Pipalia®

Software House

SPECIALIST IN SAGE AND NOTES/DOMINO DEVELOPMENT

Using JSF Framework Development Standards for your XPages Project (Part Two: Designing Your Application Model for MVC)

May 29, 2013

In the previous installment of this series we introduced the idea of using standard Java development patterns for implementing your XPages project. The best practice of implementing the Model-View-Controller (MVC) architecture was described. To recap, the layers of MVC consist of:

Model: These are your Java objects that contain and manipulate the data from the database and implement business rules for managing that data

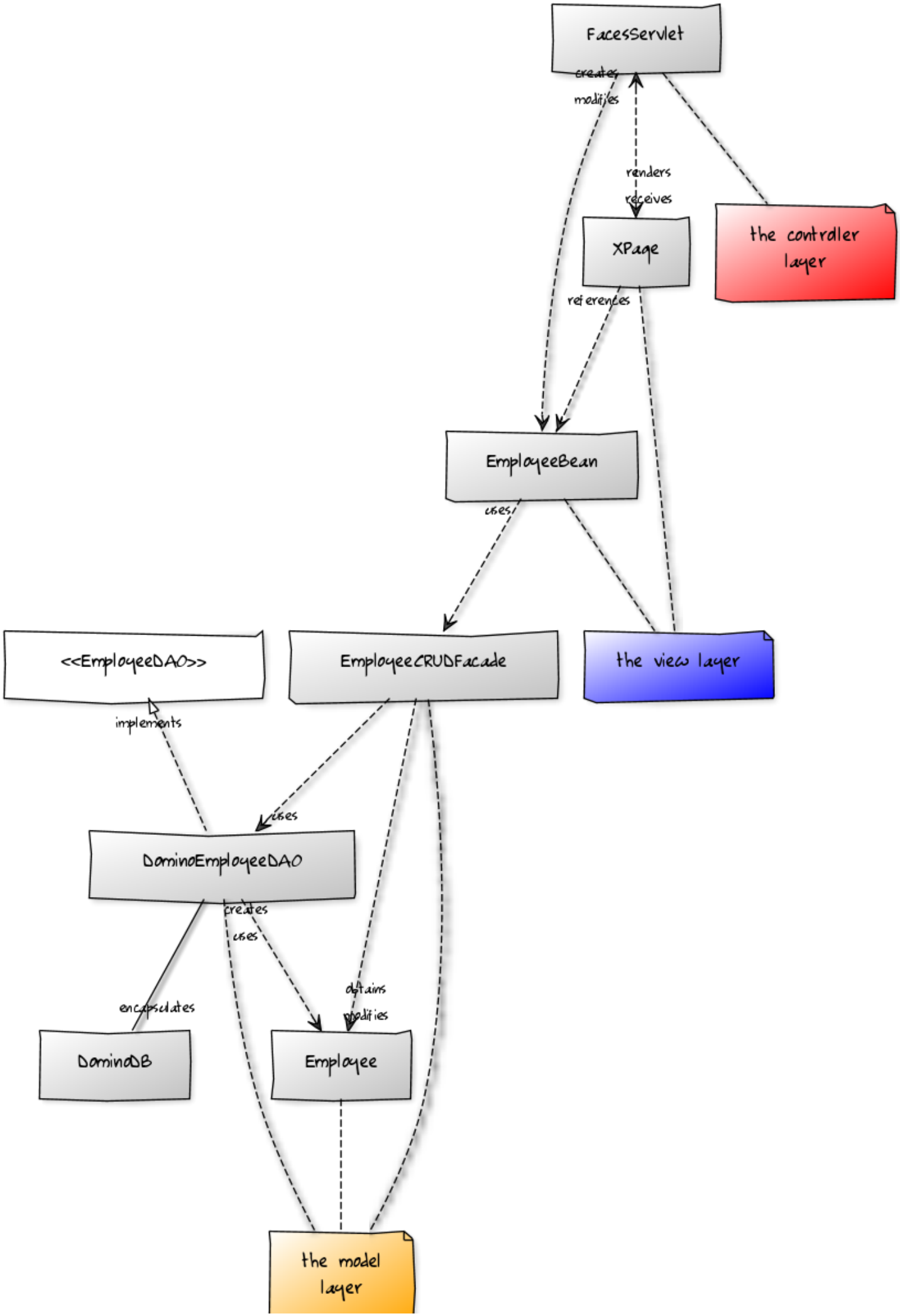
View: These are your presentation Java objects. Mostly (exceptions aside) these tends to be a one-to-one relationship between an XPage (graphical view) and a related view Bean.

Controller: This middle layer is the JSF API. No code is written here by the developer of JSF applications (including XPages). This is the FacesServlet.

