Seaside Tutorial

Software Architecture Group
Hasso-Plattner-Institut

8 - Persistence

What you are going to learn

Introduction

This chapter is all about persistence. Whenever you want your application to save data for later use you have to deal with persistence. In the following, the four main options to save your data persistently are described.

Saving All Data in the Image

This is an option that works for a single instance of your application. If you have more than one instance of your application running, maybe for load sharing, you have to think about sharing your data between the running applications.

So let us assume you have one running image. What you could do now is create a kind of database class and store all your data in class variables like all your registered users.

```
users

    ^ users ifNil: [users := OrderedCollection new]
```

StImageDatabase class>>#users

But that is not all! If your image crashes for whatever reason, your data would be lost. So, every time you save data in your image you have to save your whole image on disk. Surely you could as well save the image in fixed time intervals, but then you have to deal with possible data loss. To save your image, you could execute a method like:

```
saveImageWithoutMonitor

    SmalltalkImage current saveSession.
```

StImageDatabase>>#saveImageWithoutMonitor

Because a web application might have serveral users accessing the application simultaneously who may, moreover, be causing the image to be saved at the same time, you need to introduce a mutex as an instance variable of the class.

```
mutex

    ^ mutex ifNil: [mutex := Monitor new]
```

StImageDatabase class>>#mutex

With this mutex you can modify the image saving method like below to be sure that the image saving process is only running once at a specific time.

```
saveImage

    StImageDatabase mutex critical: [self
saveImageWithoutMonitor].
```

StImageDatabase>>#saveImage

All you have to do is call that image saving method whenever you are going to add or delete data (e.g., a user).

```
addUser: aUser

    StImageDatabase users add: aUser.
    self saveImage.
```

StImageDatabase>>#addUser:

This works pretty well, but saving an image causes a massive amount of disk writes and your application slows down during that writing process. Also, you are not able to share your saved data with other applications or another instance of your application for load balancing. So, let us look at better options.

Saving All Data in a Relational Database: Glorp

One such option is to use an O/R mapper and a relational database as its back-end. With this option, it is possible to share your data between other applications. You also no longer have to worry

about conflicts, which could occur if more than one dataset is trying to be saved at the same time. All the work is done by the relational database management system of your database server.

That is nearly almost true; you also have to invest some thorough work in your data models. Since you are programming in an object-oriented environment, you have to map your data to the relational view of the database. This is done using GLORP (Generic Lightweight Object-Relational Persistence).

In the following, we will use PostgreSQL as the relational database management system of choice.

Installing a PostgresSQL Server

If you do not have access to an existing PostgreSQL server, you have to install one on your machine. On http://www.postgresql.org/ftp/binary/, you can find installation files for Windows and Linux. A detailed description about installing the server on a Mac OS X machine can be found on http://www.entropy.ch/software/macosx/postgresql.

After the database server has been installed, make sure the authentication mechanism of the database is using password and not md5! You can find the setting in data/pg_hba.conf in your PostgreSQL directory.

Now you can create a new database user called "postgres" with password "postgres" and a database "StDatabase". Make sure that the encoding of your database is UTF-8.

Installing the GLORP Framework

To get GLORP working, two new packages, "GLORP Port" and "PostgreSQL Client for Squeak", are required. Open the SqueakMap Package Loader via the World Menu and "open...". First install the package "PostgreSQL Client for Squeak" and then "GLORP port". Installing GLORP takes some time, so do not worry.

The Error Message

If you get an error message during the installation asking you whether you would like to open a debugger, just click on Yes. Now the installation is paused. Please open the System Browser and go to the class method **#basicIsSqueak** of the **Dialect** class. Edit this method, so it looks like this:

```
basicIsSqueak
^true
    "^ (Smalltalk respondsTo: #vmVersion) and:
[(Smalltalk vmVersion copyFrom: 1 to: 6) =
'Squeak'.]"
```

Dialect class>>#basicIsSqueak

Using GLORP

Using GLORP means mapping your model's attributes to data columns of the relational database. To achieve this, you first need to create a subclass of **DescriptorSystem** called **StGlorpDatabase**. Every model has its own database table containing all attributes. The connection between a user and their tasks are given through foreign-key relations, i.e., each object, **StUser** or **StTask** has its own id for identification and every task has also a foreign key, which is the id of its owner (user).

