MALLA REDDY UNIVERSITY

R-22

III YEAR B.TECH. (CSE) / I – SEM

MR22-1CS0155

Salesforce Platform Developer

Unit 4
Secure Server-Side Development

Dr Shaik Hussain Shaik Ibrahim
Associate prof/CSE
MRU

Write Secure Apex Controllers

1. Write Secure Apex Controllers

Two ways to provide security

1. Enforcing Sharing Rules

2. Enforcing Object and Field Permissions

1. Enforcing Sharing Rules

With Sharing

- The with sharing keyword lets you specify that the sharing rules for the current user are enforced for the class.

- You have to explicitly set this keyword for the class because Apex code runs in system context

Without Sharing

- You use the without sharing keywords when declaring a class to ensure that the sharing rules for the current user are not enforced.

- For example, you can explicitly turn off sharing rule enforcement when a class is called from another class that is declared using with sharing

1. Enforcing Sharing Rules

Inherited Sharing

Apex class without a sharing declaration is insecure by default.

Designing Apex classes that can run in either with sharing or without sharing mode at runtime is an advanced technique.

Such a technique can be difficult to distinguish from one where a specific sharing declaration is accidentally omitted.

An explicit inherited sharing declaration clarifies the intent, avoiding ambiguity arising from an omitted declaration or false positives from security analysis tooling.

2. Enforcing Object and Field Permissions

User Mode Operations

Data operations (SOQL, DML, and SOSL) in Apex run in system mode by default and have full CRUD access to all objects and fields in general.

In Spring 2023, Apex introduced new access levels allowing developers to select the mode for executing data operations

2. Enforcing Object and Field Permissions

Access Records in User Mode

Access Records in User mode ensures the enforcement of sharing rules, CRUD/FLS, and Restriction Rules.

By utilizing SOQL queries with the USER_MODE keyword, such as in this example

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

2. Enforcing Object and Field Permissions

Insert Records in User Mode

In User mode, Insert Records ensures that the insert operation executes only if the user has permission for both creating a new record and edit permission on the field Opportunity.Amount (FLS check)

```
Opportunity o = new Opportunity();
// specify other fields
o.Amount=500;
insert as user o;
```

2. Enforcing Object and Field Permissions

Insert Records in User Mode

Another way to execute User mode operations:

```
Opportunity o = new Opportunity();
// specify other fields
o.Amount=500;
database.insert(o,AccessLevel.USER_MODE);
```

2. Enforcing Object and Field Permissions

Update Records in User Mode

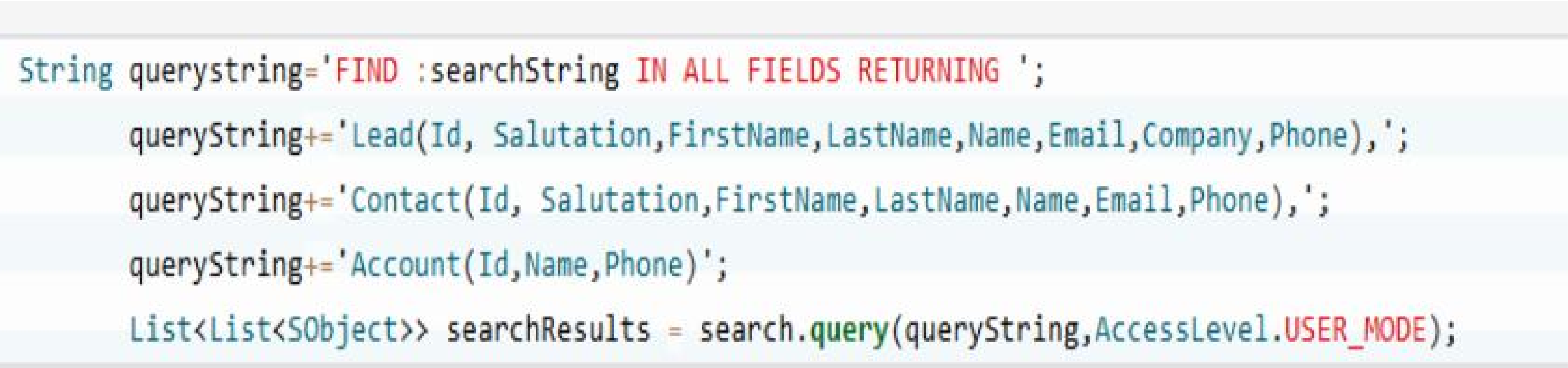
To update Records in User mode:

```
Account a = [SELECT Id,Name,Website FROM Account WHERE Id=:recordId];
// specify other fields
a.Website='https://example.com';
update as user a;
```

