REST API in APEX

Architectural constraints/Characteristics

Stateless

Each request from client to server must contain all the information necessary to understand the request, and not use any stored context on the server.

Caching behavior

Responses are labeled as cacheable or non-cacheable.

Uniform interface

All resources are accessed with a generic interface over HTTP.

Named resources

All resources are named using a base URI that follows your Lightning Platform URI.

Layered components

allows for the existence of such intermediaries as proxy servers and gateways to exist between the client and the resources.

Authentication

Uses Oath 2.0.

Support for JSON and XML

JSON is the default. You can use the HTTP ACCEPT header to select either JSON or XML, or append json or xml to the URI

Compression

The REST API allows the use of compression on the request and the response, using the standards defined by the HTTP 1.1 specification. Compression is automatically supported by some clients, and can be manually added to others. For better performance.

Request compression: Add Accept-Encoding: gzip or Accept-Encoding: deflate in a request Header.

Response compression: in response header Content-Encoding: gzip/Content-Encoding: deflate

Authorization Through Connected Apps and OAuth 2.0

A connected app requests access to REST API resources on behalf of the client application.

OAuth 2.0 is an open protocol that authorizes secure data sharing between applications through the exchange of tokens.

OAuth authorization flows grant a client app restricted access to REST API resources on a resource server. Each OAuth flow offers a different process for approving access to a client app, but in general the flows consist of three main steps.

- 1. a connected app, on behalf of a client app, requests access to a REST API resource.
- 2. In response, an authorizing server grants access tokens to the connected app.
- 3. A resource server validates these access tokens and approves access to the protected REST API resource.

Perform Cross-Origin Requests from Web Browsers

Suppose a user visits http://www.example.com and the page attempts a cross-origin request to fetch the user's data from http://service.example.com. A CORS-compatible browser will attempt to make a cross-origin request to service.example.com.

In Salesforce, add the origin serving the code to a CORS allowlist.

- 1. enter CORS in the Quick Find box, then select CORS
- 2. New
- **3.** Enter a URL Pattern

You can add (*) wildcard at 2nd level domain, Example: https://*.example.com.

If a browser that supports CORS makes a request to an origin in the allowlist, Salesforce returns the origin in the **Access-Control-Allow-Origin** HTTP header along with any additional CORS HTTP headers. If the origin isn't included in the allowlist, Salesforce returns HTTP status code 403.

Connected App and OAuth Terminology

https://help.salesforce.com/articleView?id=sf.remoteaccess_terminology.htm&type=5

Access Token: Instead of using the user's Salesforce credentials, a consumer (connected app) can use an access token to gain access to protected resources on behalf of the user. For OAuth 2.0, the access token is a session ID and can be used directly.

Authorization Code: In Oath 2.0, The authorization code is used to obtain an access token and a refresh token. It expires after 15 minutes.

Authorization Server: server that authorizes a resource owner, and upon successful authorization, issues access tokens to the requesting consumer.

Callback URL: A callback URL is the URL that is invoked after OAuth authorization for the consume.

OAuth Endpoint: OAuth endpoints are the URLs that you use to make OAuth authorization requests to Salesforce.

Consumer: A consumer is the website or app that uses OAuth to authorize both the Salesforce user and itself on the user's behalf. Referred to as client in OAuth 2.0.

Consumer Key: A consumer uses a key to identify itself to Salesforce. Referred to as client_id in OAuth 2.0.

Consumer Secret : A consumer uses a secret to establish ownership of the consumer key. Referred to as client_secret in OAuth 2.0.

Refresh Token : Only used in OAuth 2.0, a consumer can use a refresh token to obtain a new access token, without having the end user approve the access again.

Resource Owner: The resource owner is the entity (usually the end user) that grants access to a protected resource.

Resource Server : The resource server is the server that hosts the protected resource. Your Salesforce org is the resource server that protects your data.

Token Secret: A consumer uses this secret to establish ownership of a given token, both for request tokens and access tokens.

Connected Apps

A connected app is a framework that enables an external application to integrate with Salesforce using APIs and standard protocols, such as Security Assertion Markup Language (SAML), OAuth, and OpenID Connect. Connected apps use these protocols to authorize, authenticate, and provide single sign-on (SSO) for external apps.

