Software House

SPECIALIST IN SAGE AND NOTES/DOMINO DEVELOPMENT

Using JSF Framework Development Standards for your XPages Project (Part Three: Creating the DAO)

June 12, 2013

In the previous installment of this series we described the object model that will be used to help illustrate the Model-View-Controller (MVC) architecture. Included in the article was an object diagram (not quite UML) which provided a representation of a very simple employee database application. To recap, the objects involved consist of:

Model Layer:

- Employee object: This is our low level data. In Domino this comes from the data saved to a Domino employee document. In relational terms, this is a row of data.
- DominoEmployeeDAO (implements EmployeeDAO) : A Data Access Object (DAO) which provides a concrete implementation of an interface to the Employee data for a Domino database.
  - DominoDB: These are the instructions for working with our Domino database. They are encapsulated by the DominoEmployeeDAO.
- EmployeeCRUDFacade: The business logic of retrieving data from the system.

View Layer:

- EmployeeBean: Holds the information you would like to display on an XPage or read from the XPage.
- XPage: Controls the look and feel of your web page, captures and displays data, and acts as the set of instructions for the FacesServlet.

Controller Layer:

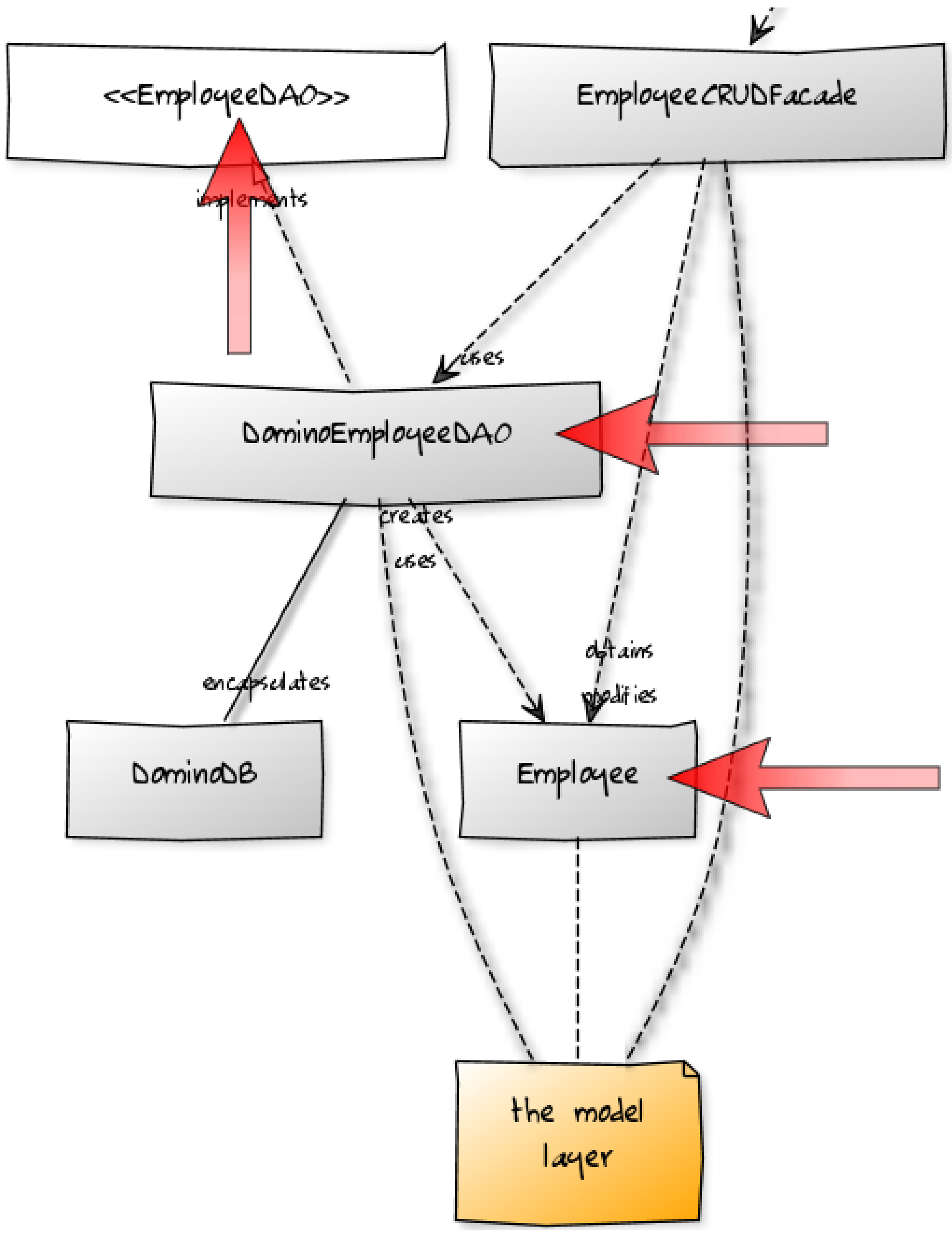
- FacesServlet: It acts as the go between for your XPages and Beans.

