Forms to Struts 2, Spring and Hibernate: Value and "Gotcha's"

A NEOS and Vgo Software Whitepaper

Authors:

Robert Nocera and Jigar Parsana

Optimizing the Busniness-Technology Evolution

Forms to Struts 2, Spring and Hibernate: Value and "Gotcha's"

Robert Nocera, NEOS & Vgo Software
Jigar Parsana, NEOS & Vgo Software

Introduction

In today's corporate environments, many legacy applications still exist and are being used daily to perform critical business operations. The technology used for these applications ranges from mainframe to client/server applications using Oracle Forms or even PowerBuilder. The reasons for the reluctance to modernize these applications have been many, but most often the deciding factor is cost. If an application is working and supports the business, the business will not see much need to bear the expense of modernizing the application.

For the Information Technology departments supporting legacy applications, it is a different story. The IT folks deal with the day to day reality of these legacy applications, including maintaining the environment they run in, fixing bugs in the application, and implementing enhancements for the application. As people with the expertise in legacy applications and platforms leave the company, it becomes increasingly difficult for IT departments to maintain the legacy applications. These people are difficult to replace. This is one reason that companies are starting to modernize or at least consider modernizing their applications and platforms.

The other drivers forcing modernization are coming from the business side rather than the technology side. End users have greater expectations of the tools that they work with, especially if those tools run on their computers. Whereas several years ago, users interacted with high-end equipment and applications at their workplace and less so at home, the opposite is now true. In their personal lives, people use very powerful and intelligent systems that allow them to interact with computers in ways they can only hope to do at their jobs. Therefore, there is mounting pressure from the business user side of companies to enhance existing applications to have enhanced capabilities.

Client/Server technologies are still improving, version 12 of Oracle Forms is due for release sometime and PowerBuilder is still out there. In spite of the upgrades, the types of interactions that those applications are being asked to perform do not fit within the client/server model.

Today a lot of the new business systems are deployed as web applications due to the various benefits offered. Cost, maintenance, accessibility are just a few of the motivating factors. From a technology perspective, the Java ecosystem offers mature tools and technologies, like Struts 2 and Spring MVC, to build and maintain web applications. This paper outlines the use of Struts 2, Spring, and Hibernate as a target platform for the conversion of Oracle Forms applications. This target platform is then compared to Oracle's proprietary Java based offering, Application Development Framework (ADF).

With the acquisition of Sun by Oracle, the use of Java-based technologies and frameworks has become a viable alternative for Oracle customers looking to modernize Oracle Forms applications. Unlike Oracle ADF, Struts 2, Spring, and Hibernate provide a more Java-centric approach to application development, allowing a different level of control for IT departments.

Struts 2, Spring and Hibernate

Today there exist numerous tools, technologies and frameworks that were built using the Java programming language. Struts 2, Spring and Hibernate are some of the well-known names in the Java ecosystem. These technologies are mature and widely used by businesses around the world to develop applications. Best of all, these technologies come with open source licenses.

Struts 2 is a web application framework that enables the use of the Model, View, Controller (MVC) design pattern and allows for simplified development of web applications. The libraries included with the framework allow one to simultaneously build rich GUI's quickly and give programmers the flexibility to customize the layout, look and feel as they choose. The framework architecture automates a lot of tasks that traditionally cost the programmer extra time and effort. As a result, the programmer is able to focus on building user interfaces and implementing business rules and, ultimately, use configuration files to tie the two together. The architecture also provides components to define request processing workflow tasks that cut across the entire application. Struts 2 also supports additional customization using external plugins that can be integrated to provide specific functionality that may be desired but not shipped with the framework.

The Spring Framework is another application development framework that consists of specialized components that developers can use to create web applications. At its core, Spring provides an Inversion of Control (IOC) container, which can be used for managing resources and dependencies within an application. The Spring IOC container uses Dependency Injection (DI) to manage the life cycle of framework and application objects by using reflection. The motivation for having such a resource management service is driven by the gains achieved due to the resultant loose coupling between the objects. Having resource management is a well-accepted and standard component of web applications. In this paper, we will focus on Spring in terms of a resource manager and review its integration into the Struts 2 framework using the Struts 2 Spring Plug-in.