2. Enforcing Object and Field Permissions

SOSL in User Mode

To execute SOSL in User mode:



```
String querystring='FIND :searchString IN ALL FIELDS RETURNING ';
        queryString+='Lead(Id, Salutation,FirstName,LastName,Name,Email,Company,Phone),';
        queryString+='Contact(Id, Salutation,FirstName,LastName,Name,Email,Phone),';
        queryString+='Account(Id,Name,Phone)';
        List<List<SObject>> searchResults = search.query(queryString,AccessLevel.USER_MODE);
```

Using WITH SECURITY_ENFORCED

You can integrate the **WITH SECURITY_ENFORCED clause into your SOQL SELECT queries** within Apex code to validate field- and object-level security permissions automatically.

Strategic Placement:

1. **Insert the clause after the WHERE clause** (if present) or after the FROM clause if no WHERE clause exists.

2. **Place it before ORDER BY, LIMIT, OFFSET, or aggregate function** clauses

Using WITH SECURITY_ENFORCED

Example:

```
List<Account> act1 = [SELECT Id, (SELECT LastName FROM Contacts) FROM
Account WHERE Name like 'Acme' WITH SECURITY_ENFORCED]
```

Result:

returns Id and LastName for the Acme account entry if the user has field access for LastName

Using CRUD/FLS Check Methods

You can also enforce object-level and field-level permissions in your code by explicitly calling the sObject describe result methods and the field describe result methods

Schema.DescribeSObjectResult

Schema.DescribeFieldResult

Using CRUD/FLS Check Methods

Let’s walk through theDescribeSObjectResult class helper functions that you can use to verify a user’s level of access. These include:

IsCreateable()

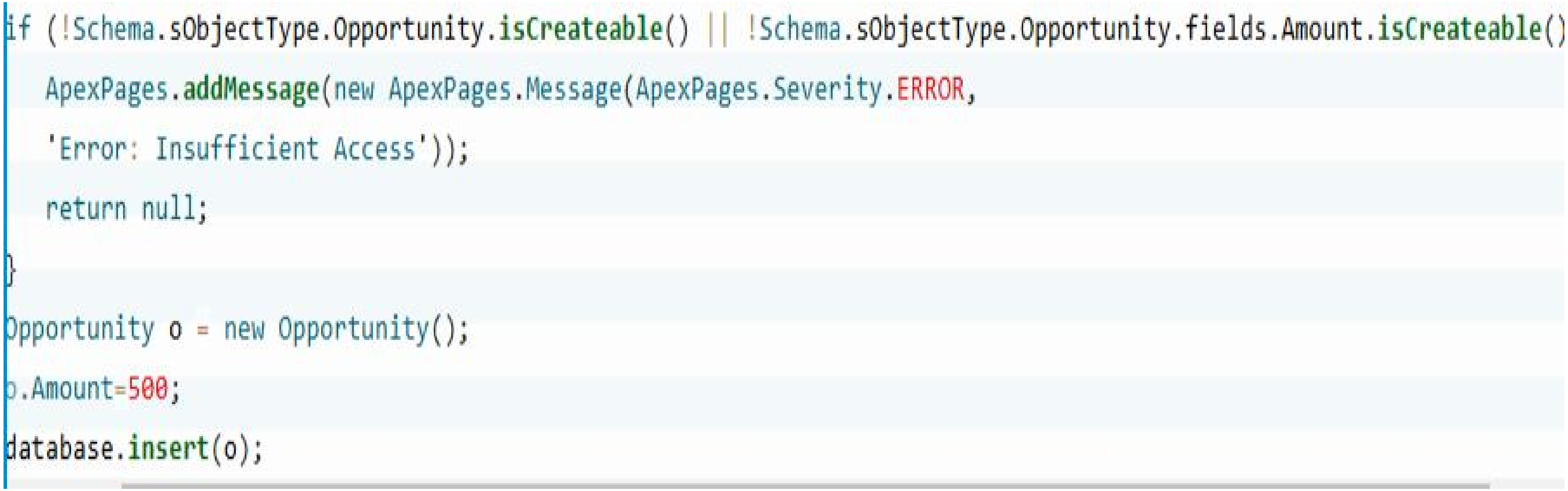
IsAccessible()

