MALLA REDDY UNIVERSITY

R-22

III YEAR B.TECH. (CSE) / I – SEM

MR22-1CS0155

Salesforce Platform Developer

UNIT–V

Apex Integration. & Deployment.

Project with Case Study: Hands on real world Projects and case study on Travel App, Dream house App etc.

Apex Integration - steps to add remote sites to the remote site settings to allow callouts to external sites.

Apex callouts come in two flavors:

Web service callouts to SOAP web services use XML, and typically require a Web Services Description Language (WSDL) document for code generation.

HTTP callouts to services typically use REST with JSON.

- These two types of callouts are similar in terms of sending a request to a service and receiving a response.
- But while WSDL-based callouts apply to SOAP Web services, HTTP callouts can be used with any HTTP service, either SOAP or REST.
- HTTP services are typically easier to interact with, require much less code, and utilize easily readable JSON.

Apex Integration - steps to add remote sites to the remote site settings to allow callouts to external sites.

Authorize Endpoint Addresses:

- Any time you make a callout to an external site we want to make sure that it is authorized.
- We can't have code calling out to any endpoint without prior approval.
- Before you start working with callouts, update the list of approved sites for your org on the Remote Site Settings page.
- We'll be making calls to the following sites.

https://th-apex-http-callout.herokuapp.com

https://th-apex-soap-service.herokuapp.com

Apex Integration - steps to add remote sites to the remote site settings to allow callouts to external sites.

Authorize both of these endpoint URLs by following these steps.

1. From **Setup**, enter **Remote Site Settings** in the Quick Find box, then click **Remote Site Settings.**
2. Click **New Remote Site**.
3. For the remote site name, enter **animals_http**.
4. For the remote site URL, enter **https://th-apex-http-callout.herokuapp.com**
5. This URL authorizes all subfolders for the endpoint, like

   https://th-apex-http-callout.herokuapp.com/path1 and

   https://th-apex-http-callout.herokuapp.com/path2

Apex Integration - steps to add remote sites to the remote site settings to allow callouts to external sites.

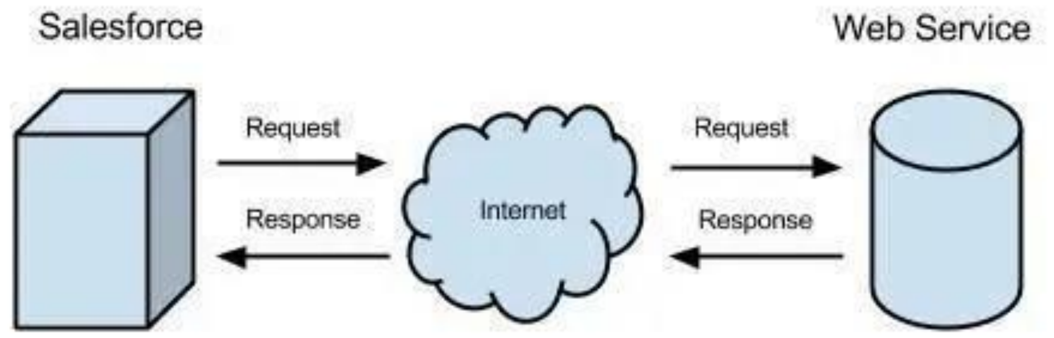
6. For the **description**, enter **Trailhead animal service: HTTP**.
7. Click **Save & New**.
8. For the **second remote site name**, enter **animals_soap**.
9. For the remote site URL, enter **https://th-apex-soap-service.herokuapp.com**
10. For the **description**, enter **Trailhead animal service: SOAP**.
11. Click **Save**.

Apex Integration

Apex REST Callouts:

HTTP and Callout Basics:

REST callouts are based on HTTP. To understand how callouts work, it's helpful to understand a few things about HTTP. Each callout request is associated with an HTTP method and an endpoint. The HTTP method indicates what type of action is desired.



Apex Integration

- The simplest request is a **GET request** (GET is an HTTP method). A **GET request** means that the **sender wants to obtain information** about a resource from the server.
- When the server receives and processes this request, it returns the request information to the recipient. A GET request is similar to navigating to an address in the browser.
- When you visit a web page, the browser performs a GET request behind the scenes. In the browser, the result of the navigation is a new HTML page that’s displayed.
- With a callout, the result is the response object.

