Using JSF Framework Development Standards for your XPages Project (Part Four: Finishing the Model Layer)

July 2, 2013

In the previous installment of this series we began to build the Model layer described in the Model-View-Controller (MVC) architecture. In that article we covered low level database interaction classes as well as the representation of the data object for our sample Employee application. To recap, we designed the following classes four our Model layer:

- **Employee:** A base foundation class representing the employee data. The fields in this class, for our simple application, correspond to the fields on a Notes form.
- **DominoEmplyeeDAO (and EmployeeDAO interface):** The DominoEmplyeeDAO contains all of the code that processes the low level Domino database specific language for creating, reading, updating, and deleting an employee. The concrete class DominoEmployeeDAO implements the EmployeeDAO interface. The interface defines the generic method signatures that must be created by any class implementing that interface. This allows us to define a structured set of methods for working with any database without the requirement that client code have knowledge of the type of database implemented.

In a "real world" application there will be many "Employee" type classes that represent different documents (records). For example, in the order portion of a simple Order Entry application, there might be the Order class, the OrderItem class, and a Product class. But even there, these classes may actually be divided into sub-classes. Your application might have multiple types of products. For a clothing store, there may be a Footwear product, a Dress product, a Shirt product, a Trouser product, a Jacket product, and an Accessory product. Regardless of how many types, you would end up with a class to represent each unique table (Form for Domino) of data that you want to track. And there would be a corresponding DAO interface and implementation for each model data object.

You may be thinking about model object relationships. For example, an Order has one or more OrderItem classes. In that case you would actually build each of the classes (Order and OrderItem), but then create a relationship in the Order and have it contain a list of OrderItem objects. You would write your DominoOrderDAO to handle loading that

relationship based on whatever data relationships you have in place to connect an OrderItem to an Order. When you get to the XPage, you would be able to deal directly with the Order knowing it contained its related OrderItem objects.

Spend your time designing this layer!

If you have ever done relational database modeling, you know the importance of spending time in design. Modeling your data objects is very similar to database modeling. You want to think of your data as unique groups of fields.

As Domino developers we have been mostly free from data modeling; largely because Domino does not do relational data very well. It is common to encounter Domino applications where related data is all on one form, but the data really represents two or more model objects. There is no concept of a "join" for viewing data together in Domino. We have design features that allow us to visually connect the data, but it is not joined as it would be in a relational database. I do acknowledge that we can create response documents. That does provide help, some of the time, but when we have to perform complex calculations against the data in a child and show the result in a view, the solution tends to be storing that data on the form at save time.

Don't mimic the non-relational tricks on your Notes form in your java model!

