ReStore: Relational Database Persistency for Smalltalk Objects

John Aspinall

December 3, 2021

Layout and typography based on the sbabook LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Introduction

ReStore is a framework for Dolphin Smalltalk and Pharo which enables objects to be stored in and read from relational databases (SQLite, PostgreSQL, MySQL etc.).

ReStore aims to make relational persistency as simple as possible, creating and maintaining the database structure itself and providing access to stored objects via familiar Smalltalk messages. This allows you to take advantage of the power and flexibility of relational storage with no specialist knowledge beyond the ability to install and configure your chosen database.

CHAPTER 2

Getting Started

To install ReStore in your image follow the instructions on the GitHub project page for your Smalltalk dialect:

- Dolphin Smalltalk - https://github.com/rko281/ReStore
- Pharo - https://github.com/rko281/ReStoreForPharo

The class `SSWReStore` represents a ReStore session/connection; following installation a default singleton instance of `SSWReStore` is created and assigned to the global variable `ReStore`. We will use this global default throughout most of this document (see chapter 8 for information on working with multiple ReStore instances).

2.1 Choosing a Database

ReStore supports several different databases via the SSWSQLDialect class hierarchy. Currently defined SQL Dialects are:

- SQLite
- MySQL / MariaDB
- PostgreSQL
- SQL Server
- Access

Each subclass defines the different behavior, data types, functions etc. supported by a particular database. ReStore automatically selects the appropriate subclass after connecting to your chosen database; this ensures your application code is independent of database choice, enabling you to switch

databases easily if required. For example, for simplicity and speed you may use SQLite during development, then deploy to PostgreSQL for better scalability.

2.2 Configuring ReStore

After choosing and installing your database you must tell ReStore how to connect to it; the method for doing this varies by Smalltalk dialect:

Dolphin Smalltalk

ReStore for Dolphin accesses databases via ODBC. You must first create a Data Source Name (DSN) using the driver for your chosen database via the ODBC control panel. Since Dolphin is a 32-bit application ensure that you use the 32-bit ODBC control panel – you can open this from your Dolphin image by evaluating

```
ReStore openODBC
```

Once the DSN is created you can configure ReStore to use it as follows:

```
ReStore dsn: ‘’MyDataSourceName
```

Pharo

Pharo currently supports SQLite, MySQL and PostgreSQL. You must create the appropriate connection object then assign this to ReStore as follows:

```
"SQLite – see https://github.com/pharo-rdbms/Pharo-SQLite3 for more
    information"
ReStore connection: (SSWSQLite3Connection on: (Smalltalk
    imageDirectory / 'test.db') fullName)
```

```
"PostgreSQL – see https://github.com/svenvc/P3 for more information"
ReStore connection: (SSWP3Connection new url:
    'psql://user:pwd@192.168.1.234:5432/database')
```

```
"MySQL – see https://github.com/pharo-rdbms/Pharo-MySQL for more
    information"
ReStore connection:
(SSWMySQLConnection new
  connectionSpec:
    (MySQLDriverSpec new
      db: 'database'; host: '192.168.1.234'; port: 3306;
      user: 'user'; password: 'pwd';
    yourself);
yourself)
```

2.3 Connecting and Disconnecting

Once you have configured ReStore for your chosen database you may connect to and disconnect from the database as follows:

```
ReStore connect.
ReStore disconnect.
```

CHAPTER 3

Defining the Object Model

The first step in creating a ReStore application is to define your data model – the structure of your model classes. This allows ReStore to automatically create database rows from your objects, and also to create the actual database table in which those rows will exist.

Defining the structure of a class is done with the class method `reStoreDefinition`. This method should list the name of each persistent instance variable in the class, and define the type of object held in that instance variable. The `'type'` of object will be a class, a parameterized class, or a collection. These different types are highlighted in the following example for a hypothetical CustomerOrder class:

```
reStoreDefinition

  ^super reStoreDefinition
    define: #orderDate as: Date;  "Class"
    define: #customer as: Customer; "Class"
    define: #items as: (OrderedCollection of: CustomerOrderItem);
    "Collection"
    define: #totalPrice as: (ScaledDecimal withPrecision: 8 scale:
    2);  "Parameterized Class"
    yourself
```

3.1 Simple Classes

In the simplest case, just the class of object held is needed. Supported classes are:

- Integer
- Float
- Boolean
- String
- Date
- Time
- DateAndTime

Example for a hypothetical `Person` class:

```
define: #surname as: String;
define: #dateOfBirth as: Date;
define: #salary as: Float;
define: #isMarried as: Boolean;
```

Additionally, any other class defining a `reStoreDefinition` method may be used. This allows your classes to reference each other or even themselves:

```
define: #gender as: Gender;
define: #address as: Address;
define: #spouse as: Person;
```

3.2 Parameterized Classes

A parameterized class defines not only the class of an object but also additional information that may be required in relation to the class.

