



# "New JSON Schema"

Clemens Vasters  
Principal Architect  
Messaging & Real-Time Intelligence  
[clemensv@microsoft.com](mailto:clemensv@microsoft.com)

# Who are we? (My team)

- My team owns/builds several eventing and messaging services
  - Azure Event Grid – CloudEvents-centric event broker (CloudEvents HTTP, MQTT 3.1.1./5.0)
  - Azure Event Hubs – Event stream engine (Kafka compatible, AMQP 1.0)
  - Azure Service Bus – Transactional Queue & Pub/Sub Broker (JMS 2.0 compatible, AMQP 1.0)
  - Azure Stream Analytics – Real-time stream processing engine (SQL)
  - Microsoft Fabric Eventstreams – SaaS offering integrating all of the above for mortals
- We care about Standards and de-facto standards
  - We (co-)drive: OASIS AMQP TC, OASIS MQTT TC, CNCF CloudEvents, CNCF xRegistry
  - We use: Apache Kafka RPC, JMS 2.0, JSON, JSON Schema, Avro Schema, HTTP, ...
- Very significant push towards "type-safe messaging"
  - CNCF xRegistry as metadata model for asynchronous messaging
  - Polyglot, multi-protocol, multi-encoding, multi-schema code generation and validation

# JSON Schema as it exists

- The de-facto standard for JSON Schema is "Draft 07" (2018)
  - Very inconsistent uptake of later drafts in the tooling landscape. Indicates that "draft 07" is "good enough" from a feature perspective and later drafts didn't move the needle for the majority of users.
- Everyone who tries to define data structures with JSON Schema eventually struggles
  - There is no tool generating code or deriving schemas (e.g. databases) that can handle every valid JSON Schema.
  - Tools give up at very different levels of complexity and features
  - Mapping from/to common programming language constructs is inconsistent
  - Schemas are not consistently typed and named.
- The conditional composition constructs (*anyOf*, *allOf*, *oneOf*, *if/then/else*, *not*) are pattern-matching rules that users attempt to (and fail to) use as a data structure definition language.
  - For instance: *oneOf* is used as an alternate way to define type unions, *allOf* is used to model inheritance
- JSON Schema's primitive type system model is as poor as JSON's even though it could and should do better, being an overlaid rule set. "format" is a partial, but incomplete way to address this.
- Cross-referencing across schema file boundaries (*\$ref*) causes very brittle interdependencies with file locations and protocols being significant.

# "New JSON Schema"

- "New JSON Schema" is a refactoring of the existing JSON Schema.
  - The project name choice (in quotes) indicates that we see this as a complete functional replacement for the existing JSON Schema and a pragmatic way forward.
- **Goals:**
  - The core specification focuses on data structure definitions, names and namespaces, a rich primitive type system, an extended set of compound types (sets, maps, tuples), and explicit definition sharing/reuse mechanisms that align with common programming-language constructs.
  - Companion specs introduce internationalization support for names and symbols and descriptions, alternate names for special purposes, and scientific and currency unit annotations. Better data quality and more formalized context, also for LLMs.
  - Cross-file references ("import") are enabled by a companion spec.
  - All pattern-matching validation rules are factored into optional companion specs, with conditional composition (anyOf, allOf, oneOf, if/then/else, not) separate from other rules.

# New JSON Schema – Core

- "The parts of JSON Schema most devs care about, but better"

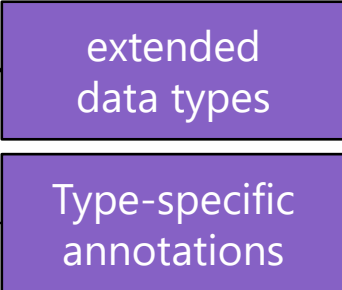
```
{
  "$schema": "https://schemas.vasters.com/experimental/json-schema-core/v0",
  "type": "object",
  "name": "Person",
  "properties": {
    "firstName": { "type": "string" },
    "lastName": { "type": "string" },
    "dateOfBirth": { "type": "date" }
  },
  "required": ["firstName", "lastName"]
}
```

The diagram illustrates three key features of the new JSON Schema Core with callouts from a sample schema:

- "type" now required**: A callout points to the `"type": "object"` property in the root schema object.
- every type must be named**: A callout points to the `"name": "Person"` property in the root schema object.
- more primitive data types**: A callout points to the `"type": "date"` property within the `dateOfBirth` property definition in the `properties` object.

