Asynchronous Processing in Force.com

Asynchronous Processing in Force.com

Table of Contents

ASYNCHRONOUS PROCESSING IN FORCE.COM

Introduction

Who Should Read This

This paper is for experienced technical architects who work with Salesforce deployments who want to have a better understanding of Force.com asynchronous processing. This will help an architect build effective design patterns around asynchronous processing.

Asynchronous Overview

An asynchronous process is a process or function which does not require interaction with a user. It can be used to execute a task "in the background" without the user having to wait for the task to finish. Force.com features such as Asynchronous Apex (`@future`), Batch Apex, Bulk API, Reports and other features use asynchronous processing to efficiently process requests.

Asynchronous processing provides a number of benefits including:

- *User Efficiency* – Making a user wait for a long running process is not an efficient use of the user's time. It can be done in the background and the user can see the results at their convenience.
- *Resource Efficiency* – Each Salesforce instance has finite set of resources. Under normal load patterns, these resources can be managed for asynchronous processes which results typically results in faster request processing.
- *Scalability* – By allowing some features of the Force.com to execute asynchronously, resources can be managed and scaled quickly. This allows the Salesforce instance to handle more customer jobs using parallel processing.

There are over 200 different types of asynchronous requests on Force.com, some are user initiated and some are internal housekeeping functions. Force.com asynchronous processing makes a best effort to complete requests as quickly as possible, however there is no guarantee on wait or processing time

What's in This Paper

The section Best Practices on page 7 lays out techniques to design better asynchronous processes on the platform. The section Underlying Concepts on page 1 provides details of Salesforce mechanisms and implementation that affect asynchronous processing in non-obvious ways.

Underlying Concepts

Underlying Concepts

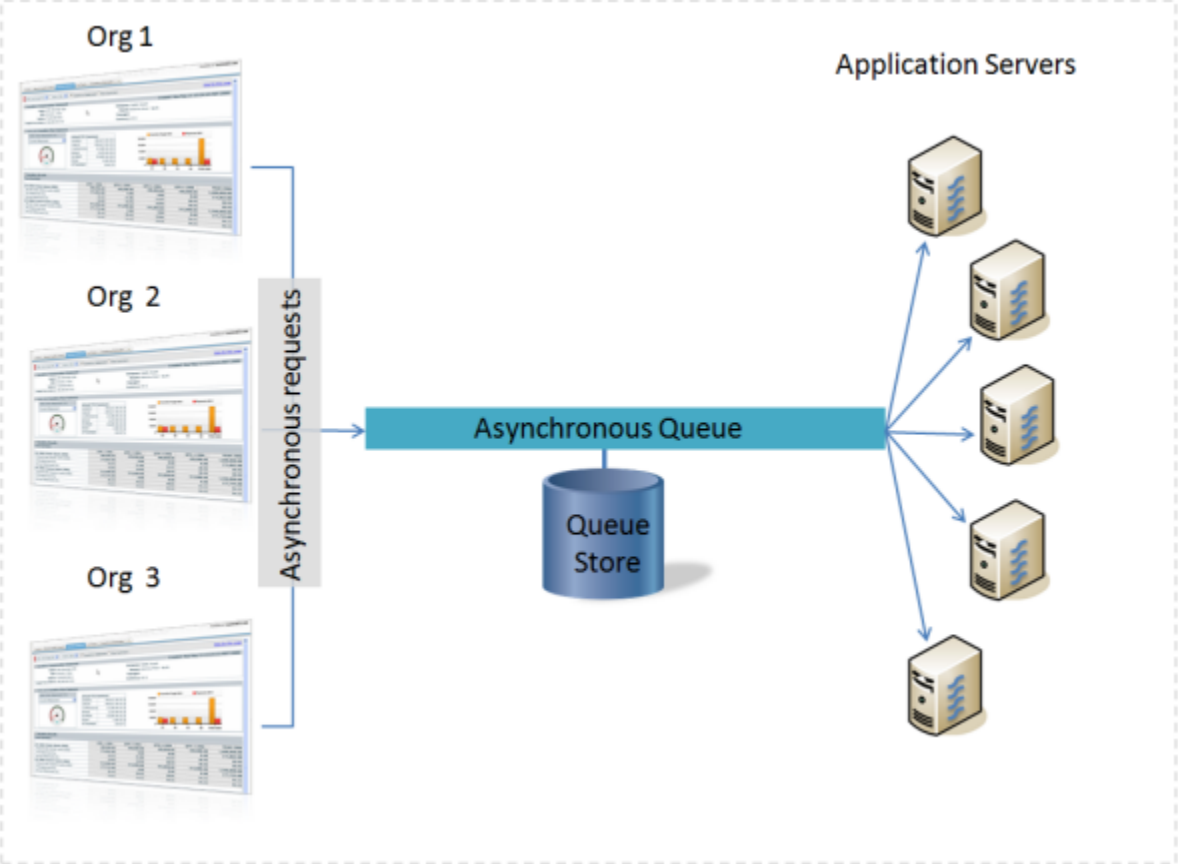
The material in this section provides a background for understanding the best practices in this paper. It will help you adapt these practices to your own situation.