Let us look at the models you have created. For each model, you have to create a method like this:

```
tableForSTUSER: aTable

    aTable
        createFieldNamed: 'id'
        type: platform sequence.
    aTable
        createFieldNamed: 'userName'
        type: platform text.
    aTable
        createFieldNamed: 'email'
        type: platform text.
    aTable
        createFieldNamed: 'password'
        type: platform text.

    (aTable fieldNamed: 'id') bePrimaryKey.
```

StGlorpDatabase>>#tableForSTUSER:

The naming convention is **#tableForTABLENAMEINUPPERCASE**. With these methods, you are specifying which tables and columns should be created in the relational database and which data types these columns should have. All available data types can be found in the method category "types" of the class **PostgreSQLPlatform**. Each column name and type should match an attribute of your model. The object itself is identified by its id. Since each row in a relational database has to be somehow unique, we choose to specify the id as primary key, so this id is unique for every table row.

Next, you have to specify the mapping between the model's attributes and the database tables' columns. For each model, create a descriptor that looks like this:

```
descriptorForStUser: description

    | table |
    table := self tableNamed: 'stuser'.
    description table: table.
```

```
    (description newMapping: DirectMapping)
        from: #id
        to: (table fieldNamed: 'id').
    (description newMapping: DirectMapping)
        from: #userName
        to: (table fieldNamed: 'userName').
    (description newMapping: DirectMapping)
        from: #email
        to: (table fieldNamed: 'email').
    (description newMapping: DirectMapping)
        from: #password
        to: (table fieldNamed: 'password').
    (description newMapping: OneToManyMapping)
        attributeName: #tasks;
        referenceClass: StTask;
        collectionType: OrderedCollection;
        orderBy: #id.
```

StGlorpDatabase>>#descriptorForStUser:

Above, you can see that most attributes are directly mapped to columns of the database table "stuser". All attributes of the object are accessed through symbols, e.g., **#userName** accesses the instance variable userName of the user object. Be sure to create accessors for your instance variables. What you can also see is that we need a **OneToManyMapping** instead of a **DirectMapping** for the user's tasks. Every user has several tasks which are collected in the user object's **#tasks** instance variable. So, we need to define a **OneToManyMapping** of the attribute **#tasks**, which is an **OrderedCollection** containing instances of the referenceClass, here **StTask**. Additionally, we want to order the collection by task ids.

You now need a class models enumerating all attributes. Do not forget to tell the model that the attribute **#tasks** is a collection of **StTask** instances.

```
classModelStUser: model

    model
        newAttributeNamed: #id;
        newAttributeNamed: #userName;
        newAttributeNamed: #email;
        newAttributeNamed: #password;
        newAttributeNamed: #tasks collectionOf:
StTask.
```

StGlorpDatabase>>#classModelStUser:

Create the method **#allTableNames**, which should return a collection of all used tables:

```
allTableNames

    ^ #('stuser' 'sttask')
```

StGlorpDatabase>>#allTableNames

Create the method **#constructAllClasses**, which should return a collection of all used classes:

```
constructAllClasses

    ^ super constructAllClasses
        add: StUser;
        add: StTask;
        yourself.
```

StGlorpDatabase>>#constructAllClasses

Now just add the instance variables "glorpSession", "connectionFailed", add their accessors. The glorpSession variable will contain the database connection as an instance of **GlorpSession**, and connectionFailed is a **Boolean**, which is true if anything is wrong with the database connection.

