Best Practices Guide

Written By: David E. Jones, jonesde@apache.org

If you look for contributor best practices here you go: OFBiz Contributors Best Practices

Table of Contents

See Also

- HTML and CSS Best Practices
- Managing Your Source Differences
- Methodology Recommendations
- User Interface Layout Best Practices

Introduction

This documents presents best-practices for development and architecture related to The Open For Business Project.

It will not try to cover all of the practically limitless options available for using tools in OFBiz and the many related open source projects and standard APIs. It will also not try to cover all development best-practices, just the ones most closely related to creating and modifying OFBiz based components.

It will try to cover the best practices for every layer of architecture and the best tools to use, and for steps that should be taken prior to even laying out what code should be written. In some cases there will be close seconds to the best practices that will be presented because in some cases these second-best-practices will be more appropriate.

Note that this document assumes some knowledge of the OFBiz Core Framework. With some basic knowledge it can answer a lot of questions about how certain things should be used. If you are not familiar with the OFBiz Core Framework this will give you an introduction to it, but some things may not make a lot of sense to you.

General Concepts

Reduce Code Complexity, Redundancy and Size

There are various techniques that can be applied to this problem that produce good results. The most common, but least effective, practice is code generation. Because it is not something that I consider to be a best practice, I won't discuss it further here.

The best practices that are used in various places in the OFBiz framework are dynamic APIs such as the Entity and Service Engines, and special purpose languages like the Workflow and Rule Engines and the MiniLang library.

The dynamic API pattern is characterized by a generic API with simple operations that behave differently based on configuration and domain definition files. These are usually XML files. This is an alternative to code generation and the input or domain description files used for generating code can often be used unchanged to drive a dynamic API resulting in much less code and much more dynamic, ad-hoc control.

A special purpose language is used to create logic in a language or using a tool that fits the specific need more appropriately than a general purpose procedural language like Java. This reduces code because it is easier to describe what you want in a context that closely matches the problem that needs to be solved than it is to use a generic language. High level conepts can be expressed without the need to write a lot of code.

Generic Versus Special Purpose Artifacts

For OFBiz "out-of-the-box" (OOTB) the point for the generic artifacts (in all tiers of the application) is to be as inclusive as possible. There is one reason for this: it is easier to know something is there and decide you don't want it and then to remove it than it is to not know that it exists and either try to find it or end up implementing it anew.

How do you decide which fields you want on a screen? It depends on the organization, and even more so on the role within the organization.

There is no one answer for all users of OFBiz. There isn't even an answer for everyone in a single organization (unless it is REALLY small).

This has been thought through and the OFBiz framework was designed very specifically to handle this sort of thing. It is relatively easy to create user interfaces that are specific to certain roles within an organization, and that is what OFBiz customization or creation of derivative works or creation of special purpose user interfaces is all about. That's the very definition of it.

In all part of the framework the best practice is to first create a generic or general purpose artifact if it does not already exist. For anything that is specific to a certain role within an organization or that is for a special purpose the best practice is to derive the artifact from the generic one with as little redundant code as possible using the many override mechanisms that exist in OFBiz. For the most part this can be done without changing ANY of the base code from the open source project, making it easier to maintain over time.

Data Layer

The best-practice tool to use in the data layer is the OFBiz Entity Engine. For most applications the Entity Engine will elegantly do the work for 99% of your database interaction needs. In the few cases where the Entity Engine is not sufficient I recommend using custom JDBC code for your queries or other commands. That would be one of the second-best-practices that are sometimes needed.

When using the Entity Engine refer to entity and field names using inline strings. This makes it much easier to read and maintain your code. If you need to prepare a large Map or EntityCondition to pass to an EE method it is generally cleaner to do it on a separate line, or on various separate lines before the actual EE method call.

Use a simple, normalized data model based on the needs of your applications. Usually the data model can be driven directly by the requirements for the functionality that will use the entities. This usually results in a highly normalized data model which will make your life much easier. When you need combined data for reporting, use the view-entity feature to accomplish joins and grouping and summarizing data.

Always use primary keys and avoid the use of generic sequenced primary keys when a more descriptive composite key is possible. Always use relationship definitions to document how entities are used together, to make it easier to get at related data, to constrain field by foreign keys, and to improve performance through automatic foreign key based indexes.

Logic Layer

The best tool to use for invoking logic is the OFBiz Service Engine. Nearly all business logic should be implemented as a service to improve reusability and facilitate component based development.

Even though services are very flexible there are cases where the service model is not appropriate, even for business logic. In some cases calling a script or Java method directly is necessary and in those cases using the service model would not make sense and should not be used.

The Service Engine reduces code size by providing many ways to use logic implemented as a service. You can call your logic synchronously, asynchronously or on a schedule. When you are calling a service you don't need to know where it is located or how it is implemented. This makes it easy to effectively leverage remote services and services written using different languages. Being able to transparently call logic in different languages through the Service Engine is important for the effective use of special purpose languages.

When defining your services try to keep them as simple as possible using interface services or other techniques. Whenever a service is based on an entity be sure to use the auto-attributes tag to create attributes from the entity field definitions. Also, rather than redefining attributes using the attribute tag, use the override tag and only specify the information you want to change.

Always implement your service using the easiest and most appropriate language or tool. You can implement services with many different languages including Java, Groovy, Beanshell or any BSF compliant scripting language (Jython, Jacl, JavaScript, etc), OFBiz Workflow Engine processes, OFBiz MiniLang simple-methods, and various others. Additional languages can be supported by writing simple adapters.

Most services are oriented around data mapping and handling and the simple-method is the best way to implement them. Writing a service with a simple-method is the primary best practice for writing a service. There are some cases where simple-method scripts are too restrictive or cumbersome, and in those cases a more general purpose language should be used. The two recommended secondary best practice tools for this case are Groovy and Java, with Groovy being preferred because of flexibility/extensibility features, no recompilation needed to test changes, and various other benefits.