Apex Integration

The following are descriptions of common HTTP methods.

HTTP Method	Description
GET	Retrieve data identified by a URL.
POST	Create a resource or post data to the server.
DELETE	Delete a resource identified by a URL.
PUT	Create or replace the resource sent in the request body.

Apex Integration

- In addition to the HTTP method, each request sets a URI, which is the endpoint address at which the service is located.
- For example, an endpoint can be **http://www.example.com/api/resource**
- In the example in the “HTTP and Callout Basics” unit, the **endpoint** is **https://th-apex-http-callout.herokuapp.com/animals**
- When the server processes the request, it sends a status code in the response. The status code indicates whether the request was processed successfully or whether errors were encountered.
- If the **request is successful**, the server sends a status code of **200.** You’ve probably seen some other status codes, such as 404 for file not found or 500 for an internal server error.

Apex Integration

- In addition to the endpoint and the HTTP method, you can set other properties for a request.
- **For example**, a request can contain headers that provide more information about the request, such as the content type.
- A request can also contain data to be sent to the service, such as for POST requests.
- A POST request is similar to clicking a button on a web page to submit form data.
- With a callout, you send data as part of the body of the request instead of manually entering the data on a web page.

Create an Apex REST class that contains methods for each HTTP method (GET, POST)

```
public class AnimalsCallouts {
    public static HttpResponse makeGetCallout() {
        Http http = new Http();
        HttpRequest request = new HttpRequest();
        request.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals');
        request.setMethod('GET');
        HttpResponse response = http.send(request);
if(response.getStatusCode() == 200) {
Map<String,          Object>          results          =          (Map<String,          Object>)
JSON.deserializeUntyped(response.getBody());
           List<Object> animals = (List<Object>) results.get('animals');
            System.debug('Received the following animals:');
            for(Object animal: animals) {
                System.debug(animal);            }        }
        return response;    }
```

Create an Apex REST class that contains methods for each HTTP method (GET, POST)

```
public static HttpResponse makePostCallout() {
    Http http = new Http();
    HttpRequest request = new HttpRequest();
    request.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals');
    request.setMethod('POST');
    request.setHeader('Content-Type', 'application/json;charset=UTF-8');
    request.setBody('{"name":"mighty moose"}');
    HttpResponse response = http.send(request);
      if(response.getStatusCode() != 201) {
      System.debug('The status code returned was not expected: ' +
        response.getStatusCode() + ' ' + response.getStatus());
    } else {
      System.debug(response.getBody());       }
    return response;    }}
```

Comparison Table: Web Services vs HTTP Callouts

Aspect	**Web Services**	**HTTP Callouts**
Definition	A way to expose Salesforce functionality to external systems or consume external services.	A mechanism to make HTTP requests to external services/APIs.
Direction	Can be both outbound (consume external services) and inbound (expose Salesforce data).	Outbound only (Salesforce sends requests to external systems).
Protocols	SOAP (XML-based), REST (JSON/XML-based)	REST (JSON/XML-based), SOAP (XML-based)
Usage	Exposing Salesforce data and methods to external systems, or consuming external web services in Salesforce.	Sending HTTP requests from Salesforce to external APIs or systems (e.g., payment gateways, external databases).
Common Use Cases	- Exposing Salesforce data to external systems. - Integrating with external CRMs or services.	- Interacting with third-party APIs. - Integrating with external services like payment gateways, social media, etc.
Salesforce Classes	@WebService, @RestResource for exposing services.	Http, HttpRequest, HttpResponse for sending HTTP requests.
Authentication	SOAP or REST APIs often use authentication tokens, certificates, or API keys to secure communication.	Typically uses API keys, OAuth, or session-based authentication for securing requests.

REST and SOAP Web Services Explained

Web services are a way for different applications to communicate with each other over the internet. Two of the most widely used types of web services are **REST** (Representational State Transfer) and **SOAP** (Simple Object Access Protocol). Both are protocols or architectural styles that allow applications to send and receive data, but they do so in different ways, with different advantages and trade-offs.

1. SOAP Web Services

SOAP (Simple Object Access Protocol) is a protocol that defines a set of rules for structuring messages. It is an XML-based messaging protocol used to send data between clients and web services. SOAP web services rely on XML format for both request and response messages.