String

In Smalltalk an instance of String can be any size, with (to all intents) no upper bound. Within relational databases, however, there are usually three different types of Strings:

1. Fixed sized - usually referred to as a CHAR
2. Variable sized with some upper limit on the number of characters - this is usually referred to as a VARCHAR
3. An unbound, variable sized String - names vary; LONGTEXT, TEXT, MEMO etc.

For these reasons ReStore allows you to parameterize a String definition to enable the best choice of database type to be made. For Strings of a known, fixed number of characters (e.g. a postal/zip code, or a product code), you can specify a CHAR-type String using the String class method fixedSize:

```
define: #productCode as: (String fixedSize: 8);
```

For a variable sized String with a known maximum number of characters (VARCHAR) the method maxSize: is used:

```
define: #surname as: (String maxSize: 100);
```

Finally, if you just specify String (i.e. unparameterized) then a default value will be used as the maximum size of that String. This value will vary from database to database, but the net effect is usually to cause a LONGTEXT-type String to be used, although some databases may use an intermediate type with a large upper limit (e.g. MEDIUMTEXT). Example:

```
define: #notes as: String;
```

ByteArray

ReStore offers support for storing ByteArrays in a BLOB-type database column:

```
define: #imageData as: ByteArray;
```

Similar to with Strings, you may optionally specify a maximum size for the ByteArray – this will help ReStore choose the most appropriate BLOB type where the database offers multiple types with different (or no) maximum size:

```
define: #thumbnailImageData as: (ByteArray maxSize: 8192);
```

ScaledDecimal

Within Smalltalk an instance of ScaledDecimal has a scale - this defines the number of digits after the decimal point. In ReStore, when defining an instance variable as a ScaledDecimal as a minimum you must give the scale of that ScaledDecimal:

```
define: #totalPrice as: (ScaledDecimal withScale: 2);
```

Most relational databases support a type similar to ScaledDecimal (NUMERIC, DECIMAL etc.) but in addition to scale there is usually also precision - the total number of digits that may be held, including the scale. If you specify just a scale (as in the above example) a default precision of 15 will be used. Alternatively, you may specify the precision yourself:

```
define: #totalPrice as: (ScaledDecimal withPrecision: 8 scale: 2);
```

3.3 Unique IDs

Within ReStore every persistent object is automatically allocated an auto-incremented integer ID, unique to itself within its class. This happens completely transparently – you do not need to define this or store it within an

instance variable of your class. However there may be times where you wish to access this unique ID from your application code, for example to use as a customer or order reference.

Where this is the case you can declare the corresponding instance variable in your class as follows:

```
defineAsID: #customerNo;
```

This defines the instance variable customerNo as an Integer that holds the unique ID of the object. You do not need to instantiate this value yourself – ReStore will automatically allocate the next available ID when the object is persisted. Should you wish to allocate the unique ID yourself however, you can simply assign it prior to storing the object and ReStore will use the assigned value instead. In this latter case it is up to your application code to ensure the ID remains unique.

3.4 Collections

Defining a Collection in ReStore is done by specifying an example (template) collection. Supported collection classes are:

- OrderedCollection
- SortedCollection
- Array
- Set
- Dictionary

In addition to the collection class, the template collection must also specify the class of object held within that collection which can be either another persistent class, or a base or parameterized class. This is done through the Collection method of: (for convenience, this is defined as both a class and instance method). Example:

```
define: #middleNames as: (OrderedCollection of: (String maxSize:
  100));
define: #friends as: (OrderedCollection of: Person);
```

For a SortedCollection, by default the implementation of <= in the referenced class is used to define the sort order. This can be changed by specifying a sort block:

```
define: #children as: ((SortedCollection sortBlock: [ :c1 :c2 | c1
  age > c2 age]) of: Person);
```

For an Array, since it is a fixed-size collection its size must be specified:

```
define: #parents as: ((Array new: 2) of: Person);
```

Note that collections in ReStore are homogenous, i.e. all elements must be of the same class.

3.5 Owned Collections

In ReStore, collections specified as described above are termed General Collections. For readers with knowledge of relational databases they are equivalent to a many-to-many relationship. The relationships are stored using an intermediate table; ReStore takes care of creating this table and automatically adding/removing the mapping entries.