Always call remote logic through the Service Engine. You can call remote services through various mechanisms including HTTP, SOAP, JMS, and others. You can also add remote service invocation mechanisms by creating a simple adapter.

Most of the time you will want to let the Service Engine automatically wrap your service call in a transaciton so that the whole thing will succeed, or the whole thing will fail. Note that if you call a service inside an existing transaction it will recognize the current transaction and use it instead of trying to create another one.

Presentation Layer

Always separate input processing logic, view data preparation logic, and view presentation templates. This will make it easy to reuse logic not only in web applications, but also for independent fat client applications. It will also make it easier to organize your code and find a specific piece of functionality when debugging or exploring to find out how a component works.

For each of the three separate pieces there are special best-practices tools to use.

Input Processing Logic

Input processing logic should always be associated with a request in the controller.xml file and never with a view. Input processing logic should generally be implemented as a service and called through the service event handler which will automatically pull data from request parameters or attributes and convert it from a string to the object type defined in the service definition. This makes it easy to specify which parameters you care about processing just using the service definition, and let the framework get them ready for you.

There are various cases where input processing logic cannot be implemented as a service. There are various other types of event handlers for logic associated with requests that give you more to the request context and are not environment agnostic like services are. One example is receiving uploaded data. Another good example is doing special pre-processing and validation on parameters before passing them to a service for processing. Note that you can always call services from these custom events and wherever possible generic logic should be implemented in services.

Always let the Control Servlet configuration handle decisions about the appropriate response to take for a request given the result string from an event. In most cases the response will be the generation of a view, but sometimes it will make sense to chain requests together to achieve logic reuse or more advanced flow control.

View Data Preparation Logic

View data preparation logic should always be associated with the view template it is meant to prepare data for. This should be done through the OFBiz Screen Widget in the screen definition XML file by specifying a script action. When a screen is split up into multiple templates or screens the data preparation action should be associated only with the individual small screen that it prepares data for. This makes it easier to move templates and content pieces around and reuse them in many places.

View data preparation logic should be specified as actions in the XML screen definition or if necessary implemented in a dynamic scripting language such as Groovy, JavaScript, BeanShell, Jython or Jacl to make it easy to modify the user interface on the fly. Generic data retrieval should be implemented as services which should be called from these dynamic action scripts. This makes it easier to share and reuse this functionality in multiple pages and in other types of user interfaces.

The best practice in OFBiz for data preparation is to use one of these tools:

1. for simple data retrieval: the entity* tags in the screen action section
2. for more complex data preparation and manipulation: simple-method or Groovy
3. for larger data preparation and more reusability: implement a service and call in a screen action tag

When preparing data in view actions you should make the data available to the view template by putting it in the "context" object. All attributes in the context object will be made available in the context of the template, if the template language supports that.

While the use of JSP is not recommended in OFBiz, it is supported. Note that when using JSP for a view template you cannot use the Screen Widget so the actions facility will not be available. Our recommendation for JSPs is to have a single scriptlet at the top of the page that prepares the data. In this case try to call worker services or worker Java methods to do most of the work and to keep as much logic as possible out of the page.

View Presentation Templates

The best-practice template engine that we recommend for HTML and other text generation is FreeMarker. It is like Velocity from Jakarta, but much more flexible and fits in nicely with other OFBiz Core Framework tools. Rather than running FreeMarker templates directly we strongly recommend using the OFBiz Screen Widget so that actions can be associated with screens and they can be decorated with common templates. We'll describe how to best use this below.

The view presentation templates should always be kept as simple as possible and common content such as headers, footers, sidebars, and so on should be added at run-time using the decoration pattern. The template file that should be used to decorate each page is specified in the screen definition XML file.

Always use the view generation tool that most closely matches your needs. FreeMarker is the recommended tool for generating text output, but there are many situations where other tools are more appropriate. If you want to use other text generation tools such as Velocity or XSLT we recommend you do so through the Screen Widget, especially if you want it to be decorated and have actions run to prepare for the templates.

When you want to display report like views we recommend using the Screen Widget along with FreeMarker to generate an XSL:FO XML file, and use the Screen Widget FOP view handler in the Control Servlet configuration to transform it into a PDF to send to the client. This approach allows you to use the same tools for these reports as with other views in your applications and offers essentially unlimited flexibility. As an alternative you can use other more traditional reporting tools such as Jasper Reports or DataVision and mounting those reports through a view-map in the Control Servlet configuration file (controller.xml).

If you have UI patterns that are repeated frequently such as forms, query data displays, tab or menu bars, expanding tree views, and so forth we currently recommend using an XML file to describe the UI pattern and then transform it to HTML or other output using a rendering engine for that specific XML format, or using XSLT/FreeMarker/etc to transform it into the desired output.

For most forms the best way to represent them is to use the OFBiz Form Widget which accepts a generic form definition in XML that is usable with multiple user interface types, and that takes advantage of existing assets in your OFBiz based code such as Service and Entity definitions. This results in code that is much smaller and easier to maintain.

When using FreeMarker is not possible or practical we recommend using another dynamic templating language such as Velocity. When that is also not possible or practical we recommend using JSPs. But, note that when using JSPs you cannot take advantage of the actions or decoration templates because you cannot run it through JPublish. This is thanks to limitations in the JSP specification. Even through you cannot use the decorator patter, you can use the composite view pattern with the OFBiz Regions framework. Regions are specified in the regions.xml file. Note that these are not as easy to use as Screen Widget composite views, and they do not support actions. But the Regions framework does offer a lot of flexibility and is very useful in many cases.