Key Characteristics of SOAP:

XML-based: SOAP messages are formatted in XML, which makes it platform-agnostic.

Strict Standards: SOAP has a strict message format and relies on a set of rules, such as message encoding and security.

Works over Multiple Protocols: SOAP can work over multiple transport protocols such as HTTP, SMTP, TCP, etc., but it is most commonly used with HTTP/HTTPS.

Reliability and Security: SOAP supports built-in standards for security (WS-Security), transactions, and message reliability, which make it more suitable for enterprise-level applications.

REST and SOAP Web Services Explained

Example of a SOAP Web Service

Let's say you are building a service that allows users to fetch account information by providing an account ID. The SOAP web service might expose a method like getAccountInfo.

SOAP Request (XML format):

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/
"
    xmlns:web="http://www.example.com/webservice">
   <soapenv:Header/>
   <soapenv:Body>
      <web:getAccountInfo>
         <web:accountId>12345</web:accountId>
      </web:getAccountInfo>
   </soapenv:Body>
</soapenv:Envelope>
```

SOAP Response (XML format):

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"

xmlns:web="http://www.example.com/webservice">
   <soapenv:Header/>
   <soapenv:Body>
      <web:getAccountInfoResponse>
         <web:accountName>Acme Corp</web:accountName>
         <web:accountBalance>5000</web:accountBalance>
      </web:getAccountInfoResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

REST and SOAP Web Services Explained

2. REST Web Services

REST (Representational State Transfer) is an architectural style that defines a set of constraints for creating web services. RESTful web services use standard HTTP methods (GET, POST, PUT, DELETE) and are typically lightweight, using formats such as **JSON** or **XML** for transmitting data.

Key Characteristics of REST:

HTTP-based: RESTful web services use HTTP methods (GET, POST, PUT, DELETE, PATCH) to perform CRUD operations on resources (data).

Stateless: Each REST request from a client to a server must contain all of the information the server needs to fulfill the request. The server does not store any state between requests.

Lightweight: REST is usually simpler, more flexible, and faster compared to SOAP, especially when JSON is used as the data format.

Resource-Oriented: In REST, resources (such as accounts, products, or customers) are identified by URLs, and operations are performed on these resources.

REST and SOAP Web Services Explained

Example of a REST Web Service
Let’s say we are building a RESTful web service to fetch account information.

REST API Endpoint:
GET /api/accounts/{accountId}

This endpoint retrieves account information by the account ID.

REST Request (HTTP GET):
GET /api/accounts/12345

REST Response (JSON format):
```
{
  "accountId": "12345",
  "accountName": "Acme Corp",
  "accountBalance": 5000
}
```

Role of @HttpGet, @HttpPost, @HttpPut, and @HttpDelete annotations in Apex REST

Annotation	Action	Details
@HttpGet	Read	Reads or retrieves records.
@HttpPost	Create	Creates records.
@HttpDelete	Delete	Deletes records.
@HttpPut	Upsert	Typically used to update existing records or create records.

@HttpGet :

The @HttpGet annotation in Apex is used to define a method that handles HTTP **GET** requests for your Apex REST web services.

This annotation tells Salesforce that the method will respond to GET requests, typically used for **retrieving data** from Salesforce.

When you annotate a method with @HttpGet, Salesforce will route any incoming GET requests to that method.

This is commonly used for reading records or returning information, and it maps to the standard HTTP method used to fetch data.

Role of @HttpGet, @HttpPost, @HttpPut, and @HttpDelete annotations in Apex REST

Example: Apex REST with @HttpGet

Let's say we want to expose an API endpoint that allows external clients to retrieve the details of a specific Account by its Id.

Apex Class with @HttpGet:

```
@RestResource(urlMapping='/accounts/*')
global with sharing class AccountRESTService {