*Note: These are only the basic objects of a JSF application implementing the MVC pattern. There are more advanced objects that fit in the scheme. For example you can write an ActionListener which is an event handler that can respond to user events in your JSF XPage (such as clicking a button). You can also write a PhaseListener which allows you to insert custom processing logic into the JSF lifecycle. We will not cover these (and other) advanced topics in this series. For now, we will be writing a very basic JSF application.

In later posts, we can explore interesting things that can be done with an ActionListener to create conditional behavior from an XPage action as well as how to interact with the FacesServlet through the use of a PhaseListener. Notice the phrase “interact with the FacesServlet”. You will actually never write any controller code directly. But through a PhaseListener you can tell the controller what to do through its exposed API methods. But the FacesServlet is still the controller and you are only able to ask it to behave according to its pre-defined capabilities.

This might be a good place to mention one related but off-topic point. Because you are using java, it is often possible to extend and override entire features of an implementation of some sort of API (the FacesServlet in this case), if the developers have not finalized the class(es) involved. And of course you could just from scratch write your own FacesServlet based on the JSF specification (which is what IBM has done – somewhat sidestepping the standard specifications delivered in each version of JSF – but that is a different conversation altogether). In those cases you are then writing your own custom controller. The best you can equate that to in our world would be if IBM allowed you to override or implement your own custom HTTP task on the server. Java does tend to allow you to do those things, just by the nature of the open source approach often used. But let’s move forward based on the fact that 99% of us will never write our own custom FacesServlet and simply utilize the behavior as delivered in the implementation of the JSF specification we are using.

We will spend the next two installments of this series covering the Model layer and the DAO pattern. In this installment, we will create the Employee object and the DominoEmployeeDAO (and EmployeeDAO interface). These are the objects annotated with a red arrow in the diagram below.



Employee

If you look at the Model portion of our object diagram, you will notice that the Employee is at the bottom and our DominoEmployeeDAO is above that. In terms of objects relying on each other, the Employee is our base foundation object. Without an Employee object, we would have no place to hold Domino data for use on our XPage; or anywhere else that may implement employee processing code. The Employee object is synonymous with our Domino Document. If you also use a Notes client in your application, the Domino Document may be backed by a single Notes Form, or more advanced techniques of creating the Document data (ex. LotusScript routines kicked off by workflow). Regardless, the Employee object represents the entire Domino Document. If we were discussing a relational database, this would be considered the columns of a single row.

Our sample application is going to remain very “thin” so as to keep focus on the JSF, MVC, and DAO concepts. We will only have a few fields in our Employee object. As Notes developers, we all know how to create a form and view, so that will not be covered here.