IsUpdateable()

IsDeleteable()

Using CRUD/FLS Check Methods

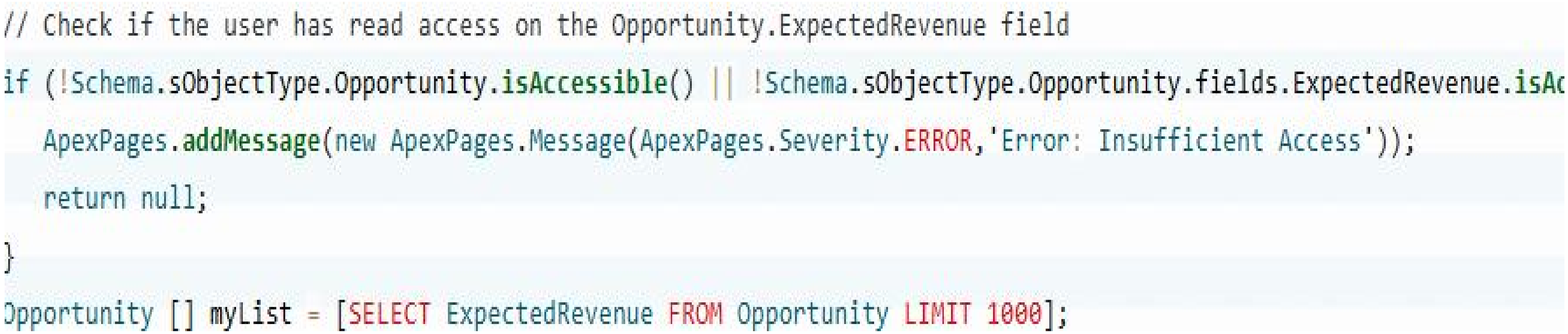
IsCreateable()



```
if (!Schema.sObjectType.Opportunity.isCreateable() || !Schema.sObjectType.Opportunity.fields.Amount.isCreateable()
    ApexPages.addMessage(new ApexPages.Message(ApexPages.Severity.ERROR,
    'Error: Insufficient Access'));
    return null;
}
Opportunity o = new Opportunity();
o.Amount=500;
database.insert(o);
```