The next step is to overwrite **#initialize**. The platform we use is PostgreSQL, which we need to describe the datatypes of the database tables. We also want every new instance of our database class to be connected to the database, so call **#connect**.

```
initialize

    super initialize.
    self platform: PostgreSQLPlatform new.
    self connect.
```

StGlorpDatabase>>#initialize

What does **#connect** do? This method creates at first a login object, containing user name, password, host, database name and our platform. The next step is to define an accessor for the database for the newly created login. **#glorpSession** is then set to an instance of **GlorpSession** which needs a **DescriptorSystem**, which in turn is our instance of **StGlorpDatabase** containing all necessary descriptions about our models. We also need some exception handling in case anything goes wrong. In case of an error while connecting to the database our **#connectionFailed** variable is true, otherwise, false.

```
connect

    | accessor |
    accessor := DatabaseAccessor forLogin: self
createLogin.
    self glorpSession: (GlorpSession forSystem:
self).
    self glorpSession accessor: accessor.
    self connectionFailed: false.
    [accessor login] ifError: [:err | self
connectionFailed: true].
```

StGlorpDatabase>>#connect

```
createLogin
```

```
    ^ Login new
        database: platform;
        username: 'postgres';
        password: 'postgres';
        connectString: 'localhost_STDatabase';
        yourself.
```

StGlorpDatabase>>#createLogin

When we can connect we also need to be able to disconnect. If the current database connection has not timed out, we can simply log out.

```
disconnect

    self glorpSession
        ifNotNil: [self glorpSession accessor
logout].
```

StGlorpDatabase>>#disconnect

The next method does just one thing: it creates **PGSequence** objects representing the id sequence of each model. If anything is wrong, the error will be shown in the Transcript.

```
createAllSequences

    self glorpSession system platform
areSequencesExplicitlyCreated
        ifFalse: [^ self].
    self glorpSession system allSequences do:
[:each |
        self glorpSession accessor
            createSequence: each
            ifError: [:error | Transcript show:
error messageText]].
```

StGlorpDatabase>>#createAllSequences

In the next step, we can finally create our tables. First, we create the tables with their names, columns and data types. Second, we create all the indexes we need, which allows the database to reorder the table rows, so that a row can be found faster if we search about an attribute whose column has been indexed. The third step is to create foreign key relations between the tables. For example, the "sttask" table will have a foreign key, which is the id of the owner of the specific task, because one user can have several tasks.

```
createAllTables

    | accessor errorBlock allTables |
    accessor := self glorpSession accessor.
    errorBlock := [:errorx | Transcript show:
errorx messageText].
    allTables := self glorpSession system
allTables.
```

```
    allTables
        do: [:each | accessor createTable: each
ifError: errorBlock.];
        do: [:each | accessor createTableIndexes:
each ifError: errorBlock.];
        do: [:each | accessor
createTableFKConstraints: each ifError:
errorBlock].
```

StGlorpDatabase>>#createAllTables

Finally, **#dropAllThenCreateSchema** will recreate all tables by deleting all existing tables and creating them again.

```
dropAllThenCreateSchema

    self sess accessor dropTables: self allTables.
    self
        createAllSequences;
        createAllTables.
```

StGlorpDatabase>>#dropAllThenCreateSchema

You can now open a workspace and execute

```
StGlorpDatabase dropAllThenCreateSchema
```

If your descriptions are fine all tables should be added. You can check this by using some of the PostgreSQL Tools, like PGAdmin

So you learned to create your schema for saving your objects. But how do you save objects? That is easy, all you have to do is registering your objects in an unit of work:

```
addUser: aUser

    self glorpSession
        inUnitOfWorkDo: [self glorpSession
register: aUser].
```

StGlorpDatabase>>#addUser:

The "unit of work"-block commits all changes made during the block execution. **#register** checks whether the object was read from the database. If it was read, only the changes are saved. If it was not read, the object is saved as a new object.

Adding a task to a user is simple now:

```
addTask: aTask toUser: aUser

    self glorpSession
        inUnitOfWorkDo: [aUser addTask: aTask.
                         self glorpSession
register: aUser].
```

StGlorpDatabase>>#addTask:toUser:

Now you can save objects. Reading objects is easy too.

```
findUserByEmail: anEmailAddress

    ^ self glorpSession
        readOneOf: StUser
        where: [:each | each email =
anEmailAddress].
```

StGlorpDatabase>>#findUserByEmail:

#readOneOf: aClass where: aBlock reads the first object which is an instance of aClass and matches the where-block. More reading operations can be found in the "api/queries"-category of the **GlorpSession** class.