NOTE: You will want to create your Java source folder as part of the web application structure so the entire source is moved with the database when you copy it around. Here are a few steps from IBM on how to do this:

Creating a Java Control in a NSF on www-10.lotus.com

Once you have a source folder in your project, you can start adding the code. This is the Employee.java file which represents our employee record (Domino Document):

```
package com.data;

/**
 * Employee object will hold the data for each employee.
 */
public class Employee {

        /**
         * Employee unique identifier
         */
        private String employeeNumber;

        /**
         * Employee full name
         */
        private String employeeFullName;

        /**
         * Employee work telephone phone number
         */
        private String employeeWorkPhone;

        public String getEmployeeNumber() {
                return employeeNumber;
        }

        public void setEmployeeNumber(String employeeNumber) {
                this.employeeNumber = employeeNumber;
        }

        public String getEmployeeFullName() {
                return employeeFullName;
        }

        public void setEmployeeFullName(String employeeName) {
                this.employeeFullName = employeeName;
        }

        public String getEmployeeWorkPhone() {
                return employeeWorkPhone;
        }

        public void setEmployeeWorkPhone(String employeePhone) {
                this.employeeWorkPhone = employeePhone;
        }

}
```

What should stand out is that the Employee class has no business logic. It simply holds the values of fields from our document and our XPage (and possibly the result of custom process… say calculating the next available employee number). No methodology exists to get the data from our database or read the data from our XPage. That work is delegated to the other parts of our MVC implementation. This particular part of the Model is simply the representation of a row of data (our Domino Document).

DominoEmplyeeDAO (and EmployeeDAO interface)

Up one layer from our Employee object, within our Model layer, is the DominoEmployeeDAO. The DominoEmployeeDAO contains all of the code for reading an employee from our database and storing an employee in our database. It encapsulates all of the Domino Database (DominoDB) instructions for performing reads and writes (and deletes if necessary).

One item of note, in a larger application, it would be more traditional to separate out all of the core methods for opening a database, reading a database, and closing a database, into a separate class by which all Domino<Data Type>DAO objects extend. But in our small one form application, for ease of understanding the layers, we have encapsulated those instructions within the DominoEmployeeDAO object.

Notice also that our object diagram makes reference that the DominoEmployeeDAO implements an interface called EmployeeDAO. The concept of using an interface to define the access method signatures will allow you to potentially extend an implementation of an Employee DAO for another data source (say LDAP for example). This would then give you great flexibility in where you retrieve employee data if it is possible that the information is not centralized.

Here is our EmployeeDAO interface:

```
package com.dao;

import java.util.List;

import com.data.Employee;

/**
 * Interface for implementing access methods for employee DAO concrete classes
 */
public interface EmployeeDAO {

        /**
         * Read an employee from the database and load it into an Employee object
         * @param employeeNumber        The employee number of the employee to read
         * @return      An employee object loaded with the record data or null
         * if the employee was not found.
         */
        public Employee loadEmployee(String employeeNumber);

        /**
         * Save employee information to the database. This will overwrite existing
         * data if the employee already exists or create a new record if the employee
         * is not currently in the database.
         * @param employee      An employee object loaded with employee data
         */
        public void saveEmployee(Employee employee);

        /**
         * Retrieve a list of all employees in the system
         * @return      A list of Employee objects representing all employees in the system
         * or an empty list if no employees are found
         */
        public List getAllEmployees();

}
```

Notice our interface only contains the method signatures and some java documentation on what behavior each method will provide. The documentation is important as this will tell anyone implementing a concrete Employee DAO exactly what their implementation should do. Also, the documentation will tell any developers of client classes of the DAO exactly what to expect from the method use. For example, notice in the "loadEmployee" that if an employee is not found, NULL will be returned. That would be important to know when implementing business rules for working with data returned by the DAO.

