Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2020, 7(4):45-52

Research Article ISSN: 2394 - 658X

Simplifying Backend Queries in Salesforce by Implementing a Selector Framework

Chirag Amrutlal Pethad

PetSmart.com, LLC, Stores and Services
Phoenix, Arizona, USA
ChiragPethad@live.com,
ChiragPethad@gmail.com,
Cpethad@petsmart.com

ABSTRACT

As Salesforce applications size grows and complexity increases, maintaining clean, efficient, and reusable code becomes increasingly important. One such challenge and complexity is associated with managing data access. This document discusses the implementation of Selector framework in Salesforce, highlighting the importance of modularized and reusable code, its key components, some of the challenges in implementing the framework, Step by Step guide to implementing the framework components and provides pseudocodes for the implementations, emphasizing application maintainability and performance.

Keywords: Maintainability, Performance, Debugging, Framework, Querying, Optimization, Backend, Database, Centralized, Modularized.

INTRODUCTION

Data access refers to the methods and mechanisms used to retrieve, manipulate, and secure data. Effective data access strategies are critical for building reliable, scalable, and secure applications. Same principles apply to applications running on Salesforce. As applications grow in complexity, managing data access across different classes and layers becomes challenging. This involves understanding Salesforce's data architecture, query languages, security models, and best practices for optimizing performance. The Selector Framework is a design pattern that addresses this challenge by centralizing and standardizing data access logic. This centralized component/data retrieval logic helps encapsulate queries to the database. The paper provides an overview of the Selector Framework, its benefits, best practices for implementation, and practical examples of how to implement this pattern in Salesforce using Apex.

KEY COMPONENTS OF DATA ACCESS IN SALESFORCE

A. Salesforce Data Model

• Objects and Fields: Salesforce organizes data into objects (similar to tables in a database), which consist of fields (similar to columns). There are two main types of objects:
O Standard Objects: Predefined objects provided by Salesforce, such as Account, Contact, Opportunity, and Lead.
O Custom Objects: User-defined objects created to store data specific to an organization's needs.
• Relationships: Objects can have relationships with one another, such as lookup and master-detail relationships, which define how data is related and interact between objects.
• Governor Limits Monitoring: Salesforce imposes limits on resource usage to ensure that no single tenant can monopolize shared resources. Monitoring governor limits helps prevent performance degradation and ensures compliance with Salesforce's multi-tenant architecture.

B. SOQL (Salesforce Object Query Language)

• Purpose: SOQL is Salesforce's query language, similar to SQL, used to retrieve data from Salesforce objects.

• Usage: SOQL allows you to query data from one or more objects, filter results using WHERE clauses, sort results, and aggregate data. It's commonly used in Apex classes, triggers, Visualforce controllers, and Lightning components.
• Examples

```
// Retrieve all Accounts with a specific name
List<Account> accounts = [SELECT Id, Name FROM Account WHERE Name = 'Acme'];

// Join query to retrieve related records
List<Contact> contacts = [SELECT Name, Account.Name FROM Contact WHERE Account.Industry = 'Technology'];
```

C. SOSL (Saleforce Object Search Language)
• Purpose: SOSL is used to search for text across multiple objects and fields simultaneously, which is different from SOQL's record-focused querying.
• Usage: SOSL is ideal for scenarios where you need to search across various objects and fields, such as implementing global search functionality.
• Example

```
List<List<SObject>> searchResults = [FIND 'Acme' IN ALL FIELDS RETURNING Account(Id, Name), Contact(Id, Name)];
```