Developers create and configure authorization flows for connected apps, and admins set policies and permissions to control connected app usage.

- **1. Access Data with API Integration**: web-based or mobile applications that need to pull data from your Salesforce org, you can use connected apps as the clients to request this data. Use OAuth 2.0 **Security**.
 - a. Web Server integration Web -> connected app -> Salesforce
 - b. Mobile integration -> mobile to salesforce/ can use mobile SDK instead of connected app.
 - c. Server to Server integration -> no need to be real time, can do authorization and authentication in advance by sending JSON WEB TOKEN or JWT
 - d. IOT integration -> for devices like TV, having limited input capabilities.

2. Integrate Service Providers with Salesforce: SSO

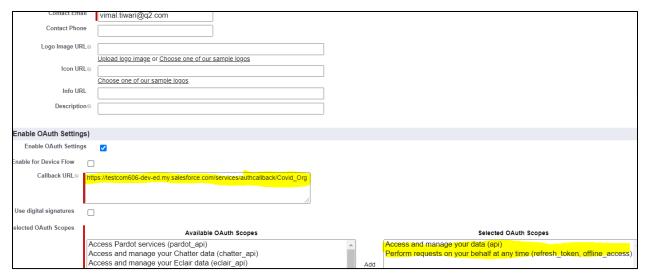
- a. You want your users to be able to log in to their app with their Salesforce credentials. Using SAML 2.0 for user **authentication**.
- b. OpenID Connect is a protocol that enables SSO between two services, like SAML2.0. It adds an authentication layer on top of OAuth 2.0 to enable secure exchange of ID tokens that contain user information alongside OAuth access tokens.
- **3. Provide Authorization for External API Gateways:** Using OpenID Connect dynamic client registration, resource servers can dynamically create client apps as connected apps in Salesforce, by sending request to authorization server. The authorization server

verifies the resource server's request and creates the connected app, giving it a unique client ID and client secret.

Basic Rest Based data movement Demo using Connected App and OAuth2.0

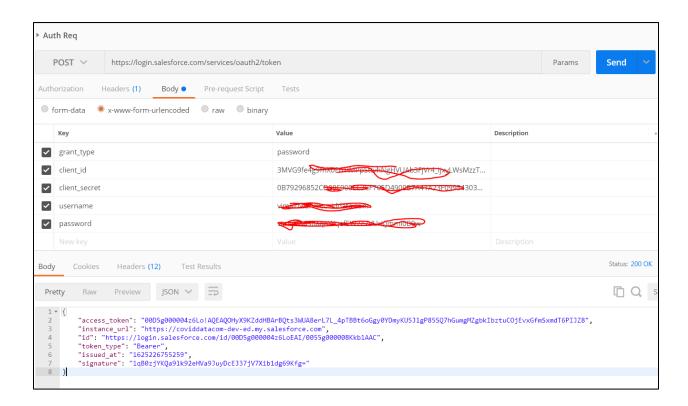
Steps for Org 1 acting as Server.

- 1. In response Server i.e. your org providing service, create Apex class for <u>Account Rest Services</u>
- 2. Create a connected App there.



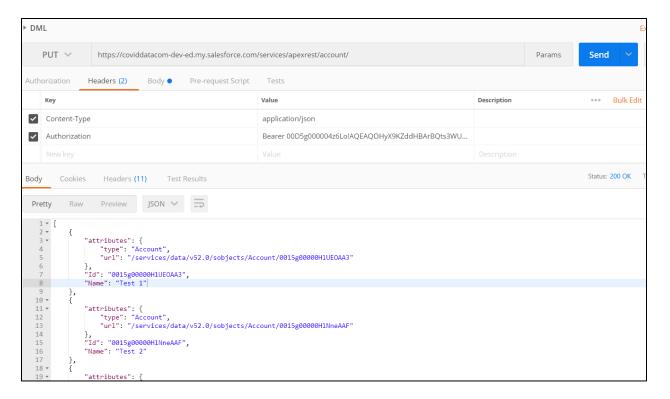
3. Use <u>Basic Tutorial of Rest API with Oauth2.0</u> for creation and testing Rest Services.