    // Handle GET requests to retrieve an Account by its ID
    @HttpGet
    global static Account doGet() {
        // Extract the account ID from the URL path
        RestRequest req = RestContext.request;
        String accountId = req.requestURI.substring(req.requestURI.lastIndexOf('/') + 1);
                // Query Salesforce for the Account using the provided ID
        Account acc = [SELECT Id, Name, Phone, Website FROM Account WHERE Id = :accountId LIMIT 1];
                // Return the Account object
        return acc;
    }    }
```

Role of @HttpGet, @HttpPost, @HttpPut, and @HttpDelete annotations in Apex REST

@HttpPost :

The @HttpPost annotation in Apex is used to define a method that handles HTTP POST requests for your Apex REST web services. A POST request is typically used to create new resources or submit data to be processed, such as creating a new Salesforce record or submitting form data to an external system.In Apex, when you use @HttpPost, you are defining an endpoint where external clients (or other systems) can send data to create new records in Salesforce or trigger some action. This is commonly used for creating records, processing information, or interacting with external systems.

Example: Apex REST with @HttpPost We'll create an endpoint that allows external systems to create a new Account in Salesforce by sending data such as the Account's Name, Phone, and Website.

Apex Class with @HttpPost:

```
@RestResource(urlMapping='/accounts')
global with sharing class AccountRESTService {
    // Handle POST requests to create a new Account
  @HttpPost
  global static String doPost(String name, String phone, String website) {
    // Create a new Account object with the provided data
    Account newAccount = new Account(
      Name = name,
      Phone = phone,
      Website = website
    );
    // Insert the new Account into Salesforce
    insert newAccount;
        // Return the ID of the newly created Account
    return newAccount.Id;  }}
```

Role of @HttpGet, @HttpPost, @HttpPut, and @HttpDelete annotations in Apex REST

@HttpPut :

The @HttpPut annotation in Apex is used to define a method that handles HTTP PUT requests in an Apex RESTful web service. The HTTP PUT method is typically used for updating an existing resource on the server. In the context of Salesforce, you can use @HttpPut to update existing Salesforce records via a REST API.When you use @HttpPut, the method is invoked when an external system sends a PUT request to the endpoint. This request typically includes the ID of the record being updated, along with the new data for the fields that need to be changed.

Example:

Apex Class with @HttpPut:

