Software House

SPECIALIST IN SAGE AND NOTES/DOMINO DEVELOPMENT

Using JSF Framework Development Standards for your XPages Project (Part Five: The View)

August 14, 2013

In the previous installment of this series we completed our Model layer described in the Model-View-Controller (MVC) architecture. In that article we covered the façade (or service) layer:

- **EmployeeCRUDFacade:** A service layer class to handle data manipulation and data business rules for access by client code that should not work directly with the model layer.

Examples of data manipulation code and data business rule code include:

- Calculate the age of an employee. Your database holds a birthdate, but not an 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.
- Calculate the number of approvals required for a time off request that exceeds a certain number of hours.

Examples of client code that should never access the model directly include:

- Workflow processing code.
- External system integration code.
- UI code (a managed bean…covered in this article).

It may appear on the initial design of an application that this layer is just adding extra code. In other words, why not just let my view bean access the DAO? Well, ask yourself, is that single view bean going to be the only code to ever access the data in question? Is it foreseeable that some other code may need to also get at the data stored in the model layer? If the answer is yes, or maybe, or even “I'm not sure”, then build the service layer. Redesigning your application later will be a far more resource expensive exercise than spending a little more time up front. According to Martin Fowler's book “Patterns of Enterprise Architecture”:

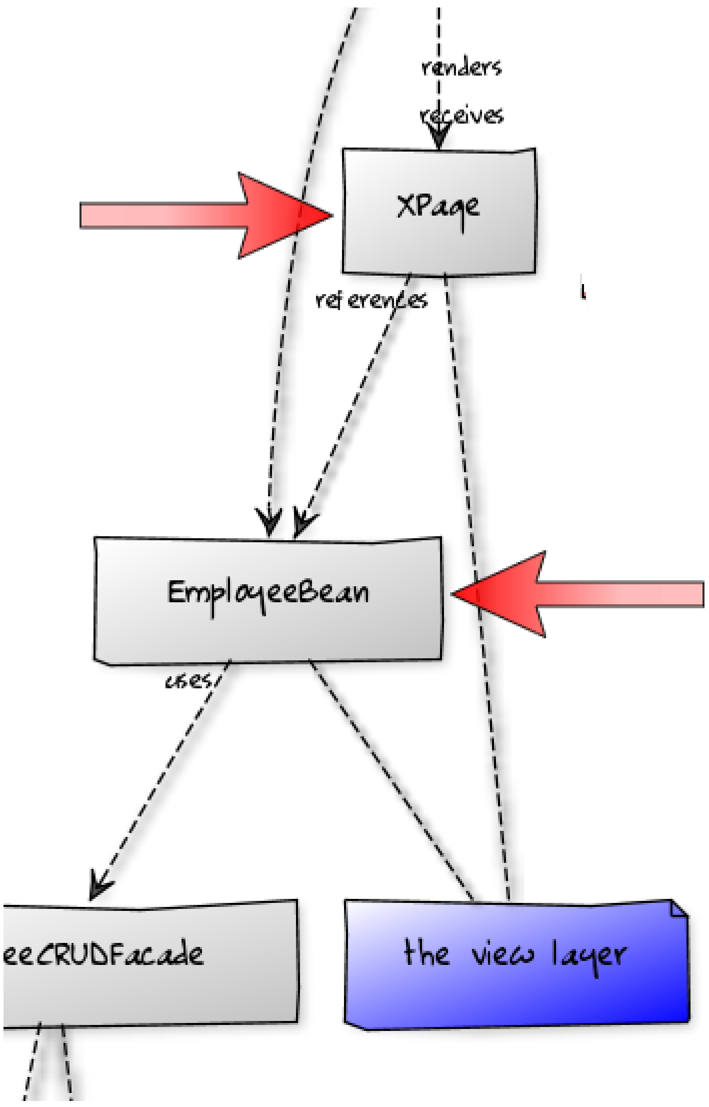
The easier question to answer is probably when not to use it. You probably don't need a Service Layer if your application's business logic will only have one kind of client – say, a user interface – and its use case responses don't involve multiple transactional resources. [...]