The third and final piece of the puzzle is Hibernate. Hibernate is a Java-based object relational mapping library that allows a framework to automate the persistence of Java objects to tables in relational databases. Hibernate uses XML-based meta data to describe the mapping between Java objects and database tables. This meta data is used to transform data from tables to objects and vice-versa. Using a solution like this in web applications improves developer productivity by eliminating the need to write boilerplate code. The code is more maintainable as most of the code related to handling persistence and data retrieval is not scattered all over the application code base. Finally, Hibernate gives the application the option to stay vendor neutral by allowing it to choose a dialect that should be used depending on the targeted database.

Case Study

To understand the solution architecture better, we present a summary of one of our recent modernization engagements at Vgo / NEOS. The client is an international car rental company with branches across North America and Europe. The client heavily relied on the company's Rental Car Management System (RCMS) to support sales and maintenance operations. The system was built using Oracle Forms technology and was approaching the end of its product life cycle, thus forcing the client to come up with a strategy to update the system. The challenge was to pick a strategy that allowed the client to update the system with room for redesigning certain parts as needed while still securing long-term support leveraging in-house resources, thus reducing external vendor dependency. The solution comprised developing an enterprise architecture that aligned with the overall IT strategy from a cost and maintainability perspective. In addition, the solution included using modernization tools to convert the Oracle Forms application to a web application that conformed to the enterprise architecture and leveraged Java-based technologies like Spring, Struts2 and Hibernate. The result was that the client got flexibility without overhauling existing resources.

From Oracle Forms to a Java-based Web Application

To modernize an Oracle Forms application, it is essential to have an understanding of Oracle Forms. Before delving into exactly how Forms are modernized to a Java-based framework, this section will give the reader an overview of how Oracle Forms applications are built. Figure 1 outlines the basic building blocks of a Forms application.

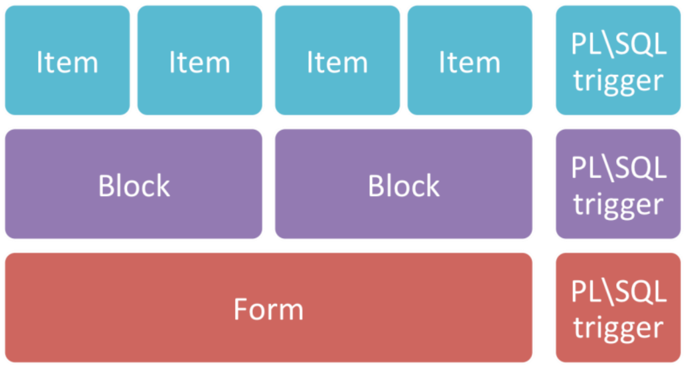

Figure 1. Form building blocks

The basic "building blocks" of Oracle Forms are Blocks (the pun was intended). A Block is a Forms object that is typically bound to a database object, usually a table, but sometimes a view or a query, or even a set of stored procedures. A Block consists of Items, which are based on columns in a table, calculated values or fields to hold transient data input by a user.

A Block and its Items contain various properties that both define the data within the Item or Block and how it gets displayed to the user, including which Canvas or Window it gets displayed on. Since Forms is a 4GL, it provides a lot of built-in functionality. By creating a Block based on a Table and dragging and dropping Items onto a layout, a developer can create a simple CRUD screen. Forms will provide all of the necessary plumbing to do search, inserts, updates, and deletes on the data associated with the underlying table.

A Form, Block, or Item can have an event associated with it; that event is called a Trigger. A Trigger is written in PL/SQL, and it can perform any sort of business logic or affect the UI in some way, perhaps changing the color of an input to indicate that it is an invalid value. Triggers can also be associated with Items that are of a “Button” type. These types of Triggers sometimes call more involved business logic, a Stored Procedure on the database, a Program Unit in the Form (another piece of PL/SQL code, typically more involved than just an event), or a Procedure or Function in an associated PL/SQL Library.

