Microsoft

JSON Structure (json-structure.org)

Clemens Vasters
Principal Architect
Messaging & Real-Time Intelligence
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

- 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.

JSON Structure

- JSON Structure is a refactoring and extension of JSON Schema.
  - JSON Structure is in many ways a feature superset of JSON Schema and a pragmatic way forward, but the projects are independent. JSON Structure has a different focus.
- Goals:
  - The core specification focuses on data structure definitions, names and namespaces, a rich primitive type system, an extended set of compound types (**set**, **map**, **tuple**, **choice**), 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.

JSON Structure – Core

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

```
{
   "$schema": "https://json-structure.github.io/meta/core/v0/#",
   "type": "object",
   "name": "Person",
   "properties": {
      "firstName": { "type": "string" },
      "lastName": { "type": "string" },
      "dateOfBirth": { "type": "date" }
   },
   "required": ["firstName", "lastName"]
}
```

"type" now required

every type must be named

more primitive data types

JSON Structure – Extended Primitive Types

```
{
   "$schema": "https://json-structure.github.io/meta/core/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 data types

Type-specific annotations

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

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

JSON Structure – Compound Types: Set and Map

```
{
  "$schema": "https://json-structure.org/meta/core/v0/#",
  "$id": "https://example.com/schema/setmap-example",
  "name": "SetMapExample",
  "type": "object",
  "properties": {
    "tags": {
      "name": "tags",
      "type": "set",
      "values": {
        "type": "string"
      }
    },
    "translations": {
      "name": "translations",
      "type": "map",
      "values": {
        "type": "string"
      }
    }
  }
}
```

set: array with unique values

map: first class dictionary type

JSON Structure – Compound Types: Tuple

```
{
  "$schema": "https://json-structure.org/meta/core/v0/#",
  "$id": "https://example.com/schema/tuple-example",
  "name": "PersonTuple",
  "type": "tuple",
  "properties": {
    "name": {
      "type": "string"
    },
    "age": {
      "type": "int32"
    },
    "email": {
      "type": "string"
    }
  },
  "tuple": ["name", "age", "email"]
}
```

Tuple is similar to *object* but serializes into an array without labels, with fields being positional

```
["Alice", 42, "alice@example.com"]
```

JSON Structure – Compound Types: Choice

```json
{
  "$schema": "https://json-structure.org/meta/core/v0/#",
  "$id": "https://example.com/schema/vehicle-choice",
  "name": "Vehicle",
  "type": "choice",
  "selector": "type",
  "choices": {
    "car": { "$ref": "#/definitions/Car" },
    "bike": { "$ref": "#/definitions/Bike" }
  },
  "definitions": {
    "Car": {
      "name": "Car",
      "type": "object",
      "properties": {
        "make": { "type": "string" },
        "doors": { "type": "int32" }
      }
    },
    "Bike": {
      "name": "Bike",
      "type": "object",
      "properties": {
        "brand": { "type": "string" },
        "gears": { "type": "int32" }
      }
    }
  }
}
```