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.

Ok, so let's move on to our client of the service layer EmployeeCRUDFacade object, our EmployeeBean.

EmployeeBean and XPage

The EmployeeBean is our client code of the service layer. It is also our interface for access by the XPage. In this final installment, we will create our EmployeeBean code and wire it up to a very simple XPage. While we only have this one bean and simple XPage, envision our application has many of these beans that must access the employee data, which is why we created the service layer.



EmployeeBean

The EmployeeBean will contain all of the UI (XPage) interaction code:

- What to do when someone navigates to a page that is supposed to present a list of employees.
- What to do when someone clicks a “New Employee” button.
- What to do when someone clicks an “Edit Employee” button.

The EmployeeBean can also contain special code to help the UI present the data with special effects. Examples of such effects might include displaying an image on the employee detail page when someone views a record of an employee that has been with the company for more than 10 years. Or maybe an alert message when the employee is past due for their annual performance evaluation.

Please don’t confuse this last bit with the “data manipulation” code I mentioned that the service layer would handle. Let’s dive deeper into how the various layers handle different parts of the equation that might lead to a special image appearing for employees that have worked with the company for more than 10 years:

- Model Layer: The database will hold a field called “EmployeeHireDate”. This is the date the employee was hired. The database can’t track a running calculation of how long the employee has worked for the company. Time is ever moving and the database only works with data at the time it is requested.
- Service Layer: Some sort of client requests data by asking the service layer to pull one or more records from the database. The service layer client does not know anything about the database. The service layer client only has visibility to the construction of the Employee model object. The service layer has no idea what the client wants to do with the data. It only knows it received a request to pull the employee. So the code in the service layer pulls the employee and adds a bit of data to the Employee model. It adds the number of years the employee has worked for the company as of that point in time.
- Bean Layer: It asks for and receives a populated employee object. This layer, unlike the service layer, knows that the UI wants to display a special image when the employee has worked for the company for more than 10 years. So the bean layer has a Boolean property called “displayMilestoneImage”. The bean layer looks at the employee object, tests the number of years, and sets the property to true.

On a side note, some of you may be thinking, “why not put the test in the actual XPage?” Well, here developers will probably all be split on a best practice.

The guidance we are providing is that the XPage should contain as little programming code as possible. The cleaner you can keep the page with pure tags, the easier it will be to maintain and the easier it will be for a web graphic designer to create an engaging interface. This is not a hard rule, just a practice to follow as often as possible. You will definitely have code syntax in your XPage. It’s nearly unavoidable. But anywhere you can keep the code thin, the easier it will be for non-developers to “spice-up” your UI.

In the example above, all we have to do is add a “render” attribute to our XPage tag and wire the bean’s getter for the “displayMilestoneImage” to the attribute. Yes, the “render” attribute could also be wired to a simple test against the number of years. But let’s think beyond the image. What if other UI properties were to render or behave certain ways based on the same number of years? All of those pages would have a copy of the test (which means the number of years gets hard coded into all those pages). Or, they all wire up to the Boolean property, and only one place has the number of years to test against (the bean method).

XPage

The XPage represents your pure view layer. It provides a mechanism for the user to interact with information. If you look back at all of the other layers involved, the code we have written, and the explanation of each of the objects at the various layers, you should now see there is nothing left for the XPage to do except provide an interface to the information. There will be no data processing, database interaction, data manipulation, external system integration, or anything else. All of that work has been delegated to the various layers covered in the previous articles.