There are various types of Triggers that can be implemented in a Form and can be executed at various times during use of the application. It is beyond the scope of this paper to discuss them all, but some important ones include: WHEN-VALIDATE-ITEM (an event fired when the user leaves an editable Item field); PRE-QUERY and POST-QUERY (events fired before and after bound block data is retrieved, respectively); and ENTER-QUERY (an event that sets up the page for searching data).

Solution Architecture

The solution leveraged the Model-View-Controller design pattern, which is commonly used by most web applications. This will form the basis for an architecture that is separated by the concerns of Business Logic (Model), User interface (View) and Flow of Control (Controller) thereby making it easier to maintain. Figure 2 provides a high level view of the architecture and control flow for how a typical use case request is processed.

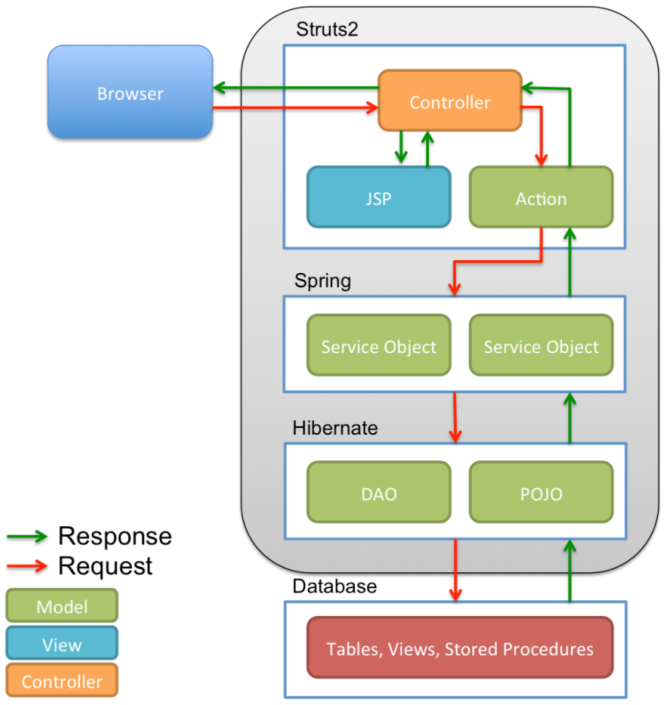

Figure 2. Solution Architecture

Looking at the solution from a Forms application perspective, the various components (building blocks) of a Forms application do not *directly* map to the solution architecture components. This is due to the fundamental differences between the Architectures and their resultant concerns. The traditional client server model breaks the application into two simple concerns: 1) the database; and 2) the client accessing the database. This model results in the client doing a lot of heavy lifting as it contains the components for UI and Business Logic. With the evolution in technologies and advent of Internet and Internet-based applications, the burden on the client has gotten lighter, and heaviness has been moved to a middle tier that is responsible for interacting with one or more data sources on one end and multiple clients on the other end. Furthermore, by using the MVC pattern in the middle tier, it is possible to further divide the components into three major concerns, which makes the architecture more flexible and maintainable. The flexibility gives the IT departments more control over each concern and the ability to make changes to one and minimally affect the other.

The perspective to take for understanding how Forms components map to a Java web application that uses MVC is that of one where different parts of an individual form's component, say a block or an item, are disintegrated and placed/mapped to one or more concerns of the MVC. For example, an Item from a forms perspective is a single unit that could contain field UI information and validation logic. However from an MVC perspective, the UI information belongs to the View piece and validation logic resides within the Model piece. Figure 3 gives a simplified view of the mapping between Forms and MVC concerns.

