Data-transformer: an example of data-centered tool set

Michael A. Raskin*
Moscow Center for Continuous Mathematical Education
119002 Moscow
Bolshoy Vlasyevskiy 11
Russia
raskin@mccme.ru

ABSTRACT

This paper describes the data-transformer library, which provides various input and output routines for data based on a unified schema. Currently, the areas of the library's use include storage and retrieval of data via CLSQL; processing CSV and similar tabular files; interaction with user via web forms. Using the supplied schema, the data-transformer library can validate the input, process it and prepare it for output. A data schema may also include channel-specific details, e.g. one may specify a default HTML textarea size to use when generating the forms.

DOI: 10.5281/zenodo.3254742

1. INTRODUCTION

Processing medium and large arrays of data often encounters the problem of the poor data quality. Use of human input may even lead to incorrect data formats. The same problem may occur when automated systems written by independent teams for completely unrelated tasks have to interact. Usage of wrong data formats may happen completely unexpectedly: spreadsheet processors sometimes convert 555-01-55 to 555-01-1955 just in case.

Finding and fixing such mistakes (both in the manually entered data and in the exchange format handling) usually relies on the automated verification. Of course, verification is implemented differently depending on the needs of the application in question.

Many software products use formally defined data schemas for codifying the structure of the data being handled. We could name XML Schemas and SQL database schemas among the examples of such formal schemas. XML schemas are declarative and SQL schemas are usually understood in a declarative way; this improves portability but sometimes restricts functionality.

*This work was partially supported by RFBR grant 12-01-00864-a

The data-transformers library is a library with the opposite approach: it tries to provide consistent handling with many small features without any hope for portability. Simple things are defined declaratively, but in many cases pieces of imperative code are included inside the schema. Moreover, declarative schemas for the separate data exchange interfaces can be generated out of a single data-transformer schema.

2. SCOPE

Initially this library has been written to support complex validations when parsing CSV files. So the focus of this specific library is on validating simple records one-by-one; validating hierarchical data sets is not in the scope. Handling of more complex data structures is done by wrappers which use the data-transformer library functions to handle each specific level of the hierarchy.

Interface of the data-transformer library supports many various operations, but all of them are applied to a single record at a time (although the record fields may contain complex data types for many of the operations).

3. DATA MODEL

The data transformers are defined by the schemas, usually written as s-expressions. For actual data processing a data-transformer class instance is created. It holds the data format definition, the data entry currently processed, and the errors found during processing.

The record format is defined as an array of fields; each field description (instance of the field-description class) holds the specified parameters from the schema and a cache of the computed default parameter values. All supported field parameters and the rules for deducing default values when necessary are defined inside the data-transformer library, although it is easy to add additional field parameters after loading the main data-transformer library.

The default values for the field parameters usually depend on the values of the other parameters.

The data held inside a data-transformer instance is an array of the values (in the same order as the fields); this array is supposed to contain the current representation of data according to the current needs of the application. A data-

transformer instance is not meant to store the data for a long period of time, it only keeps the performed conversions uniform.

4. INPUT VERIFICATION

Let us consider a typical data flow. The data-transformer library is used to validate the data in a CSV file and put it into an SQL database. The process goes as follows. Check that the text looks like something meaningful, then parse it into the correct data type, check that this value doesn't violate any constraints, check that there are no broken cross-field constraints, combine the fields when the input requirements and the storage requirements are different (think of the date formats), generate an SQL statement to save the data or pass the list of errors to the UI code.

For example, a birth date is usually split into three columns in our CSV forms. The text format validation ensures that the day, the month and the year represent three integer numbers; parsing is done by parse-integer; the single-field validation ensures that year is in a generally reasonable range; the cross-field validation ensures that such a date exists (i.e. 2013, 04 and 31 are legal values for a year, a month, and a day number, but 2013-04-31 is not a legal date); and finally the data is formatted into the YYYY-MM-DD ISO format for the insertion into the database.