The Code

EmployeeBean

So let’s take a look at some code. Let’s build the EmployeeBean first. The bean will contain methods for the XPage to access data retrieved from the database and add & update data captured in the UI. We know the bean won’t actually do the heavy lifting for all that work low level work. But it will provide mechanism for the UI to kick off the various tasks.

```
package com.bean;

import java.io.Serializable;
import java.util.List;

import com.dao.facade.EmployeeCRUDFacade;
import com.data.Employee;

/**
 * This is our EmployeeBean used to connect the employee information to an XPage
 * This bean will provide standard getters & setters for dealing with lists and individual
 * employees.
 *
 * This bean will be defined in the faces-config.xml as SessionScoped.
 * The bean name for XPage access will be 'employeeBean'
 *
 */
public class EmployeeBean implements Serializable {

        /**
         * All beans must implement the Serializable interface
         */
        private static final long serialVersionUID = 1L;

        /**
         * The list of employees to display on the XPage
         */
        private List employees;

        /**
```

```
         * An instance of an employee object to be added, updated, or deleted
         */
        private Employee employee;

        /**
         * The CRUD facade used for accessing and upding our employee information.
         * There will be no public getter or setter methods for this field
         * because it should not be accessible directly from the XPage
         */
        private EmployeeCRUDFacade employeeCrud;

        /**
         * All Beans must have a zero argument constructor.
         * Beans can have additional constructors, but if they do, there must
         * be a zero argument constructor explicitly defined, even if it does
         * nothing.
         */
        public EmployeeBean() {
                super();

                //Get an instance of the CRUD facade to work with the data
                employeeCrud = new EmployeeCRUDFacade();
        }

        /**
         * A mechism to load all the employees from the database to
         * a list for external access
         */
        public void loadEmployeeList() {
                try {
                        setEmployees(employeeCrud.getAllEmployees());
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }

        /**
         * Seeds the employee object with an empty new employee for creation
         */
        public void createNewEmployee() {
                setEmployee(new Employee());
        }

        /**
         * Save the new or edited employee back to the database
         * NOTE: If no id is entered for the employee this method will generate an id
         */
        public void saveEmployee() {
                try {
                        employeeCrud.saveEmployee(getEmployee());
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }

        /*
         * ########################
         * GETTERS & SETTERS
         * ########################
         */

        /**
         * Get the list of employee objects
         * @return      A list containing the employee objects
         */
        public List getEmployees() {
                return employees;
        }

        /**
         * Set the employees field with a list of employees
         * @param employees     The list of employees to set
         */
        public void setEmployees(List employees) {
                this.employees = employees;
        }

        /**
         * Get the employee to work with
         * @return      The employee object to work with
         */
        public Employee getEmployee() {
                return employee;
        }

        /**
         * Set the employee to work with
         * @param employee      The employee object to work with
         */
        public void setEmployee(Employee employee) {
                this.employee = employee;
        }

}
```

EmployeeCRUDFacade

We are going to make a small change to our EmployeeCRUDFacade. Because we are not deploying data validation, we will have to shut down the possibility of saving an employee without an employee number. To do this, we will add a bit of "data manipulation" code to the façade saveEmployee method:

```
if (employee.getEmployeeNumber() == null ||
                                        employee.getEmployeeNumber().trim().equals("")) {
                                /*
                                 * If the employee number was not entered, force a random id
                                 * Because our employee id is the key to a record
                                 */
```

```
                                    employee.setEmployeeNumber(String.valueOf(UUID.randomUUID()));
}
```

This code tests to see if the employee number is empty and if so, assigns a random employee number. Since the employee number is the key to editing an employee, we must have one.

Faces Configuration

JSF 1.x applications, including IBM XPage applications, require a faces-config.xml file that defines the Managed Beans in scope of the application. JSF v2.x actually makes the contents of the file optional in favor of annotations directly on the bean, but that is a whole other conversation. For our application, here is the faces-config.xml file which contains our EmployeeBean. This makes the bean accessible to the XPage at runtime.

```
                    employeeBean
                    com.bean.EmployeeBean
                    session
```

We are not going to use navigation rules in our faces-config.xml. Instead we will have actions directly on our XPage navigate to where we want to go.

XPage(s)

The purpose of this series is to provide a base understanding of JSF applications. XPage design, control use scenarios, control re-use practices, and other UI best practices for XPage development are outside the scope of this article. We will not be covering advanced functionality through DOJO or the extension library.