So now let's move on to the real core of our DAO; the implementation of this interface which contains all of the code for loading and saving an employee:

```
package com.dao;

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

import lotus.domino.Database;
import lotus.domino.Document;
import lotus.domino.NotesException;
import lotus.domino.View;
import lotus.domino.ViewEntry;
import lotus.domino.ViewEntryCollection;

import com.data.Employee;
import com.ibm.xsp.model.domino.DominoUtils;

/**
 * Concrete implementation of an EmployeeDAO for use with a Domino database
 */
public class DominoEmployeeDAO implements EmployeeDAO {

        /**
         * A Notes database object
         */
        private Database db;

        /**
         * A Notes view object
         */
        private View view;

        /**
```

```
         * @see com.dao.EmployeeDAO#loadEmployee(java.lang.String)
         */
        public Employee loadEmployee(String employeeNumber) {
                //Setup a null employee variable to return if nothing is found
                Employee employee = null;

                //Try to find the employee
                try {
                        //Open the database and view
                        openDatabase();

                        //Try to get a document from the view matching the key
                        ViewEntry entry = view.getEntryByKey(employeeNumber);

                        //If a document was found, create a new employee object and load it
                        if (entry != null) {
                                employee = loadEmployeeFromEntry(entry);

                                //Have to recycle Domino objects
                                entry.recycle();
                        }

                        //Close the database and view
                        closeDatabase();
                } catch (NotesException e) {
                        //Something went wrong, print messages to the server console
                        e.printStackTrace();
                }

                //Return the null variable or if found, the new employee object
                return employee;
        }

        /**
         * @see com.dao.EmployeeDAO#saveEmployee(com.data.Employee)
         */
        public void saveEmployee(Employee employee) {
                //A Domino document that will be created or overwritten if found in the database
                Document doc;

                //Try to find the employee in the database
                try {
                        //Open the database and view
                        openDatabase();

                        //Try to get a document from the view matching the key
                        doc = view.getDocumentByKey(employee.getEmployeeNumber(), true);

                        //If a document was not found, create a new document
                        if (doc == null) {
                                doc = db.createDocument();
                        }

                        //Load the values from the employee to the
                        doc.replaceItemValue("employeeNumber", employee.getEmployeeNumber());
                        doc.replaceItemValue("employeeFullName", employee.getEmployeeFullName());
                        doc.replaceItemValue("employeeWorkPhone", employee.getEmployeeWorkPhone());
                        //For simplicity, we associate the employee to its form here.
                        //You can implement logic if needed, in the CRUD facade, and modify the model to hold a form
                        doc.replaceItemValue("Form", "Employee");

                        //Save the document
                        doc.save(true, false);

                        //Have to recycle notes objects when done
                        doc.recycle();

                        //Close the database and view
                        closeDatabase();
                } catch (NotesException e) {
                        //Something went wrong, print messages to the server console
                        e.printStackTrace();
                }

        }

        /**
         * @see com.dao.EmployeeDAO#getAllEmplouyees()
         */
        public List getAllEmployees() {
                //Create an empty array list to work with
                ArrayList employees = new ArrayList();

                //Try to find the employees in the database
                try {
                        //Open the database and view
                        openDatabase();

                        //Try to get a document from the view matching the key
                        ViewEntryCollection entries = view.getAllEntries();

                        //If we have some entries, load the list
                        if (entries != null) {

                                //Loop through the entries starting with the first row
                                ViewEntry entry = entries.getFirstEntry();
                                while (entry != null) {
                                        //Create a new employee, load it from the row, and add it to the list
                                        Employee employee = loadEmployeeFromEntry(entry);
                                        employees.add(employee);

                                        //Get the next view entry, but be sure to recycle the old entry too
                                        ViewEntry oldEntry = entry;
                                        entry = entries.getNextEntry(entry);
                                        oldEntry.recycle();
                                }

                                //Have to recycle Notes objects when done
                                entries.recycle();
                        }

                        //Close the database and view
                        closeDatabase();
                } catch (NotesException e) {
                        //Something went wrong, print messages to the server console
                        e.printStackTrace();
                }

                //Return the array list (loaded or empty)
                return employees;
        }

        /**
         * Helper method to open a handle to the current database and the employee list view
         * @throws NotesException       Something went wrong, throw it back up the chain
         */
        private void openDatabase() throws NotesException {
                db = DominoUtils.getCurrentDatabase();
                view = db.getView("EmployeeList");
        }

        /**
         * Helper method to close the view and database
         * @throws NotesException       Something went wrong, throw it back up the chain
         */
        private void closeDatabase() throws NotesException {
                view.recycle();
                db.recycle();
        }

        /**
         * Helper method to load an employee from a view entry
         * @param entry The view entry to read
         * @return      A new Employee object with the values from the view entry
         * @throws NotesException       Something went wrong, throw it back up the chain
         */
        private Employee loadEmployeeFromEntry(ViewEntry entry) throws NotesException {
                //Create a new employee and load it
                Employee employee = new Employee();
                employee.setEmployeeNumber((String)entry.getColumnValues().get(0));
                employee.setEmployeeFullName((String)entry.getColumnValues().get(1));
                employee.setEmployeeWorkPhone((String)entry.getColumnValues().get(2));

                //Return the new employee
                return employee;
        }
}
```

Let’s dissect this class a little.

