Cactusforce ‘25

Advanced Apex Best Practices:

Best Practices Update, Tips and Tricks

Patrick Fendt, CEO and Chief Salesforce Architect

Platform Technology, Inc. (a.k.a. *Platform Technologies*)

Email: patrick@platformtechnology.ai

Web: https://platformtechnology.ai

LinkedIn: https://linkedin.com/company/platformtechnologies

Thank You Sponsors!

Cactusforce ‘25

Agenda (Q&A at End)

- Security Update
- Performance Optimizations
- Metadata and Custom Metadata
- Daisy-chaining Patterns
- Framework Patterns
  - Trigger Framework
  - Error Handling
- Row Locking Considerations
- Demo of AI/LLM IDE Extension for *Best Practices Trigger Framework*

Coding Exercise Steps (Please do Offline)

- **Install "Blackbelt Toolkit"** from AppExchange (trial)
  - **Note the Rules custom-metadata page layout must be edited to add all the fields (being fixed next week)**
- GitHub site: **https://github.com/ptsupport/samplecode**
- Create a TriggerHandler custom-metadata record named *OptyTriggerHandler,* and set all RecurseLevels to "1" and Active=true and, optionally turn on Diagnostics (start-date-time and set to 60 minutes) – refer to the image in GitHub
- Create the INFO Error_Handler custom-metadata record (see image in GitHub)
- Create the ArchiveOpportunity Rules custom-metadata record (see image in GitHub)
- Create the AccountCriteria Rules custom-metadata record (see image in GitHub)
- Open Developer Console
- Create new *OpportunityTrigger* trigger (1 line of code)
  - pform.TriggerHandler.run(Opportunity.sObjectType);
- Create new OptyTriggerHandler Apex Class (refer to Github sample to copy/paste)
  - Demonstrates use of Rules Engine
  - Demonstrates use of two ErrorHandler.logError() methods
- Save and ensure the *OpportunityTrigger* and *OptyTriggerHandler* class compiles
- Create and update a new Opportunity record (with Account populated) and view Log file in Dev Console
- View ErrorLogs object list-view

Apex Security Update: Page 1

- Inherited Sharing (object/field perms)
  - Reminder: inherited sharing is preferable in many situations: acts at runtime to behave as “with sharing” in these contexts:
  - UI controllers, API integrations, Asynch processes (i.e. defaults to “with sharing” unless explicitly already in without-sharing)
  - Note: omitted “sharing” declaration runs as “without sharing”
- DML: prefer *USER_MODE* rather than *SYSTEM_MODE*
  - Best practice now is to use *AccessLevel.USER_MODE* versus *WITH_SECURITY_ENFORCED*
  - Or *insert as user objectList* (for inline DML)
  - Note: can use *Security.stripInaccessible()* method prior to using USER_MODE to remove fields that user does not have access to
  - New Dev Preview in Spring-25: Database.insert(new Account(name='foo'), AccessLevel.User_mode.withPermissionSetId(permSetId));

Apex Security Update: Page 2

- SOQL
  - List<Account> acc = [SELECT Id FROM Account WITH USER_MODE];

- User_Mode benefits:
  - WITH USER_MODE accounts for polymorphic fields like Owner and Task.whatId
  - WITH USER_MODE finds all FLS errors in your SOQL query, while WITH SECURITY ENFORCED finds only the first error
  - WITH USER_MODE supports all clauses in the SOQL SELECT statement
  - Can use the *getInaccessibleFields()* method on QueryException to examine specific access errors

Performance Optimizations

- SOQL
  - Use Index fields for WHERE clause criteria
  - Use SOQL inside FOR loop when thousands of records
  - See next slide for more information on SOQL performance
- Use Asynchronous Designs when dealing with LDV, CPU time issues or Record Conflicts
  - **Batch Classes** - best for large volumes asynch record processing: nightly/weekly/etc.
  - **Queueable Classes** - best for offloading code to "sister" class that runs asynch with 6X more CPU time (can pass many complex variable via the constructor)
  - **Platform Events** - best for offloading Apex to a separate transaction to avoid record contention of overcome synchronous performance challenges
  - **@future methods** - use sparingly for simple separate-transaction finalization of DML
- Typically a last resort: skinny tables (limited to 100 fields) for LDV (>1M records)
- May want to use Apex for complex multi-record cross-object scenarios
  - Flows can potentially "auto-bulkify" themselves via the SFDC platform, but it is rare and cannot be relied on
- Learn to use the Platform Cache (see cache sample in github)
  - Typically need separate Trigger or Platform-Event to invalidate/delete the cache

