How permissions and sharing work on Salesforce

The Salesforce platform has two main ways of controlling access to records—**permissions** and **sharing**. Permissions in Salesforce focus on what you can do with a particular object in general. Sharing focuses on what records you can see for that object based on their ownership.

With Salesforce's permissions and sharing tools, you can build up a very granular set of visibility permissions that control exactly who can access the different records within your org. When you attempt to retrieve a record in Salesforce, these permissions are checked before any further processing occurs. Following this, sharing calculations are then run to verify whether you have access to the record or records that you are retrieving.

Sharing and performance

As an application grows on the platform, the volume of data stored on it will also inevitably grow. The obvious side effect of this is that querying for records and loading lists of data can become much slower. To help keep your application performant, you should reduce the amount of data every user can see to that which is strictly necessary. This will ensure that queries, list views, reports, and all manner of functionalities continue to run in an optimal way and don't slow down the user experience.

Enforcing sharing

By default, all Apex operations (and Process Builder and certain Flows) run in System Mode; that is, they execute as a generic system user that has access to all metadata and data within the org. Within our Apex configuration, sharing is enforced through the use of the with sharing keywords in our class definition. Declaring either **with sharing** or **without sharing** explicitly is a deliberate action for us to verify that we either do or do not want the sharing rules for the current user to be enforced. If we do not define a class as with sharing or without sharing explicitly (`ClassC`), then the current sharing rules remain in force.

```
public with sharing class ClassA {
        public static List<Account> getAccounts() {
                return ClassC.getAccounts();
        }
}

public without sharing class ClassB {
        public static List<Account> getAccounts() {
                return ClassC.getAccounts();
        }
}
```

```
public class ClassC {
        public static List<Account> getAccounts() {
                return [SELECT Name from Account];
        }
}
```

> If `ClassC` was the entry point to our transaction, it would operate in without sharing mode by default.

In situations where we want to default to a `with sharing` context, but enable the code to run in a `without sharing` context when called from a class defined as `without sharing`, we can utilize the `inherited sharing` option as our default. Apex without a sharing declaration is insecure by default. An explicit `inherited sharing` declaration makes the intent clear, avoiding ambiguity arising from an omitted declaration or false positives from security analysis tooling.

So whenever we are defining our Apex classes, we should apply the following rules to ensure our sharing is actually enforced as we anticipate it to be:

- Use `with sharing` when we know we want the sharing model to be enforced.
- Use `without sharing` when we know we want the sharing model to be ignored.
- Otherwise, use `inherited sharing` as a default.

Sharing records using Apex

Salesforce has several ways of sharing records with users and groups of users such as managed sharing, user-managed (or manual) sharing, and Apex managed sharing:

- Managed sharing is the point-and-click sharing that most Salesforce developers and administrators are familiar with, and relies upon record ownership, the role hierarchy in the org, and any sharing rules.
- User-managed sharing or manual sharing is when a user chooses to share a record with a user or group of users using the Share button.
- Apex managed sharing is the sharing of records with a user or group of users through the use of Apex code.

All three of the methods described store records in the share object associated with the record within the Salesforce database. For every object, there is a corresponding share object. For standard objects, it is the object API name plus share, so **AccountShare**, **ContactShare**, and so on.

For custom objects, **__c** in the object API name is replaced by **__Share**. Sharing via org-wide defaults, the role hierarchy, and permissions such as View All are not stored in these objects.

So as an example, to create an `AccountShare` record, we require to set the following information:

- The ID of the record to be shared with the **ParentId** field.
- The user of group to be shared within the **UserOrGroupId** field.
- An access level, either **Edit** or **Read**, in the **AccessLevel** field.
- A reason for sharing in the **RowCause** field. The default value is **Manual**, as we have set here, however custom reasons can be set by adding them through the setup menu.

```
AccountShare newShare = new AccountShare();
newShare.ParentId = accId;
newShare.UserOrGroupId = userId;
newShare.AccessLevel = 'Read';
newShare.RowCause = Schema.AccountShare.RowCause.Manual;
accShares.add(newShare);
```

Enforcing object and field permissions

We have two ways of enforcing our object-level and field-level permissions in Apex code, the first of which is to utilize the describe methods within Apex to verify that the user had the correct permissions. The methods to verify the permissions for an `sObject` are as follows and are utilized on the `Schema.DescribeSObjectResult` instance for the given `sObject`:

- isAccessible
- isCreateable
- isUpdateable
- isDeletable

For example, we can verify permissions on a `Contact` object as follows:

```
if(Schema.sObjectType.Contact.isAccessible()) {
   // Read Contact records
}

if (Schema.sObjectType.Contact.isCreateable()) {
    // Create Contact records
}

if (Schema.sObjectType.Contact.isUpdateable()) {
    // Create Contact records
}

if (Schema.sObjectType.Contact.isDeletable()) {
    // Create Contact records
}
```

Similarly, at the field level, on the `Schema.DescribeFieldResult` instance for a field, we have the following methods:

- isAccessible
- isCreateable
- isUpdateable

All of these are available to us to verify that we have the correct permissions to manipulate a field:

```
if(Schema.sObjectType.Contact.fields.Email.isAccessible()) {
    // Read Contact record Email
}

if (Schema.sObjectType.Contact.fields.Email.isCreateable()) {
    // Populate Contact record Email
}

if (Schema.sObjectType.Contact.fields.Email.isUpdateable()) {
    // Edit Contact record Email
}
```

We can also enforce field and object permissions using the `Security.stripInaccessible` method, which takes two parameters. The first is an access level that we wish to verify against, and the second is a `List<sObject>`. The method then removes any fields that the user does not have the stated access level for. It is also particularly useful for sanitizing records as a whole, such as when we are providing an API and receiving `sObject` data from external users. Read more about the `stripInaccessible` method here.

In the following code, we are using the `stripInaccessible` method to remove any fields that the user does not have update permissions on:

```
String jsonBody = '[{"FirstName":"Alice", "LastName":"Jones", "Email": "ajones@test.com"}]';

List<Contact> contacts = (List<Contact>)JSON.deserializeStrict(jsonBody, List<Contact>.class);

SObjectAccessDecision accessDecision = Security.stripInaccessible(AccessType.UPDATABLE, contacts);
update accessDecision.getRecords();
```

Enforcing permissions and security within SOQL

Salesforce added the `WITH SECURITY_ENFORCED` clause to the SOQL language. Unlike the `stripInaccessible` method, if the user is lacking permissions for a field, an exception is thrown rather than the field simply being removed.

To apply this clause, we simply include `WITH SECURITY_ENFORCED` after any WHERE clause and before any ORDER BY, LIMIT, OFFSET, or aggregate function clauses. For example, consider the following:

```
List<Contact> contacts = [SELECT FirstName, LastName, Secret_Field__c FROM Contact WITH SECURITY_ENFORCED];
```

In the preceding query, if the user has no access to `Secret_Field__c`, then an exception will be thrown, indicating insufficient permissions.

Avoiding SOQL injection vulnerabilities

The first and most simple is to ensure we are using Apex binding variables and static queries. By default, Apex binding variables are automatically escaped and so will ensure that the query would run as expected.

```
public String searchName {get; set;}

public PageReference search() {
   return [SELECT Id, LastName, Email FROM Contact WHERE LastName Like :searchName];
}
```

There are instances where we must use a dynamic query. In these instances, we should ensure that we escape any input from the end user using the `escapeSingleQuotes` method:

```
public String searchName {get; set;}
```