```json
{
 "car": {
   "make": "Toyota",
   "doors": 4
 }
}
```

```json
{
 "bike": {
   "brand": "Trek",
   "gears": 21
 }
}
```

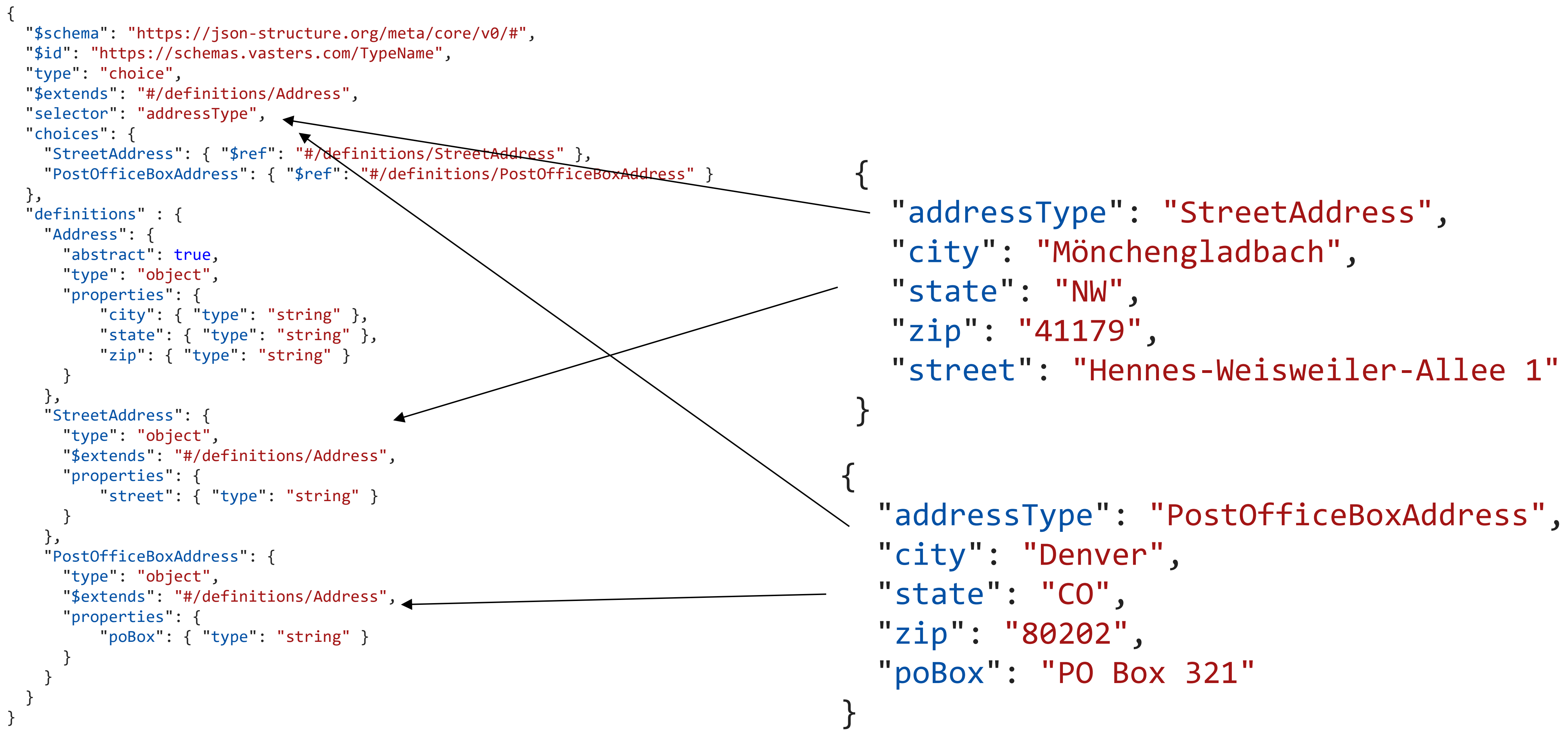
JSON Structure – Compound Types: Choice (Tagged)

```
{
  "$schema": "https://json-structure.org/meta/core/v0/#",
  "$id": "https://example.com/schema/vehicle-choice",
  "name": "Vehicle",
  "type": "choice",
  "selector": "type",
  "choices": {
    "car": { "$ref": "#/definitions/Car" },
    "bike": { "$ref": "#/definitions/Bike" }
  },
  "definitions": {
    "Car": {
      "name": "Car",
      "type": "object",
      "properties": {
        "make": { "type": "string" },
        "doors": { "type": "int32" }
      }
    },
    "Bike": {
      "name": "Bike",
      "type": "object",
      "properties": {
        "brand": { "type": "string" },
        "gears": { "type": "int32" }
      }
    }
  }
}
```