5. EXAMPLE DEFINITION

Example piece of a schema:

```
(defparameter *basic-schema*
  `(((:code-name :captcha-answer)
     (:display-name "Task answer")
     (:type :int)
     (:string-verification-error
     "Please enter a number")
     (:data-verification-error "Wrong answer")
     (:string-export ,(constantly "")))
    ((:code-name :email)
     (:display-name "Email")
     (:type :string)
     , (matcher "^(.+@.+[.].+|)$")
         (:string-verification-error
   "Email is specified but it doesn't
   look like a valid email address"))))
(let
  ((schema (transformer-schema-edit-field
              *basic-schema* :captcha-answer
       (lambda (x)
         (set-> x :data-verification
  (lambda (y)
    (and y (= y captcha-answer))))))))

  ; some code using the schema
  )
```

This description (a mangled piece of a registration form) illustrates the following attributes:
1) A code name for generating HTML, SQL and similar field identifiers and a human readable name used for generating the labels for HTML forms, CSV tables etc.
2) Types of the individual record fields. In our system the types are used mainly for generating the SQL table definitions.
3) Verification procedures. For the integer fields checking against the regular expression **" *[+-]?[0-9]+[.]? *"** is the default text format check, so it is not specified. Although the default check is used, a custom error message is specified for use when the format requirements are not met. The data verification is added to the schema later, right before the actual use. Note that the second verification step may rely on the previous ones to ensure that it is passed nil or a valid number.
4) Data formatting procedure. In this case, if the user entered a wrong CAPTCHA answer, there is no point in showing them their old answer, so we clear the field, instead.

6. CURRENTLY USED INPUT AND OUTPUT CHANNELS

Initial scope: CSV and SQL.

The only feature which is mostly specific for the CSV support is support for specifying a single date field and getting separate fields for day, month and year with verification and aggregation specified correctly by default. This is used in some of our web forms, too.

SQL-specific features are more numerous. A record definition has to contain the list of the relevant fields in the database; there is support for adding extra parameters and specifying the WHERE-conditions and the source tables as well. To simplify generating the SQL table definitions and the queries, one may specify the foreign keys as such where appropriate.

Creating a nice HTML output is quite similar to exporting data in the CSV format from the data viewpoint, one just needs a template. However, validating the web forms has some specifics. The data-transformer library supports generating the input fields for web forms, getting the field values from a Hunchentoot request object, handling the uploaded files (they are stored in a separate directory and the file paths are put into the database), preparing the data for use by the CL-Emb templates etc. Some of this functionality is also used for generating printable PDF documents (CL-Emb supports generating the TEXcode just as well as generating HTML).

7. WHAT DOES AND DOESN'T WORK FOR US

The data-transformer library has been started to avoid a mismatch between two data format descriptions (the code that validates the CSV files and the SQL schema) and unify validation. It grows organically, and therefore it is sometimes inconsistent.

When we started the project that includes the data-transformer library, we were looking for a good validation library designed to support many data exchange channels and have not found any that met our needs. I guess I shall look better if I have to start next big project from scratch. Or maybe I will just take this code.

It is nice to have the field definitions always in sync, of course. Although it is still possible that data storage and data loading procedures are slightly mismatched, this problem almost never occurs.

As one can see, we freely use Lisp code inside the data format definition. This means that we don't care about portability. On the bright side, this means that we can easily perform any needed check. For example, some of our online registrations for various events can check whether there is enough room for one more participant via a DB query. It is done inside the verification procedure.

It turned out that the lack of portability means that the data schemas are tied not only to the Common Lisp language itself. The code is also coupled with the tools we use (Hunchentoot, CLSQL, CL-Emb etc.) as well.

The excessive flexibility helps in an understaffed project. For example, there is some code for the event registrations. The main procedures of this code are almost never changed; the changes are mostly restricted to the page templates and fields lists, where all the definitions are relatively short.

The main problem is the frequent lack of time to find out a way to remove small annoyances. Sometimes some code repeatedly uses the library almost in the same way multiple times, and it is hard to find a good way to express these patterns. But it is probably not specific to our library.

Some functionality is still implemented in a wasteful way. For example, currently validating web form submission iterates over the field list and checks whether a validation procedure is defined for each field. It would be nice to allow generating a progn with all the verifying code included to remove the unnecessary branching (and the iteration over the fields lacking the verification procedures as well).

8. SOURCE

Data-transformer library is a part of the MCCME-helpers library (which serves as a core of one of our sites). Code may be found at
`http://mtn-host.prjek.net/viewmtn/mccme-helpers/`
`/branch/changes/ru.mccme.dev.lisp.mccme-helpers`