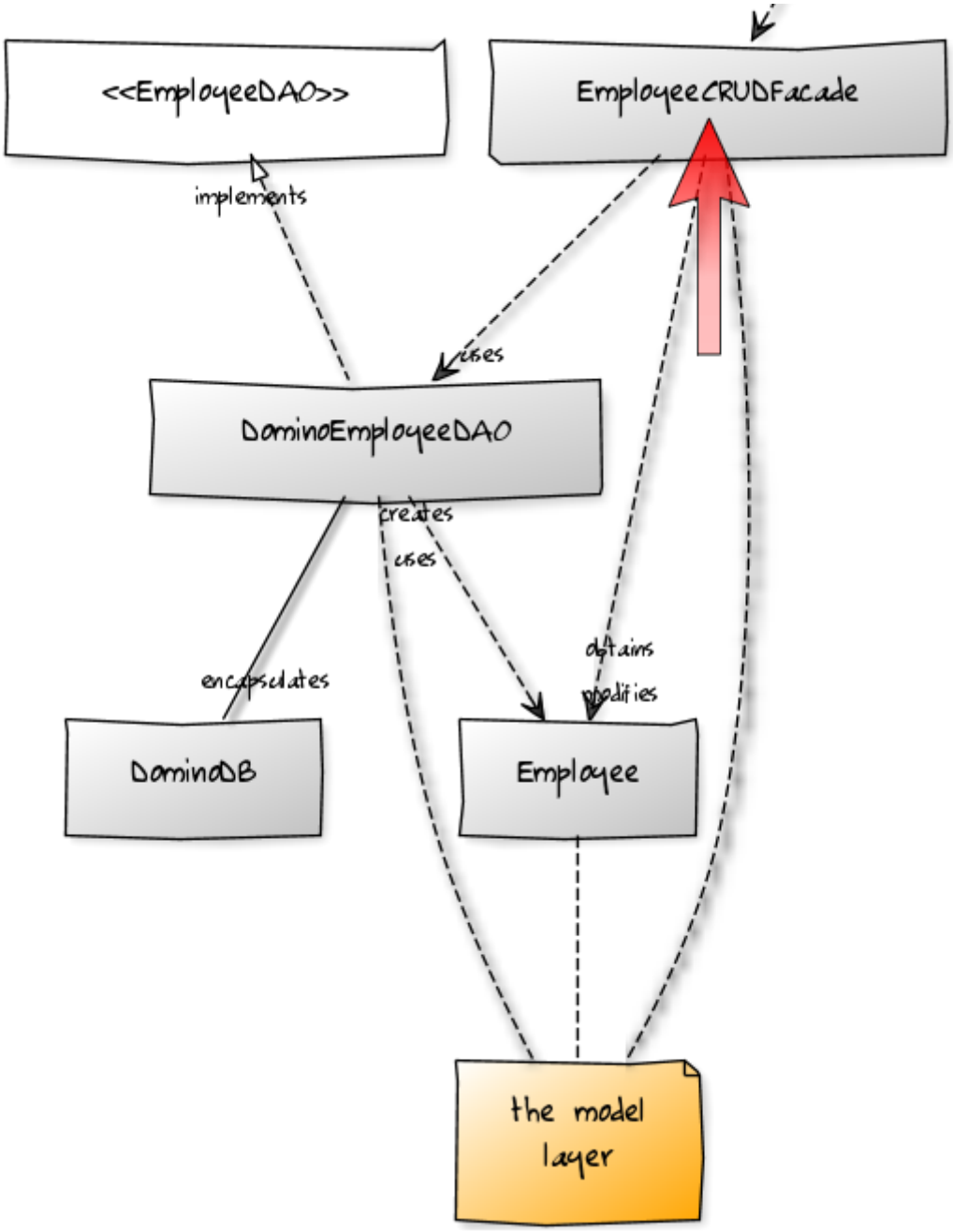
Though we have to create some coding tricks to provide a relational feel to our application, you should not mimic that in your java model classes. When thinking about that statement, consider this example.

We need to create a Holiday request form for our new Employee application. In the form we want to know each day to be taken off and the number of hours. And on the form we want a manager approval section to show if it is approved or denied, by whom, and on what date. In the relational database world, that would end up being three tables. One for the employees involved (both manager and requester), one for the request containing the requester relationship and manager relationship, and one containing each day related to the request. In Domino though, that is not as easy to pull off with a nice UI look and feel. It most definitely can be done, but with Domino, sometimes it is easier to just put a bunch of repetitive fields on the form. Most of us at one time or another have run across the application where the above design would produce something like this for the days off portion of the form: Day01, Hours01, Day02, Hours02, Day03, Hours03, …. up to some ceiling where it is expected that the normal workflow would never reach the number of fields you designed (say Day20, Hours20).

If that design exists anywhere in your code and splitting the data into a separate form is not a choice, you should still keep your Java classes separate. In addition to our Employee class, have a HolidayRequest class and separately a HolidayRequestDay class. Do not add 20 java class fields for each day and 20 java class fields for the number of hours taken on a particular day.

Let the HolidayRequestDominoDAO contain the data specific logic to split the request from the related days and create a list of HolidayRequestDay objects in the request. You will be doing yourself a huge favor by doing this at the data layer. When you get to the view layer, the fact that they are separate opens a whole world of controls that can be applied to each.

Ok, we've gone on about the data classes and hopefully impressed upon you the importance of great model class design. But now let's finish the "M" in MVC and create our final class, the EmployeeCRUDFacade.



