



WebObjects Database Connectivity: Part II

by Theresa Ray of Tensor Information Systems

Sponsored by Apple Computer, Inc.
for Apple's Worldwide Developer Relations Group

media

WebObjects Database Connectivity: Part II



by Theresa Ray of Tensor Information Systems

Creating powerful websites for your users often involves integration with a database for storage and retrieval of information. WebObjects provides efficient access to nearly every database on the market through its Enterprise Object Frameworks (EOF). EOF allows you to map the information stored in your database to business objects used by your applications - whether web based or windowed. And by putting your business logic into one set of objects used by all your applications, behavior such as validation, data conversion, or security restricted access is guaranteed to be the same no matter which application is accessing the data.

The Database Wizard and Enterprise Object Frameworks

One of the best resources for all WebObjects programmers, regardless of experience level, is Apple's on-line technical documentation. One of the tutorials provided at their site is a database access example using the database wizard. The documentation provided there is an excellent introduction to the Enterprise Object Frameworks, creating an eomodel (the mapping of data from your database to your business objects), and the terminology necessary for continued database connectivity discussions. This survival guide assumes that the reader has done the on-line tutorial, and is familiar with the terminology introduced there. Apple's database application tutorial can be found at <http://gemma.apple.com/techinfo/techdocs/enterprise/WebObjects/GettingStarted/Movies/MoviesTOC.html>.

Relationships Between Tables In a Database

In most applications involving a database, several tables will contain related data. For example, there may be tables named Employee and Department which are related. Each Employee record has a related Department record - the home department for that employee. Usually the Employee table will have a column which contains a unique Department identifier, most typically the Department table's primary key attribute. For example, if the Department table uses the departmentNumber attribute for the

primary key, the Employee table would contain a departmentNumber attribute so that the employee could be mapped to a distinct department. This mapping is called a join or relationship. Relationships may be one-to-one, one-to-many or many-to-many. In the case of a many-to-many relationship, an intermediate table is used to resolve the unique relationships. See your database administrator for more information on these intermediate tables.

Relationship in Enterprise Object Frameworks: Referential Integrity

Enterprise Objects Frameworks provides powerful and convenient access for joined tables. If the database administrator has already set up related tables when you start a new project, the eomodeler wizard will read the primary key/foreign key relationships identified in the database and define the relationship in the model for you. You will also be able to specify the referential integrity rules regarding the relationship.

The first rule you can define is whether the parent table owns the destination table - if so, when the code deletes one of the items in a relationship, it is also deleted from the database. You may have two tables called Company and Store, where Company is the parent. If the code indicates that a store is to be deleted from the relationship with Company, it is likely that you wish the Store to be removed from the database entirely. However, if the two tables are Department and Employee, where Department is the parent, you probably don't want to delete the Employee from the database if the relationship with a Department is removed - the employee may have transferred to a different Division, but still exist within the company.

If you define this first rule to be true, then you can choose from two additional rules regarding the parent entity. The cascade rule states that when a parent entity (such as a Company) is deleted, all children associated with that parent (such as all Stores for a Company) are deleted as well. The deny rule states that if an attempt to delete a parent entity is made when children still exist for that parent, the request is denied and an exception is raised. For example, if a Company still had Stores associated with it and the code tried to delete that Company, EOF would deny the request and raise an exception.

If you define the first rule to be false, you can still choose the cascade or deny rules regarding the parent entity, or you can choose the nullify rule. The nullify rule must be supported by the referential constraints of the database, and nils the child's reference to the parent when the parent is deleted. If a Company was deleted that still had Stores associated with it, the attribute in the Store entity that identified a unique parent Company would be set to nil. Many database administrators do not set up the database to allow nil foreign keys. It really depends on your business logic for the given relationship. If the database does not allow this key to be nil, the nullify rule is probably not the best option for this relationship's referential integrity rule.

The Relationship Inspector

You may also choose that no action be taken to the child upon deletion of the parent entity, but you must go into the relationship inspector within the eomodel to do so. The relationship inspector allows you to view and edit the properties associated with a relationship, including the join type (join types are defined in the Enterprise Object Frameworks Developers Guide appendix and essentially define what to do when an attempt to fetch through a relationship returns some nil results), batch faulting options (if fetching one item through this relationship, go ahead and fetch the next 'n' items) and optionality rules (is the destination for this relationship optional or mandatory).



Traversing a Relationship

Now that the relationship has been defined, how do you use the relationship when writing code? Suppose you have a Company which has a one-to-many relationship to Store. The relationship from Company to Store is named toStores, and Store has an attribute named storeName which you wish to display. The code to access the relationship might look like:

```
// Assume the variable someCompany exists and is a pointer to a Company object
storeList=[someCompany.toStores]; //storeList is an array of Stores associated
with this Company
// storeList=someCompany.toStores in WebScript syntax
// storeList=someCompany.toStores() in Java syntax
name=[[storeList objectAtIndex:0] name]; //name is the name of the first Store
item in the array
```

What is wrong with this code? If the relationship is defined to be optional, there may not be any stores in the storeList array (remember that the result of the relationship evaluation is an array since it is a one-to-many relationship - a one-to-one relationship would return a single object with the appropriate class). Attempting to access [storeList objectAtIndex:0] will result in an exception. The array exists, but it is empty and objectAtIndex:0 does not exist. This is obviously not desirable. When working with optional to-many relationships, you must be careful that you don't try to access an item that may not exist.

