Apex coding patterns

This page provides a brief overview of how we expect our Apex to be structured. As much as possible all apex development should follow these guidelines. The patterns here help with some common problems encountered when developing with Apex on the Salesforce platform. Equally by following these we help to maintain a consistent approach which should make it easier for another developer now or in future to quickly follow the flow of any feature and understand what is going on. Obviously that will be mitigated by a legacy code, which will likely want to be migrated over at some point.

At times it may be that we have to go outside these suggestions, but where that is the case please discuss with a Senior developer. Some of the patterns here are not really Apex patterns as such, but much more general patterns that have long been considered good practise for software engineering in general.

A link to a slide deck used as part of presentation on some of this stuff (with a few other bits) Basic Apex Patterns.pptx - mainly bullet points, but has a few skeleton example code snippets.

High level overview

Layers

For almost as long as people have been coding it has been considered good practise to divide code into modules and layers, not just conceptually but also technically. One of the most common concept here is that UI (presentation) specific code should be kept apart from the business logic, and that the business logic should be kept separate from the database logic. These layers go a good way to making the code easier to follow, test and maintain. UI behaviour can be changed far easier without impacting business logic, database models can be more readily changed without impacting business logic etc.

Presentation layer

Any Apex that is specific to a UI (e.g. controllers) should have no code other than UI specific logic, e.g. controlling panel visibility, toggle state, navigation state or http parameter handling etc. Anything else should be part of a callout to some service.

However, it is worth bearing in mind that from a layering point of view it is not only UI logic that is found at this layer. Global entry points (effectively our visible API), Triggers, Batch/Queueable classes, WebServices, Invocable methods and others are all part of this ‘presentation’ layer. Presentation in this respect does not mean a UI presentation, it means anywhere that ‘presents’ a way into our coded logic.

Batches, for example, are part of this top layer. A batch should do nothing that is not related to running over a bulk set of records. The actual business logic being performed should be in some service. This service might be the same service that a UI calls, or a trigger handler calls etc.

Triggers also fall into this category. They are often the point at which our Apex code is first invoked by some external actor. A trigger should use some from of generic interfaces etc to move away from the trigger itself and into code that can control what happens. For the most part the trigger won’t have any other code in it.

See Entry Points. Triggers, Batches, Controllers etc. for some more discussion on these.

Service Layer

All business logic should be in its own set of classes, accessed by some service or domain style class. These classes should have no knowledge or dependency on what called them. These classes could be doing a whole range of stuff. They may do simple things like setting default values on fields or validating records. Equally they may do complex things like create an invoice by pulling in data from multiple other records.

It is important that the interfaces to these classes are not based on a specific caller, in particular no instances of classes from the presentation layer should be passed to them; for example no controllers nor controller state, no BatchableContexts etc. Just as important is that the method signatures should not mirror some specific caller type, e.g. services should not take a list of old and new records just because it is called by a trigger. The service should take only what is needed to perform its work, if a trigger needs to call into it then a trigger handler should only pass the relevant collection and not just pass in both sets of old and new.

A class at this layer should not be accessing the database directly. This means that all SOQL and DML should be done via another set of classes that implement some known interface.

Database Layer

All database access should be done from a set of classes specific to accessing the database. Both reads and writes should be avoided in any other code, except for a couple of places where Salesforce make that more or less impossible.

In Apex terms this layer can be broken down into 2 distinct areas, SOQL and DML, and there will usually be distinct classes to do each. The DML aspect is usually very generic and hence likely to be done by just one common class that everything else can rely on. The needs of reading from the database are more specific to each service and therefore each service will normally define some Selector interface that it requires. Though that is not the only pattern that works, it is probably all that we need for the most part.

The primary benefits of this pattern are easier security handling, easier database changes etc. However, one of the more important benefits is hugely improved testing ability. When services call into some interface to read or write to the database we gain the ability to mock out the database and hence most of the pain of setting up data for tests disappear. Not only are tests easier to write they run a lot faster.