Forms Client	Model			View	Controller
	Action	Service	Entity Pojo	JSP / TagLibs / CSS	Struts XML Config
Form		✔			✔
Block	✔	✔	✔		
Canvas				✔	
Item		✔	✔	✔	

Figure 3. Forms to MVC mapping

As we look over the various entities that make up each of the concerns, we can start to see how this architecture can be leveraged to house common functionality that would otherwise need to be redundantly developed. It is this common functionality that will become the 'plumbing' that allows programmers to focus on coding the business logic and UI functionality, but at the same time, provide extensibility.

The Action class serves as the entry point to the model. This class contains the reference to the domain entity class, which serves as the object on which the CRUD operations ultimately occur. As we are using Spring's IOC container, we inject the Service interface implemented by the Service class that contains the business logic into the Action class using a setter injection. The Action class contains methods that handle common actions or events common to your application various screens. Some examples are Search or List, Save, and Validation. Since the Service containing the business logic is injected at runtime, the Action can call the relevant service methods based on the event it is processing (search, save, etc.).

Just as the Action class contains a reference to the service class, the DAO can be injected into the Service class at runtime. One could expose the DAO directly or expose the DAO methods by writing pass through methods in the Service class that call the DAO methods. It is a design choice; however, in our experience, the latter gives the flexibility to package multiple operations: e.g.: Pre-Update, Update and Post-Update within one transaction.

In summary we now have an Action class that contains the references to the persistent Entity and the Service, which holds the business logic along with the methods that expose CRUD, Pre and Post CRUD functionality. All Table based blocks converted will contain Action and Service classes that extend these 'base' classes holding the common functionality. This makes the architecture complete and flexible. Complete in the sense that basic CRUD operations can be supported with very minimal coding effort, and flexible in the sense that it can be extended to implement additional functionality that is unique to a form.

Our solution architecture requires that for each Table based block we have an Action and Service class because each action acts on one persistent entity. Hence, from a controller perspective each Form that is being converted should contain an individual configuration file. This file would map all the Action classes that together support the form's events or operations with the various views components. The View components are made up of JSP pages that utilize Struts2 tag libraries, JQuery and JavaScript snippets and CSS for styling. Using a JavaScript library like JQuery allows programmers to develop screens with dynamic behavior that might be custom to each screen. We found most of the JQuery libs useful for doing AJAX related heavy lifting.

Challenges and Lessons Learned

Most of the technical challenges we faced were encountered once the conversion process had started. One of the challenges we ran into was that of using Annotation based constraints on shared Entity POJO's. We leveraged Java and Hibernate's validation constraints to do data validation on Entity POJOs. This worked well until the number of converted forms started to increase, and we found that sometimes there were different forms that acted on the same table. For instance, a scenario like this translated to the POJO fields being required on one screen but not on the other. The solution to the problem was being able to create a custom constraint annotation where we could specify the Action class for which the validation was applicable. Since the Action class is where the validation occurred, with a property on the constraint indicating the action to which this validation is applicable, we now had a conditional mechanism in place for when to check for constraints.

Another challenge we discovered early on was the need to be able retrieve data efficiently. When it comes to data retrieval in a Forms application, the inbuilt infrastructure takes care of retrieving it in a manner that doesn't overload the client, be it LOV or records being displayed on the block. Our initial approach to converting LOV's entailed dropdown fields. We quickly discovered that some LOV's queries were fetching thousands of records, which translated to slower page loading times as the browser waited until all the data was fetched. As a result, we introduced a server side paging mechanism to our Model and from that point forward any data retrieval operation had the ability to leverage server side paging. On the view end we developed a finder component that would be used instead of dropdowns where the number of records retrieved was beyond 50 or 100. The finder is a popup that allows the user to specify search criteria and, once retrieved, allows the user to select a value that would be set a value for finder field.