D. Apex and Data Manipulation Language
• DML Operations: Apex allows you to perform Data Manipulation Language (DML) operations such as INSERT, UPDATE, DELETE, UPSERT, and MERGE to modify Salesforce data.
• Transaction Control: Salesforce processes DML operations within a transaction context, ensuring data integrity. You can use Apex methods like Database.insert to control transaction behavior, handling partial successes or failures.
• Bulkification: To optimize performance and avoid governor limits, it's essential to bulkify DML operations, processing records in batches rather than one at a time.
E. Data Security
• Object-Level Security: Controls which objects a user can access, managed via Profiles and Permission Sets.
• Field-Level Security: Controls which fields within an object a user can access, also managed via Profiles and Permission Sets.
• Record-Level Security: Controls access to individual records, managed through Organization-Wide Defaults (OWDs), Role Hierarchies, Sharing Rules, and Manual Sharing.
• CRUD and FLS Enforcement: Apex developers need to enforce CRUD (Create, Read, Update, Delete) and FLS (Field-Level Security) checks in code to ensure that users have the necessary permissions before accessing or modifying data.
F. Data Access Best Practices
• Governor Limits: Salesforce enforces governor limits to ensure efficient use of resources. Understanding and working within these limits is crucial for performance and reliability.
• Selective Queries: Write selective queries to limit the number of records returned and avoid performance issues. Use indexed fields in WHERE clauses to improve query performance.
• Avoiding SOQL in Loops: To prevent performance issues, avoid placing SOQL queries inside loops. Instead, query all necessary records outside the loop and process them within the loop.
• Lazy Loading: Implement lazy loading for data that is not immediately needed to optimize performance and reduce unnecessary queries.
• Caching: Use caching mechanisms to store frequently accessed data temporarily, reducing the need for repeated queries.
G. Advanced Data Access Techniques
• Selector Framework: A design pattern that encapsulates data access logic into dedicated Selector Classes, improving code maintainability, reusability, and scalability.
• Service Layer: Implementing a service layer to handle business logic and data access coordination, promoting a clean separation of concerns.
• Batch Apex: Use Batch Apex to process large volumes of records asynchronously, allowing you to bypass certain governor limits and handle complex data processing tasks.
H. External Data Access
• Salesforce Connect: Allows you to access external data in real-time using External Objects, without importing the data into Salesforce.
• Callouts: Apex HTTP callouts enable Salesforce to interact with external systems via REST or SOAP APIs, allowing for data integration and synchronization.

NEED FOR A SELECTOR FRAMEWORK

The need for a Selector Framework in Salesforce arises from the challenges and complexities associated with managing data access in large, scalable applications. Some of the reasons include:

A. Seperation of Concerns

• In many Salesforce applications, business logic and data access logic are often intertwined. This makes the codebase harder to maintain and test, as changes to one aspect can inadvertently affect others.

• The Selector Framework separates data access logic from business logic. By encapsulating all data retrieval operations in dedicated Selector Classes, you isolate concerns, making the codebase more modular and easier to manage.

B. Code Reusability

• Without a standardized approach to data access, developers may write similar SOQL queries across different parts of the application, leading to redundancy and inconsistencies.

• The Selector Framework centralizes common query logic, enabling reuse across multiple classes and modules. This reduces duplication, ensures consistency, and accelerates development by providing ready-to-use query methods.

C. Improved Maintainability

• As applications evolve, so do the underlying data models. Without a centralized approach to data access, making changes to the data model (e.g., adding new fields or altering query criteria) can require extensive updates across the codebase.

• By using Selector Classes, changes to data access logic need to be made in only one place, significantly reducing the effort required to maintain and update the application.

D. Enhanced Testability

• When data access logic is scattered throughout the codebase, it becomes difficult to isolate and test specific components. This can lead to inadequate test coverage and unanticipated bugs.

• The Selector Framework improves testability by encapsulating data retrieval logic in isolated classes. These Selector Classes can be unit tested independently of the business logic, ensuring that queries return the expected results without side effects.

E. Scalability and Performance [8]

• As applications scale, inefficient or redundant queries can lead to performance issues and resource contention. Without a framework to manage data access, it can be challenging to optimize query performance across the application.

• The Selector Framework allows developers to optimize queries centrally, ensuring that data access is efficient and scalable. Advanced techniques like bulkification, lazy loading, and caching can be implemented within Selector Classes to further enhance performance.