```
@RestResource(urlMapping='/accounts/*')
global with sharing class AccountRESTService {
    // Handle PUT requests to update an Account by its ID
  @HttpPut
  global static String doPut(String accountId, String name, String phone, String website) {
    // Query Salesforce for the Account record using the provided accountId
    Account accToUpdate = [SELECT Id, Name, Phone, Website FROM Account WHERE Id = :accountId LIMIT 1];
        // Update the fields with the new data
    accToUpdate.Name = name;
    accToUpdate.Phone = phone;
    accToUpdate.Website = website;
        // Update the Account record in Salesforce
    update accToUpdate;
        // Return the ID of the updated Account
    return accToUpdate.Id;    }}
```

Role of @HttpGet, @HttpPost, @HttpPut, and @HttpDelete annotations in Apex REST

@HttpDelete :

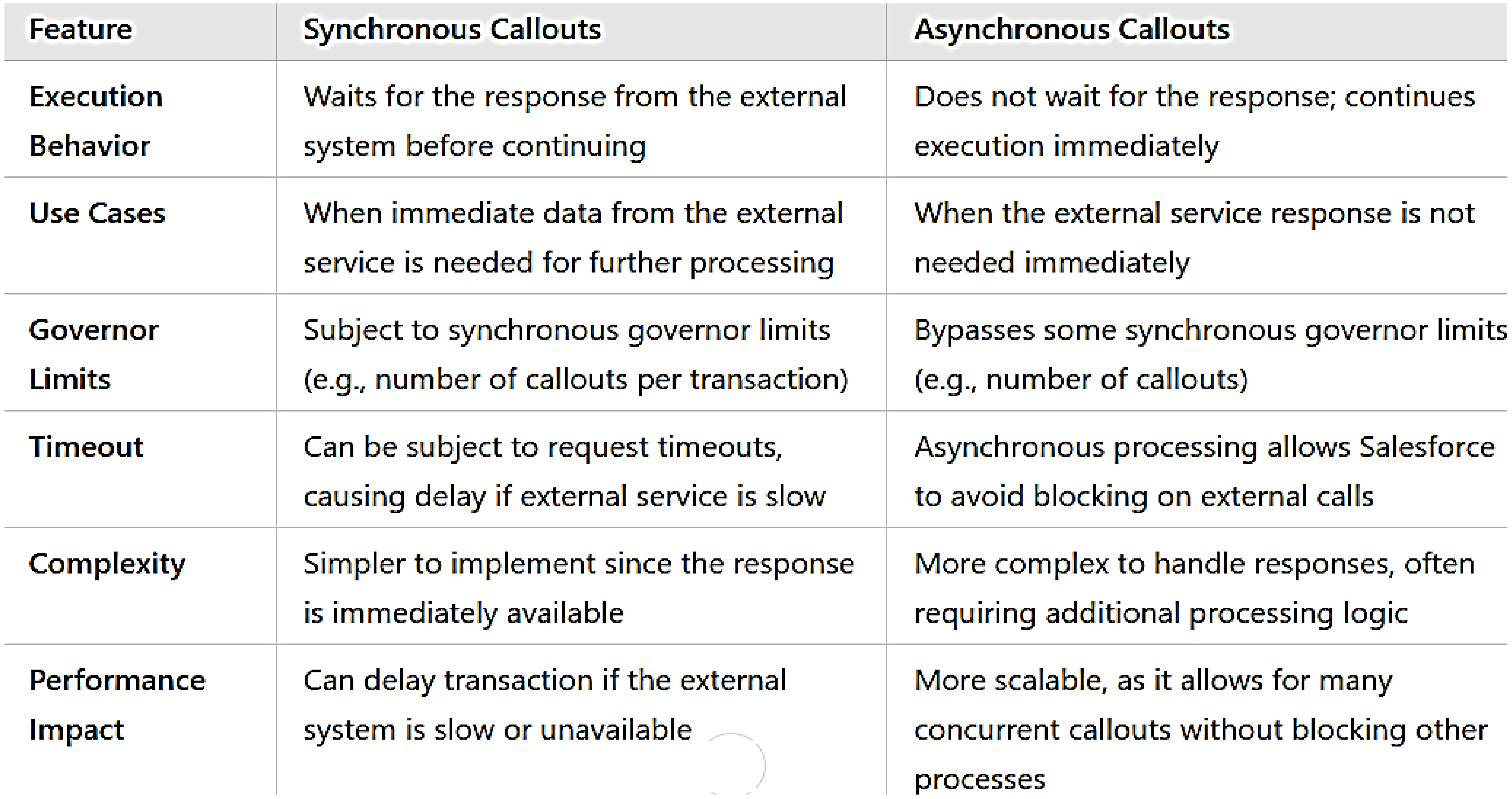
The @HttpDelete annotation in Apex is used to handle HTTP DELETE requests in Apex REST web services. The HTTP DELETE method is typically used to delete an existing resource on the server. In the context of Salesforce, you can use the @HttpDelete annotation to delete records such as Account, Contact, or any other Salesforce object via a RESTful API.When you use @HttpDelete, the method is invoked when an external system sends a DELETE request to the endpoint. This request typically includes the ID of the record to be deleted.

Example:

Apex Class with @HttpDelete :

```
@RestResource(urlMapping='/accounts/*')
global with sharing class AccountRESTService {
    // Handle DELETE requests to delete an Account by its ID
  @HttpDelete
  global static String doDelete() {
    // Extract the AccountId from the URL
    RestRequest req = RestContext.request;
    String accountId = req.requestURI.substring(req.requestURI.lastIndexOf('/') + 1);
        // Query Salesforce for the Account to be deleted
    Account accToDelete = [SELECT Id FROM Account WHERE Id = :accountId LIMIT 1];
        // Delete the Account record from Salesforce
    delete accToDelete;
        // Return a success message
    return 'Account with ID ' + accountId + ' has been deleted.';    }}
```

Difference between synchronous and asynchronous callouts in Apex



Feature	Synchronous Callouts	Asynchronous Callouts
Execution Behavior	Waits for the response from the external system before continuing	Does not wait for the response; continues execution immediately
Use Cases	When immediate data from the external service is needed for further processing	When the external service response is not needed immediately
Governor Limits	Subject to synchronous governor limits (e.g., number of callouts per transaction)	Bypasses some synchronous governor limits (e.g., number of callouts)
Timeout	Can be subject to request timeouts, causing delay if external service is slow	Asynchronous processing allows Salesforce to avoid blocking on external calls
Complexity	Simpler to implement since the response is immediately available	More complex to handle responses, often requiring additional processing logic
Performance Impact	Can delay transaction if the external system is slow or unavailable	More scalable, as it allows for many concurrent callouts without blocking other processes