From a project perspective, there were some non-technical challenges that were also encountered. As a service provider, we went against our better judgment and offered to do the first set of work without first creating a proof of concept (POC) and reference architecture. Since the client had presumably been working on this project already and had a multitude of documents describing their desired architecture, after some review (but not a code review) we allowed the project to proceed assuming that the major pieces of the architecture were in place. What we found as the project progressed was that though the major components had been chosen (Struts 2, Spring, and Hibernate) and there were plenty of standards in place, there was not a design in place to support the basics of an Oracle Forms application.

In addition to that core functionality, as in any conversion project it is critical to communicate with the business customer the decisions that need to be made regarding the functionality of the modernized application. It typically is not feasible to maintain the exact workflow that the original application had, as screens are sometimes split up to provide the same functionality with better usability. Keyboard shortcuts are probably not going to be as prominent in a web-based application as they were in a character-based application. Transaction handling needs to be decided, transactions are normally shorter in a web application than in a client/server application. There are other details that also need to be decided upon.

While the big picture did not change, our team spent a good deal of time filling in the blanks in the architecture and design and creating a design that could support a lot of the normal forms functionality out of the box. This was time and money that was not originally planned, since assurances were made that these were already in place.

ADF Comparison

As we mentioned, ADF is an Oracle proprietary framework. Contrary to popular belief, however, it is also Open Source and therefore somewhat free. It is not “Free as in Beer”, nor is it really “Free as in Speech” either, but if you are a licensed customer you can get the source code if you want it or need it.

The ADF stack of technology includes ADF Rich Faces (Rich Faces) and ADF Business Components (ADF BC) and JDeveloper. Though JDeveloper is not strictly a requirement, it is currently the best IDE to build ADF applications in and the wizards provided by JDeveloper are part of what make using ADF an attractive alternative to other Java frameworks, including the Struts 2 / Spring / Hibernate trifecta discussed in this paper.

ADF was built by Oracle with Forms developers in mind. The building blocks within ADF are eerily similar to the building blocks in Forms (again, no pun intended). Most likely a complete ADF application will encompass the functionality of many Oracle Forms. Each individual Form, however, maps well to a Task Flow in ADF.

A Task Flow is essentially a work flow, or piece of a work flow. It maps out screens (JSP pages or fragments) and functions and shows how they interact with each other. A Task Flow can be extremely simple consisting of only one or two screens, or very complex, consisting of many with even embedded calls to other Task Flows. Also, much like a typical Form, a bounded Task Flow encompasses a transaction allowing the user to commit or rollback at the end of the work flow.

The functionality of the main components in Forms, the Blocks, is split into a few components in ADF. Since the View and the Model are separated in ADF but not in Forms, this is expected as it is in any other typical MVC framework. In ADF, the component that talks to an updateable object database is an Entity. A developer will create an ADF View component that uses the Entity but may contain more or less fields; it is the View component that supports the presentation on the page. In our S/S/H implementation, this is analogous to the Hibernate entity object (hbm and POJO) and the Hibernate GridModel object (hbm and POJO).

The functionality of the Block Items themselves is also represented in more than one layer. The binding to the database occurs in the Entity object, with a View object used for presentation. Validation on that attribute can go in either the Entity or the View. To display the information to the user or allow input from a user, a control bound to a View attribute is used in a JSP page or fragment. The input control may also have validation associated with it.

Again, this is similar to the S/S/H implementation. The main difference is that in ADF BC the View component is normally described by XML whereas the POJO (the analogous component in S/S/H) is written in Java. For ADF BC, a Java implementation is only necessary if some sort of complex validation or calculation is being done.

The pages in the S/S/H implementation require some amount of JavaScript to provide the AJAX functionality needed by most customers these days. ADF Rich Faces contains AJAX enabled components and an architecture that supports these components so that many times, no additional work is required to get the desired functionality. It is assumed that when something changes in the ADF Model, the new values should be displayed to the user and a lot of this is done without any developer work at all. There are cases where the developer will need to fine tune some of the AJAX interaction, but almost all of it can be done by adjusting properties of the components and without any coding.

