SCHEMATA

Mariano Montone (marianomontone@gmail.com)

Table of Contents

1 Introduction

Generic purpose schema library for serialization and validation of data.

This library is used by CL-REST-SERVER for API serialization and validation.

It features:

- Validation using schemas.
- Serialization and unserialization using schemas.
- Integration with Common Lisp type system.
- A schema class metaclass.

2 Installation

With Quicklisp:

```
(ql:quickload "schemata")
```

3 Usage

3.1 Schema definition

Use [DEFINE-SCHEMA], page 8for defining schemas.

Schema example:

```
(schemata:define-schema customer
    (object "customer"
            ((id string :external-name "id" :accessor
                customer-id :documentation "customer id")
             (number string :external-name "number" :required nil
                             :accessor customer-nr :documentation
                  "customer number")
             (name string :external-name "name" :accessor
                  customer-name :documentation "customer name")
             (address-1 string :external-name "address1"
                             :required nil :documentation
                  "customer first address")
             (address-2 string :external-name "address2"
                             :required nil :documentation
                  "customer second address")
             (postal-code string :external-name "postalcode"
                             :required nil :documentation
                  "postal code")
             (postal-area string :external-name "postalarea"
                             :required nil :documentation
                  "postal area")
             (country string :external-name "country" :required nil
                  :documentation "country code")
             (phone string :external-name "phone" :required nil
                        :documentation "phone")
             (fax string :external-name "fax" :required nil
                        :documentation "fax")
             (email string :external-name "email" :required nil
                        :documentation "email"))
            (:documentation "customer data fetched")))
```

3.2 Validation using schemas

Use [VALIDATE-WITH-SCHEMA], page 9.

3.3 Serialization using schemas

Via *generic-serializer* library.

Use [SERIALIZE-WITH-SCHEMA], page 9.

```
(with-output-to-string (s)
```

```
        (gs:with-serializer-output s
            (gs:with-serializer :json
               (serialize-with-schema *schema* user))))
    (with-output-to-string (s)
      (gs:with-serializer-output s
        (gs:with-serializer :json
          (serialize-with-schema
           (find-schema 'user-schema) *user*))))
```

3.4 Unserialization using schemas

Use [UNSERIALIZE-WITH-SCHEMA], page 8.

```
      (unserialize-with-schema
        (find-schema 'user-schema)
        (json:decode-json-from-string data)
        :json)
```

3.5 Patch and updates

See [PATCH-WITH-SCHEMA], page 9and [POPULATE-WITH-SCHEMA], page 9.

4 Schema types

4.1 Type schemas

Schemas can be built from Common Lisp types:

```
SCHEMATA> (defparameter *s* (schema string))
*S*
SCHEMATA> *s*
#<TYPE-SCHEMA STRING {1006FBBD13}>
SCHEMATA> (validate-with-schema *s* "22")
NIL
SCHEMATA> (validate-with-schema *s* 22 :error-p nil)
#<VALIDATION-ERROR "~s is not of type: ~a" {100152EB13}>
```

4.2 Object schema

Object schemas are built using the syntax: `(object name attributes options)`. Attributes are specified as: `(attribute-name attribute-type &rest options)`.

The `attribute-type` is parsed as a schema.

Possible attribute options are: 'required', 'required-message', 'default', 'accessor', 'writer', 'reader', 'parser', 'validator', 'add-validator', 'formatter', 'external-name', 'serializer', 'unserializer', 'slot'.

Example:

```
SCHEMATA> (schema (object person
                          ((name string)
                           (age integer :required nil))))
#<OBJECT-SCHEMA {1001843543}>
```

4.3 List schema

Homogeneous list of schemas are specified via `list-of`.

Example:

```
SCHEMATA> (defparameter *s* (schema (list-of integer)))
*S*
SCHEMATA> (validate-with-schema *s* '(1 2 "foo"))
; Evaluation aborted on #<SCHEMATA:VALIDATION-ERROR "~s is not of type: ~a" {1006ECA323}>.
SCHEMATA> (validate-with-schema *s* '(1 2 3))
NIL
```

4.4 Schema references

Defined schemas can be referenced via either '(schema schema-name)' or '(ref schema-name)' (they are identical).

Example:

```
SCHEMATA> (define-schema person
```

```
                    (object person
                            ((name string))))
#<OBJECT-SCHEMA {1006F8A813}>
SCHEMATA> (defparameter *list-of-person* (schema (list-of (ref person))))
*LIST-OF-PERSON*
SCHEMATA> *list-of-person*
#<LIST-SCHEMA {1006F8C2A3}>
SCHEMATA> (parse-with-schema *list-of-person* '((("name" . "Mariano")) (("name" . "Peter"))
((("name" . "Mariano")) (("name" . "Peter"))
(((NAME . "Mariano")) ((NAME . "Peter")))
SCHEMATA> (validate-with-schema *list-of-person* '((("name" . 22)) (("name" . "Peter"))))█
; processing (DEFMETHOD SCHEMA-VALIDATE ...); Evaluation aborted on #<SB-PCL::NO-APPLICABLE
SCHEMATA> (validate-with-schema *list-of-person* '((("name" . 22)) (("name" . "Peter"))))█
; Evaluation aborted on #<SCHEMATA:VALIDATION-ERROR "~s is not of type: ~a" {10082EB883}>.█
SCHEMATA> (validate-with-schema *list-of-person* '((("name" . "Mariano")) (("name" . "Peter
NIL
SCHEMATA> (validate-with-schema *list-of-person* '((("names" . "Mariano")) (("name" . "Pete
; Evaluation aborted on #<SCHEMATA:VALIDATION-ERROR "Attributes not part of schema: ~a" {10
```

