

Async Apex Framework Documentation

The two main classes of the Async Apex Framework app are the **RecordProcessorBatch** and the **RecordProcessorQueueable** classes, both containing a `submit()` function, with which the user can start a trackable batch job, or a trackable queued job. Both job types support callouts. The submit functions need a **RecordProcessorConfig**, additionally the batch job needs the batch size as well (the default is 2000). For the **RecordProcessorConfig** the user needs to supply it with:

- source ID (the Id field of the SObject, that the job is attached to)
- parent ID (the id of a parent log or parent job for hierarchical sorting)
- query string (which queries all the data needed for the object processing)
- bind map (for the query string)
- a **RecordProcessorAction** (the processing logic)
- an **AsyncCallback** (the finishing algorithm)
- an object of any type (for any additional data needed)
- and a boolean (for determining if the job is retryable)

The **RecordProcessorAction** is an interface, and the implementing class will have the logic of the job. It has an `execute()` function, which supplies the user with an ID map of the objects (gathered with the query above), and the additional data object to work with. It also has a `getAsyncApexJobDisplayName()`, to get the name of the type of job for a nice UI display.

AsyncCallback is also an interface which also has an `execute()` function, which gives the user the source ID, and the additional data object. This callback will run after the job is done successfully, but before the job log status is updated. The package ships an **EmptyCallback** implementation for jobs that need no finishing logic.

Both implementing classes must be declared **global** with a global no-argument constructor, because the retry feature instantiates them by class name from the package namespace, and only global classes are visible across namespaces. Subscriber classes can live in the local namespace of the org or in another managed package: the framework resolves the class in its own namespace first, then in the local namespace, and a namespaced name (`ns.ClassName`) is resolved directly. If the class cannot be resolved, the retry fails with the `ASYNC_APEX_JOB_RETRY_CLASS_NOT_FOUND` error.

In case the job fails mid-executing, an **AsyncJobError** object is created which has all the relevant information about the failed objects, the causes, and which **AsyncJobLog** is associated with the failed job. For queueable jobs the errors are created by the finalizer (one for every log of the job), while for batch jobs they are created from the **BatchApexErrorEvent** platform event, storing the failed scope and the error message, so the failed scope of a batch job can be retried as well.

The handling of these is with the **AsyncJobLogSelector** class which has five important methods:

- **selectAsyncJobLogs**: it takes in the record ID and a filter list with the log IDs. It returns all the top-level (non-child ones) logs for the given record, filtered to the logs given in the filter, if it is not empty or null.
- **selectChildrenAsyncJobLogs**: it needs the id of the parent log, and it gives all the child logs of it.
- **selectRefresh**: it takes in a list of IDs of the logs, the ID of the SObject that the job is attached to, and a boolean determining, if the refresh should include all the top-level logs of the given object not just the logs given in the list. It returns the fresh data of all the logs in an ID map.
- **retry**: tries to rerun the log with the given ID, if it is retryable. Only the failed scope (taken from the **AsyncJobError** records) is rerun. If the job ran with an empty scope, a meaningful error is thrown (`ASYNC_APEX_JOB_RETRY_EMPTY_SCOPE` label), and if the action or callback class cannot be resolved, the `ASYNC_APEX_JOB_RETRY_CLASS_NOT_FOUND` label is thrown.
- **getEventChannelInfo**: returns the platform event channel name and the payload field name of the source ID, resolved with the package namespace, so client-side components can subscribe to the

`AsyncJobLogEvent` platform event without hard-coding the namespace.

These five are all `@AuraEnabled` methods, meaning the user can use them for client-side custom components. But there is also a built-in component, the **`asyncApexJobRelatedList`**, which takes in the ID of the record associated with the jobs, and optionally the log IDs to filter. The component displays the data of the logs in a table, in the order of their creation, with a parent-child dropdown hierarchy for logs that have child logs. It also gives an option to easily retry a failed job, by clicking the retry button on the failed job's row. The component refreshes automatically: by default it subscribes to the `AsyncJobLogEvent` platform event, but with the polling property it can be switched to a 5 second polling mode, which stops automatically when every displayed log is completed (useful in contexts where platform event subscriptions are not available). After every refresh the component fires an `asyncapexjobrelatedlistrefreshed` event, so custom components can react to the changes. The creation dates are displayed in the time zone of the user.

Logs can also be created and updated manually with the **`AsyncJobLogUtil`** class:

- **`createBasicAsyncJobLog`**: creates a single log for a record with a display name. A list-based `createAsyncJobLog` overload taking `AsyncJobLogUtilInput` objects is also available for bulk creation.
- **`createAsyncJobLog`**: creates a log for a job, either from a `RecordProcessorConfig` or from the individual parameters. If the source ID is not given, it falls back to the job ID.
- **`createAsyncJobLogs`**: creates one log for each given source record under the same job, and returns the IDs of the created logs.
- **`updateStatus` / `updateStatusWithException`**: update every log of the given job based on the `AsyncApexJob` status (or the given exception), and return the IDs of the updated logs. Both are tolerant of jobs without logs.

Security model: every SOQL query of the framework runs in user mode (`WITH USER_MODE`), and the queries supplied in the `RecordProcessorConfig` are executed with `AccessLevel.USER_MODE` , so the async jobs only see the records and fields the submitting user has access to. All DML operations run in user mode as well, combined with `Security.stripInaccessible()` , and a meaningful error is thrown if the user has no access to the `AsyncJobLog` or `AsyncJobError` objects. The necessary object and field permissions are included in the **`AsyncApexFrameworkPermissions`** permission set.

Localization: the objects, fields, picklist values and the labels of the related list component are delivered in English with Hungarian translations. The translations are applied in subscriber orgs where Translation Workbench is enabled.

Note: the `BatchApexErrorEvent` trigger runs as the Automated Process user by default. Because of the user mode enforcement it is recommended to configure a `PlatformEventSubscriberConfig` with a dedicated running user that has the framework permission set assigned.