have a subregistry of allowed values.

The registry entries sort alphabetically in ascending order by the following columns: "Property Name" first, "Property Context" second, and "Since Version" third. The enum values sort alphabetically in ascending order. Equal entries sort in any order.

The registry process for a new enum value or adding a new enumerable property is outlined in Section 3.3.

3.7.1. JSContact Enum Values Registry Property Template

This template is for adding a subregistry for a new enumerable property to the "JSContact Enum Values" registry.

Property Name: The name(s) of the property or properties where these values may be used. This **MUST** be registered in the "JSContact Properties" registry.

Context: The list of allowed object types where the property or properties may appear, as registered in the "JSContact Properties" registry. This disambiguates where there may be two distinct properties with the same name in different contexts.

Since Version: The JSContact version on which the enum value definition is based. The version **MUST** be one of the allowed values of the `version` property in the "JSContact Enum Values" registry (see Table 1).

Until Version: The JSContact version after which the enum value definition was obsoleted; therefore, the enum value definition **MUST NOT** be used in later versions. The Until Version value either **MUST NOT** be set or **MUST** be one of the allowed values of the `version` property in the "JSContact Enum Values" registry (see Table 1).

Change Controller: This is who may request a change to the entry's definition (`IETF` for RFCs from the IETF stream).

Initial Contents: The initial list of defined values for the enum, using the template defined in Section 3.7.2. A subregistry will be created with these values for this property name/context tuple.

3.7.2. JSContact Enum Values Registry Value Template

This template is for adding a new enum value to a subregistry in the "JSContact Enum Values" registry.

Enum Value: The verbatim value of the enum.

Since Version: The JSContact version on which the enum value definition is based. The version **MUST** be one of the allowed values of the `version` property in the "JSContact Enum Values" registry (see Table 1).

Until Version: The JSContact version after which the enum value was obsoleted; therefore, the enum value **MUST NOT** be used in later versions. The Until Version value either **MUST NOT** be set or **MUST** be one of the allowed values of the `version` property in the "JSContact Enum Values" registry (see Table 1).

Reference or Description: A brief description or RFC number and section reference for the semantics of the value.

3.7.3. Initial Contents of the JSContact Enum Values Registry

For all entries in each subregistry created in this section, the Since Version is `1.0`, the Until Version is not set, the Change Controller is `IETF`, and RFC section references are for RFC 9553.

Property Name: contexts
Context: Address
Initial Contents:

Enum Value	Reference or Description
billing	Section 2.5.1
delivery	Section 2.5.1
private	Section 1.5.1
work	Section 1.5.1

Table 6: JSContact Enum Values for contexts (Context: Address)

Property Name: contexts
Context: Calendar, CryptoKey, Directory, EmailAddress, LanguagePref, Link, Media, Nickname, OnlineService, Organization, Phone, Pronouns, SchedulingAddress
Initial Contents:

Enum Value	Reference or Description
private	Section 1.5.1
work	Section 1.5.1

Table 7: JSContact Enum Values for contexts (Context: Calendar, CryptoKey, Directory, EmailAddress, LanguagePref, Link, Media, Nickname, OnlineService, Organization, Phone, Pronouns, SchedulingAddress)

Property Name: features
Context: Phone
Initial Contents:

Enum Value	Reference or Description
fax	Section 2.3.3
main-number	Section 2.3.3

Enum Value	Reference or Description
mobile	Section 2.3.3
pager	Section 2.3.3
text	Section 2.3.3
textphone	Section 2.3.3
video	Section 2.3.3
voice	Section 2.3.3

Table 8: JSContact Enum Values for features (Context: Phone)

Property Name: grammaticalGender
Context: SpeakToAs
Initial Contents:

Enum Value	Reference or Description
animate	Section 2.2.3
common	Section 2.2.3
feminine	Section 2.2.3
inanimate	Section 2.2.3
masculine	Section 2.2.3
neuter	Section 2.2.3

Table 9: JSContact Enum Values for grammaticalGender (Context: SpeakToAs)

Property Name: kind
Context: AddressComponent
Initial Contents:

Enum Value	Reference or Description
apartment	Section 2.5.1
block	Section 2.5.1
building	Section 2.5.1