Asynchronous Processing Overview

Asynchronous processing, in a multi-tenant environment, presents some challenges that need to be handled:

- Ensure fairness of processing – Make sure every customer gets a fair chance at processing resources.
- Ensure transactional capabilities – Ensure equipment or software failures allow continued processing at reduced capacity and requests are persisted until completion.

The following diagram provides a high level overview of Force.com's asynchronous processing technology:

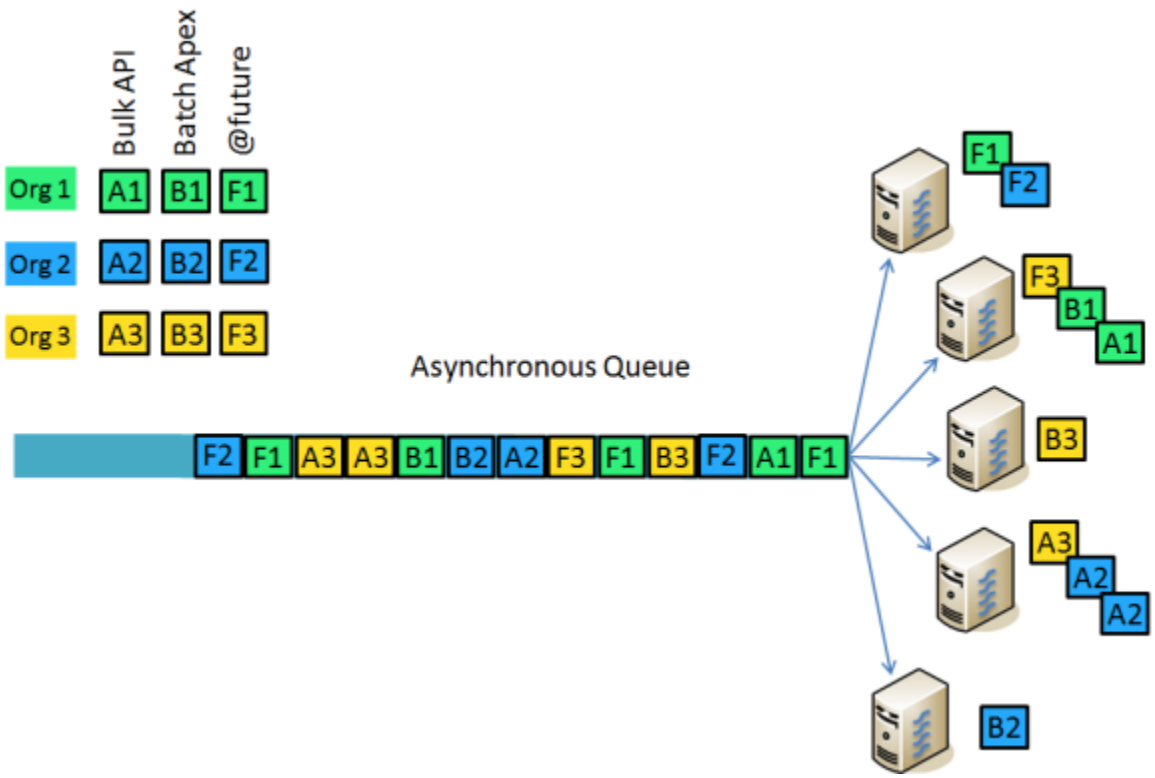


Salesforce.com uses a queue-based asynchronous processing framework. This framework is used to manage asynchronous requests within each instance. The request lifecycle is made up of three parts:

1. Enqueue – This is the act of putting asynchronous requests in the queue. This could be an Apex batch request, `@future` request or one of many others. The Salesforce application or a custom application will enqueue requests along with the appropriate data to process that request.
2. Persistence – The requests are stored in persistent storage for failure recovery and to provide transactional capabilities.
3. Dequeue – This is the act of taking requests off the queue and processing them. Transaction management occurs in this step to assure messages are not lost if there is a processing failure.

Each request is processed by a *handler*. The handler is the code that performs functions for a specific request type.

The handlers are executed by *worker threads* on each of the application servers that make up an instance. Each application server supports a finite amount of threads. Each thread can execute most any type of handler and the handler determines how many threads are available to process requests for its given request type. The threads request work from the queuing framework and when received starts a specific handler to do the work. The following diagram shows the asynchronous processing in action:



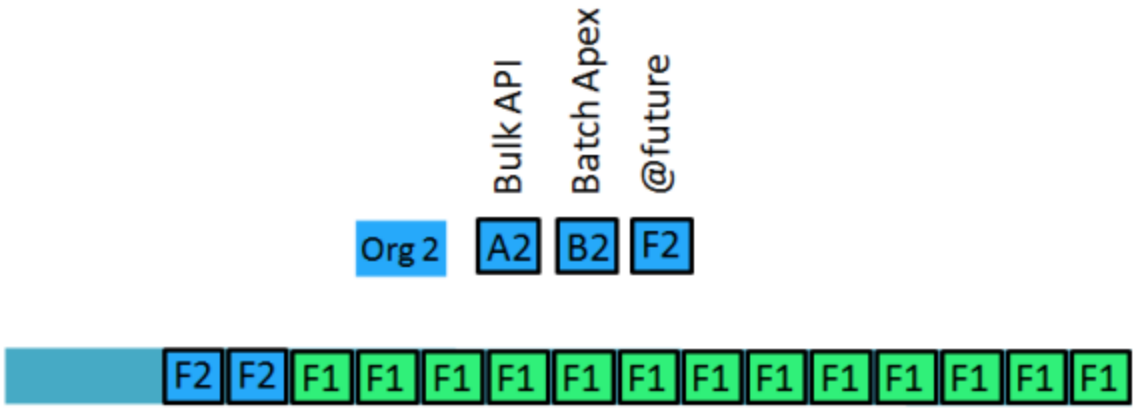
Typically requests from different organizations for different functions will be in the queue. As one request is completed, another request is removed from the queue and processed. Error handling and failure recovery is built in so the requests are not lost if a queue failure or handler failure occurs.

In the scenario where one organization adds a large number of requests to the queue, it could prevent other customers from getting access to the worker threads. The queuing framework implements *flow control* which prevents a single customer from using all of the available threads.

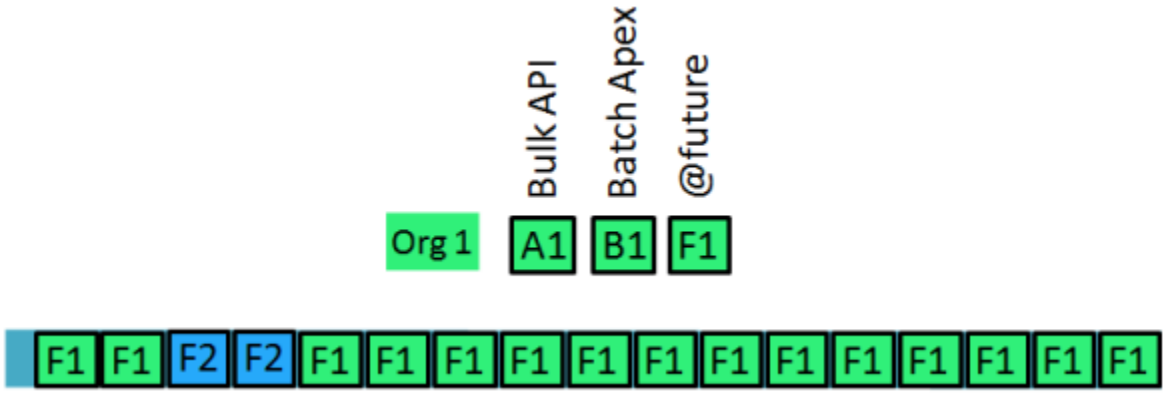
When a worker thread is available to process a request, the queuing framework will determine if the maximum number of worker threads (as determined by the handler) is being used by a single organization. If so, the framework will "peek" into the queue to see if other organizations have requests waiting. The set of requests is called the *peek set* and is limited to a fixed number of requests at the front of the queue (currently set at 2000 requests). The framework will look for the requests for a different organization and process those (as long as that organization isn't currently consuming all of its allocated threads for a given handler).