F. Consistency in Data Access

• Inconsistent data access patterns across an application can lead to data integrity issues, unexpected behavior, and increased complexity in troubleshooting and debugging.

• The Selector Framework enforces consistent data access patterns by standardizing how queries are constructed and executed. This consistency makes the application more predictable and easier to debug.

G. Simplified Data Access Management

• Managing and optimizing data access becomes increasingly difficult as the number of queries and the complexity of the application grows.

• The Selector Framework simplifies the management of data access by centralizing it into specific, well-defined classes. This makes it easier to monitor, optimize, and modify how data is retrieved within the application.

CHALLENAGES OF IMPLEMENTING SELECTOR FRAMEWORK

Implementing a Selector Framework in Salesforce provides many benefits, such as improved code organization, reusability, and maintainability. However, there are also several challenges that developers might face during the implementation. Below are some of the key challenges:

A. Complexity in Initial Setup

Setting up a Selector Framework requires careful planning and design, which can be complex and time-consuming. Developers must define consistent patterns, organize data retrieval logic, and ensure that the framework is adaptable to various use cases. The initial setup can be resource-intensive, especially in large projects with numerous objects and relationships, potentially leading to delays in development.

B. Performance Overhead

While a Selector Framework aims to centralize and optimize data access, improper implementation can lead to performance issues. For instance, if caching is not correctly managed or bulk operations are not adequately handled, it could result in excessive SOQL queries or DML operations. Poorly optimized Selector Classes may lead to

governor limit violations, slow query performance, or unnecessary data retrieval, impacting the overall performance of the Salesforce application.

C. Balancing Flexibility and Simplicity

A well-designed Selector Framework needs to balance flexibility (to accommodate various querying needs) with simplicity (to ensure ease of use and maintainability). Striking this balance can be difficult, as overly complex selectors can become hard to maintain, while overly simplistic selectors might not meet all business requirements. If the framework is too complex, it can lead to increased maintenance effort and potential errors. If it's too simplistic, it may require frequent modifications or fail to meet the needs of different modules or teams.

D. Ensuring Consitent Usage

Once the Selector Framework is implemented, it's essential that all developers adhere to its usage. Inconsistent usage, where some developers bypass the framework and write their own queries, can lead to fragmented codebases and undermine the benefits of the framework. Inconsistent usage can result in code duplication, increased maintenance burden, and a lack of uniformity in data access logic, making it harder to manage and troubleshoot.

E. Maitaining Security Compliance

Ensuring that CRUD (Create, Read, Update, Delete) and Field-Level Security (FLS) checks are consistently enforced in Selector Classes can be challenging, especially as the framework evolves and expands. Developers need to be vigilant in implementing security checks across all selectors. Failure to enforce security checks can lead to unauthorized data access, compliance issues, and potential data breaches, posing significant risks to the organization.

F. Handling Complex Query Requirements

Some business scenarios require complex queries involving multiple joins, subqueries, or dynamic conditions. Implementing these in a Selector Framework while keeping the code clean and maintainable can be difficult. If not handled properly, complex query requirements can lead to bloated Selector Classes, making the framework harder to manage and increasing the likelihood of bugs and performance issues.

G. Testing and Debugging

While the Selector Framework promotes reusability, it can make testing and debugging more challenging. Errors in a centralized selector can affect multiple parts of the application, making it harder to isolate and fix issues. Increased difficulty in testing and debugging can lead to longer development cycles and increased risk of undetected bugs in production.

H. Scalability of the Framework

As the Salesforce application grows, the Selector Framework must evolve to handle new objects, fields, and relationships. Ensuring that the framework scales effectively without becoming overly complex or cumbersome is a significant challenge. If the framework does not scale well, it can become a bottleneck in development, leading to inefficiencies, increased maintenance effort, and potential limitations in handling new business requirements.

I. Training and Adoption

Implementing a Selector Framework often requires a shift in how developers approach data access. Ensuring that the entire development team understands and adopts the framework can be challenging, particularly in larger teams or organizations with varying levels of Salesforce expertise. Without proper training and buy-in from the development team, the framework may not be used consistently, leading to a fragmented approach to data access and diminishing the benefits of the framework.