In this installment and over the next few installments, we are going to dive deeper into each of the layers of MVC. To help illustrate each part the MVC pattern and applying it to a Notes & Domino XPage project, I am going to create very simple database application that tracks employee information. The dataset will be tiny in terms of the number of fields and documents. The form design and XPage design will be very simple and will not contain the layout standards usually found in a Domino application. Its purpose is to help illustrate how to implement a pure JSF framework for XPage development.

The Employee Database Application Model

I have designed the following model diagram which represents all of the components involved in creating the employee database Domino JSF application:


FacesServlet
creates
modifies
renders
receives
XPage
the controller layer
references
EmployeeBean
uses
<<EmployeeDAO>>
EmployeeCRUDFacade
the view layer
implements
uses
DominoEmployeeDAO
creates
uses
encapsulates
obtains
modifies
DominoDB
Employee
the model layer

Framework Diagram

You may be looking at the model and thinking to yourself that there appears to be an excessive number of classes just to have a simple employee tracking XPage. As Notes & Domino developers we are all very used to placing fields on a Form and/or components on an XPage, specifying a few attributes of the fields, maybe some data validation LotusScript on the Form or Server Side Java Script on the XPage, and being done. And for such a simple example, I will concede that you are mostly correct. But the point here is to illustrate the model and to understand that when you design large scale enterprise systems, the model will not change. This in turn means your coding decreases substantially and is actually less than if you built everything in native Notes/XPage language. Also, the model supports industry standard methodologies for opening, working with, and closing a database vs. live connections from your XPage and lingering database sessions on the server.

Understanding the Employee Database Application Model

So what does this model all mean? I will break out the components and explain the functionality. But before I do that, take note of the color coded boxes (the red, blue, and orange boxes). Those annotations show you where each of the model objects fit into the MVC pattern. If you remember from my first post, I had described the MVC pattern in terms of the JSF architecture and the code you write.

FacesServlet: The core of the JSF architecture is the FacesServlet which is part of the web application server (Domino in our case) API. You don't write any code here. It is the controller of the group and acts as the go between for your XPage and Bean. You can influence its behavior in terms of rendering, navigation, and lifecycle processing, but you won't write any of the core code that receives & renders information.

The FacesServlet has many important jobs. It renders your XPage, it receives requests from your XPage, it creates your Managed Beans, and it updates your beans based on requests from XPage submissions (or component Ajax requests):

- *Render:* When the FacesServlet renders the XPage, it reads all of the instructions in the form of the XPage tags and bean references. It will execute all of the Java code behind the tags and also the code of your Managed Bean. The result will be standard HTML (possibly mixed with JavaScript depending on what you have coded).
- *Create:* During the rendering of your XPage, if there are references to a Managed Bean the FacesServlet will create the bean if it does not exist.
- *Receive:* When an XPage is submitted (or Ajax request made) the FacesServlet responds to the submission/request and processes the instructions from the XPage. You may notice that the result of the rendering of the XPage, all of your tags that had no ID have a generated ID value for the HTML tag. This is because the FacesServlet will process all of the underlying tag code for the element by ID. Remember above I mentioned that a page is rendered as simple HTML. The FacesServlet needs a method to map each HTML component back to the XPage tag it came from. It uses the IDs to perform that connection and processes the original XPage tag instructions.
- *Updates:* During the processing of the request from the submitted XPage (or Ajax request) the FacesServlet will update the existing bean with data from the XPage. Whatever code you have written in the methods referenced in the XPage from your Managed Bean, will be executed by the FacesServlet.

** Earlier I had mentioned that the FacesServlet can be influenced in how it works and what it does. That conversation is more advanced and not covered in this overview. But I will mention that throughout all of the work described above, it is

possible for you to capture the FacesServlet at certain stages (the JSF LifeCycle) of what it is doing and add additional instructions, manipulate the beans, modify the rendering, and even divert the flow of the application and cause it to work with a completely different XPage.

XPage: You mostly know what the XPage does. Beyond controlling the look and feel of your web page, the XPage acts as the set of instructions for the FacesServlet to do its job. All of those tags you place on the XPage have related Java code behind the scenes. That code, which may be as simple as HTML rendering information, or as complex as processing instructions for a dataset, is executed by the FacesServlet.

Take note of the model annotation that I made about the XPage referencing your Managed Bean. The XPage only contains a textual reference to your managed bean via the placement in the XPage tags. The XPage itself does nothing to or with your managed bean. The bean reference is simply more instructions for the FacesServlet to process. It is not the XPage that actually reads your bean or sets a bean value, it is the instruction of the reference to your bean that tells the FacesServlet to place bean data in the HTML and to store request data back into your bean. The next time you reference an invalid method in your bean take note of the Java Exception you receive. It is a FacesServletException. That is because the FacesServlet was attempting to access the referenced method from your XPage design, not the other way around. The XPage was only a set of instructions for the FacesServlet to process.