Enum Value	Reference or Description
country	Section 2.5.1
direction	Section 2.5.1
district	Section 2.5.1
floor	Section 2.5.1
landmark	Section 2.5.1
locality	Section 2.5.1
name	Section 2.5.1
number	Section 2.5.1
postcode	Section 2.5.1
postOfficeBox	Section 2.5.1
region	Section 2.5.1
room	Section 2.5.1
separator	Section 2.5.1
subdistrict	Section 2.5.1

Table 10: JSContact Enum Values for kind (Context: AddressComponent)

Property Name: kind
Context: Anniversary
Initial Contents:

Enum Value	Reference or Description
birth	Section 2.8.1
death	Section 2.8.1
wedding	Section 2.8.1

Table 11: JSContact Enum Values for kind (Context: Anniversary)

Property Name: kind

Context: Calendar
Initial Contents:

Enum Value	Reference or Description
calendar	Section 2.4.1
freeBusy	Section 2.4.1

Table 12: JSContact Enum Values for kind (Context: Calendar)

Property Name: kind
Context: Card
Initial Contents:

Enum Value	Reference or Description
application	Section 2.1.4
device	Section 2.1.4
group	Section 2.1.4
individual	Section 2.1.4
location	Section 2.1.4
org	Section 2.1.4

Table 13: JSContact Enum Values for kind (Context: Card)

Property Name: kind
Context: Directory
Initial Contents:

Enum Value	Reference or Description
directory	Section 2.6.2
entry	Section 2.6.2

Table 14: JSContact Enum Values for kind (Context: Directory)

Property Name: kind
Context: Link
Initial Contents:

Enum Value	Reference or Description
contact	Section 2.6.3

Table 15: JSContact Enum Values for kind (Context: Link)

Property Name: kind
Context: Media
Initial Contents:

Enum Value	Reference or Description
logo	Section 2.6.4
photo	Section 2.6.4
sound	Section 2.6.4

Table 16: JSContact Enum Values for kind (Context: Media)

Property Name: kind
Context: NameComponent
Initial Contents:

Enum Value	Reference or Description
credential	Section 2.2.1
generation	Section 2.2.1
given	Section 2.2.1
given2	Section 2.2.1
separator	Section 2.2.1
surname	Section 2.2.1
surname2	Section 2.2.1
title	Section 2.2.1

Table 17: JSContact Enum Values for kind (Context: NameComponent)

Property Name: kind
Context: PersonalInfo

Initial Contents:

Enum Value	Reference or Description
expertise	Section 2.8.4
hobby	Section 2.8.4
interest	Section 2.8.4

Table 18: JSContact Enum Values for kind (Context: PersonalInfo)

Property Name: kind
Context: Title
Initial Contents:

Enum Value	Reference or Description
role	Section 2.2.4
title	Section 2.2.4

Table 19: JSContact Enum Values for kind (Context: Title)

Property Name: level
Context: PersonalInfo
Initial Contents:

Enum Value	Reference or Description
high	Section 2.8.4
low	Section 2.8.4
medium	Section 2.8.4

Table 20: JSContact Enum Values for level (Context: PersonalInfo)

Property Name: phoneticSystem
Context: Address, Name
Initial Contents:

Enum Value	Reference or Description
ipa	Section 1.5.5
jyut	Section 1.5.5

Enum Value	Reference or Description
piny	Section 1.5.5

Table 21: JSContact Enum Values for phoneticSystem (Context: Address, Name)

Property Name: relation
Context: Relation
Initial Contents:

Enum Value	Reference or Description
acquaintance	Section 2.1.8
agent	Section 2.1.8
child	Section 2.1.8
colleague	Section 2.1.8
contact	Section 2.1.8
co-resident	Section 2.1.8
co-worker	Section 2.1.8
crush	Section 2.1.8
date	Section 2.1.8
emergency	Section 2.1.8
friend	Section 2.1.8
kin	Section 2.1.8
me	Section 2.1.8
met	Section 2.1.8
muse	Section 2.1.8
neighbor	Section 2.1.8
parent	Section 2.1.8
sibling	Section 2.1.8
spouse	Section 2.1.8

Enum Value	Reference or Description
sweetheart	Section 2.1.8

Table 22: JSContact Enum Values for relation (Context: Relation)

4. Security Considerations

Contact information is very privacy sensitive. It can reveal the identity, location, credentials information, employment status, interests and hobbies, and social network of a user. Its transmission and storage must be done carefully to protect it from possible threats such as eavesdropping, replay, message insertion, deletion, modification, and on-path attacks.