J. Upgrading and Refactoring

As Salesforce evolves and new features are introduced, the Selector Framework may require upgrades or refactoring to take advantage of new capabilities or to address deprecated functionality. Managing these changes without disrupting the application can be challenging. Frequent upgrades or refactoring can increase the maintenance burden and introduce risks of breaking changes, potentially affecting the stability and reliability of the application.

KEY FEATURES OF A SELECTOR FRAMEWORK

An effective Selector framework in Salesforce Apex should include the following features:

A. Centralized Data Access [4]:

All SOQL queries related to a specific Salesforce object are centralized in dedicated Selector Classes. This ensures that all data retrieval logic for an object is maintained in one place, making it easier to manage and update. It helps reduces code duplication and promotes consistency across the application, simplifying maintenance and updates.

B. Reusable Quey Logic [4]:

The framework provides reusable methods for common queries, such as retrieving records by ID, name, or custom fields. These methods can be called from anywhere in the application, avoiding the need to rewrite the same SOQL queries multiple times. It helps enhances code reusability, reduces redundancy, and speeds up development by providing a set of pre-defined, reliable methods.

C. Support for Customizable Queries:

The framework allows for the customization of queries through method parameters. Developers can pass in conditions, sorting options, and limits to tailor the query results to specific needs. It helps increase flexibility and adaptability of the framework, allowing it to cater to a wide range of business requirements without extensive modifications.

D. Bulk Query Support:

Selector Classes are designed to handle bulk operations by accepting lists of IDs or other parameters. This enables efficient querying of large datasets, ensuring that the framework can scale with the application's needs. It helps improve performance by minimizing the number of SOQL queries and reduces the risk of hitting governor limits.

E. Enforcement of Security Best Practices [4]:

The framework includes built-in checks for CRUD (Create, Read, Update, Delete) and Field-Level Security (FLS) permissions within the Selector Classes. This ensures that only authorized users can access or manipulate data. It helps enhance security and compliance by ensuring that all data access operations adhere to the organization's security policies.

F. Caching Mechanism

The framework can incorporate caching strategies, such as Platform Cache or custom in-memory caches, to store frequently accessed data. This reduces the number of databases queries and improves overall application performance. It also optimizes performance by reducing the load on the Salesforce database and speeding up data retrieval times.

G. Consistent Naming Conventions

The framework follows consistent naming conventions for classes, methods, and variables. For example, Selector Classes might be named after the Salesforce object they represent (e.g., AccountSelector), and methods might follow a pattern like getById(), getByName(), etc. It also improves code readability and makes it easier for developers to understand and navigate the codebase.

H. Support for Complex Query Logic

The framework can handle complex query logic, such as joins, subqueries, and dynamic conditions. This is achieved by breaking down complex queries into smaller, reusable methods that can be composed as needed. Also provides the ability to handle sophisticated data retrieval scenarios without compromising code maintainability or readability.

I. Unit Testability

The framework is designed with testability in mind. Selector Classes are isolated from business logic, making them easier to unit test. Each method can be tested independently to ensure it functions correctly under various conditions. It also improves code quality by facilitating thorough testing, making it easier to detect and fix issues early in the development process.

J. Extensibility [4]

The framework is designed to be easily extensible. New Selector Classes can be added as new objects are introduced to the Salesforce environment, and existing classes can be extended to support additional queries or features. It ensures that the framework can grow and adapt with the application, providing long-term scalability and flexibility.

K. Error Handling and Logging

The framework can include standardized error handling and logging mechanisms to capture and manage exceptions that occur during data retrieval. This ensures that issues are logged and can be addressed promptly. And it improves reliability and makes it easier to troubleshoot and resolve issues in production environments.