EmployeeCRUDFacade

If you've been reading this and the previous blog post intently – as I am sure you have -, or even just skimming through, hopefully something stuck out when we spoke about the model classes representing the data and model classes to read and write the data in the database. There was no data manipulation or business rule code involved!

Data manipulation and business rule code belongs almost anywhere except the data classes and database interaction classes. Those classes only read, write, and hold data.

Examples of data manipulation code:

- Calculate the age of an employee. Your database holds a birthdate, but not an age. The DAO should not calculate the age.
- Calculate the number of years an employee has worked for your company. Your database holds a hire date but not a length of time. The DAO should not calculate the employment time.

How about a more advanced concept. Earlier we talked about a Holiday request feature for our Employee application. Well, what if there was a rule that stated all requests over 40 hours (1 week) require both a Manager's and Director's approval? That means when the employee submits the request, something should be marked in the request that two approvals are required (now we really need a whole other "approvals" table and object, but let's not go there for ease of this discussion). At save time, where would that code go? The DAO? No.

The DAO has only one purpose: create, read, update, and delete (CRUD). It contains the database specific code to perform that function and knows nothing about where the data came from or what the data even means. Our code to trigger two levels of approval should go outside of the DAO in a higher level class.

If you do a little research, you may notice that this middle layer is often referred to by multiple names. Sometimes you will see it called the business layer, other times the service layer, and yet other times a façade layer. Personally, we have chosen the term façade to name this layer. In the end, and at the risk of leading to a debate about semantic, they all generally mean the same thing. The pattern lays the foundation for encapsulated implementation of an application's business logic and invocation of that logic by various clients in a consistent manner. This means you also reduce/remove duplication of code, as your clients share the same common services.

We mentioned "clients" of the service layer. What do we mean? A client of this layer is any code that must invoke the common business logic contained therein. Some example client code:

- UI code
- Workflow processing code
- External system integration code

You may be thinking that you won't have any other code except the UI access the data, so why create this service layer. You are not off in your thinking. According to Martin Fowler's book "Patterns of Enterprise Architecture":

The easier question to answer is probably when not to use it. You probably don't need a Service Layer if your application's business logic will only have one kind of client – say, a user interface – and it's use case responses don't involve multiple transactional resources. [...]

But as soon as you envision a second kind of client, or a second transactional resource in use case responses, it pays to design in a Service Layer from the beginning.

In his statements please take note of a couple of sentences: "....**probably** don't need..." and also "...as soon as you **envision** a second kind of client...". The main point to take away is that you are better off simply creating the layer in the event that someday a "nice to have" feature becomes a coding reality. It is far easier to spend the extra time now then it would be to decouple all of your business code from the UI layer later.

This layer is not a one-to-one with the DAO. If you want to perform several actions on several DAOs for a single business process in the same transaction, then you have all the code contained in one method of a single façade object doing the orchestration, and managing the transactions.

Let's take a look at our Holiday request again. When the employee submits a new request, we create a new HolidayRequest object. That class contains an Employee object and multiple HolidayRequestDay objects that hold the date and hours for each day. In a façade to process this, the following objects would be incorporated into a method possibly called "SaveEmployeeHolidayRequest":

- HolidayRequest DAO to save the HolidayRequest object.
- HolidayRequestDay DAO to save the HolidayRequestDay objects.
- EmployeeDAO to find the employee's direct supervisor and possibly the director's record if the hours exceed 40 for the entire request.

You see, the façade encapsulated the work of reading the request, breaking it into its parts, and processing the 40 hour business rule. All of the DAOs involved and the business rule code was contained in one method of one façade.