# New JSON Schema – Extended Primitive Types

```
{
  "$schema": "https://schemas.vasters.com/{...}/v0",
  "type": "object",
  "name": "UserProfile",
  "properties": {
    "username": { "type": "string" },
    "dateOfBirth": { "type": "date" },
    "lastSeen": { "type": "datetime" },
    "score": { "type": "int64" },
    "balance": {
      "type": "decimal",
      "precision": 20,
      "scale": 2
    },
    "isActive": { "type": "boolean" }
  },
  "required": ["username", "birthdate"]
}
```



## Extended Primitive Types

binary

float8

int8

float

uint8

double

int16

decimal

uint16

date

int32

datetime

uint32

time

int64

duration

uint64

uuid

int128

jsonpointer

uint128

uri

All extended primitive types map to the base primitive types *number* or *string* in a well-defined way.

# New JSON Schema – Reusable Types

```
{
  "$schema": "https://schemas.vasters.com/{...}/v0",
  "type": "object",
  "name": "UserProfile",
  "properties": {
    "username": { "type": "string" },
    "dateOfBirth": { "type": "date" },
    "lastSeen": { "type": "datetime" },
    "score": { "type": "int64" },
    "balance": { "type": "decimal", "precision": 20, "scale": 2 },
    "isActive": { "type": "boolean" },
    "address": { "type": { "$ref": "#/$defs/Address" } }
  },
  "required": ["username", "birthdate"],
  "$defs": {
    "Address": {
      "type": "object",
      "properties": {
        "street": { "type": "string" },
        "city": { "type": "string" },
        "state": { "type": "string" },
        "zip": { "type": "string" }
      },
      "required": ["street", "city", "state", "zip"]
    }
  }
}
```

- Reusable types **must** be defined in *\$defs*.
- Far stricter *\$ref* rules:
  - Must only be used as a nested schema expression underneath "type" or where a type expression is expected (e.g. unions).
  - Must refer to a definition underneath \$defs
  - Must be a relative URI and cannot refer to outside the document

\* Given the stickiness of "#/definitions" there may be an alias \$defs/definitions

# New JSON Schema – Namespaces

```
{
  "$schema": "https://schemas.vasters.com/experimental/json-schema-core/v0",
  "type": "object",
  "name": "UserProfile",
  "properties": {
    "username": { "type": "string" },
    "dateOfBirth": { "type": "date" },
    "networkAddress": { "type": { "$ref": "#/$defs/Network/Address" } },
    "physicalAddress": { "type": { "$ref": "#/$defs/Physical/Address" } }
  },
  "required": ["username", "birthdate"],
  "$defs": {
    "Network": {
      "Address": {
        "type": "object",
        "properties": {
          "ipv4": { "type": "string" },
          "ipv6": { "type": "string" }
        }
      }
    },
    "Physical": {
      "Address": {
        "type": "object",
        "properties": {
          "street": { "type": "string" },
          "city": { "type": "string" },
          "state": { "type": "string" },
          "zip": { "type": "string" }
        }
      },
      "required": ["street", "city", "state", "zip"]
    }
  }
}
```

- Namespaces are an explicit construct inside *\$defs*
- Top level in *\$defs* and the root type belong to the empty namespace
- Namespaces have the same function as in common programming languages: disambiguate and group concepts and types to avoid collisions.
- Namespaces enable organizing *\$import*