Adding and Deleting with Relationships

What about creating a new relationship? One should always use the method addObject:toBothSidesOfRelationshipWithKey: to add or replace an object to a

relationship. This method guarantees that all aspects of the forward and reverse relationships are synchronized. For example, if an Employee was transferred to a new Department, the code might update the toDepartment relationship from Employee by:

```
// Assume that newDepartment and thisEmployee exist and are initialized  
  
[thisEmployee addObject:newDepartment  
toBothSidesOfRelationshipWithKey:@"toDepartment"];
```

The key you specify is the name of the relationship in your eomodel (case sensitive). This message replaces thisEmployee's old Department association with the new Department association, and updates Department's toEmployees relationship appropriately. Even though the toDepartment relationship is a one-to-one relationship from Employee to Department, this is still the appropriate message. The structure is the same if you are adding a new Employee to an existing Department. The message would update the Department's toEmployees relationship, and would set the newEmployee's Department appropriately.

Similarly, the method removeObject:fromBothSidesOfRelationshipWithKey: should be used when removing an object from a relationship. This is where the rules defined above come into action. The appropriate action between child and parent will be taken, as defined by the rule specified by the developer of the application.

Custom Validation with Enterprise Object Frameworks

In addition to defining relationship rules in eomodeler, a developer can implement validation logic for relationship or any attribute in a model explicitly. First, you must make a custom class for the entity which requires the validation. Choose Generate ObjC or Java files from the Property menu in eomodeler. A skeleton template in the appropriate language will be made from the model. Only attributes flagged as class properties will be included in the custom class. Accessor methods will be automatically coded, allowing users of the class to read and update the value of those attributes.

Usually, the first method a developer adds to this skeleton framework is the init method. Implementing your own init method allows you to set reasonable default values for a new object of this class. The first line in your init method MUST be a call to the super class's init method. In ObjC syntax, the last line must be a return self so that the pointer is passed back to the object which instantiated it.

A second method frequently implemented in a custom class is the validateForSave method. By extending the default behavior of validateForSave, you can perform additional validation of the data before generating and sending SQL to the database.

Most apps have rudimentary validation logic when the data is collected, but some validation logic shouldn't be recoded every place in the application that data is gathered. It is more efficient and robust to include business-level validation here. For example, if the business rules specify that the entry for a field have a specific format, this might be a good place to check for that format. By placing the validation logic at this level, you guarantee that if any new forms gather this data, it is still formatted correctly without having to reimplement the validation logic within the form directly. The method should return an exception (the exception really is the return value for the method - EOF will raise it automatically) if any business rules have been violated - hopefully with a clearly defined error reason. Otherwise, the method should return nil.



Reformatting Data

The custom class's accessor methods are a prime place to perform global reformatting of values when necessary. For example, your business logic says that all names are to be stored in upper case in the database. The `setFirstName` method would probably convert the entry into an uppercase version of whatever was entered by the user and store that result. By forcing the conversion in the custom class, you again guarantee that any new application or set of code saving this field will uppercase the value before it is saved to the database. Usually, the set of custom classes created by eomodeler and extended by the developer are grouped and placed as a separate Framework (choose Framework from Project type when you start a new project in ProjectBuilder). That framework can be loaded by any application accessing the database.

Conclusion

Relationships are a very convenient, very powerful mechanism for traversing related tables in the database without constructing a separate qualifier and `fetchSpecification` each time. The use of referential integrity rules ensures that the data stored in the database does not become “dirty” or corrupted as records are added and deleted to the database. Additional validation and reformatting of data before it is saved to the database by a single set of classes also helps keep the data “pure”. A developer need not worry about all the validation rules if a new form is to be added to the application - if the framework used by the application is set up to perform the validation and formatting instead of the application itself.

References

<http://gemma.apple.com/techinfo/techdocs/enterprise/WebObjects>

WebObjects Developer's Guide

Enterprise Objects Framework Developer's Guide

<http://www.omnigroup.com/MailArchive/WebObjects>

<http://www.omnigroup.com/MailArchive/eof>

<http://www2.stepwise.com/cgi-bin/WebObjects/Stepwise/Sites>

About the Author

Theresa Ray is a Senior Software Consultant for Tensor Information Systems in Fort Worth, TX. She has worked as a consultant on WebObjects projects for a wide variety of clients including the U.S. Navy and the United States Postal Service. Her experience spans all versions of WebObjects, from 1.0 to 3.5, several versions of EOF, from 1.1 to 2.1, and NEXTSTEP 3.1 to OPENSTEP 4.2. In addition, she is an Apple-certified instructor for WebObjects courses.

Tensor Information Systems is a systems integrator providing enterprise solutions to its customers. Tensor's employees are experienced in all NeXT/Apple technologies including OPENSTEP, NEXTSTEP, EOF and WebObjects. Tensor also provides Apple-certified training in WebObjects, Oracle consulting and training, as well as systems integration consulting on HP-UX.

You may reach Theresa by e-mail: theresa@tensor.com

<http://www.tensor.com>