General Collections closely match the flexible nature of Smalltalk collections. Frequently however collections actually fulfill the following more limited criteria:

- each member object holds a reference to the owner of the collection
- each member object appears in the collection only once

In this case a more efficient form of collection can be used - an Owned Collection. This is equivalent to a one-to-many relationship in relational database terminology.

As an example, consider the hypothetical classes Customer and CustomerOrder. Each CustomerOrder knows its customer through its customer instance variable, defined as follows:

```
define: #customer as: Customer;
```

A Customer has a collection of CustomerOrders each of which appears only once. Thus the Customer's orders collection can be specified as:

```
define: #orders as: (OrderedCollection of: CustomerOrder owner:
  #customer);
```

This defines the Customer's orders instance variable as an owned collection of CustomerOrders, with the CustomerOrder instance variable customer holding a reference to the owning Customer.

Why use an owned collection? Owned collections are usually quicker to store and retrieve from the database compared to general collections since there is no intermediate mapping table – the relationship is defined solely by the instance variables of the two objects.

3.6 Dictionaries

The main difference between Dictionaries and the other collection classes detailed previously is that each element of a Dictionary is essentially storing

two objects - the key and the value – so the specification of a Dictionary must define the class of both objects. This is done by providing an Association between the key class and the value class as the parameter to of:

```
define: #profitByDate as: (Dictionary of: Date -> Float);

define: #productQuantities as: (Dictionary of: Product -> Integer);
```

Like other collections, the class of elements for both key and value can be any other persistent class, and will be the same for all elements of that collection (except in the case of inheritance).

Cache Dictionaries

For reasons of logic or efficiency you may have a Dictionary where the key is some attribute of the value. For example, say you have a Company object - this has a potentially large collection of addresses of individual offices:

Company	**Offices**
Widgets International, Inc.	Schupstraat 24, Antwerp
	...
	The Widgets Centre, Zagreb
Transdiffussion Worldwide	Westerstraat 93, Amsterdam
	...
	Transdifussion House, Zurich

Your application has a requirement to quickly find the Office in a particular city - hence you would like to store the collection of CompanyOffices as a Dictionary mapping city name to actual CompanyOffice. Since this arrangement is effectively a lookup cache, ReStore terms such structures Cache Dictionaries.

In this scenario ReStore offers an optimised form of Dictionary which will read a collection of objects from the database and automatically assemble a cache ready for fast and efficient lookup in your application code. To specify such a Dictionary, the key of the Association which is the parameter to of: should be the name of the instance variable of the value class which holds the key object, for example:

```
define: #cityOffices as: (Dictionary of: #city -> CompanyOffice);
```

Cache dictionaries can also take advantage of any owner link in the target object in the same way as other owned collections:

```
define: #cityOffices as: (Dictionary of: #city -> CompanyOffice
    owner: #company);
```

Multi-Value Cache Dictionaries

In the above Company-Office Dictionary example it is implicit that there can only be one CompanyOffice per city. If this is not guaranteed to be the case we need to use a different cache implementation. A common approach to a scenario like this is to use a Dictionary with a collection of objects as its values, often implemented like this:

```
addOffice: aCompanyOffice

  ^(self cityOffices at: aCompanyOffice city ifAbsentPut:
    [OrderedCollection new]) add: aCompanyOffice
```

```
removeOffice: aCompanyOffice

  ^self cityOffices at: aCompanyOffice city ifPresent: [ :offices |
    offices remove: aCompanyOffice]
```

ReStore terms such structures Multi-Value Cache Dictionaries and allows you to specify them in your class's reStoreDefinition as follows:

```
define: #cityOffices as: (Dictionary of: #city -> (OrderedCollection
    of: CompanyOffice));
```

Again, an owner link can be included in the specification:

```
define: #cityOffices as: (Dictionary of: #city -> (OrderedCollection
    of: CompanyOffice) owner: #company);
```

By specifying a cache dictionary or multi-value cache dictionary, ReStore will read a collection of objects from the database and automatically assemble a cache dictionary ready for efficient and fast lookup in your application code.