# New JSON Schema – \$import

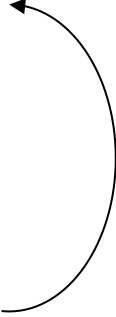
```
{
  "$schema": "https://schemas.vasters.com/{...}/v0",
  "$id": "https://example.com/people.json",
  "name": "Person",
  "type": "object",
  "properties": {
    "firstName": { "type": "string" },
    "lastName": { "type": "string" },
    "address": { "$ref": "#/$defs/Address" }
  },
  "$defs": {
    "Address": {
      "type": "object",
      "properties": {
        "street": { "type": "string" },
        "city": { "type": "string" }
      }
    }
  }
}
```

```
{
  "$schema": "https://schemas.vasters.com/{...}/v0",
  "type": "object",
  "properties": {
    "person": {
      "type": { "$ref": "#/$defs/People/Person" }
    },
    "shippingAddress": {
      "type": { "$ref": "#/$defs/People/Address" }
    }
  },
  "$defs": {
    "People": {
      "$import": "https://example.com/people.json"
    }
  }
}
```

- *\$import* loads all definitions of an external schema into a namespace
- The result of *\$import* behaves like a local copy of all imported types
- *\$importdefs* only loads the *\$defs* section of the external schema, skipping the root type.
- After types have been imported, they may be used via *\$ref* as if local
- *Shadowing* allows imported types to be overridden
- *\$import* is an optional extension

# New JSON Schema – *abstract* and *\$extends*

```
{
  "$schema": "https://schemas.vasters.com/{...}/v0",
  "$defs": {
    "AddressBase": {
      "abstract": true,
      "type": "object",
      "properties": {
        "city": { "type": "string" },
        "state": { "type": "string" },
        "zip": { "type": "string" }
      }
    },
    "StreetAddress": {
      "type": "object",
      "$extends": "#/$defs/AddressBase",
      "properties": {
        "street": { "type": "string" }
      }
    },
    "PostOfficeBoxAddress": {
      "type": "object",
      "$extends": "#/$defs/AddressBase",
      "properties": {
        "poBox": { "type": "string" }
      }
    },
    "Address": {
      "type": "object",
      "$extends": "#/$defs/AddressBase"
    }
  }
}
```



- *abstract: true* declares extensible types
- Extensible types cannot be used (*\$ref*) directly
- Extensible types can be extended by concrete types for sharing definitions
- No polymorphism, only definition sharing

# New JSON Schema – Add-Ins

```
{
  "$schema": "https://schemas.vasters.com/{...}/v0",
  "$id": "https://schemas.vasters.com/Addresses",
  "$root": "#/$defs/StreetAddress",
  "$offers": {
    "DeliveryInstructions": "#/$defs/DeliveryInstructions"
  },
  "$defs" : {
    "StreetAddress": {
      "type": "object",
      "properties": {
        "street": { "type": "string" },
        "city": { "type": "string" },
        "state": { "type": "string" },
        "zip": { "type": "string" }
      }
    },
    "DeliveryInstructions": {
      "abstract": true,
      "type": "object",
      "$extends": "#/$defs/StreetAddress",
      "properties": {
        "instructions": { "type": "string" }
      }
    }
  }
}
```

```
{
  "$schema": "https://schemas.vasters.com/Addresses",
  "$uses": ["DeliveryInstructions"],
  "street": "123 Main St",
  "city": "Anytown",
  "state": "QA",
  "zip": "00001",
  "instructions": "Leave at the back door"
}
```

- Add-Ins are extensions that can be plugged into types on demand.
  - You can activate add-ins at the document instance or meta-schema level.
  - Used to turn on companion specification features
- Offered in schema via *\$offers*, enabled in instance or metaschema via *\$uses*
- Activated add-ins are merged into the target type as if they were local definitions
- Add-ins are *abstract* and *\$extend* another type

# New JSON Schema – Internationalization & Aliasing

```
{
  "$schema": "https://schemas.vasters.com/{...}/v0",
  "$uses": ["Altnames"],
  "Person": {
    "type": "object",
    "altnames": {
      "json": "person_data",
      "lang:en": "Person",
      "lang:de": "Person"
    },
    "properties": {
      "firstName": {
        "type": "string",
        "altnames": {
          "json": "first_name",
          "lang:en": "First Name",
          "lang:de": "Vorname"
        }
      },
      "lastName": {
        "type": "string",
        "altnames": {
          "json": "last_name",
          "lang:en": "Last Name",
          "lang:de": "Nachname"
        }
      }
    },
    "required": ["firstName", "lastName"]
  }
}
```

- The *altnames* extension allows defining alternate names for types and properties.
- The keys in the *altnames* table indicate the purpose.
- *lang:{code}* is reserved for display names
- *json* is reserved for identifiers that are outside the permitted character range for identifiers.
- *altnums* exists for alternates to enum symbols.