For example, assume organization 1 creates 13 `@future` requests that are at the head and adjacent in the queue as shown in the diagram below:

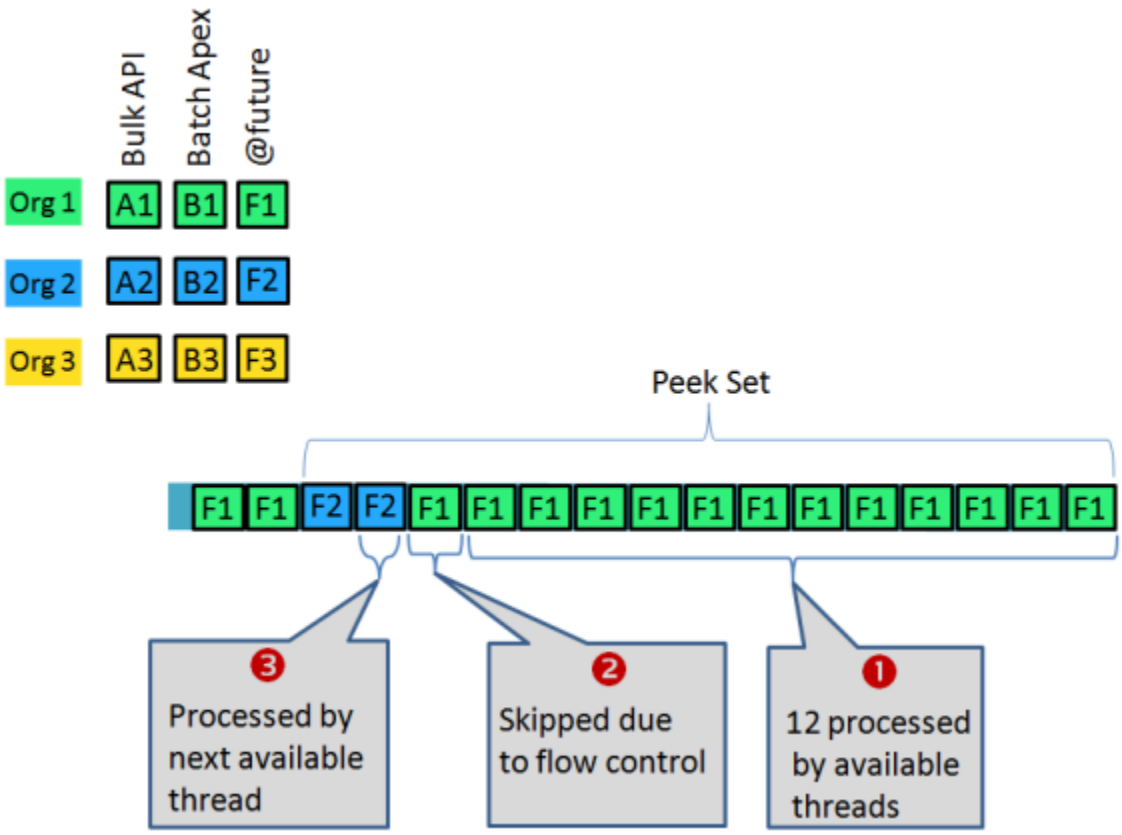
Organization 2 adds two `@future` requests to the queue:



And two more organization 1 `@future` requests are en-queued. At this point, the queue looks like this:



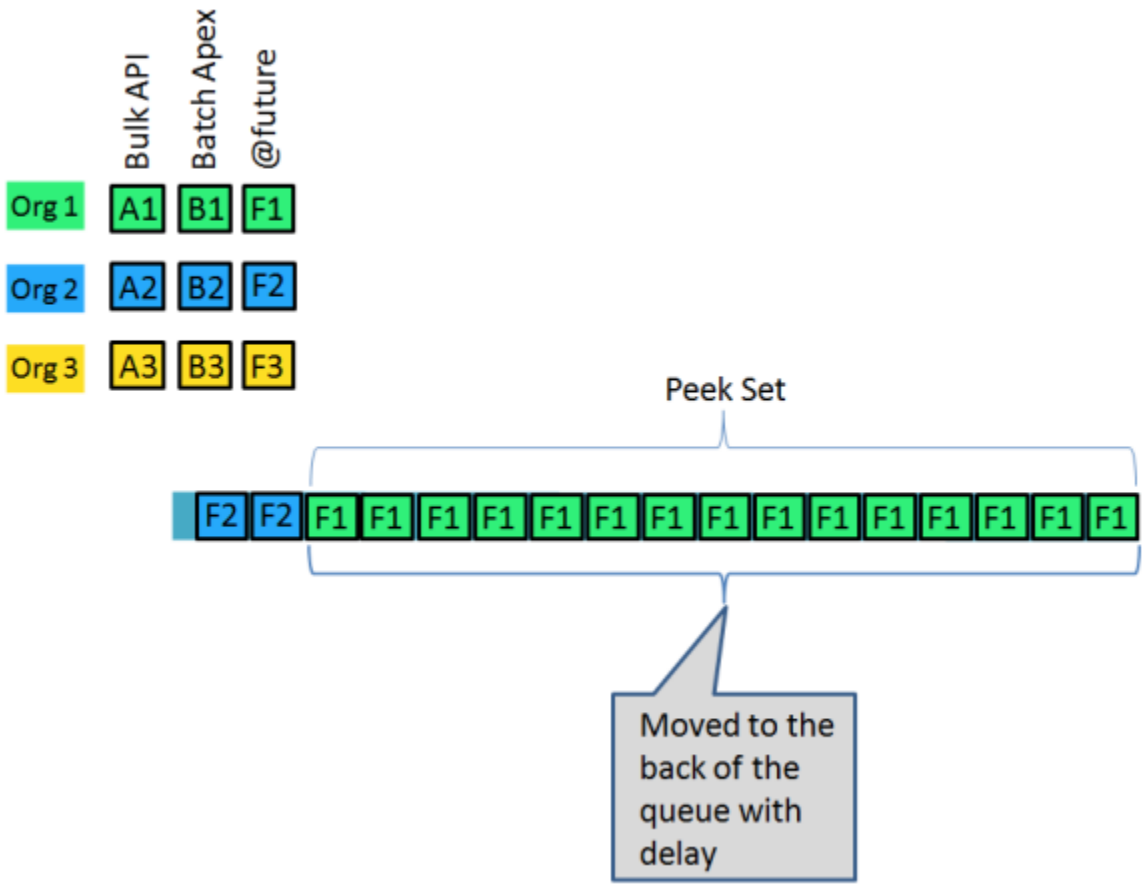
Assuming the `@future` handler specifies 12 maximum worker threads per organization. This indicates that a maximum of 12 threads can process requests from a single organization.



Assuming 13 threads are available, a maximum of 12 threads per organization and no other requests are being processes for organization 1 or organization 2 and a peek set size of 15, the processing will be as follows (see diagram above):

1. 12 threads will take the first 12 requests from organization 1.
2. The 13th thread will not process a request from organization 1 although it is the next one in the queue. This is because organization 1 has taken its allotted amount of threads. This request will remain in the queue at its current position until one of the 12 threads becomes available. This request is *delayed*.
3. The framework will scan for requests from other organizations within the peek set. It will find an organization 2 request and begin processing the first request for organization 2, skipping the 13th request for organization 1.