EmployeeBean: This is a MVC view level Java object that you will design to provide whatever information you would like display on or read from the web. As it is a view level object, no data processing business logic will exist here. There may be navigation business logic, but not data processing business logic. Beans here fall into various categories based on the type of work they do (provide/store data, control navigation, common page processing rules, etc…). But the primary goal in designing the bean will be only to work at the view layer and all data business processing requests will be delegated to the Model layer.

EmployeeCRUDFacade: In our model, I have implemented the Façade pattern to encapsulate the business logic of retrieving data from the system. There are several patterns that can be used here. I personally prefer the Façade pattern as it allows me to centralize my data business logic into re-usable logical units that can be easily accessed by other client code (beans act as the client to the Façade – notice the uses reference from the EmployeeBean to this Façade object, in the diagram). The "CRUD" portion of the object name stands for the standard database "Create", "Read", "Update", and "Delete" actions. The actual code of that work is not in the Façade. The Façade's job is to provide a client interface to a logically grouped set of instructions executed on other Java objects. In this example, the low level work and implementation of those instructions is delegated to the classes that make up the Data Access Objects (DAO Pattern).

DominoEmployeeDAO: To get at data a best practice is to implement some sort of data pattern. Given that Domino is not readily exposed for Entity Management (another Java conversation in itself), we will fall back on the long standing Data Access Object (DAO) pattern. This pattern provides an abstract interface to the database. It provides specific data operation capabilities without exposing or requiring clients to have detail knowledge of the database. This centralizes the responsibility of working with the data layer and provides the interface to external clients for obtaining/updating the data objects. In our example application, the EmployeeCRUDFacade uses the DominioEmployeeDAO to work with employee data. The DAO does the heavy lifting and the Façade makes simple calls to tell the DAO what it needs. The Façade then hands the result back to the MVC view object.

EmployeeDAO (interface): You will notice that the DominoEmployeeDAO implements an interface (EmployeeDAO). The DominoEmployeeDAO is the implementation level object that will contain specific instructions for working with an employee in a Domino database. But it is possible to have an application where employee data might also be stored in an Oracle database. In that case there would be an OracleEmplyeeDAO with specific instructions for dealing with the

Oracle database. The common practice in this case is to have a Java interface that defines common methods that each DAO must implement. This is so that client code (the Façade in this case) can use either DAO and always expect to be able to execute the same method between the two. The Façade should not care which database is used or how to talk to the database; its only job is to execute methods from the DAO that will work with an employee dataset.

Employee: This is our low level data. In Domino this comes from data saved with the employee form. Each field that you place on the form is represented by a Java field in this object. There is no real logic in this object. It is a simple set of fields with matching getters and setters. The DAO does all the heavy lifting to create this object and update/read from the object as processing occurs. The Façade obtains the object from the DAO and passes the object back to the DAO after fields are changed by the UI or other methodology.

DominoDB: This object contains database instructions that are not specific to any data object. For example, before we read a document we have to get a handle to the database. So the implementation of this DominoDB object will contain common instructions that all DAO implementations can use.

In this installment of "Using JSF Development Framework Standard for your XPages Project" we have covered the high level model design of a simple employee application. I have not yet provided hard coding solutions or concrete implementations of the classes, but instead provided the structure of our application as will exist in Java. In the next installment I am going to begin to implement the specific classes and working code that can interact with the Domino database for our employee application.

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)

May 29, 2013 [http://www.pipalia.co.uk/rethinking-xpages-part-two/] | by Gary Glickman | in jsf, mvc, xpages .

4 Comments

4 thoughts on “Using JSF Framework Development Standards for your XPages Project (Part Two: Designing Your Application Model for MVC)”

Mark Crosby
May 30, 2013 at 8:35 pm

So far so good, keep ‘em coming!
I’ve been developing Notes/Domino apps for 20 years, but am just getting into XPages now. It’s articles like these that I believe will help me “get it” and not just do the same thing with new tools.

Gary Glickman Post author
June 1, 2013 at 2:32 pm

Hi Mark,

Thank you for taking the time to read the post. It sounds like you go way back with Notes (R2.0 even?). I myself date back to R3.0 so I have also come a long way with this product. I was also lucky enough in the early part of the last decade to gain Java training and work on some very exciting pure JSF projects. So when XPages was designed, I was re-invigorated in terms of Notes & Domino development.

I hope you enjoy the remainder of the series as it comes out.

Thank you,
Gary

Cameron Gregor
June 3, 2013 at 12:04 am

It is great to see an XPages topic like this addressed from someone with pure JSF experience.

Looking forward to the next one!

Gary Glickman Post author
June 3, 2013 at 2:49 pm

Hi. Thank you for reading the article. I am glad you are enjoying the series.