```
{
  "car": {
    "make": "Toyota",
    "doors": 4
  }
}
```

```
{
  "bike": {
    "brand": "Trek",
    "gears": 21
  }
}
```

JSON Structure – Compound Types: Choice (Inline)



```
{
  "$schema": "https://json-structure.org/meta/core/v0/#",
  "$id": "https://schemas.vasters.com/TypeName",
  "type": "choice",
  "$extends": "#/definitions/Address",
  "selector": "addressType",
  "choices": {
    "StreetAddress": { "$ref": "#/definitions/StreetAddress" },
    "PostOfficeBoxAddress": { "$ref": "#/definitions/PostOfficeBoxAddress" }
  },
  "definitions" : {
    "Address": {
      "abstract": true,
      "type": "object",
      "properties": {
          "city": { "type": "string" },
          "state": { "type": "string" },
          "zip": { "type": "string" }
      }
    },
    "StreetAddress": {
      "type": "object",
      "$extends": "#/definitions/Address",
      "properties": {
          "street": { "type": "string" }
      }
    },
    "PostOfficeBoxAddress": {
      "type": "object",
      "$extends": "#/definitions/Address",
      "properties": {
          "poBox": { "type": "string" }
      }
    }
  }
}
```

```
{
  "addressType": "StreetAddress",
  "city": "Mönchengladbach",
  "state": "NW",
  "zip": "41179",
  "street": "Hennes-Weisweiler-Allee 1"
}
```

```
{
  "addressType": "PostOfficeBoxAddress",
  "city": "Denver",
  "state": "CO",
  "zip": "80202",
  "poBox": "PO Box 321"
}
```

JSON Structure – Reusable Types

```
{
   "$schema": "https://json-structure.github.io/meta/core/v0/#",
   "type": "object",
   "name": "UserProfile",
   "properties": {
      "username": { "type": "string" },
      "dateOfBirth": { "type": "date" },
      "lastSeen": { "type": "datetime" },
      "score": { "type": "int64" },
      "balance": { "type": "decimal", "precision": 20, "scale": 2 },
      "isActive": { "type": "boolean" },
      "address": { "type" : { "$ref": "#/definitions/Address" } }
   },
   "required": ["username", "birthdate"],
   "definitions": {
      "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 *definitions*.
- 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 definitions
  - Must be a relative URI and cannot refer to outside the document

JSON Structure – Namespaces

```
{
   "$schema": "https://json-structure.github.io/meta/core/v0/#",
   "type": "object",
   "name": "UserProfile",
   "properties": {
      "username": { "type": "string" },
      "dateOfBirth": { "type": "date" },
      "networkAddress": { "type" : { "$ref": "#/definitions/Network/Address" } },
      "physicalAddress": { "type": { "$ref": "#/definitions/Physical/Address" } }
   },
   "required": ["username", "birthdate"],
   "definitions": {
      "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 *definitions*
- Top level in *definitions* 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*

JSON Structure – $import

```
{
  "$schema": "https://json-structure.github.io/meta/core/v0/#",
  "$id": "https://example.com/people.json",
  "name": "Person",
  "type": "object",
  "properties": {
    "firstName": { "type": "string" },
    "lastName": { "type": "string" },
    "address": { "$ref": "#/definitions/Address" }
  },
  "definitions": {
    "Address": {
      "type": "object",
      "properties": {
        "street": { "type": "string" },
        "city": { "type": "string" }
      }
    }
  }
}
```

```
{
  "$schema": "https://json-structure.github.io/meta/core/v0/#",
  "type": "object",
  "properties": {
    "person": {
      "type": { "$ref": "#/definitions/People/Person" }
    },
    "shippingAddress": {
      "type": { "$ref": "#/definitions/People/Address" }
    }
  },
  "definitions": {
    "People": {
      "$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
- *$importdefs* only loads the *definitions* section of the external schema, skipping the root type.
- After types have been imported, they may be used via *$ref* as if local
- *Shadowing* allows imported types to be overridden
- *$import* is an optional extension