The data being stored and transmitted may be used in systems with real-world consequences. For example, a malicious actor might provide JSContact data that uses the name of another person but insert their contact details to impersonate the unknown victim. Such systems must be careful to authenticate all data they receive to prevent them from being subverted and ensure the change comes from an authorized entity.

This document only defines the data format; such considerations are primarily the concern of the API or method of storage and transmission of such files.

4.1. JSON Parsing

The security considerations of [RFC8259] apply to the use of JSON as the data interchange format.

As for any serialization format, parsers need to thoroughly check the syntax of the supplied data. JSON uses opening and closing brackets for several types and structures, and it is possible that the end of the supplied data will be reached when scanning for a matching closing bracket; this is an error condition, and implementations need to stop scanning at the end of the supplied data.

JSON also uses a string encoding with some escape sequences to encode special characters within a string. Care is needed when processing these escape sequences to ensure that they are fully formed before the special processing is triggered, with special care taken when the escape sequences appear adjacent to other (non-escaped) special characters or adjacent to the end of data (as in the previous paragraph).

If parsing JSON into a non-textual structured data format, implementations may need to allocate storage to hold JSON string elements. Since JSON does not use explicit string lengths, the risk of denial of service due to resource exhaustion is small, but implementations may still wish to place limits on the size of allocations they are willing to make in any given context, to avoid untrusted data causing excessive memory allocation.

4.2. URI Values

Several JSContact properties contain URIs as values, and processing these properties requires extra care. Section 7 of [RFC3986] discusses security risks related to URIs.

Fetching remote resources carries inherent risks. Connections must only be allowed on well-known ports, using allowed protocols (generally, just HTTP/HTTPS on their default ports). The URL must be resolved externally and not allowed to access internal resources. Connecting to an external source reveals IP (and therefore often location) information.

A maliciously constructed JSContact object may contain a very large number of URIs. In the case of published address books with a large number of subscribers, such objects could be widely distributed. Implementations should be careful to limit the automatic fetching of linked resources to reduce the risk of this being an amplification vector for a denial-of-service attack.

5. References

5.1. Normative References

[IANA-TZ] IANA, "Time Zone Database", <https://www.iana.org/time-zones>.

[IANA-vCard] IANA, "vCard Elements", <https://www.iana.org/assignments/vcard-elements>.

[ISO.3166-1] International Organization for Standardization, "Codes for the representation of names of countries and their subdivisions -- Part 1: Country codes", ISO 3166-1:2020, August 2020.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <https://www.rfc-editor.org/info/rfc1034>.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <https://www.rfc-editor.org/info/rfc1035>.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <https://www.rfc-editor.org/info/rfc2046>.

[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/info/rfc2119>.

[RFC2426] Dawson, F. and T. Howes, "vCard MIME Directory Profile", RFC 2426, DOI 10.17487/RFC2426, September 1998, <https://www.rfc-editor.org/info/rfc2426>.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/info/rfc3339>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/info/rfc4122>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5234] 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/info/rfc5234>.

[RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, DOI 10.17487/RFC5322, October 2008, <https://www.rfc-editor.org/info/rfc5322>.

[RFC5646] Phillips, A., Ed. and M. Davis, Ed., "Tags for Identifying Languages", BCP 47, RFC 5646, DOI 10.17487/RFC5646, September 2009, <https://www.rfc-editor.org/info/rfc5646>.

[RFC5870] Mayrhofer, A. and C. Spanring, "A Uniform Resource Identifier for Geographic Locations ('geo' URI)", RFC 5870, DOI 10.17487/RFC5870, June 2010, <https://www.rfc-editor.org/info/rfc5870>.

[RFC6350] Perreault, S., "vCard Format Specification", RFC 6350, DOI 10.17487/RFC6350, August 2011, <https://www.rfc-editor.org/info/rfc6350>.

[RFC6901] 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/info/rfc6901>.

[RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <https://www.rfc-editor.org/info/rfc7493>.