Using CRUD/FLS Check Methods

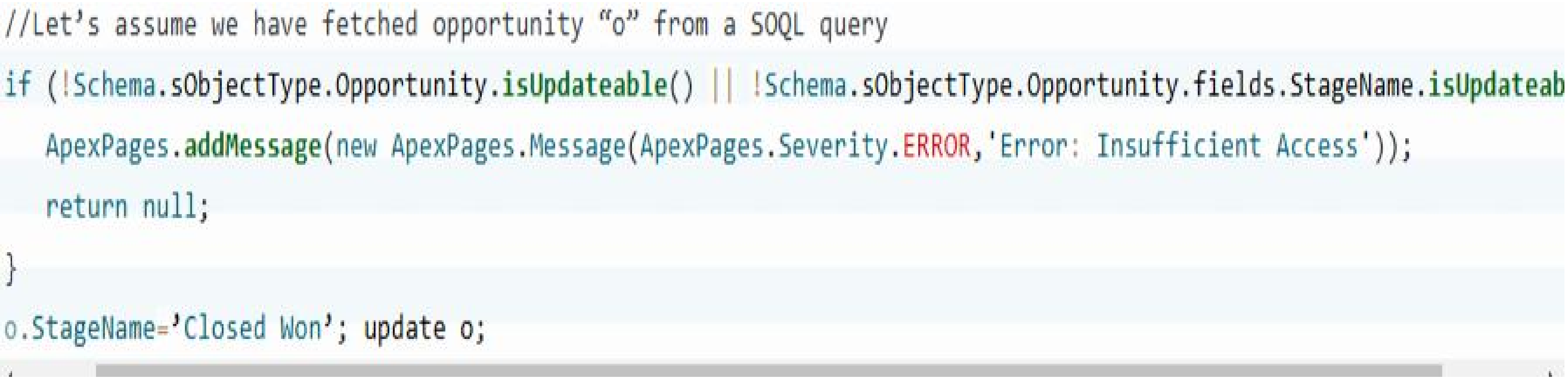
isAccessible()



```
// Check if the user has read access on the Opportunity.ExpectedRevenue field
if (!Schema.sObjectType.Opportunity.isAccessible() || !Schema.sObjectType.Opportunity.fields.ExpectedRevenue.isAc
    ApexPages.addMessage(new ApexPages.Message(ApexPages.Severity.ERROR,'Error: Insufficient Access'));
    return null;
}
Opportunity [] myList = [SELECT ExpectedRevenue FROM Opportunity LIMIT 1000];
```

Using CRUD/FLS Check Methods

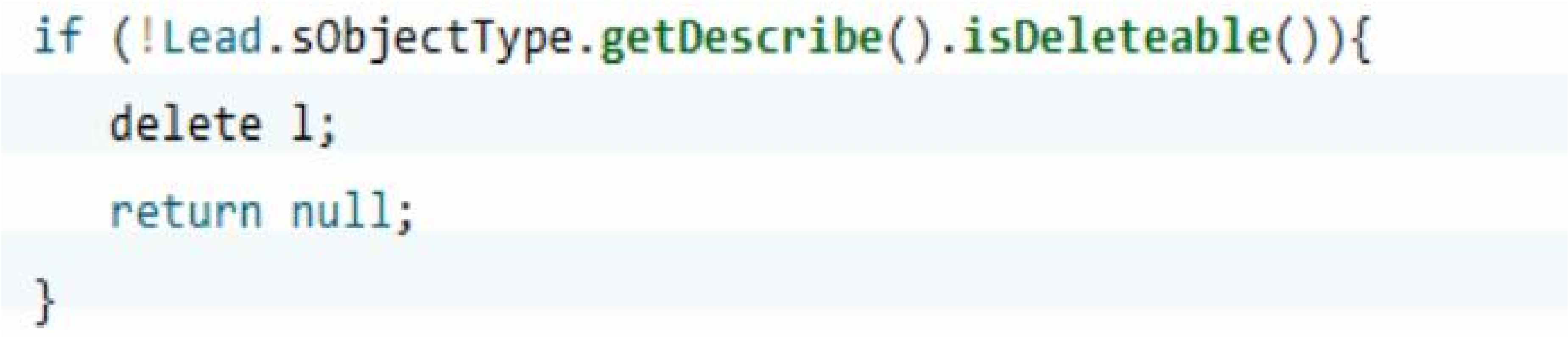
isUpdateable()



```
//Let’s assume we have fetched opportunity “o” from a SOQL query
if (!Schema.sObjectType.Opportunity.isUpdateable() || !Schema.sObjectType.Opportunity.fields.StageName.isUpdateab
   ApexPages.addMessage(new ApexPages.Message(ApexPages.Severity.ERROR,'Error: Insufficient Access'));
   return null;
}
o.StageName=’Closed Won’; update o;
```