# New JSON Schema – Scientific Units and Currencies

```
{
  "$schema": "https://schemas.vasters.com/{...}/v0",
  "$uses": ["Units"],
  "type": "object",
  "name": "Price",
  "properties": {
    "value": {
      "type": "decimal",
      "precision": 20,
      "scale": 2,
      "currency": "USD" }
  }
}
```

---

```
{
  "$schema": "https://schemas.vasters.com/{...}/v0",
  "$uses": ["Units"],
  "type": "object",
  "name": "Pressure",
  "properties": {
    "value": { "type": "double", "unit": "hPa" },
    "volume": { "type": "double", "unit": "m^3" },
  }
}
```

- We use schemas to exchange data structures between applications, including AI agents.
- Currencies and scientific units are extremely common points of miscommunication and confusion
- *currency, unit, and symbol* are extensions with well-defined value spaces aiming to limit that confusion

# New JSON Schema – Validation

- [3. Validation Keywords](#3-validation-keywords)
  - [3.1. Numeric Validation Keywords](#31-numeric-validation-keywords)
    - [3.1.1. `minimum`](#311-minimum)
    - [3.1.2. `maximum`](#312-maximum)
    - [3.1.3. `exclusiveMinimum`](#313-exclusiveminimum)
    - [3.1.4. `exclusiveMaximum`](#314-exclusivemaximum)
    - [3.1.5. `multipleOf`](#315-multipleof)
  - [3.2. String Validation Keywords](#32-string-validation-keywords)
    - [3.2.1. `minLength`](#321-minlength)
    - [3.2.2. `pattern`](#322-pattern)
    - [3.2.3. `format`](#323-format)
  - [3.3. Array and Set Validation Keywords](#33-array-and-set-validation-keywords)
    - [3.3.1. `minItems`](#331-minitems)
    - [3.3.2. `maxItems`](#332-maxitems)
    - [3.3.3. `uniqueItems`](#333-uniqueitems)
    - [3.3.4. `contains`](#334-contains)
    - [3.3.5. `maxContains`](#335-maxcontains)
    - [3.3.6. `minContains`](#336-mincontains)
  - [3.4. Object and Map Validation Keywords](#34-object-and-map-validation-keywords)
    - [3.4.1. `minProperties` and `minEntries`](#341-minproperties-and-minentries)
    - [3.4.2. `maxProperties` and `maxEntries`](#342-maxproperties-and-maxentries)
    - [3.4.3. `dependentRequired`](#343-dependentrequired)
    - [3.4.4. `patternProperties` and `patternKeys`](#344-patternproperties-and-patternkeys)
    - [3.4.5. `propertyNames` and `keyNames`](#345-propertynames-and-keynames)
    - [3.4.6. `has`](#346-has)
  - [3.5. Default Values](#35-default-values)
    - [3.5.1. `default`](#351-default)

- The optional *Validation* extension spec has all the validation constraints of JSON Schema you love.
- Enable with *\$uses* : ["Validation"]

# New JSON Schema – Conditional Composition

- [3. Composition and Evaluation Model](#3-composition-and-evaluation-model)
- [4. Conditional composition Keywords](#4-conditional-composition-keywords)
  - [4.1. `allOf`](#41-allof)
  - [4.2. `anyOf`](#42-anyof)
  - [4.3. `oneOf`](#43-oneof)
  - [4.4. `not`](#44-not)
  - [4.5. `if`/`then`/`else`](#45-ifthenelse)

- The optional *Conditional Composition* extension spec contains *allOf*, *anyOf*, *oneOf*, *not*, and *if/then/else*.
- Enable with  
*\$uses : ["Conditionals"]*
- Enabling conditionals means that you're no longer defining data types but a matching pattern.