In our Employee application, the façade is going to be very simple and only work with the employee entries. We don't have complex business rules or multiple DAOs to incorporate. But as we grow the application, we may, and we may also need to provide external access to employee data. So we will create the façade layer.

```
package com.dao.facade;

import java.util.ArrayList;
import java.util.List;

import org.apache.commons.lang.StringUtils;

import com.dao.DominoEmployeeDAO;
import com.dao.EmployeeDAO;
import com.data.Employee;

/**
 * Service layer class implementing business rules for working with
 * employee data.
 */
public class EmployeeCRUDFacade {

	/**
	 * An instance of a DAO to work with an employee
	 */
	private EmployeeDAO dao = new DominoEmployeeDAO();

	/**
	 * Find an employee based on the employee number
	 * @param employeeNumber	The employee number to search
	 * @return	An employee object loaded with the data or null of no employee found
	 * @throws Exception	The employee number was invalid
	 */
	public Employee findEmployee(String employeeNumber) throws Exception {
		//Make sure the employee number is valid
		if (StringUtils.isEmpty(employeeNumber)) {
			throw new Exception("The employee number is not valid");
		}

		//Setup a null employee object to hold the found employee (if the employee exists)
		Employee employee = null;

		//Get the employee from the database
		employee = dao.loadEmployee(employeeNumber);

		//Return the employee to client code
		return employee;
```

```
}

/**
 * Save an employee
 * @param employee      The employee to save
 * @throws Excption     The employee object did not pass validation
 */
public void saveEmployee(Employee employee) throws Exception {

        //Test the employee object for validity and throw an exception if there is an issue
        if (employee == null ||
                        StringUtils.isEmpty(employee.getEmployeeFullName()) ||
                        StringUtils.isEmpty(employee.getEmployeeNumber())) {
                throw new Exception("The employee is missing required information!");
        }

        //Save the employee
        dao.saveEmployee(employee);
}

/**
 * Get the full list of employees
 * @return      A list of Employee objects
 */
public List getAllEmployees() {
        //Simply return the results of the DAO work - no extra validation here
        return dao.getAllEmployees();
}

/**
 * Find all employees who name begins with a specific letter
 * @param letterToFind  The letter to search
 * @return      A list of employees starting with the specific letter (or an empty list of no employees)
 * @throws Exception    An invalid letter was specified
 */
public List findEmployeesStartingWith(String letterToFind) throws Exception {
        //Check the letter was passed in and it is not a number
        if (StringUtils.isEmpty(letterToFind) || !StringUtils.isAlpha(letterToFind)) {
                throw new Exception("An invalid letter was specified!");
        }

        //Seed an empty list to return
        List employees = new ArrayList();

        //Loop through all employees and pull the matching names
        for (Employee employee : dao.getAllEmployees()) {
                //If the employee name is not empty and a lower case match finds the name starts with the le
                //then add the employee to the list
                if (!StringUtils.isEmpty(employee.getEmployeeFullName()) &&
                                employee.getEmployeeFullName().toLowerCase().startsWith(letterToFind.toLower
                        employees.add(employee);
                }
        }

        //Return the list of employees that meet the criteria
        return employees;
}
```

```
}
```

You will notice that our EmployeeCRUDFacade does not implement a lot in terms of business rules or data processing. Our sample application is small and only meant for illustration. But there are some business rules and some data processing. The idea is that instead of placing that code in the UI related code, by having the façade any client code can simply call one method to get an employee and any processing rules for getting the employee, such as validating the employee number, will be enforced. The code inside the method is not repeated over and over in client code. The only call a client makes is to the one method; a single line to call on the part of the client, but many lines of code in the method.

I threw in a processing method. The "findEmployeesStartingWith" method provides a way to search for an employee by name. Again, this is not the strongest illustration given our sample application is small. In reality, we may find an employee by searching a view instead. The idea of providing this method is to show that the façade may contain methods that interrogate the data and provide a subset of information based on what meets the business rule. The business rule may be something not easily searched. For example, if we had our holiday request feature, we might want to find all employees that have taken off more than 40 hours so far this year. In that case we have to pull multiple DAOs to create lists of information and then summarize and filter the results. Something a simple database search could not provide, but code that should be available for any client request.

Next up...EmployeeBean

In the next entry in the series, we will cover the EmployeeBean. We are making our way to the presentation and the bean will allow us to provide some UI access to our lower layers for each user of the system.

Articles in this series:

July 2, 2013 [http://www.pipalia.co.uk/rethinking-xpages-part-four-finishing-the-model-layer/] | by Gary Glickman | in Bean, mvc, xpages .

10 Comments

10 thoughts on “Using JSF Framework Development Standards for your XPages Project (Part Four: Finishing the Model Layer)”

Murray
July 2, 2013 at 9:25 pm

Awesome series – thanks for doing this. It’s helping me a lot with a current project.
Cheers.

