vlingo/schemata
Specification

This component is a schema registry. It provides the means for services built using the vlingo/platform to publish standard types that are made available to client services. The published standard types are known as schemas, and the registry hosts the schemas for given organizations and the services within. This is the basic hierarchy:

```
Organization
  Unit
    Context
      Commands
        Schema
          SchemaVersion (Specification, Version, Status)
          ...
        ...
      Data
        Schema
          SchemaVersion (Specification, Version, Status)
          ...
        ...
      Documents
        Schema
          SchemaVersion (Specification, Version, Status)
          ...
        ...
      Envelopes
        Schema
          SchemaVersion (Specification, Version, Status)
          ...
        ...
      Events
        Schema
          SchemaVersion (Specification, Version, Status)
          ...
        ...
```

From the top of the hierarchy the nodes are defined as follows:

- **Organization:** The top-level division. This may be the name of a company or the name of a prominent business division within a company. If there is only one company using this registry then the Organization could be a major division within the implied company. There may be any number of Organizations defined, but there must be at least one.
- **Unit:** The second-level division. This may be the name of a business division within the Organization, or if the Organization is a business division then the Unit may be a department or team within a business division. Note that there is no reasonable limit on the name of the Unit, so it may contain dot notation in order to provide additional organizational levels. In an attempt to maintain simplicity we don't want to provide nested Unit types because the Units themselves

can become obsolete with corporate and team reorganizations. However, renaming Units is potentially dangerous since it can break current dependencies. Thus, it is best to name a Unit according to some non-changing business function rather than physical departments, etc.

- **Context:** The logical application or (micro)service within which schemas are to be defined and for which the schemas are published to potential consumers. You may think of this as the name of the *Bounded Context,* and it may even be appropriate to name it the top-level namespace used by the Context, e.g. `com.saasovation.agilepm`. Within each Context there may be a number of parts used to describe its *Published Language* served by its *Open Host Service.* Currently these include: Commands, Data, Documents, Envelopes, and Events. Some of the parts are meant to help defined other parts, and so are building blocks. Other parts are the highest level of the *Published Language.* These are called out in the following definitions.
- **Commands:** This is a top-level schema type where Command operations, such as those supporting CQRS, are defined by schemas. If the Context's *Open Host Service* is REST-based, these would define the payload schema submitted as the HTTP Request body of POST, PATCH, and DELETE methods. If the *Open Host Service* is asynchronous-message-based mechanism (e.g. RabbitMQ or Kafka), these would define the payload of Command messages sent through the messaging mechanism.
- **Data:** This is a building-block schema type where general-purpose data records, structures, or objects are defined and that may be used inside any of the other schema types (e.g. type `Token`). You may also place metadata types here (e.g. type `Metadata` or more specifically, type `CauseMetadata`).
- **Documents:** This is a top-level schema type that defines the full payload of document-based operations, such as the query results of CQRS queries. These documents are suitable for use as REST Response bodies and messaging mechanism payloads.
- **Envelopes:** This is a building-block schema type meant to define the few number of message envelopes that wrap message-based schemas. When sending any kind of message, such as Command messages and Event messages, it is common to wrap these in an Envelope that defines some high-level metadata about the messages being sent by a sender and being received by a receiver.
- **Events:** This is a top-level schema type that conveys the facts about happenings within the Context that are important for other Context's to consume. These are known as *Domain Events* but may also be named *Business Events.* The reason for the distinction is that some viewpoints consider Domain Events to be internal-only events; that is, those events only of interest to the owning Context. Those holding that viewpoint think of events of interest outside the owning Context as Business Events. To avoid any confusion the term Event is used for this schema type and may be used to define any event that is of interest either inside or outside the owning Context, or both inside and outside the owning Context.
- **Schema:** Under every top-level schema Category (or type, such as Commands and Events) are any number of Schema definitions. Besides a category, a Schema has a name and description. Every Schema has at least one Schema Version, which holds the actual Specification for each version of the Schema. Thus, the Schema itself is a container for an ordered collection of Schema Versions that each have a Specification.
- **Schema Version:** Every Schema has at least one Schema Version, and may have several versions. A Schema Version holds the Specification of a particular version of the Schema, and also holds a Description, a Semantic Version number, and a Status. The Description is a

textual/prose description of the purpose of the Schema Version.