More Information about GLORP can be found on http://glorp.org/documentation.html.

Saving All Data in an Object-Oriented Database: GOODS

Another option to make your data persistent and share it with other applications is to use an object-oriented database like GOODS (Generic Object Oriented Database System).

Installing the GOODS Database

There are two ways to install GOODS: the first is to download a binary of the GOODS server, the second, to download the sources and compile the server on your own.

A binary for Windows platforms can be found on http://wiki.squeak.org/squeak/3492.

If you would like to compile the server yourself, download the sources from http://www.garret.ru/~knizhnik/goods.html, extract the sources and start compiling.

To setup and start the server, you have to create a configuration file named sttd.cfg in the directory where the server binary resides. Edit the file to make it look like this:

```
1
0:localhost:6100
```

This tells GOODS to start a single instance of the server listening on port 6100 of localhost. You are now ready to start the server with the command:

```
goodsrv sttd
```

To terminate the server later on, just type "logout" in the

execution window.

Installing the GOODS Framework

The next step is to install the GOODS Client Framework in Squeak. So just open the "SqueakMap Package Loader" (WorldMenu/open../) and install the package "GOODS".

Using GOODS

First we need a new database class, like **StGOODSDatabase**, which has an instance variable **#db**. We also need some methods for connecting, disconnecting and saving data. Let us create them:

```
connect

    self db: (KKDatabase
        onHost: self localhost
        port: self defaultPort).
```

StGOODSDatabase>>#connect

```
disconnect

    self db logout.
```

StGOODSDatabase>>#disconnect

```
initialize

    self connect.
```

StGOODSDatabase>>#initialize

We created and edited the **#initialize** method in the way shown above to allow us to be connected to the database every time we instantiate a new database object. Furthermore, we need to create a root object, from which all other objects to be made persistent in the database are referenced. Let us take a dictionary for that example:

```
createRoot

    | users root |
    users := OrderedCollection new.
    root := Dictionary with: ('users' -> users).
    self db root: root.
    self db commit.
```

StGOODSDatabase>>#createRoot

What you see above is that we create a new root object (which is a **Dictionary** containing all users using the key "users") and commit the changes to the database. There are several other database-related methods like **#refresh** or **#rollback** which can also be used. They can be found in the class **KKDatabase**, but for now the above code should be sufficient.

Now we need a method to get all our users,

```
users

    ^ self db root
        at: 'users'
        ifAbsent: [self error: 'Database root not
initiated!']
```

StGOODSDatabase>>#users

and a method to add users, do not forget to commit the changes!

```
addUser: aUser

    self users add: aUser.
    self db commit.
```

StGOODSDatabase>>#addUser:

This works pretty well, but be aware of critical situations when two or more different database connections want to write their data. GOODS does not have a very good database management system. And all object structures are stored as they are, not in other database-internal structures. This means that, if, for example, you want to store a **BTree** and edit this tree at the same time with several connections, the database could crash. Please keep this in mind.

More Information about GOODS can be found on http://www.garret.ru/~knizhnik/goods.html.

Saving All Data in an Object-Oriented Database: Magma

Another object-oriented database is Magma. Other than GOODS, it provides you with a full-blown Smalltalk-only implementation, which is capable of persisting your objects either locally or on a remote Magma server. To install Magma, you should open the Package Universe and go to the 'Persistence' category. Here you will find (among other packages) three versions of Magma: client, server and tester. They should be at least version 1.0. Client is the smallest and contains only the amount of Magma code required to connect to a remote server. The server package contains the client package and, additionally, code to establish your own server. The tester package contains the complete Magma distribution and includes a large test code base. As we will set up our own server here, the server package is recommended. However, be aware that Magma and Glorp have a naming collision with the class Cache. Although not verified, we believe that it is not possible use both of them seamlessly at the same time.

The next thing you have to do is to decide whether you want your database running in the same Squeak image your client will run in,

or in another image. The former is a little bit easier but limits the scalability of your web application: If you want to do things like load balancing you will need more than one image connecting to the server and thus one extra image running the Magma server. The only difference between these two solutions relevant in the scope of this tutorial is the way connections to Magma are established. Switching between the two alternatives is easy.