Samir Pipalia
July 2, 2013 at 10:26 pm

Thank you kindly, we are really glad to know that you are enjoying the series and more importantly it’s making a difference.

Steve
July 3, 2013 at 5:31 pm

I can’t seem to get #3 in the series to load… something about too many redirects. Really looking forward to the rest of the series.

Samir Pipalia
July 3, 2013 at 7:16 pm

Many thanks Steve for informing us about this issue, this has now been resolved.

Martin Rolph
July 18, 2013 at 7:55 am

Fantastic series. Am finding it very useful.
Am looking forward to what you say the about the EmployeeBean and how you bind your controls on the XPages. I guess you don't use the out of the box DataSource objects and create your own

Gary Glickman Post author
July 18, 2013 at 2:29 pm

Thank you Martin. I am glad you are finding the series helpful.

Yes you are correct, as a JSF developer, my practice is to keep the data connectivity stuff off of the web page so the out of box datasoure stuff is not helpful to me. The Model layer described in the early part of the series does all the heavy lifting of working with the Domino document. It follows a more traditional web application architecture. One huge advantage of doing it this way in Domino is the unlimited potential for real-time external system integration (for example Web Services connectivity to another application).

The next installment of the series will be available on this site the early part of next week.

Thanks again!
Gary

John Dalsgaard
July 19, 2013 at 11:48 am

This is exactly the way I would like to code

I have a "real" project where I have a data model with 30+ entities. One thing, however, that I am wondering is how "far" you would go with this pattern? I have done it for all entities including lists of values to select from (e.g. a list of universities and different types of keywords). The problem with these is that the do not have a "natural" key (like an employee number). However, I still give them keys (being UNIDs). But I am not sure whether this is a sound approach or I should either have no key – or just treat the name as a key? I am going to use these types of data for selects in the application (dropdown lists etc.). The data are treated as ketwords and as thus can be maintained in the application – but is not used for anything outside of this….

Looking very much forward to the next article.

/John

Gary Glickman Post author
July 19, 2013 at 2:02 pm

Hi John,

Great question. It sounds like you have a good size project ahead of you with some interesting challenges. Those are always fun; they keep the job interesting.

Let me see if I can give you an answer in two parts.

For the first part of the answer ("how far" to go): If we remember that JSF & MVC are standard web application frameworks unrelated to Lotus Notes & Domino, and implementing data as unique objects is best practice in utilizing those frameworks, then really there is no such thing as "too far". The concept of creating objects (often referred to as entities) for each record of data is standard practice, even in POJO type applications. It is not uncommon to see 20, 30, 40, 50, etc… number of entity objects. Now to be fair, they each don't need a separate DAO because many are relationship objects with parents and the DAO of the parent has the job of loading its related children. But that is advanced functionality that was not covered in the article series. Maybe for a future series though.

For the second part of the answer ("natural" key issues in Domino): This concept of keys in Domino is really an impacting topic unrelated to XPages and Beans. Domino, as we know, is not a relational database system. It is a variable length document storage facility. While the flexibility in how data is managed adds to the RAD approach to creating applications, it can sometimes leave a little to be desired in terms of data integrity. For example, given your problem of keyword type documents, what happens when data changes. Imagine a list of lookup documents that controlled workflow for archiving. And the archive rule document status titles were:

"To Be Reviewed"
"Reviewed – Approved for Archive"
"Reviewed – Denied for Archive"
"Reviewed – Re-Review Required"
"Archived"

Then a decision maker comes along and says, please change "Reviewed – Re-Review Required" to "Reviewed – Pending Re-Review". To ourselves we say: ok, I will have to write an agent to update all that data. But if we design our application data using relational database rules, then we would have had a key/value relationship. And if that is the rule we use for all of our data, then changing the title as the person wanted would be as simple as updating the master record with the "Reviewed – Re-Review Required" value. We would not have to run an agent.