What happens when a particular organization requests occupy the entire peek set when the queue is scanned in step 3 above?



Assume 12 threads are processing requests from organization 1. Also, assume organization 1 has 15 requests remaining in the queue and organization 2 has two requests in the queue as shown in the diagram above.

Since all of the requests in the peek set are from a single organization (organization 1 in this case), those 15 requests will be moved to the back of the queue with a specific delay. This is called an *extended delay*.

The delay is different for each message. For example, for `@future` requests, the delay is 5 minutes. That means a minimum of 5 minutes must elapse before those requests are eligible for processing.

When the requests become eligible for processing, it's possible for these requests to be acted upon by flow control and again get moved to the back of the queue and delayed. Therefore requests can be delayed several times before they are completed.

Additionally, when those requests are moved, they will be put back into the queue in the same logical order and they may have other requests intermingled with them.

Resource Conservation

Asynchronous processing in Force.com is very important but has lower priority over real-time interaction via the browser and API. Message handlers run on the same application servers that process interactive requests, so it's possible that asynchronous processing or increased interactive usage can cause a sudden increase in usage of computing resources. To ensure there are sufficient resources to handle a sudden increase, the queuing framework will monitor system resources such as server memory and CPU usage and reduce asynchronous processing when thresholds are exceeded. This will give resource priority to interactive requests. Once the resources fall below thresholds, normal asynchronous processing will continue.

Best Practices

Best Practices for Using Asynchronous Processing

The Force.com platform provides user controlled features that use asynchronous processing. Each feature has its own values for the number of worker threads per organization and the delay time for requests that are moved to the back of the queue.

Apex `@future`

Every @future invocation adds one request to the asynchronous queue. Design patterns that would add large numbers of @future requests over a short period of time should be avoided unless absolutely needed. If your design has the potential to add 2000 or more requests at a time, requests could get delayed due to flow control. Best practices include:

- Ensure that the @future requests execute as fast as possible. The longer the request executes the more likely flow control will occur when there are a large number of @future requests. This includes minimizing web service call out times if utilized.
- Conduct thorough testing at scale of the @future design. This will help determine if delays may occur given the design at current and future volumes.
- Consider using Batch Apex instead @future to process large number of records asynchronously. This will be more efficient then creating a @future request for each record.
- Consider submitting smaller batches of @future requests as not to exceed the peek set limit of 2000.

Extended delay time is 5 minutes.

Case studies:

- Consumer web page is created using Force.com Sites. Each person registering at the site requires three non related web service call outs to validate the consumer. These validations need to occur asynchronously after the consumer submits their information. Designed solution was to use one @ future call for each web service call thereby creating large volumes of @future calls. A better design is to create a single @future request for each consumer that will handle the three call outs. This solution creates a much lower volume of requests.
- For each new lead created, outside data validation was required via web services call out. The designed solution was to create one @future call for every record which would do the web service call out. This created large volume of @future requests with a single call out. a better design pattern is to use batching features to make more efficient call outs. Create a call out that could accept multiple records and then use @future to process multiple records in one call out. This solution creates a much lower volume of requests

Batch Apex

Ensure that the Batch Apex process executes efficiently as possible and minimize the batches submitted at one time. Like @future requests, batch Apex needs to execute as fast as possible. Best practices include:

- Tune any SOQL query to gather the records to execute as quickly as possible.
- Minimize web service call out times if utilized.

Extended delay is not applicable to batch Apex.

Bulk API

The Bulk API allows a company to submit large volumes of data for processing asynchronously into Salesforce. Batches of records are submitted via the Bulk API and those batches are then enqueue and processed. If too many batches are enqueue at once, they may be subject to flow control therefore minimize the number of batches if possible.

Summary

Summary

When using the asynchronous features of Force.com, remember these keys points:

- Make `@future` and Batch Apex code is efficient as possible. Long execution times increase the chance of delays and extended delays.
- Minimize the number of the asynchronous requests created to minimize the chance of delay and extended delays.
- Asynchronous requests are processed as quickly as possible governed by available resources and flow controls. There is no guaranteed processing time.