Using CRUD/FLS Check Methods

isDeleteable()



```
if (!Lead.sObjectType.getDescribe().isDeleteable()){
    delete l;
    return null;
}
```

Mitigate SOQL Injection

2. Mitigate SOQL Injection

SOQL Injection Prevention

1. Static queries with bind variables
2. String.escapeSingleQuotes()
3. Type casting
4. Replacing characters
5. Allowlisting

2. Mitigate SOQL Injection

1. Static Query and Bind Variables

The first and most recommended method to prevent SOQL injection is to use static queries with bind variables. Consider the following query.

```
String query = ‘select id from contact where firstname =\’’+var+’\’’;
queryResult = Database.execute(query);
```

2. Mitigate SOQL Injection

1. Static Query and Bind Variables

As you’ve learned, using user input (the var variable) directly in a SOQL query opens the application up to SOQL injection. To mitigate the risk, translate the query into a static query like this:

```
queryResult = [select id from contact where firstname =:var]
```

2. Mitigate SOQL Injection

2. Typecasting

Another strategy to prevent SOQL injection is to use typecasting.

By casting all variables as strings, user input can drift outside of expectation.

By typecasting variables as integers or Booleans, when applicable, erroneous user input is not permitted

The variable can then be transformed back to a string for insertion into the query using **string.valueOf()** method

2. Mitigate SOQL Injection

2. Typecasting

```
public String textualAge {get; set;}
[...]
whereClause+='Age__c >'+textualAge+'';
whereclause_records = database.query(query+' where '+whereClause);
```

2. Mitigate SOQL Injection

After Typecasting

```
whereClause+='Age__c >'+string.valueOf(textualAge)+";
```

2. Mitigate SOQL Injection

3. Escaping Single Quotes

Another cross-site scripting (XSS) mitigation option that is commonly used by developers who include user-controlled strings in their queries is the platform-provided escape function **string.escapeSingleQuotes()**.

2. Mitigate SOQL Injection

3. Escaping Single Quotes

```
String query = 'SELECT Id, Name, Title__c FROM Books';
String whereClause = 'Title__c like \'%'+textualTitle+'%\' ';
List<Bookswhereclause_records = database.query(query+' where '+whereClause);
```

2. Mitigate SOQL Injection

3. Escaping Single Quotes

In the example, replacing the where clause with the following code wrapping **textualTitle with String.escapeSingleQuotes()** will prevent an attacker from using SOQL injection to modify the query behavior.

```
String whereClause = 'Title__c like \'%'+String.escapeSingleQuotes(textualTitle)+ '%\'
';
```

2. Mitigate SOQL Injection

4. Replacing Characters

A final tool in your tool belt is character replacement, also known as blocklisting.

This approach **removes “bad characters” from user input**.

2. Mitigate SOQL Injection

4. Replacing Characters

```
String query = 'select id from user where isActive='+var;
```

Example:

before replacing

true AND ReceivesAdminInfoEmails=true

after replacing

trueANDRecievesAdminInfoEmails=true

2. Mitigate SOQL Injection

4. Replacing Characters

The code to remove all spaces from a string can be written as follows:

```
String query = 'select id from user where isActive='+var.replaceAll('[^\\w]','');
```

2. Mitigate SOQL Injection

5. Allowlisting

Another way to prevent SOQL injection without string.escapeSingleQuotes() is allowlisting

Create a list of all “known good” values that the user is allowed to supply.