- **Specification:** A Schema Version's Specification is a textual external DSL (code block) that declares the data types and shape of the Schema at a given version. Any new version's Specification must be backward compatible with previous versions' of the given Schema if the new version falls within the same major version. The DSL is shown in detail below.
- **Semantic Version:** A semantic version is a three-part version, with a major, minor, and patch value, with each subsequent version part separated by a dot (decimal point), such as **1.2.3** for example. Here 1 is the major version, 2 is the minor version, and 3 is the patch version. If any two Schema Versions share the same major version then it is required that their Specifications must be compatible with each other. Thus, the newer version, such as 1.2.x, must be compatible with the Specification of 1.1.x, and 1.1.x must be compatible with 1.2.x. In this, the x is any patch version.
- **Status:** The Schema Version Status has three possible values, Draft, Published, and Removed. The Draft is the initial status and means that the Specification is unofficial and may change. Dependents may still use a Draft status Schema Version for test purposes, but with the understanding that the Specification may change at any time. When a Schema Version is considered production-ready, it's status is upgraded to Published. A Published Schema Version may never be removed and all future versions that share the same major Semantic Version must be backward compatible. Marking a Schema Version as Published is performed manually by the Context team after it has satisfied it's team and consumer dependency requirements. If, for some reason, it is necessary to forever remove a Draft status Schema Version, it can be marked as Removed status. Since this Removed status is a soft removal, it may be restored to Draft and later promoted to Published, but only if another Schema Version has not superseded it.

Schema Version Specification DSL

The following is an example of a generic, bulk DSL that demonstrates all the features supported by the typing language. Note that the leading **`type`** keyword is one of the concrete category types: **`command data document envelope event`**

```
type TypeName {
  type typeAttribute;
  version versionAttribute;
  timestamp timestampAttribute;

  boolean booleanAttribute = true
  boolean[] booleanArrayAttribute { true, false, true }
  byte byteAttribute = 0
  byte[] byteArrayAttribute { 0, 127, 65 }
  char charAttribute = 'A'
  char[] charArrayAttribute = { 'A', 'B', 'C' }
  double doubleAttribute = 1.0
  double[] doubleArrayAttribute = { 1.0, 2.0, 3.0 }
  float floatAttribute = 1.0
  float[] floatArrayAttribute = { 1.0, 2.0, 3.0 }
```

```
  int intAttribute = 123
  int[] intArrayAttribute = { 123, 456, 789 }
  long longAttribute = 7890
  long[] longArrayAttribute = { 7890, 1234, 5678 }
  short shortAttribute = 32767
  short[] shortArrayAttribute = { 0, 1, 2 }
  string stringAttribute = "abc"
  string[] stringArrayAttribute = { "abc", "def", "ghi" }
  TypeName typeNameAttribute1
  category.TypeName typeNameAttribute2
  category.TypeName:1.2.1 typeNameAttribute1
  category.TypeName:1.2.1[] typeNameArrayAttribute1
}
```

The following table describes the available types and a description of each.

Datatype	Description
`type`	The datatype specifically defining that the type-name of the specification type should be included in the message itself with the given attribute name. (Note that this may be placed instead on the Envelope.)
`version`	The datatype specifically defining that the semantic version of the given Schema Version should be included in the message itself with the given attribute name. (Note that this may be placed instead on the Envelope.)
`timestamp`	The datatype specifically defining that the timestamp of when the given instance was created to be included in the message itself with the given attribute name. (Note that this may be placed instead on the Envelope.)
`boolean`	The boolean datatype with values of true and false only, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and a true or false: `boolean flag = true`
`boolean[]`	The boolean array datatype with multiple values of true and false only, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and an array literal containing a