In the relational database world, even when there are "natural" keys, such as an employee number, there tends to be a generated record key (primary key). It is usually incremental because relational databases have built in capability to do that. All related records get the primary key, not what we think of as the natural key (employee number). This allows us to change anything about the employee, even the employee number, and all related data remains intact.

You are on the right path in your thinking of the UNID as the generated key. A best practice, in our opinion, is to create a hidden field on every single form called "UNID". Make it computed when composed, with @Text(@DocumentUniqueID)

as the value. Then any related document that needs to keep a connection to another document, stores the other documents UNID in some related field like RelatedKeywordUNID, as an example. All of your code works with the UNID. It is a little more work upfront but saves 10-fold the amount of time at the back end when it's time to do advanced data relationship work (ex. reporting, trend analysis, data management, etc…).

We use the computed when composed field approach to capture the UNID because this allows us (if ever needed) to copy the database without having to worry about relational integrity of changed DocumentUniqueID values in the new copy. The relationship will withstand the database copy because the UNID field is just a unique set of alpha-numeric characters and is not truly the UniqueID anymore (and does not interfere with Note's use of DocumentUniqueID). We just took advantage of how Domino has a built in capability of creating a unique identifier.

This was not covered in the series as it seemed out of scope of the concept of JSF and more of a best practice kind of article for Domino in general.

Other concerns for your project will include when to load data, how to load data (tricks for not going back to the database for simple lists), how to use pre-loaded data, etc… All of that is advanced expertise outside of the scope of these posts; but architectural concerns that will have a big impact on the success of your project. This is where additional resource expertise to help consult on those issues could help ensure the success of your development efforts.

I hope this answer helps you.

Thank you,
Gary

John Dalsgaard
July 19, 2013 at 4:05 pm

Hi Gary

Thank you for your reply. I agree with the issues in changing keywords – especially if they have any processing connected to them – in I tend to use keyword aliases (in normal Notes apps). This also allows for translating the application easily.

I agree on the UNID way of doing it – I just create it backend instead of using a computed field.

And then you mentioned a couple of interesting things:

1. DAO classes for these "keyword" like entities
I have thought of creating something like the CRUD level skipping the DAO for each of these – which I think is sort of the same you are referring to. As an example I have three entities that are closely related:

CoastArea (major parts of the coast)
CoastLocalArea (local parts of the coast)
CoastLocation (specific places on the coast)

This is a simple hierarchical structure where there is a name for each, a key, and a key to the level above. At the moment I have implemented this as 3 data classes, 3 interfaces, and 3 DAO implementations. However, this seems a little overkill… This is where I am thinking a little "too far" Terefore, I was considering not having a DAO for each but perhaps just have one DAO that knows the relations since I would always need to know all three levels when using the data in the app. Is this the kind of situations you are thinking of? If so, what would be a "right" way of doing that?

2. Accessing data in an efficient way
Now, the good thing about using Java and XPages is that we can easily cache static data (like keywords which would normally not need to be re-read within the session). So I definitely have some ideas as to how to do this efficiently.

The next thing is lookups where I would normally create extra views either sorted the "right" way (e.g. by name for keyword type of data) instead of reading data by key (unid…) and then get an arbitrary order – or having to sort in memory afterwards. In this case I have thought of leaving out the view by unid altogether – but I have kept it for the same reasons that you have the computed unid on documents. Other types of views are categorized by "parent key" (the order unid for order items). Is this a path that is loyal to the pattern you are describing? As I understand it the CRUD level will need some of these views to perform efficiently. You mention an example of a method to find employees starting with a character/string by reading all entries and then look at them in memory. Depending on the size of the dataset another view sorted by the name may be more efficient.

One thing I have found is that I needed to implement a mechanism that knows a view index is dirty (after a save/delete) to allow the next load/getAll to refresh the view – or I get old data. I found this was a problem when running unit tests on my DAO classes…

As a side note I can mention that I am developing all my DAO classes using the org.openntf.domino api – which solves many of the memory/handles issues we have had to deal with (….recycle() – anyone….?).

/John

Cameron Gregor
July 23, 2013 at 4:07 am

Thanks again for this series, it is well structured and this 'episode' answered a few questions I had in my head about this approach.
Eagerly awaiting the next installment