If the user enters anything else, you reject the response

Mitigate Cross-Site Request Forgery

What Is CSRF?

CSRF is a common web application vulnerability where a malicious application causes a user's client to perform an unwanted action on a trusted site for which the user is currently authenticated.

What Is CSRF?

Let’s start with the idea that we have built an application that lists all of the current students in our network of schools. In this application, there are two important things to note.

Only the admin or the superintendent can access the page allowing users to promote students to the honor roll.

The page automatically refreshes if you click the Honor Roll link. If you’ve added a student, an alert will be noted that your student has been added to the honor roll.

What Is CSRF?

What Is CSRF?

What is happening behind the scenes is that the Honor Roll button makes a GET request to /**promote?UserId=<userid>**

As the page loads, it reads the URL parameter value and automatically changes the role of that student to the honor roll

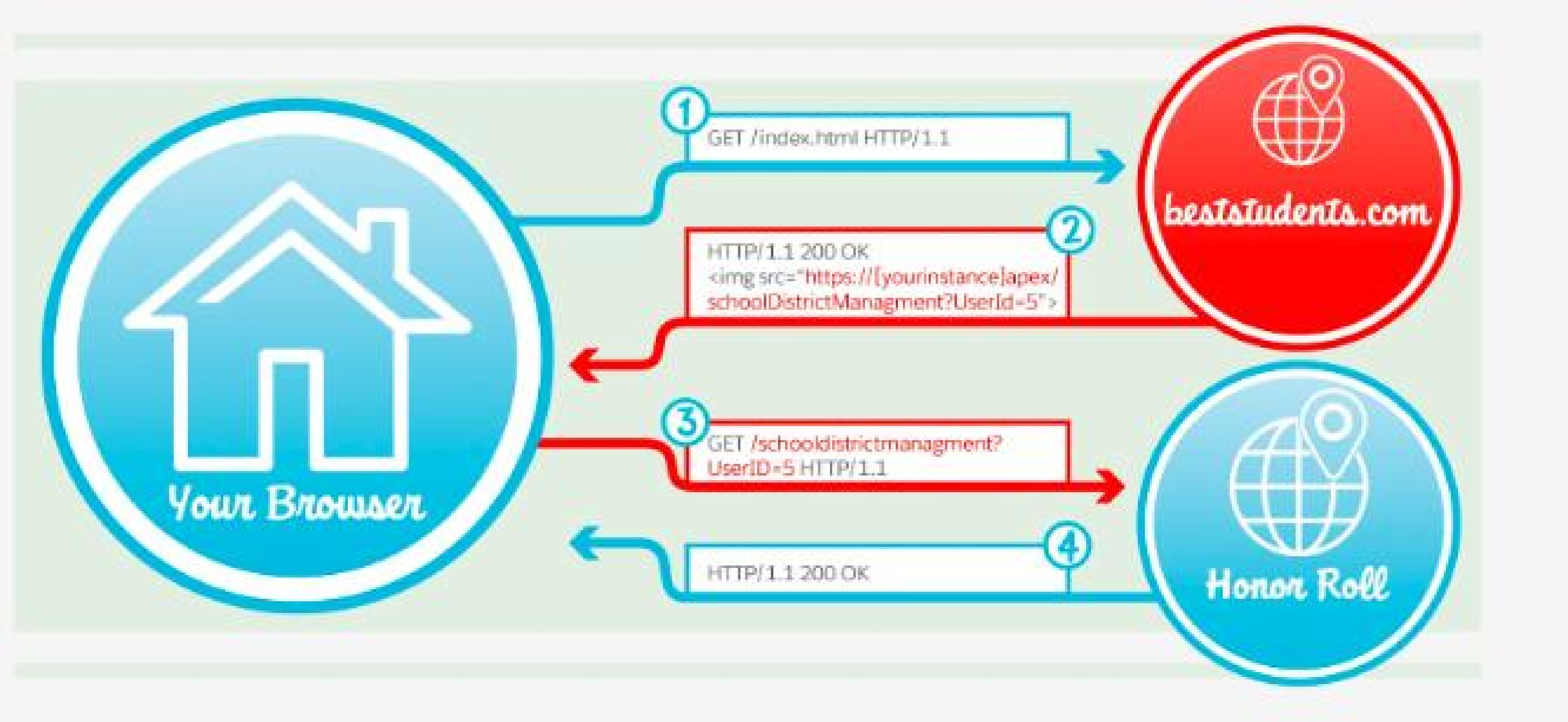
CSRF- Attack

This time, imagine that after logging in to your School District Management org, you decided to browse another website.

While on this website, you click a hyperlink. This hyperlink redirects to a link to www.beststudents.com/promote?user_id=123.

This malicious link is executed on behalf of the admin (your signed-in account), thereby promoting a student to the honor roll without you realizing it.

CSRF- Attack



1
```
GET /index.html HTTP/1.1
```

2
```
HTTP/1.1 200 OK
<img src="https://[yourinstance]apex/
schoolDistrictManagment?UserId=5">
```

3
```
GET /schooldistrictmanagment?
UserID=5 HTTP/1.1
```

4
```
HTTP/1.1 200 OK
```

Prevent CSRF Attacks

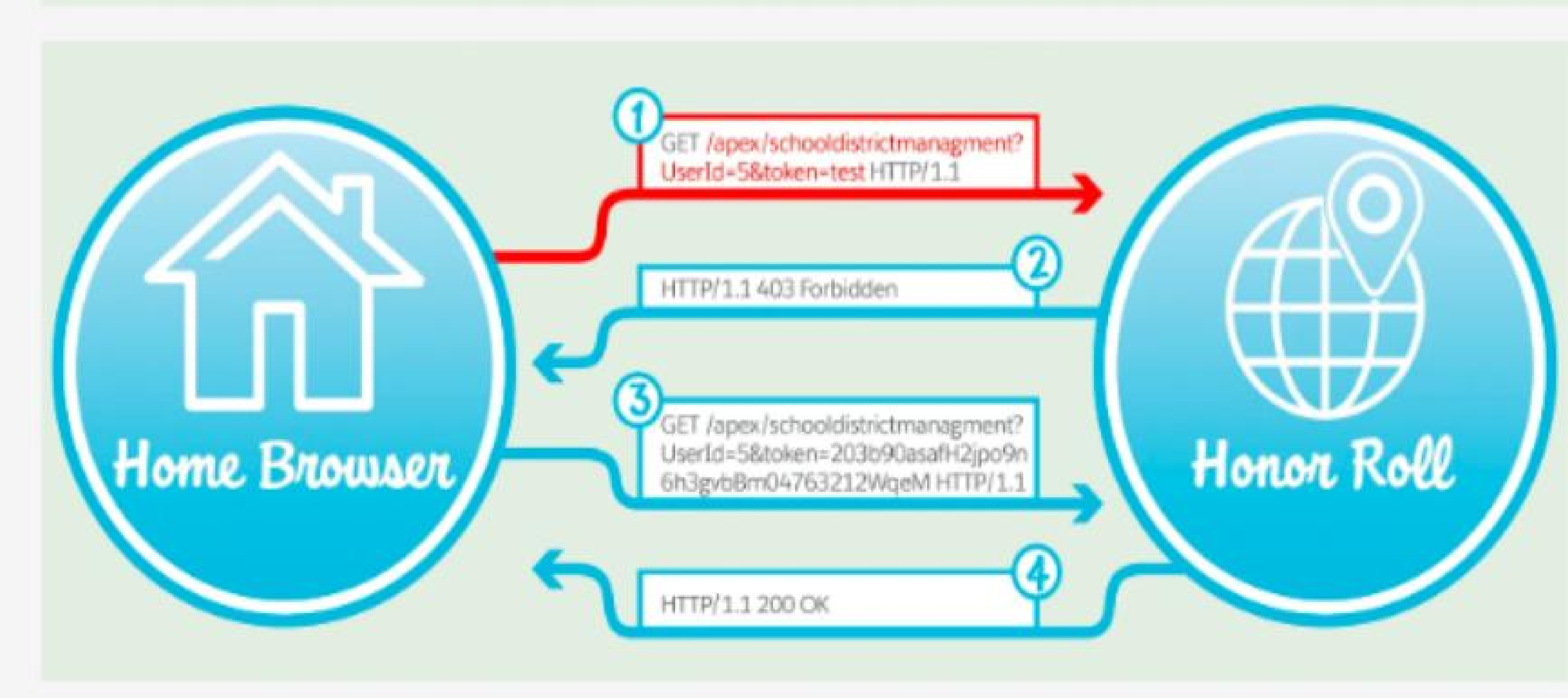
Consider a slightly different version of the page that has **two required URL parameters: userId and token**.