Derived Keys

In addition to defining a cache based on an instance variable (as in the above examples), ReStore also allows you to specify a Block which derives the key from the value object. For example, if city isn't held directly by the CompanyOffice object but instead within its address instance variable you could define the cache as follows:

```
define: #cityOffices as: (Dictionary of: [ :office | office address
    city] -> CompanyOffice);
```

Blocks can also be used with multi-value and owned Dictionary specs, for example:

```
define: #cityOffices as:
  (Dictionary
of: [ :office | office address city] -> (OrderedCollection of:
    CompanyOffice)
owner: #company);
```

3.7 Dependent Relationships

When defining your object model, you may notice subtle differences in the relationships between objects. As an example, consider the following partial specification of class Person:

```
define: #address as: Address;
define: #gender as: Gender;
```

At first glance the relationships described here are the same – each instance of Person will hold an instance of Address in its address instance variable, and an instance of Gender in gender. However there is a difference – an instance of Gender will likely be referenced by many different Person objects, whereas the object held in address will only belong to one individual Person.

The significance of this is as follows: when the contents of a Person's address instance variable is overwritten (by another instance of Address, or nil) the replaced Address object is no longer referenced from any other persistent object, and so should be removed from the database (deleted). Further, when an instance of Person is itself removed from the database, then its address object should also be deleted for the same reason. On the other hand, since the object held in gender is shared by other objects it should not be deleted in either of these cases.

In ReStore, relationships such as that between a Person and its Address are termed dependent relationships – the object held in address is dependent on the owning instance of Person for its existence. By explicitly declaring this dependency ReStore knows to delete the instance of Address when overwritten, or when the owning Person instance is deleted.

Declaring a dependent relationship is done by sending the message dependent to the class of dependent object in the class specification method:

```
define: #address as: Address dependent;
define: #gender as: Gender;
```

Collection-based relationships may also be dependent. Returning to our CustomerOrder example class, this contains an owned collection of CustomerOrderItem instances which define the component parts of the order (product, quantity etc.). This collection can also be declared dependent by sending the message dependent to the class of dependent object in the collection definition:

```
define: #items as: (OrderedCollection of: CustomerOrderItem
  dependent owner: #order);
```

3.8 Inlined Classes

We have previously seen ReStore instance variable definitions that reference other persistent model classes, for example:

```
define: #address as: Address dependent;
```

Within the database, an instance of Person will be stored in the person table, with a reference to its instance of Address, which will be stored separately in the address table.

This is a fairly standard scenario, however it does mean that to fetch a Person and its Address from the database requires two read operations (one for the Person and one for the Address). This is probably not an issue if you do not always require a person's address (indeed it may be an advantage, as it avoids fetching data that isn't required), however there may be cases where this is not optimal for your application.

Where this is the case you can ask ReStore to store the instance of the referenced class within the database row of the owning object. This is termed inlining and is done using the method inlined:

```
define: #address as: Address inlined;
```

With the address instance variable defined as inlined, the database representation of the Address object will be stored as part of the owning Person object, directly within the person table. This means only one read operation is necessary to fetch a Person and its Address.

Note that since a Person's address is now stored within the Person record in the database, it is automatically dependent so there is no need to declare this separately.

3.9 Inheritance

At the start of this chapter, it was mentioned that defining a class for ReStore enables a database table to be automatically constructed for that class. With a hierarchy of related classes it is necessary to decide whether these should share a single database table, or if each class exist in its own individual table.

By default, a hierarchy of classes will share a single table – this opens up some important possibilities.

Firstly, if you define an instance variable or collection in another class as holding instances of the superclass of a hierarchy then that instance variable/collection can hold an instance of the superclass or any of the subclasses sharing the superclass table. To illustrate this we will return to the

CustomerOrderItem class introduced in the previous section. Let's say this contains a product instance variable defined as follows:

```
define: #product as: Product;
```

Product is the superclass of a number of classes representing different product types:

```
Product      - productCode, supplier, stockLevel
  Book       - author, title, isbn, publisher
  MusicProduct  - artist, title, catNo
    Vinyl    - vinylWeight
    CD
```

If Product and its subclasses share a table then CustomerOrderItem's product instance variable can hold an instance of Book, CD, Vinyl or any other subclass of Product.

A second advantage is that when you need to query for objects in the database, a query for instances of the superclass of the hierarchy will also find instances of any subclasses sharing that table:

"Find all Products from a particular supplier. Uses ReStore querying methods – see X.X" Product storedInstances select: [:item | item supplier = aSupplier]

This will find instances of Book, CD, Vinyl or any other subclass of Product supplied by aSupplier.

Disadvantages of Sharing a Table

When constructing the single shared table for a hierarchy of classes, ReStore will allocate a column for every non-collection instance variable in each class. Thus, when for example an instance of CD is stored in the table, the columns corresponding to author, isbn, publisher (Book instance variables) and vinylWeight (Vinyl) will be empty. This is effectively wasted space in the table.