We will be using two very simple XPages which have a sole purpose of showing you how the UI connects to the underlying framework.

employeelist.xsp

This is our primary list of employees currently in the database. This will contain functionality to show the list, edit an item in the list, and add an item to the list.

A few of notes about the code:

- Server Side JavaScript (SSJS): In our buttons for adding and editing an employee, we used SSJS. JSF 1.x had limited access functionality to method invocation of beans. Especially when dealing with passing information from the UI to the bean. JSF 2.x has resolved all that with nearly full Java behavior for method invocation directly from the UI (enum objects are still missing…sadly). Since Domino is based on the JSF 1.1 specification (with some enhancements) we are still limited with our method invocation. No fear though, Domino has brought SSJS to the table which returns the full Java method access that it's enhanced v2.x can perform.
- After we click our buttons, we are starting a new employee or editing the employee at the row index. The next action is to redirect to the editor page. Since each button has first called a method, the bean is ready with a new employee or existing employee stored in its local 'employee' field, which the editor will use.
- Notice that when the code references Expression Language (EL), the full method name for getters and setters is not used. The JSF framework will convert the reference to the standardized getter and setter method names and invoke those. So for example, where you see 'employeeNumber' the JSF framework is actually executing 'getEmployeeNumber()'. For any field reference on a JSF (XPage) page, you must have a zero argument getter & setter pair that follows the standard naming convention of capitalizing the first letter of the field and adding a prefix of 'get' or 'set'.
- For SSJS, the previous note does not hold true. When calling a bean method, use the full name with parenthesis.
- The 'beforeRenderResonse' tag is used to force the XPage to invoke the loadEmployeeList() method of the bean so that it always pulls the latest data for the page to show.

employeeeditor.xsp

This is our form for adding a new employee and updating an existing employee.

```
Employee Number:



Full Name:



Work Phone:
```

Not a lot of magic on this page. Basically, the page will access the 'getEmployee()' method of the employeeBean for each field in the table. That returns the current Employee object stored in the 'employee' field. Then, with that object we are referencing each field of the Employee object.

When rendering a page, JSF knows to call the "getter" method of each field. And when submitting a page, JSF knows to call the "setter" method. During the JSF Lifecyle, the values will be read in from the UI and passed to the methods referenced at the UI component.

One note, you will notice that the employeeNumber is disabled if the employee already has a number. Since this is our key field we do not want to allow the user to change our key otherwise we would end up with a new record when the user clicked save. In a real-life application we would create code to handle that scenario, but this is a sample application to demonstrate JSF functionality, so we will keep the logic portion very simple.

Wrap-Up

That concludes our series "Using the JSF Framework Development Standards for your XPages Project". The application was extremely simple, purposely. We wanted to show you very basic code and how to wire up Java as your JSF infrastructure, instead of the Domino data controls. There is still a lot of "JSF" stuff that was not covered:

- Validation
- PhaseListeners
- The JSF LifeCycle
- Advanced data retrieval and display methodology for large datasets.
- Organizing the work of beans into logical reusable objects (ex. utility type bean vs a UI bean).
- More...

Some of these topics have been covered in articles previously posted in this blog. Some are still yet to come. We hope you enjoyed the series and found it useful in providing a base understanding of the JSF framework inside of Domino XPages.

We have attached the full working database for you to download:

Click to download

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

August 14, 2013 [http://www.pipalia.co.uk/rethinking-xpages-part-five-the-view/] | by Gary Glickman | in Beans, java, jsf, mvc, xpages .

12 Comments

12 thoughts on "Using JSF Framework Development Standards for your XPages Project (Part Five: The View)"

John Dalsgaard
August 14, 2013 at 10:16 pm

Thanks for the great work that gives very good inspiration to create better XPages applicatons 😉

/John

Gary Glickman Post author
August 15, 2013 at 2:04 pm

Hi John,

Thank you for the kind words. I am glad you found the article useful.