What if you made **the token parameter value a random, unique value that changed on every request**?

This would make it next to **impossible for an attacker to guess the current value, preventing the attack**.

This example is the most common prevention technique for CSRF.

Prevent CSRF Attacks



1. GET /apex/schooldistrictmanagment?
UserId=5&token=test HTTP/1.1

2. HTTP/1.1 403 Forbidden

3. GET /apex/schooldistrictmanagment?
UserId=5&token=203b90asafH2jpo9n
6h3gvbBm04763212WqeM HTTP/1.1

4. HTTP/1.1 200 OK

Prevent CSRF Attacks

For this prevention technique to be successful, four things must happen.

All sensitive state-changing requests (anything performing database operations) must include a token.

A token must be unique to the request or user’s session.

A token must be difficult to predict (long with advanced encryption).

The server must validate a token to ensure the request originated from the intended user

Mitigate Server Side Request Forgery

What Is Server Side Request Forgery?

Server-side request forgery (SSRF) is a security vulnerability in web applications where an attacker can make unauthorized requests, both internal and external, on behalf of the server.

In an SSRF attack, the malicious application tricks the server into making requests to internal and external services or systems, potentially leading to unauthorized access or data exposure

What Is Server Side Request Forgery?

The application is designed to make a GET request to the server whose address is contained in the API request to retrieve the student's details, for example

studentApi=https://192.168.0.1/student

An attacker would intercept the API call from the client, replace the endpoint value of the student service with a call to the metadata service and exfiltrate sensitive service configuration data from the internal metadata endpoint.

Preventing SSRF Attacks

Validate and Sanitize Inputs

Ensure that input values, such as the studentApivalue in our example, are properly validated and sanitized to prevent the injection of malicious URLs.

Implement Allowlisting

Restrict the allowed destinations for outgoing requests by enforcing the URL schema, port, and destination allowlist, disabling HTTP redirections. Only allow requests to specified, trusted endpoints

Preventing SSRF Attacks

Use URL Parsing Libraries

Utilize URL parsing libraries to parse and validate URLs before making requests. This helps ensure that the requested URLs conform to expected patterns.

Network Segmentation

Implement network segmentation to restrict the server's ability to make requests to internal resources, limiting the impact of any potential SSRF attacks

Salesforce Platform Protections Against SSRF

Avoid GET Requests

Similar to CSRF prevention, developers should avoid using HTTP GET requests. Instead, prefer using POST or PUT requests to minimize risk of SSRF data exfiltration.

Validate Origin Headers

When integrating Salesforce Lightning applications with third-party APIs, validate the origin header in HTTP requests. Ensure that the request originates from a trusted source to prevent potential SSRF exploits.

Implement Anti-SSRF Tokens

Developers can add custom anti-SSRF tokens to XMLHttpRequests within Lightning by using setRequestHeader(). This adds an additional layer of protection against SSRF attacks