4.5 SATISFIES-SCHEMA type

Schemata integrates with the Lisp type system via the SATISFIES-SCHEMA type. Schemas
can be thought as types over data. Defined schemas can be checked using TYPEP and
CHECK-TYPE with the type '(satisfies-schema schema-name)'.

    Example:
```
SCHEMATA> (define-schema string-schema string)
#<TYPE-SCHEMA STRING {10019DA8B3}>
SCHEMATA> (typep "foo" '(satisfies-schema string-schema))
T
SCHEMATA> (typep 22 '(satisfies-schema string-schema))
NIL
SCHEMATA> (let ((x "foo"))
            (check-type x (satisfies-schema string-schema))
            x)
"foo"
```

4.6 SCHEMA-CLASS metaclass

SCHEMA-CLASS classes get an schema attached.

    Example:
```
SCHEMATA> (define-schema-class person ()
            ((name :type string :initarg :name)
             (age :type integer :required nil :initarg :age)))
#<SCHEMA-CLASS SCHEMATA::PERSON>

SCHEMATA> (validate-with-schema (find-class 'person) '(("name" . "Mariano") ("age" . 22)))█
NIL
```

```
SCHEMATA> (validate-with-schema (find-class 'person) '(("name" . "Mariano") ("age" . 'asdf)
#<VALIDATION-ERROR 'ASDF is not of type: INTEGER {100109F833}>

SCHEMATA> (generic-serializer:with-serializer :json
             (generic-serializer:serialize (make-instance 'person :name "Mariano" :age 44)))
{"name":"Mariano","age":44}
```

5 Reference

5.1 SCHEMATA package

SCHEMATA   [PACKAGE]

External definitions

Macros

SCHEMATA:SCHEMA (*schema-def*)   [Macro]
    Wrapper macro for schema definitions.

SCHEMATA:DEFINE-SCHEMA (*name schema*)   [Macro]
    Register SCHEMA under NAME. The schema can then be accessed via
    FIND-SCHEMA.

Generic functions

SCHEMATA:PARSE-WITH-SCHEMA (*schema string-or-data*)   [Generic-Function]
    Parses the string to an association list using the schema

SCHEMATA:ATTRIBUTE-NAME (*sb-pcl::object*)   [Generic-Function]

SCHEMATA:UNSERIALIZE-WITH-SCHEMA (*schema data format*) [Generic-Function]

SCHEMATA:SCHEMA-DOCUMENTATION (*sb-pcl::object*)   [Generic-Function]

SCHEMATA:OBJECT-CLASS (*sb-pcl::object*)   [Generic-Function]

SCHEMATA:ATTRIBUTE-PARSER (*sb-pcl::object*)   [Generic-Function]

SCHEMATA:ATTRIBUTE-VALIDATOR (*sb-pcl::object*)   [Generic-Function]

SCHEMATA:ATTRIBUTE-ADD-VALIDATOR (*sb-pcl::object*)   [Generic-Function]

SCHEMATA:ATTRIBUTE-EXTERNAL-NAME (*sb-pcl::object*)   [Generic-Function]

SCHEMATA:SCHEMA-TYPE (*sb-pcl::object*)   [Generic-Function]

SCHEMATA:ATTRIBUTE-FORMATTER (*sb-pcl::object*)   [Generic-Function]

SCHEMATA:OBJECT-NAME (*sb-pcl::object*)   [Generic-Function]

SCHEMATA:OBJECT-ATTRIBUTES (*sb-pcl::object*)   [Generic-Function]

SCHEMATA:ATTRIBUTE-ACCESSOR (*sb-pcl::object*)   [Generic-Function]

SCHEMATA:ATTRIBUTE-TYPE (*sb-pcl::object*)   [Generic-Function]

Functions

SCHEMATA:ATTRIBUTE-READER (*attribute*) [Function]

SCHEMATA:POPULATE-WITH-SCHEMA (*schema object data* **&key** [Function]
       exclude)
    Populate CLOS objects from data + schema. Attributes members of EXCLUDE
    parameter are not populated.