If there are large differences in the instance variables held by different subclasses within a hierarchy you may want to store instances of the subclasses in different tables to avoid this waste of space. This does mean that the advantages of table sharing listed above are lost. However, if there are large differences between the classes in the hierarchy then it is likely that they are not related in a meaningful way and so the loss of these advantages may not be important.

A common situation where this is the case is where there is one abstract superclass, defining a few common attributes for all model object classes in a particular application:

```
MyModel    - description, dateCreated
  Person   - firstName, surname, address
  Address  - line1, postcode
```

In a case like this it is highly unlikely that you would want instances of Person and Address to share a table. To turn off the default behavior of hierarchies sharing a table, MyModel should implement the following class method:

```
shouldSubclassesInheritPersistency

  ^ false
```

Should you wish to turn table sharing 'on' again in a sub-hierarchy, you would simply override this method to return true.

Individual subclasses may also choose to 'opt out' of sharing a table by implementing the following method:

```
shouldInheritPersistency

  ^ false
```

Note, however, that a subclass cannot 'opt in' to table sharing where it has been specifically turned off by a superclass implementation of shouldSubclassesInheritPersistency.

3.10 Creating and Maintaining the Database

Once you have successfully defined the classes forming your object model, you are ready to begin working with ReStore.

Let's assume you have connected the global ReStore instance as described in section 1. You then need to tell it which classes are to be persistent - these are the classes for which you have defined reStoreDefinition methods. To do this, you use the methods `addClass:` or to add a whole hierarchy of classes, `addClassWithSubclasses:`

```
ReStore
  addClass: Customer;
  addClass: CustomerOrder;
  addClass: CustomerOrderItem
  addClassWithSubclasses: Product
```

You now have a ReStore object, connected to the database and containing all required classes. However, none of the corresponding tables exist in the database. Fortunately, ReStore can create all the tables for you with one simple instruction:

```
  ReStore synchronizeAllClasses
```

Maintaining Tables

One of the advantages of Smalltalk is its highly interactive nature, which allows you to rapidly develop and refine applications. Over time your classes are likely to change as you redevelop and refactor your code. Unfortunately, this evolutionary development process can leave stored data (in files, or a relational or object database) in an incompatible state, requiring manual intervention or additional coding to bring it up to date with your current object model.

ReStore helps overcome these problems and preserve the rapid development benefits of Smalltalk by automatically reconfiguring your database tables to match your object model. When you have made changes to your object classes, simply re-add them to ReStore (as above) and re-evaluate:

```
ReStore synchronizeAllClasses
```

Resynchronizing in this way allows ReStore to create new tables (for new classes), add new columns (for new instance variables) and remove redundant columns (where you have removed instance variables).

Renaming a Class

A change that ReStore cannot handle automatically with synchronizeAllClasses is where you have renamed a class. In this case, if you were to use synchronizeAllClasses, ReStore would simply add an empty table with a name based on the new name of the class, leaving the old table (and its data) in the database, but inaccessible.

To overcome this, ReStore allows you to explicitly state when you have renamed a class. As an example, let's say you renamed the class Address to PostalAddress:

```
ReStore renamedClass: Address from: #PostalAddress
```

Using this technique, the original table for Address would simply be renamed to match `PostalAddress`, preserving all its data.

Removing Classes

Similar to renaming a class, you must use a specific message to tell ReStore that you no longer require a particular class in your data model (and that its corresponding table can be removed from the database):

```
ReStore destroyClass: <redundant class>
```

In more drastic situations (e.g. to purge all data from a database), you may evaluate:

```
ReStore destroyAllClasses
```

Note that this will only remove from the database tables associated with classes known to ReStore (i.e., those that have been added with addClass: or addClassWithSubclasses:).

Renaming an Instance Variable

A further change that cannot by handled directly via synchronizeAllClasses is where an instance variable has been renamed. Using synchronizeAllClasses in this case would cause ReStore to add a new (empty) column for the new instance variable, and delete the previous column, with the loss of all data contained in that column.

Similar to renaming a class, ReStore provides a simple message which can be used to inform it of the change of instance variable name and instruct it to update the database structure accordingly. Let's say you have renamed the Customer instance variable firstName to forename; you would inform ReStore of this change as follows:

```
ReStore renamedInstVar: #forename from: #firstName in: Person
```

This method will prompt ReStore to update the table associated with Person, renaming the column previously corresponding to firstName so it matches forename.

CHAPTER 4

Storing Objects – Transactions

CHAPTER 5

Storing Objects Manually

CHAPTER 6

Query Introduction

Bibliography