In the class declaration line “**public class** DominoEmployeeDAO **implements** EmployeeDAO “ we implement the EmployeeDAO interface previously created. This then requires that the concrete implementation DominoEmployeeDAO contain at least the methods defined in that interface. This way we can access every EmployeeDAO the same way (Domino in this example, but maybe also LDAP or Oracle in a more distributed environment).

The class contains some private fields for the database connection and view we will use to retrieve data. These are class level so they can be shared across methods if need be.

You will notice there are three public methods defined in the interface and implemented in the DominoEmployeeDAO. They are loadEmployee, saveEmployee, and getAllEmployees. These each match the signatures of the EmployeeDAO interface. Within each method is the concrete implementation for performing the work on a Domino database and returning the required method object.

You will notice that the method signature does not change. The signature must match the EmployeeDAO interface so that any client using the DAO can expect what the interface has defined as the behavior. Client code using the DominoEmployeeDAO will not have to know anything about the database.

The implementation rules for how to work with the database and underlying data are encapsulated in the method implementations and not important to external code. For example, the method "loadEmployee" simply takes a Java String as a key to find an employee and then if the employee is found, returns an instance of the Employee object loaded with the data. Any external code using this class does not have to know the Domino dialect for reading a Domino document. Client code does not have to know the database name and/or location in which the document is located. The client code simply expects an Employee object with data or a java NULL.

In the DAO pattern, the only place that database specific dialect is implemented is in the concrete class that works with the records (documents for Domino) of that database.

We are not going to dive into the specific line items on how to open a database, read / save a record, or traverse a view. The purpose of this article is to discuss how to implement all of that Java code in an organized more mainstream industry approach to JSF and database interaction. But we have heavily commented each method to help you understand the underlying logic of the method implementation.

Also, you'll notice some helper methods for opening the database and view, closing the database and view, and reading view entries. It was constructed that way because those methods contained more common code that was better suited for use at the class level, across many other methods.

And finally, I mentioned earlier that larger implementations (say multiple Document types in the same database or multiple databases) would create the need for a super class that contains the entire database create, read, update, and delete (CRUD) code. In that scenario, the DominoEmployeeDAO would probably extend a class such as BaseDominoDAO and the BaseDominoDAO would have all of the dialect for opening the database, getting a view, reading view entries, etc…

Some "Domino" Points to Note

- Variable length documents: We know domino does not enforce the existence of traditional column structures like a relational database. So you may have a form that existed for a couple of years and suddenly a new field is added. Well, you wrote all your new code to deal with the fact that any reference to the field may produce an empty string. You will need to mimic that same behavior in your DAO. For example, you may need to do a check to see if the Item exists on the Document before attempting to read it. The rules in the DAO are no different because it is Java. All of the little quirks we have come to accept still require attention when working with the data via Java.
- Reading and writing the fields one-by-one. Yes, you need to load each field into the Java Employee object (or whatever your object will be), one-by-one. Something we are not used to doing in Domino (unless working with a lot of LotusScript). But there are some advanced designs that can be implemented to save on all the related code. With some ideas stolen from the Java Persistence Architecture (JPA), there are some tricks for annotating your Java object with the related Domino field name and then spending a little time writing some code to reflect the Java

objects and iterate over the annotated fields to read the field from the database. That is a far more advanced topic and I only mention it here to hopefully calm your thoughts when you consider all those forms with 100s of fields.