[RFC7529] Daboo, C. and G. Yakushev, "Non-Gregorian Recurrence Rules in the Internet Calendaring and Scheduling Core Object Specification (iCalendar)", RFC 7529, DOI 10.17487/RFC7529, May 2015, <https://www.rfc-editor.org/info/rfc7529>.

[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/info/rfc8126>.

[RFC8141] Saint-Andre, P. and J. Klensin, "Uniform Resource Names (URNs)", RFC 8141, DOI 10.17487/RFC8141, April 2017, <https://www.rfc-editor.org/info/rfc8141>.

[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/info/rfc8174>.

[RFC8259] 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/info/rfc8259>.

5.2. Informative References

[IPA]

IPA, "International Phonetic Alphabet", <https://www.internationalphoneticalphabet.org/>.

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, DOI 10.17487/RFC3261, June 2002, <https://www.rfc-editor.org/info/rfc3261>.

[RFC3966] Schulzrinne, H., "The tel URI for Telephone Numbers", RFC 3966, DOI 10.17487/RFC3966, December 2004, <https://www.rfc-editor.org/info/rfc3966>.

[RFC3986] 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/info/rfc3986>.

[RFC3987] Duerst, M. and M. Suignard, "Internationalized Resource Identifiers (IRIs)", RFC 3987, DOI 10.17487/RFC3987, January 2005, <https://www.rfc-editor.org/info/rfc3987>.

[RFC6351] Perreault, S., "xCard: vCard XML Representation", RFC 6351, DOI 10.17487/RFC6351, August 2011, <https://www.rfc-editor.org/info/rfc6351>.

[RFC6473] Saint-Andre, P., "vCard KIND:application", RFC 6473, DOI 10.17487/RFC6473, December 2011, <https://www.rfc-editor.org/info/rfc6473>.

[RFC6474] Li, K. and B. Leiba, "vCard Format Extensions: Place of Birth, Place and Date of Death", RFC 6474, DOI 10.17487/RFC6474, December 2011, <https://www.rfc-editor.org/info/rfc6474>.

[RFC6715] Cauchie, D., Leiba, B., and K. Li, "vCard Format Extensions: Representing vCard Extensions Defined by the Open Mobile Alliance (OMA) Converged Address Book (CAB) Group", RFC 6715, DOI 10.17487/RFC6715, August 2012, <https://www.rfc-editor.org/info/rfc6715>.

[RFC6869] Salgueiro, G., Clarke, J., and P. Saint-Andre, "vCard KIND:device", RFC 6869, DOI 10.17487/RFC6869, February 2013, <https://www.rfc-editor.org/info/rfc6869>.

[RFC7095] Kewisch, P., "jCard: The JSON Format for vCard", RFC 7095, DOI 10.17487/RFC7095, January 2014, <https://www.rfc-editor.org/info/rfc7095>.

[RFC8499] Hoffman, P., Sullivan, A., and K. Fujiwara, "DNS Terminology", BCP 219, RFC 8499, DOI 10.17487/RFC8499, January 2019, <https://www.rfc-editor.org/info/rfc8499>.

[RFC8605] Hollenbeck, S. and R. Carney, "vCard Format Extensions: ICANN Extensions for the Registration Data Access Protocol (RDAP)", RFC 8605, DOI 10.17487/RFC8605, May 2019, <https://www.rfc-editor.org/info/rfc8605>.

[UBiDi] The Unicode Consortium, "Unicode Standard Annex #9: Unicode Bidirectional Algorithm", Revision 48, Unicode 15.1.0, August 2023, <https://www.unicode.org/reports/tr9/>.

[UUID] Davis, K. R., Peabody, B. G., and P. Leach, "Universally Unique IDentifiers (UUID)", Work in Progress, Internet-Draft, draft-ietf-uuidrev-rfc4122bis-14, 6 November 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-uuidrev-rfc4122bis-14>.

[WHATWG-URL] WHATWG, "URL Living Standard", January 2024, <https://url.spec.whatwg.org>.

Authors' Addresses

Robert Stepanek
Fastmail
PO Box 234
Collins St. West
Melbourne VIC 8007
Australia
Email: rsto@fastmailteam.com

Mario Loffredo
IIT-CNR
Via Moruzzi, 1
56124 Pisa
Italy
Email: mario.loffredo@iit.cnr.it