SOQL Performance Optimizations

- Prefer putting in SOQL query in FOR loop if you are going to loop over the records and there are thousands of records:
  - for (Account acct : [SELECT ID,Name FROM Account WHERE ID IN: idSet]) {
    // do something - but careful about CPU usage/timeouts
    }
  - Alternate approach – especially if doing DML in loop (not good):
    - for (Account[] acctList : [SELECT ID,Name,TextField__c FROM Account]) {
      for (Account acct : acctList) {
      Other__c othr = otherObjectMap.get(acct.Id);
      othr.Field__c = 'Value';
      othrList.add(othr);
      }
      }
- Tradeoffs on Heap usage versus CPU usage:
  - non-For-loop queries use more heap space for large #'s of records (e.g. 10K-50K)
  - For-loop queries with one-record-per loop use more CPU
  - For-loop queries that retrieve a list (size 200) for each iteration are nice but depends on what you are doing with the list inside that loop
  - Use Batch processing for many hundreds of thousands of records (obviously)

Metadata and Custom Metadata

- Custom-Metadata is cached by default to some extent in the platform
  - My personal experience is the caching is very effective and reliable at improving performance
- Note: Custom-Metadata objects do not support self-relationships to themselves
- Mock your custom-metadata in Apex using this technique:
  - GitHub Class: *MockCustomMetadataTest*
- You can serialize standard and custom objects using similar technique (*JSON.serialize()* method)
  - GitHub Class: *SerializeObjects*

Metadata-Related APIs

- Metadata API
  - Asynchronous and XML-based API - deployment-CRUD-centric
  - Force.com Migration Tool (ant)
  - VSCode IDE Extension
  - SF CLI (retrieve/deploy/pull/push)
  - Java - examples on developer.salesforce.com
  - Apex - very difficult, old Metadata-XML class is unwieldy/outdated, requires special Zip library
- Tooling API
  - REST-based, very powerful and generally very fast
  - Metadata exposed as objects and uses JSON (or XML)
  - Can query about Classes, Triggers, objects, other metadata, dependencies, etc.
- User Interface API
  - CRUD operations for data; read-only for object Metadata
  - This is what SFDC UI uses (Lightning Experience)
- Refer to this page for more information on deciding which API to use:
  - https://help.salesforce.com/s/articleView?id=platform.integrate_what_is_api.htm&type=5

Daisy-Chaining Patterns

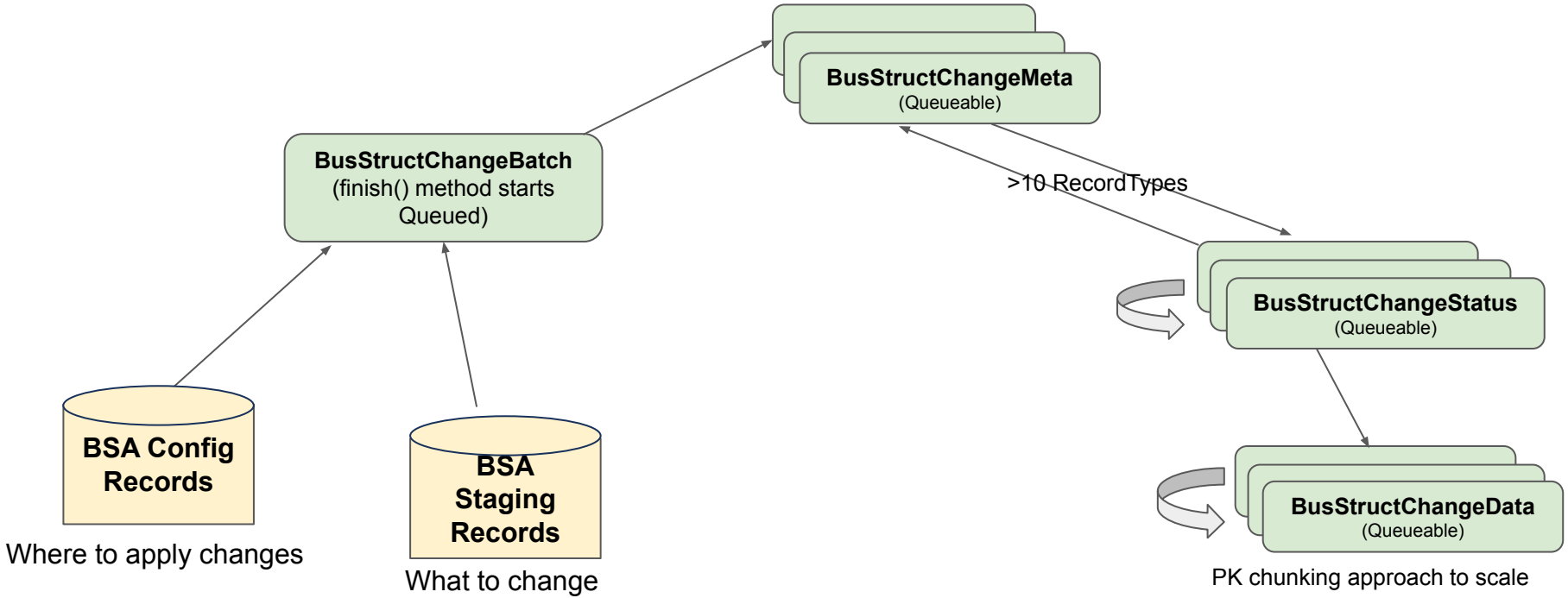
- Synchronous event (e.g. Trigger/Controller) launches asynchronous Apex, or Batch process launches additional processes
  - Queueable (need to pass a parameter so you know when you are done)
  - Batch
  - Combination of above (due to LDV and/or CPU time or other governor limits).
  - For extreme example, see forthcoming slide