	number of true and false values: `boolean[] flags = { true, false, true }`
byte	The 8-bit signed byte datatype with values of -128 to 127, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and a true or false: `byte small = 123`
byte[]	The 8-bit signed byte array datatype with multiple values of - 128 to 127, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and an array literal containing a number of byte values: `byte[] smalls = { 1, 12, 123 }`
char	The char datatype with values supporting UTF-8, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and a character literal: `char initial = 'A'`
char[]	The char array datatype with multiple UTF-8 values, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and an array literal containing a number of character values: `char[] initials = { 'A', 'B', 'C' }`
double	The double-precision floating point datatype, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and a double literal: `double pi = 3.1416`
double[]	The double-precision floating point array datatype, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and an array literal containing a number of double values: `double[] stats = { 1.54179, 7.929254, 32.882777091 }`
float	The single-precision floating point datatype, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and a float literal: `float pi = 3.14`
float[]	The single-precision floating point array datatype, to be included in the message with the given

	attribute name. This value may be defaulted if the declaration is followed by an equals sign and an array literal containing a number of float values: `float[] stats = { 1.54, 7.92, 32.88 }`
int	The 32-bit signed integer datatype with values of -2,147,483,648 to 2,147,483,647, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and an integer literal: `int value = 885886279`
int[]	The 32-bit signed integer datatype with multiple values of -2,147,483,648 to 2,147,483,647, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and an array literal containing a number of integer values: `int[] values = { 885886279, 77241514, 9772531 }`
long	The 64-bit signed integer datatype with values of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and a long literal: `long value = 15329885886279`
long[]	The 64-bit signed integer datatype with multiple values of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and an array literal containing a number of long values: `long[] values = { 15329885886279, 24389775639272, 45336993791291 }`
short	The 16-bit signed integer datatype with values of -32,768 to 32,767, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and a short literal: `short value = 12986`
short[]	The 16-bit signed integer datatype with multiple values of -32,768 to 32,767, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and an array literal containing a

	number of short values: `short[] values = { 12986, 3772, 10994 }`
`string`	The string datatype with values supporting multi-character UTF-8 strings, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and a character literal: `string value = "ABC"`
`string[]`	The string array datatype with multiple values supporting multi-character UTF-8 strings, to be included in the message with the given attribute name. This value may be defaulted if the declaration is followed by an equals sign and an array literal containing a number of string values: `string[] initials = { "ABC", "DEF", "GHI" }`
`category.TypeName`	The explicit complex Schema type of a given Category in the current Context to be included in the message with the given attribute name. The version is the tip, as in the most recent version. There is no support for default values other than null, which may be supported using the Null Object pattern.
`category.TypeName[]`	The explicit complex Schema type array of a given Category in the current Context to be included in the message with the given attribute name. The version is the tip, as in the most recent version. There is no support for default values.
`category.TypeName:1.2.3`	The explicit complex Schema type of a given Category in the current Context to be included in the message with the given attribute name. The version is the one declared following the colon (:). There is no support for default values other than null, which may be supported using the Null Object pattern.
`category.TypeName:1.2.3[]`	The explicit complex Schema type array of a given Category in the current Context to be included in the message with the given attribute name. The version is the one declared following the colon (:). There is no support for default values.

Any given complex Schema type may be included in the Specification, but doing so may limit to some extent consumption across multiple collaborating technical platforms. We make every effort to ensure cross-platform compatibility, but the chosen serialization type may be a limiting factor. We thus

consider this an unknown until full compatibility can be confirmed.

An additional warning is appropriate regarding direct domain model usage of Schema types. These Schema types are not meant to be used as first-class domain model entities/aggregates or value objects. These Events category types may be used as Domain Events in the domain model, but if so we strongly suggest keeping the specifications simple (not include complex types). In other words, define your domain model entities and value objects in your domain model code, not using a Schema Specification. Schema Specifications are primarily about data and expressing present and past intent, not behavior. Considered Schema Specification to be more about local-Context migrations of supported Domain Events and inter-Context collaboration and integration of all other Schema types.

Working with Schema Specifications and Schema Dependencies

More to follow...