1. Asynchronous Apex
2. Asynchronous Processing Basics

Asynchronous Processing Basics

Note

Attention, Trailblazer!

Salesforce has two different desktop user interfaces: Lightning Experience and Salesforce Classic. This module is designed for **Salesforce Classic**.

You can learn about switching between interfaces, enabling Lightning Experience, and more in the Lightning Experience Basics module here on Trailhead.

Learning Objectives

After completing this unit, you'll be able to:

- Explain the difference between synchronous and asynchronous processing.
- Choose which kind of asynchronous Apex to use in various scenarios.

Asynchronous Apex

In a nutshell, asynchronous Apex is used to run processes in a separate thread, at a later time.

An asynchronous process is a process or function that executes a task "in the background" without the user having to wait for the task to finish.

Here's a real-world example. Let's say you have a list of things to accomplish before your weekly Dance Dance Revolution practice. Your car is making a funny noise, you need a different color hair gel and you

have to pick up your uniform from your mom's house. You could take your car to the mechanic and wait until it is fixed before completing the rest of your list (synchronous processing), or you could leave it there and get your other things done, and have the shop call you when it's fixed (asynchronous processing). If you want to be home in time to iron your spandex before practice, asynchronous processing allows you to get more stuff done in the same amount of time without the needless waiting.

You'll typically use Asynchronous Apex for callouts to external systems, operations that require higher limits, and code that needs to run at a certain time. The key benefits of asynchronous processing include:

User efficiency

Let's say you have a process that makes many calculations on a custom object whenever an Opportunity is created. The time needed to execute these calculations could range from a minor annoyance to a productivity blocker for the user. Since these calculations don't affect what the user is currently doing, making them wait for a long running process is not an efficient use of their time. With asynchronous processing the user can get on with their work, the processing can be done in the background and the user can see the results at their convenience.

Scalability

By allowing some features of the platform to execute when resources become available at some point in the future, resources can be managed and scaled quickly. This allows the platform to handle more jobs using parallel processing.

Higher Limits

Asynchronous processes are started in a new thread, with higher governor and execution limits. And to be honest, doesn't everyone want higher governor and execution limits?

Asynchronous Apex comes in a number of different flavors. We'll get into more detail for each one shortly, but here's a high level overview.

Type	Overview	Common Scenarios
Future Methods	Run in their own thread, and do not start until resources are available.	Web service callout.
Batch Apex	Run large jobs that would exceed normal processing limits.	Data cleansing or archiving of records.
Queueable Apex	Similar to future methods, but provide additional job chaining and allow more complex data types to be used.	Performing sequential processing operations with external Web services.
Scheduled Apex	Schedule Apex to run at a specified time.	Daily or weekly tasks.

It's also worth noting that these different types of asynchronous operations are not mutually exclusive. For instance, a common pattern is to kick off a Batch Apex job from a Scheduled Apex job.

Increased Governor and Execution Limits

One of the main benefits of running asynchronous Apex is higher governor and execution limits. For example, the number of SOQL queries is doubled from 100 to 200 queries when using asynchronous calls. The total heap size and maximum CPU time are similarly larger for asynchronous calls.

Not only do you get higher limits with async, but also those governor limits are independent of the limits in the synchronous request that queued the async request initially. That's a mouthful, but essentially, you have two separate Apex invocations, and more than double the processing capability. This comes in handy for instances when you want to do as much processing as you can in the current transaction but when you start to get close to governor limits, continue asynchronously.

If you enjoy reading about heap sizes, maximum execution times and limits in general, see Execution Governors and Limits for compelling details.

How Asynchronous Processing Works

Asynchronous processing, in a multi-tenant environment, presents some challenges:

Ensure fairness of processing

Make sure every customer gets a fair share of processing resources.

Ensure fault tolerance

Make sure no asynchronous requests are lost due to equipment or software failures.

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

Enqueue

The request gets put into the queue. This could be an Apex batch request, future Apex request or one of many others. The platform will enqueue requests along with the appropriate data to process that request.

Persistence

The enqueued request is persisted. Requests are stored in persistent storage for failure recovery and to provide transactional capabilities.

Dequeue

The enqueued request is removed from the queue and processed. 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. Handlers are executed by a finite number worker threads on each of the application servers that make up an instance. The threads request work from the queuing framework and when received, start a specific handler to do the work.

Resource Conservation

Asynchronous processing has lower priority than real-time interaction via the browser and API. To ensure there are sufficient resources to handle an increase in computing resources, the queuing framework monitors system resources such as server memory and CPU usage and reduce asynchronous processing when thresholds are exceeded. This is a fancy way of saying that the multitenant system protects itself. If an org tries to “gobble up” more than its share of resources, asynchronous processing is suspended until a normal threshold is reached. The long and short of it is that there’s no guarantee on processing time, but it’ll all work out in the end.

Resources

- Execution Governors and Limits

Quiz Complete!

+100 points

Asynchronous Apex
100%
Progress: 100%
View more modules