- Can use tracking objects to track successes/failures during the asynchronous process(es)
  - Custom object with several fields:
    - Object Name (if more than object involved)
    - Process_Step - business-scenario-specific picklist values
    - Start_Date_Time and End_Date_Time
    - Status (picklist) - e.g. Pending, Failed, Retrying, Success
    - Record_Id (if appropriate to Step/Status)
    - Error_Message

Daisy-Chaining/Relevant Governor Limits

- Per-transaction
  - @future - cannot be called from an asynchronous context
  - Can only enqueue one Queueable apex process at-a-time (but can daisy-chain/restart when done)
  - Batch process can enqueue a Queueable
  - Queueable can start a Batch process
- Asynchronous vs Synchronous benefits
  - More CPU time (60 seconds vs 10 seconds)
  - More heap space (12M vs 6M)
  - 200 SOQL queries for asynchronous vs 100 for synchronous
- Synchronous concurrent transactions for long-running transactions (>5 seconds)
  - 10 → 50 (depending on # of licenses); http callouts no longer count towards this limit
- Managed Packages limits are increased as far as per-transaction totals for the following:
  - DML: extra 150 DMLs per managed-package (1650 max)
  - SOQL: extra 100 queries per namespace (1100 max)
  - And more analogous additions for other limits as well…
- Can use Limits class or *OrgLimits.getAll()* method at runtime

Daisy-chain Batch+Queued Apex (Extreme Example)



Common Framework Patterns

- A good/great **Trigger Framework** is critical:
  - Moves business logic out of Trigger and into Apex Class
  - Provides more opportunity for re-use
  - Ensures code-coverage testing is adequate (75%+)
  - See detailed capabilities on next slide

- A good/great **Error Framework** is critical:
  - Aggregate various SFDC errors into one object
  - Report on Errors (list-views and/or dashboards)
  - Provides much more consistent error-handling Org-wide
  - See detailed capabilities on following slide

- Parent/Child Field Rollup-type Batch-Frameworks are nice
  - Can avoid row-locking when processing objects with hierarchies/self-relationships

Trigger Framework: Specific Benefits

- Control recursion (ideally down at per-action-type level including testing):
  - beforeInsert, afterInsert, beforeUpdate, afterUpdate, etc...
- Control disabling Trigger logic by time-period or by user
- Provide diagnostics (per method) to give you performance information
  - Milliseconds taken per trigger handler method
  - Turn on/off for up to 90 minutes per handler (max)
- Option helper-methods can be provided
  - e.g. Which records have/have-not been processed at this point in the transaction

Error Management: Specific Benefits

- Aggregates various sources of SFDC errors into a single log object
  - Can support Apex, LWC, Flows, Batch Exceptions, Event Monitor alerts, Apex Exception Emails, etc.
- Categorize errors by Error-Type and/or Line-of-Business
- Surface errors from disparate SFDC origins and report/list details
  - SFDC provides 7 different sources of errors (requires complex code to collect)
- Take automated actions (e.g. by Error Type/Category) – such as Emailing an Administrator
  - Can also optionally throw exception automatically (e.g. many developers forget to throw exceptions and “eating an exception” is typically a bad idea
  - Can optionally Rollback a transaction automatically

Row Locking Considerations

- One of the more challenging coding tasks in SFDC is multi-record CRUD operations on records in a hierarchical or master-detail relationship (parent/child relationship)

- Reason(s): for Master:Detail and implicit master:detail (e.g. Opportunity→Account or child-account→parent-account), SFDC is often forced to lock the parent(s) above the children
  - Refer to "Record Locking Cheat Sheet" document for details on CRUD/relationships

- **Solution**: pre-process the parents and their the children (into appropriate data-structure) and only update the children that are in their parent's "tree" (but can do multiple parents in same transaction)
  - i.e. need algorithm to avoid children being "split up" from their parents in separate transactions and then those transactions lock the same parent in parallel

AI LLM IDE Extension Demo for Triggers

- Currently targeted for February 2025 release:
  - Fully supported by Platform Technology, Inc.
  - Will be on VSCode Extension Marketplace
- Current functionality as of today
  - Convert legacy triggers to our Best Practice Toolkit Trigger Framework classes/handlers
  - Generate new Best Practice Toolkit Trigger Framework handlers
  - Leverages Anthropic/Claude and/or OpenAI
  - Charges are minimal for these tasks, but are currently pass-through to customer
- On our near-term roadmap (targeting next month):
  - Convert other Trigger Framework classes/handlers to our Trigger Framework classes/handlers
  - Plan to support all the major patterns for doing this (open-source, interface-based, etc)
- Demo!

Questions?

Thanks!

Cactusforce

Platform Technologies™