L. Documentation and Inline Comments

The framework is well-documented, with clear explanations of each Selector Class, its methods, and the intended use cases. Inline comments are used to clarify complex logic or important considerations within the code. Also enhances the usability of the framework, making it easier for new developers to understand and use it effectively.

IMPLEMENTATION OF SELECTOR FRAMEWORK IN APEX

A. Implementation of Selector Class [1][7]

Begin by creating a new Apex class that follows the naming convention for Selector Classes. Define the class to encapsulate all query logic related to a specific S-Object. So in the below example we create a class called Account Selector that will encapsulate all the queries on Account Object.

```
public class AccountSelector {
    // Define query methods for retrieving Account records
}
```

Figure 1: *Selector Class Implementation*

We continue implementing common query methods on Account. We implement methods for common data retrieval operations, such as retrieving records by ID, by name, or retrieving all records[5].

```
public class AccountSelector {
    public static Account getById(Id accountId) {
        return [SELECT Id, Name FROM Account WHERE Id = :accountId];
    }

    public static List<Account> getByName(String name) {
        return [SELECT Id, Name FROM Account WHERE Name = :name];
    }

    public static List<Account> getAllAccounts() {
        return [SELECT Id, Name FROM Account];
    }
}
```

Figure 2: *Account Selector Class Common methods Implementation*

We can also allow for customizations to query by providing flexible methods that accept additional parameters or criteria [2][5].

```
public class AccountSelector {
    public static List<Account> getByIndustry(String industry,
                                List<String> additionalFields) {
        String fields = 'Id, Name, Industry';
        if (additionalFields != null && !additionalFields.isEmpty()) {
            fields += ', ' + String.join(additionalFields, ', ');
        }

        return Database.query('SELECT ' + fields + ' FROM
             Account WHERE Industry = :industry');
    }
}
```

Figure 3: *Account Selector Class with Customizable Query method.*

Implement local caching to avoid repeated queries to the database in the same transaction.

```
public class AccountSelector {
    private static Map<Id, Account> accountCache = new Map<Id, Account>();

    public static Account getByIdWithCache(Id accountId) {
        if (!accountCache.containsKey(accountId)) {
            accountCache.put(accountId, [SELECT Id, Name FROM
              Account WHERE Id = :accountId]);
        }
        return accountCache.get(accountId);
    }
}
```

Figure 4: *Local In-Memory Caching in the Selector Class*

Implement Bulk Queries. Ensure that the Selector Framework supports bulk queries by allowing methods to accept lists of IDs or other parameters. This reduces the risk of hitting governor limits and optimizes performance for large data sets [5].

```
public class AccountSelector {
  public static Map<Id, Account> getByIds(List<Id> accountIds) {
      Map<Id, Account> accountsMap = new Map<Id, Account>();
      for (Account acc : [SELECT Id, Name FROM Account WHERE Id IN :accountIds]) {
          accountsMap.put(acc.Id, acc);
      }
      return accountsMap;
  }
}
```

Figure 5: *Bulkification of Query in the Selector Class*

Enforce CRUD and FLS [6] Security[1] Checks in Selector[5] Classes to ensure that users have the necessary permissions to access the data.

```
public class AccountSelector {
  public static Account getById(Id accountId) {
      if (!Schema.sObjectType.Account.isAccessible()) {
          throw new AuthorizationException('Access to Account object is denied.');
      }
      return [SELECT Id, Name FROM Account WHERE Id = :accountId];
  }
}
```

Figure 6: *FLS Checks in Selector Class*

Next, we unit test [7] [9] each method in the selector class to ensure that queries return the expected results. Use test data factories and mock data to simulate different scenarios [3].

```
@IsTest
public class AccountSelectorTest {
    @IsTest
    static void testGetById() {
        Account testAccount = TestDataFactory.createAccount();
        insert testAccount;

        Account result = AccountSelector.getById(testAccount.Id);
        System.assertEquals(testAccount.Id, result.Id);
    }
}
```