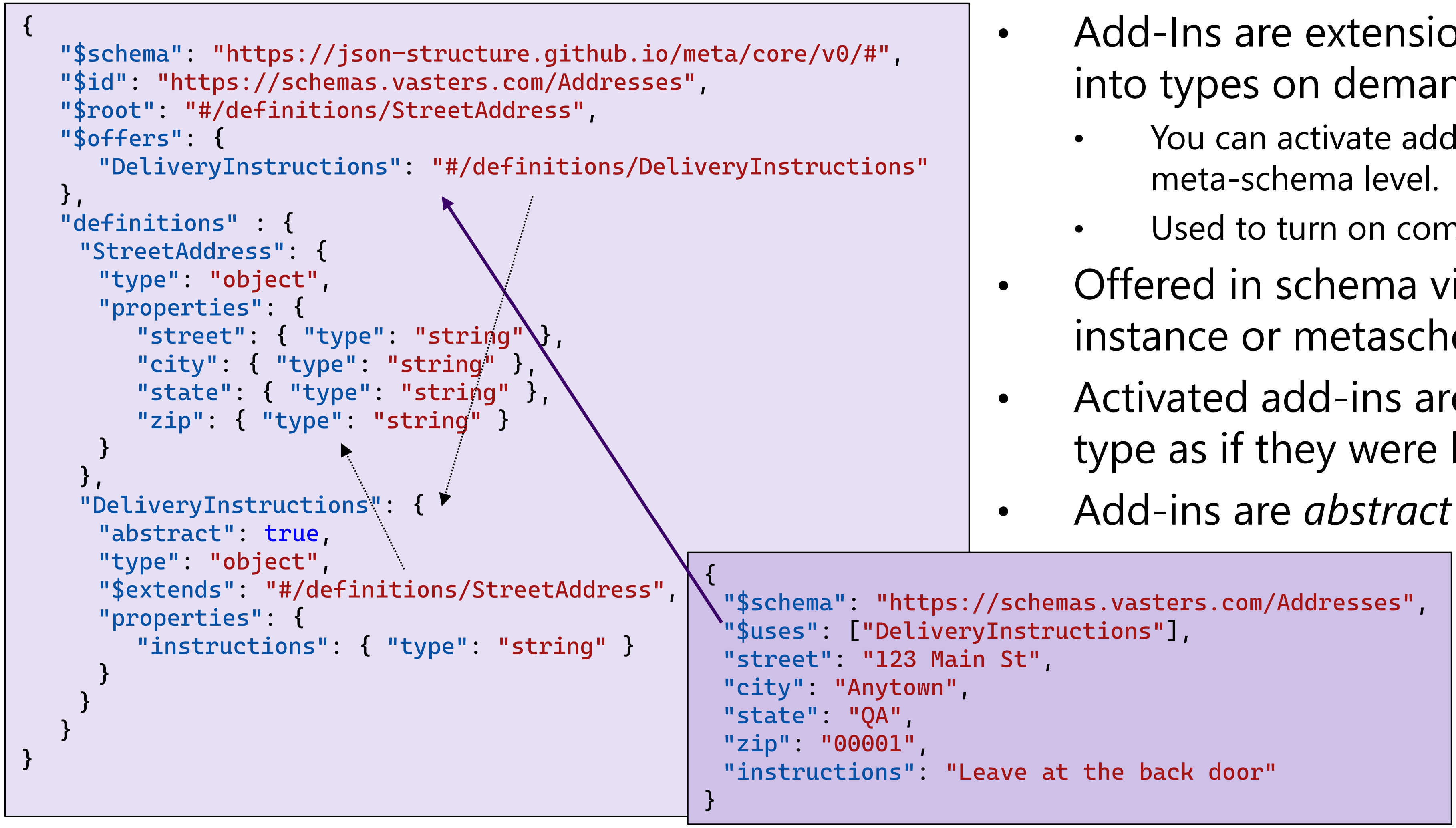
JSON Structure – *abstract* and *$extends*

```json
{
   "$schema": "https://json-structure.github.io/meta/core/v0/#",
   "definitions" : {
     "AddressBase": {
       "abstract": true,
       "type": "object",
       "properties": {
          "city": { "type": "string" },
          "state": { "type": "string" },
          "zip": { "type": "string" }
       }
     },
     "StreetAddress": {
       "type": "object",
       "$extends": "#/definitions/AddressBase",
       "properties": {
          "street": { "type": "string" }
       }
     },
     "PostOfficeBoxAddress": {
       "type": "object",
       "$extends": "#/definitions/AddressBase",
       "properties": {
          "poBox": { "type": "string" }
       }
     },
     "Address": {
       "type": "object",
       "$extends": "#/definitions/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

JSON Structure – Add-Ins