When it comes to implementing complex business logic, it is done much the same way in both frameworks, only the details differ. If implementing a POST_QUERY trigger in ADF requires modification of the fetching SQL, it can be done in the ADF View component, much the same way that it would be done in the Hibernate HBM file for the corresponding object. The same is true for business logic that needs to be written in Java, it will probably need to be written in Java in both frameworks. It is possible, however, that some of the less complex validation can be done in ADF in Groovy instead of Java eliminated some actual Java coding.

Conclusion of Comparison

Overall, ADF does provide more functionality out of the box that is very useful when modernizing a Forms application. To get to the same point in our S/S/H architecture required some months of work and fine-tuning, and it still requires more manual coding than ADF does. We supplement some of this with code generation to try to make up the gap. The functionality that ADF provides does come with a price, however. In addition to the licensing cost, it is still easier to find developers with Struts, Spring, and Hibernate knowledge than to find developers with ADF knowledge. It also takes a very good understanding of ADF to customize it as much as some customers would like. Though it probably takes a good understanding of the S/S/H architecture to do the same, that understanding is easier to come by as many people have been using these technologies for years and writing about them in blogs, books and more.

We have found that both architectures are suitable for the job. If the IT shop supporting the application has a plethora of Java experience in-house, they may be inclined to towards the open-source solution for modernizing their client/server applications. If the IT department is primarily an Oracle shop, they may choose the ADF solution. Either is fine as long as they understand the benefits and risks involved in solution they choose.

About the Authors

Robert Nocera is the CTO and co-founder of NEOS and Vgo Software. He is tasked with driving technology direction for the company's modernization solutions but has also developed a focused understanding of Oracle Fusion Middleware and ADF v11 in particular. Robert has spoken at ODTUG (2008), UKOUG (2010) and a number of regional OUGs (NYOUG, NEOUG, and more). He also has spoken on ADF and ADF security at the 2009 OOW "unconference." Robert is deeply skilled in all Java related technologies/languages. He has over twenty years of IT experience and is the author of the www.java-hair.com blog.

Jigar Parsana has been developing and implementing Web applications using Java technologies for over a decade. As a software architect, he is responsible for leading the technical aspects of modernization projects at NEOS, where he leverages that experience to modernize clients' Oracle Forms, PowerBuilder, and .Net-based systems to Web and SOA-based application architectures. Jigar combines theoretical knowledge with practical, real-world, development experience to design and deliver world-class JEE applications. Jigar holds an MS in computer science from Rensselaer Polytechnic Institute.

About NEOS and Vgo Software

NEOS is a management consulting and technology services company, offering business system modernization and enterprise data services capabilities. NEOS partners with its clients to best position their business by leveraging our collective experience, capabilities and proprietary products, across all industries and business functions.

NEOS was founded in Connecticut in 2000 and has expanded its business throughout the US, into Europe, the Middle East and Japan. NEOS clients range from large, mid-cap companies to the Global 1000 with specific industry expertise in financial services and insurance. Vgo Software, founded in 2005, was nominated as one of Connecticut's "Fast Track" companies and has enjoyed positive growth since its beginnings.

Vgo Software provides modernization solutions and professional services that support legacy application strategies, assessments, conversions, and development. Vgo utilizes proven proprietary methodologies and software products including Evo, (Oracle Forms® to Java); ART™, (Application Portfolio Assessment for Oracle

Forms® and PowerBuilder Applications); and Evolutions™ (Application Conversion Methodology). Vgo is one of only two certified third-party solutions conducting Oracle Forms® to Java modernization.

Vgo Software was created based on intellectual property developed through many professional services engagements conducted by its parent company, **NEOS LLC**. Our products and offerings are based on real-world experience from hands-on projects delivered to clients.

Vgo Software's product lines will continue to expand in functionality as well as grow in scale. It is our intent to continue to provide superior, thoughtful and innovative automation solutions to customers world-wide.

Contact

Please contact **NEOS and Vgo Software at +1-860-519-5601** for more information or visit us at www.neosllc.com and www.vgosoftware.com.