Figure 7: *Apex Test Class for Account Selector*

Finally, we implement an example of how to use the Selector Framework using the Service Layer [4][7]

```
public class AccountService {
    public void processAccountsByIndustry(String industry) {
        List<Account> accounts = AccountSelector.getByIndustry(industry, null);
        // Business logic to process accounts
    }
}
```

Figure 8: *Example Service Class for using the Framework*

BEST PRACTICES FOR SELECTOR FRAMEWORK

A. Use a Consistent Naming Convention [1]:

Adopt a clear and consistent naming convention for your Selector Classes, typically following the pattern <ObjectName>Selector (e.g. AccountSelector, OpportunitySelector). This helps in easily identifying and managing Selector Classes.

B. Define Common Query Methods:

Implement common query methods in each Selector Class, such as methods for retrieving records by ID, by specific criteria, or in bulk. These methods should be reusable across the application.

C. Lazy Loading:

Implement lazy loading in Selector Classes to defer the loading of data until it is needed. This can improve performance by reducing unnecessary database queries.

D. Use Caching:

Use caching mechanisms, such as Platform Cache, to store frequently accessed data in memory. This reduces the need to query the database repeatedly and enhances application performance.

E. Keep Methods Focused on Single Concern and Simple:

Each method within a Selector Class should perform a single, well-defined query operation. Avoid overloading methods with multiple responsibilities to maintain clarity and simplicity.

F. Handle Query Customizations:
Provide mechanisms for customizing queries, such as allowing the caller to specify additional fields or filtering criteria. This can be achieved using method parameters or through builder patterns.

G. Minimize Database Calls:
Optimize your Selector Classes to minimize the number of database calls. This can be achieved through techniques like bulkification.

H. Document Query Methods:
Clearly document each method within the Selector Class, specifying the purpose of the query, expected inputs, and outputs to facilitate collaboration and maintenance. This helps other developers understand and use the methods correctly.

I. Version Control:
Version control selector classes alongside other Salesforce metadata to track changes and manage deployments effectively.

J. Continous Refinement:
Continuously refine selector classes based on evolving business requirements, performance monitoring, and feedback from testing and production environments.

CONCLUSION

- The Selector Framework in Salesforce empowers developers to streamline data access, enhance code organization, and optimize performance within Apex applications.
- By adopting this design pattern, organizations can achieve greater efficiency, scalability, and maintainability in their Salesforce implementations.
- Leveraging the Selector Framework ensures that data access remains robust, compliant with Salesforce best practices, and aligned with business objectives.
- Improves developer productivity but also contributes to the overall success and agility of Salesforce-driven initiatives, enabling organizations to innovate and adapt to changing business needs effectively.
- By centralizing data access logic in Selector Classes, you can create a more modular and efficient codebase that is easier to manage and extend.
- Adhering to best practices, such as bulkification, governor limit awareness, and the use of design patterns like the Selector Framework, ensures that Salesforce applications remain robust, maintainable, and performant as they scale.

REFERENCES

[1]. https://codingwiththeforce.com/soc-and-the-apex-common-library-tutorial-series-part-13-the-selector-layer/
[2]. Selector Library - https://fflib.dev/docs/selector-layer/overview
[3]. Why I need Selector Layer? - https://github.com/apex-enterprise-patterns/fflib-apex-common/discussions/432
[4]. Selector Pattern Principles - https://trailhead.salesforce.com/content/learn/modules/apex_patterns_dsl/apex_patterns_dsl_learn_selector_l_principles
[5]. Salesforce SOQL and SOSL Reference - https://developer.salesforce.com/docs/atlas.en-us.soql_sosl.meta/soql_sosl/sforce_api_calls_soql.htm
[6]. Selector Framwork by Poitr Gajek - https://soql.beyondthecloud.dev/overview
[7]. Apex Developer Guide - https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_classes.htm?q=Apex%20Class
[8]. Apex Governor Limit - https://developer.salesforce.com/docs/atlas.en-us.234.0.apexcode.meta/apexcode/apex_gov_limits.htm
[9]. Apex Testing - https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_testing.htm