Now its time to set up our database. That is, we first create a Magma repository on our hard disk. In your workspace, simply execute:

```
MagmaRepositoryController
    create: 'C:\MagmaDBs\SeasideTutorial'
    root: Dictionary new
```

This will take some seconds to create the required file structure in the specified directory. If you chose to run Magma in a second image, you now have to start the Magma server in that image on some port and inspect it for later works:

```
MagmaServerConsole new
    open: 'C:\MagmaDBs\SeasideTutorial';
    processOn: 51001;
    inspect
```

To shut down the server at any time, just send it a **#shutdown** message, for example via the inspector.

Now we are able to build a class similiar to those we had with GOODS above. Name it StMagmaDatabase, give it an instance variable 'session' and create the accessors for it. Afterwards we can write the **#connect** method:

```
connect

    self session: (MagmaSession
        hostAddress: self localhost
        port: self defaultPort).
    self session connectAs: 'tutorial'.
```

StMagmaDatabase>>#connect

```
localhost

    ^ #(127 0 0 1 ) asByteArray.
```

StMagmaDatabase>>#localhost

```
defaultPort

    ^ 51001
```

StMagmaDatabase>>#defaultPort

As you can see, we simply connect to localhost:51001 under the name 'tutorial'. Of course, the host address should be the

computer on which your Magma database runs. If you want to use Magma locally only, you can use the following code:

```
connect

    self session: (MagmaSession
        openLocal: 'C:\MagmaDBs\SeasideTutorial').
    self session connectAs: 'tutorial'
```

StMagmaDatabase>>#connect for local connections

For the **#disconnect**, there are two different implementations regarding whether you work locally or remotely - i.e., with another image running Magma. As said above, we are using remote access here:

```
disconnect

    self session disconnect.
    self session: nil.
```

StMagmaDatabase>>#disconnect

For the sake of completeness, here is the implementation of **#disconnect** for the local variant, which additionally has to close the repository used by the local session:

```
disconnect

    self session
        disconnect;
        closeRepository.
    self session: nil.
```

StMagmaDatabase>>#disconnect for local connections

From now on, usage is very similiar to GOODS. As you might have seen, we created the repository with a Dictionary as the root. Via this root, we can now navigate to every object in the database via references. Magma can automatically detect changes to this structure and save them, if we surround those changes with a **#commit:**. Thus, the following methods should be very clear for you:

```
createUsers

    self session
        commit: [self session root
                    at: 'users'
                    put: OrderedCollection new].
```

StMagmaDatabase>>#createUsers

```
users

    ^ self session root
        at: 'users'
        ifAbsent: [self error: 'Database root not
```

```
initiated!'].
```

StMagmaDatabase>>#users

```
addUser: aUser

    self session commit: [self users add: aUser].
```

StMagmaDatabase>>#addUser:

Keith Hodges has written another Magma Tutorial. There you can learn more details about the Magma seasideHelper and how to integrate him in our ToDo Application.

One Database Connection per Session

It is advisable to have only one database connection per user session. Thus, every time a new user connects to your web application, a connection to the database is established. If the user's session expires or the user logs out you can disconnect from the database.

To achieve this, create an instance variable in your session class called **#db**. For illustration purposes, we will use the GLORP approach. Add the following:

```
initialize

    super initialize.
    self db: StImageDatabase new.
```

StSession>>#initialize

To terminate the database connection when the session expires, add:

```
unregistered

    self db disconnect.
    super unregistered.
```

StSession>>#unregistered

That works pretty well but you may ask what happens if the database server is down, and how to display a specified error message.

Achieving this is pretty easy: all you have to do is add the following to your session class:

```
responseForRequest: aRequest

    self db connectionFailed
        ifTrue: [^ WAResponse new nextPutAll: 'No
Database Connection'].
    ^ super responseForRequest: aRequest.
```

StSession>>#responseForRequest:

[<< External Resources](#) | [print format](#) | [Ajax >>](#)

© Copyright 2008 Software Architecture Group