- Data from multiple databases. That is no different than reading data from the current database as shown in the example above. Your DAO implementation should be given knowledge of the server and database where the related data is stored. Then use that to open the database, instead of getting the current database.
- Replication, conflicts, multiple users, locking documents, clustering, and etc...: All of those considerations still apply as if coding in the Macro language or LotusScript. Those are functionalities of the underlying database server and only require consideration in your code as any/all of those may impact your workflow. Writing the code in Java neither adds to or eliminates how those items impact your application.

Next up...EmployeeCRUDFacade

In the next entry in the series, we will cover the EmployeeCRUD Façade and how it can implement business rules surrounding the data it reads from and sends to the EmployeeDAO (DominoEmployeeDAO implementation). Hopefully it was clear that in our DAO level work, we implemented no logic for checking our data. Those rules don't belong in the DAO. The DAO has one job; work with data in the database (create, read, update, delete). The rules for how to deal with the contents of the data and any special processing based on data values belong higher up in the Model layer of the MVC pattern.

Articles in this series:

1) Using JSF Framework Development Standards for XPages Project (Part One: Rethinking the Approach to XPage Development)
2) Using JSF Framework Development Standards for your XPages Project (Part Two: Designing Your Application Model for MVC)
3) Using JSF Framework Development Standards for your XPages Project (Part Three: Creating the DAO)
4) Using JSF Framework Development Standards for your XPages Project (Part Four: Finishing the Model Layer)
5) Using JSF Framework Development Standards for your XPages Project (Part Five: The View)

Share:

MORE

Tweet Like 0 G+ Share Pin

June 12, 2013 [http://www.pipalia.co.uk/rethinking-xpages-part-three/] | by Gary Glickman | in Domino, java, jsf, mvc, xpages .

4 Comments

4 thoughts on “Using JSF Framework Development Standards for your XPages Project (Part Three: Creating the DAO)”

Cameron Gregor
June 20, 2013 at 3:13 am

Thoroughly enjoying this series and looking forward to the next post!

Gary Glickman Post author
June 20, 2013 at 4:33 pm

Hi Cameron,

Thank you again for the kind words. I’m glad you are finding this informative. The series is meant to provide some core concepts and hopefully in later articles, after the series is complete, we can expand on the different areas and show some advanced functionality to help make your applications more powerful.

Thanks again,
Gary

TomVanAken
July 23, 2015 at 12:27 pm

Hi there,
I just read and enjoyed the series.
I had some questions concerning the MVC concept and Author/Reader Access on Domino documents.
I have a database with 2 forms: Account and Contact. Each account can have multiple contacts (Account UNID used as key)
On the Account form, I have an Author field AccountManagers specifying who can edit the account (can be multiple users or groups). The Account Managers of an account are also allowed to edit the underlying Contacts. In traditional Domino development, I would have an Author field on the Contacts form which does a lookup to make sure it mimics the AccountManagers field of the related account.

In the MVC model you state that we have to avoid duplicating data on both forms. But if I do not duplicate the Author field, an AccountManager editing a contact might not have author access to the Contact Document. How would you handle this situation in your DAO?

Secondly, I want to avoid showing an Edit button on an account/contact page if the current user does not have author access to the underlying document. What’s the best way to verify if a user is in fact author of a document (taking specific

Domino features such as ACL Editor Access, UserRoles, Groups and all into account) and use that information in the View Layer?

Kind regards,
Tom

Gary Glickman Post author
July 27, 2015 at 1:17 am

Hi Tom,

Thanks for reading the post. I am glad you enjoyed it.

Regarding the authors/readers fields, in your scenario they would not be considered repeat information. While the security of your application has the main Account set with authorized account manger names/groups, it is possible (and often common) that individual contact information is accessible beyond the core team. So in that scenario, the author/reader fields have to be set individually in each document. Your application may not implement that business scenario, but you would want to remain flexible in the event of future requirements that leverage this out-of-the-box flexibility.

As far as an edit button hidden based on the user. You will use some server-side JavaScript. Specifically, use a rendered (or disabled) using a test similar to this:

return NotesContext.getCurrent().isDocEditable(notesDoc);

I hope that helps!

Thank you!