```
{
   "$schema": "https://json-structure.github.io/meta/core/v0/#",
   "$id": "https://schemas.vasters.com/Addresses",
   "$root": "#/definitions/StreetAddress",
   "$offers": {
     "DeliveryInstructions": "#/definitions/DeliveryInstructions"
   },
   "definitions" : {
    "StreetAddress": {
     "type": "object",
     "properties": {
        "street": { "type": "string" },
        "city": { "type": "string" },
        "state": { "type": "string" },
        "zip": { "type": "string" }
     }
    },
    "DeliveryInstructions": {
     "abstract": true,
     "type": "object",
     "$extends": "#/definitions/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

JSON Structure – Internationalization & Aliasing

```
{
   "$schema": "https://json-structure.github.io/meta/core/v0/#",
   "$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.
- *altenums* exists for alternates to enum symbols.

JSON Structure – Scientific Units and Currencies

```
{
   "$schema": "https://json-
structure.github.io/meta/core/v0/#",
   "$uses": ["Units"],
   "type": "object",
   "name": "Price",
   "properties": {
      "value": {
             "type": "decimal",
             "precision": 20,
             "scale": 2,
             "currency": "USD" }
   }
}
```

```
{
   "$schema": "https://json-
structure.github.io/meta/core/v0/#",
   "$uses": ["Units"],
   "type": "object",
   "name": "Pressure",
   "properties": {
      "value": { "type": "double", "unit": "hPa" },
      "volume": { "type": "double", "unit": "m^3" },
   }
}
```

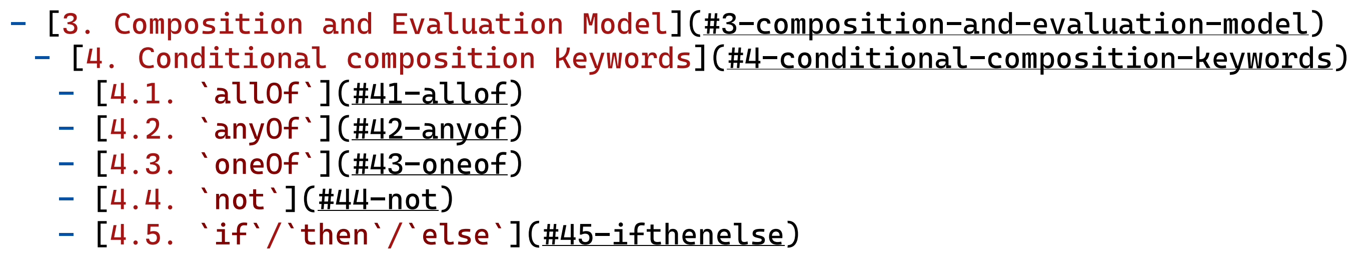
- We use schemas to exchange data structures between applications, including AI agents.
- Currencies and scientific units are extremely common points of miscommunication and confusion
- *currency, unit*, and *symbol* are extensions with well-defined value spaces aiming to limit that confusion

JSON Structure – 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"]*

JSON Structure – 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.

Microsoft