SCHEMATA:SCHEMA-CLASS-SCHEMA (*schema-class*) [Function]
    Generate a schema using the schema class meta info

SCHEMATA:SERIALIZE-WITH-SCHEMA (*schema input* **&optional** [Function]
       (serializer generic-serializer::*serializer*) (stream
       generic-serializer::*serializer-output*))

SCHEMATA:SCHEMA-SPEC (*schema*) [Function]

SCHEMATA:ATTRIBUTE-TYPE-NAME (*attribute*) [Function]

SCHEMATA:VALIDATION-ERROR (*message* **&rest** *args*) [Function]

SCHEMATA:VALIDATE-WITH-SCHEMA (*schema data* **&key** *(collect-errors* [Function]
       collect-validation-errors) (error-p *signal-validation-errors*))
    Validate input using schema. Useful for validating resource operations posted content
    (for :post and :put methods). Input can be a string or an association list.

    Args: - schema (symbol or schema): The schema - data (alist): The data to validate.
    - format (keyword): The data format. - collect-errors (boolean): If true, collect all
    the validation errors. If false, return the first validation error found. Default: true.
    - error-p (boolean): If true, when validation errors are found, a validation error is
    signaled. If false, the validation errors are returned as the function result and no
    error is signaled.

SCHEMATA:ATTRIBUTE-OPTIONAL-P (*attribute*) [Function]

SCHEMATA:ATTRIBUTE-WRITER (*attribute*) [Function]

SCHEMATA:FIND-OBJECT-ATTRIBUTE (*object attribute-name* **&key** [Function]
       (error-p t))

SCHEMATA:FIND-SCHEMA (*name* **&optional** *(errorp t))* [Function]
    Find a schema definition by name

SCHEMATA:PATCH-WITH-SCHEMA (*schema object data*) [Function]
    Populate CLOS objects from data + schema. Only populates attributes available
    in DATA, validating them. Useful for PATCH rest api operations implementations.
    DATA should be an association list.

Classes

SCHEMATA:OBJECT-SCHEMA [Class]
    Class precedence list: `object-schema, schema, standard-object, t`

Slots:

- `name` — type: `(or string symbol)`; initarg: `:name`; reader: `schemata:object-name`; writer: `(setf schemata:object-name)`

  The name of the object.

- `attributes` — type: `list`; initarg: `:attributes`; reader: `schemata:object-attributes`; writer: `(setf schemata:object-attributes)`

- `class` — type: `(or null symbol)`; initarg: `:class`; reader: `schemata:object-class`; writer: `(setf schemata:object-class)`

- `ignore-unknown-attributes` — type: `boolean`; initarg: `:ignore-unknown-attributes`; reader: `schemata::ignore-unknown-attributes`; writer: `(setf schemata::ignore-unknown-att`

- `serializer` — type: `(or null trivial-types:function-designator)`; initarg: `:serializer`; reader: `schemata::object-serializer`; writer: `(setf schemata::object-serializer)`

- `unserializer` — type: `(or null trivial-types:function-designator)`; initarg: `:unserializer`; reader: `schemata::object-unserializer`; writer: `(setf schemata::object-unserializer)`

`SCHEMATA:SCHEMA-REFERENCE-SCHEMA`   [Class]

Class precedence list: `schema-reference-schema, schema, standard-object, t`

Slots:

- `name` — type: `symbol`; initarg: `:schema-name`; reader: `schemata::schema-name`; writer: `(setf schemata::schema-name)`

`SCHEMATA:SCHEMA`   [Class]

Class precedence list: `schema, standard-object, t`

Slots:

- `documentation` — type: `t`; initarg: `:documentation`; reader: `schemata:schema-documentation`; writer: `(setf schemata:schema-documentation)`

`SCHEMATA:VALIDATION-ERROR`   [Class]

Class precedence list: `validation-error, error, serious-condition, condition, t`

`SCHEMATA:ATTRIBUTE`   [Class]

Class precedence list: `attribute, schema, attribute-properties, standard-object, t`

Slots:

- `name` — type: `symbol`; initarg: `:name`; reader: `schemata:attribute-name`; writer: `(setf schemata:attribute-name)`

- `type` — type: `schemata:schema`; initarg: `:type`; reader: `schemata:attribute-type`; writer: `(setf schemata:attribute-type)`

- `accessor` — type: `(or null symbol)`; initarg: `:accessor`; reader: `schemata:attribute-accessor`; writer: `(setf schemata:attribute-accessor)`

- `writer` — type: `(or null trivial-types:function-designator)`; initarg: `:writer`

- reader — type: (or null trivial-types:function-designator); initarg: :reader
- slot — type: (or null symbol); initarg: :slot; reader: schemata::attribute-slot; writer: (setf schemata::attribute-slot)

SCHEMATA:TYPE-SCHEMA [Class]

Class precedence list: type-schema, schema, standard-object, t

Slots:

- type — type: t; initarg: :type; reader: schemata:schema-type; writer: (setf schemata:schema-type)

SCHEMATA:SCHEMA-OBJECT [Class]

Class precedence list: schema-object, standard-object, t

SCHEMATA:SCHEMA-CLASS [Class]

Metaclass for schema objects

Class precedence list: schema-class, standard-class, class, specializer, metaobject, standard-object, t

Slots:

- schema-name — type: (or null string symbol); initarg: :schema-name; reader: schemata::schema-name; writer: (setf schemata::schema-name)

6 Index

(Index is nonexistent)

*

S