Please check back as other blog posts are created. Hopefully you will find those equally as useful.

Thank you,
Gary

Cameron Gregor
August 16, 2013 at 4:14 am

Thanks again for this extremely well presented series.
This topic is very relevant to what we are trying to achieve in our team and to have it explained to this level of detail has been invaluable.

Gary Glickman Post author
August 16, 2013 at 1:53 pm

Hi Cameron,

I am glad you found it helpful. While we tried to keep the example as simple as possible, the hope was to expose the major layers in the framework.

Please check back for future posts…hopefully you will find those helpful as well.

Thank you,
Gary

Luis
August 21, 2013 at 7:50 pm

Thanks you very very much for all the episodes so far!! I am new to xpages but not to lotus notes and lotusscript.
I was looking for a such material that you have provided to us.
I feel that is the path to follow in order to create and develop better XPages applicatons!!

Thanks again, great job!!

Gary Glickman Post author
August 22, 2013 at 2:12 pm

Hi Luis,

Thank you for the feedback. We are glad you found the series helpful. Please check back in the future for more related articles.

Thank you,
Gary

Luis
September 2, 2013 at 7:32 pm

Hello again Mr. Glickman, I have a question about how I can open and populate a dialog form eith the properties of a bean object?
I have seen in your example how to set de bean and open another xpage binding the fields with the previous bean.
I am woriking in an example binding a datatable with a property bean, following your articles and these work pretty good so far.
Now, I created a form using de dialog control from the extended library and my question is how can I populate the fields of the form with the bean properties of the current row in the datatable?

Thanks you very much in advance, sincerely,

Luis

Gary Glickman Post author
September 4, 2013 at 2:38 pm

Hi Luis,

Unfortunately, without knowing the specifics of your application or the design of the code, it is hard to say exactly how you should implement the connection. But taking information from a dialog based form and plugging the results into the bean or getting the information from the bean, is no different than when using a main page.

Please feel free to reach out to us if you wish to engage some consultancy for assistance with your application.

Regards,
Gary

George
March 6, 2014 at 12:58 pm

Hello Mr. Glickmen,

fantastic professional series. Am finding it very very useful 😃

I am interested in the topic "Advanced data retrieval and display methodology for large datasets."

Why: In my real world application I have to handle the presentation (sorting, filtering, etc…) of thousands of documents and therefore I am using the xp:dominoView control because it is – in my opinion – the only way to do this job in a performant manner. The problem of this solution is well known => model layer "operations" in the presentation (view) layer therefore => no MVC 🙁

Thanks in advance for any response
George

Gary Glickman Post author
March 6, 2014 at 5:04 pm

Hi George,

I'm glad you are finding the series helpful. Thank you for taking the time to read it.

We do understand your dilemma regarding large datasets. Yes, using the built in functionality of XPages by placing a view on the XPage is a direct violation of the principles of MVC.

The available solutions all require lower level coding at the service layer working with document collections.

Pipalia Software House would be happy to provide consulting for you and help you determine the best approach to implement a large dataset in MVC, given your application requirements. We have extensive knowledge both in and out of Domino based solutions. If you are interested, please feel free to contact Samir Pipalia @ +44 (0)20 8922 1665 or info@pipalia.co.uk.

Thank you,
Gary

George
March 7, 2014 at 10:32 am

Thanks for your prompt answer,

we will contact you if we do a refactoring of our code 😃

George

Mark Maden
January 13, 2015 at 10:08 pm

Great series, thank you very much.

I was disappointed to find that you had not used an example using the Parent/Child or one to many relationship which you describe as being one of the key reasons for using Java in XPages, i.e. order and line items.

I have found lots of resource on Java programming but for the life of my cannot find any real life examples of best practice when using XPages to replace standard Notes Forms with multiple embedded views for 'line items' etc related by a parent ID.
If you could point me to any examples of provide any tips on the basic concepts that would be fantastic.