Auth:



Rest Testing

In authorization: Bearer <access-token>// bearer is a prefix



Steps for Org 2 acting as Client

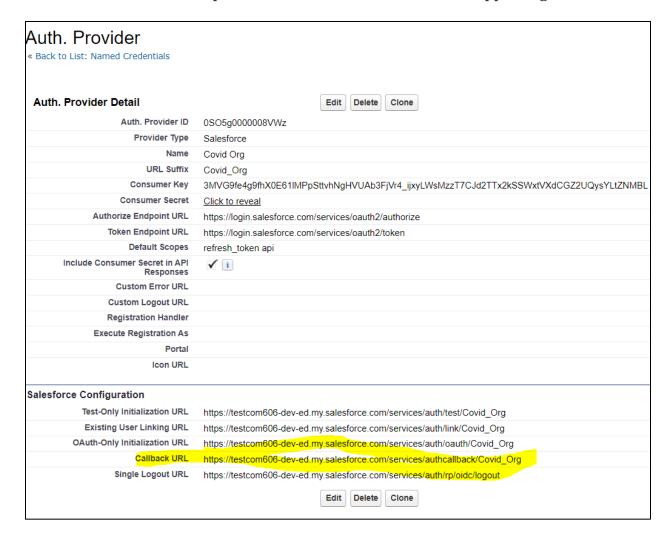
- 1. Create Authorization Provider req. in Named Credentials
 - a. Setup -> Auth. Provider
 - b. Default scope refresh_token full

refresh_token means – It will get updated token each time the existing token expires for authentication.

full means - full access,

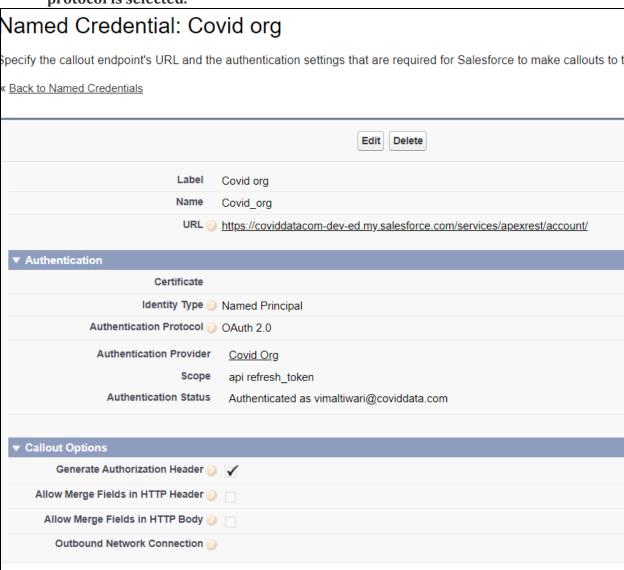
Api - Api access.

Call back url - Need to be updated in call back field of connected app in org1.

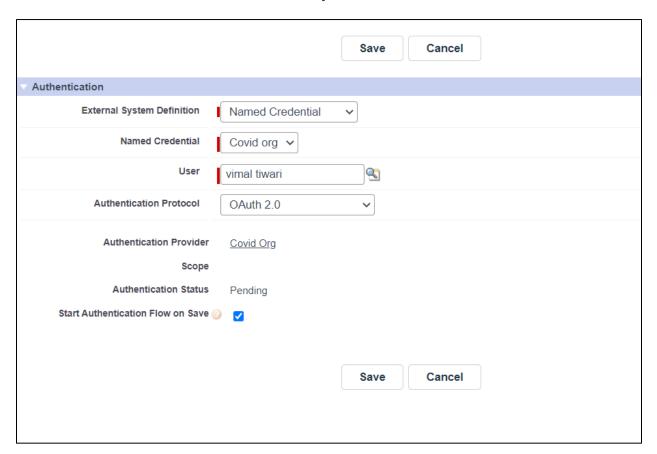


2. Create Named Credentials

- a. Provide Auth. Provider for Oauth 2.0
- b. Identity type
 - i. Per user Api callout based on calling user's context
 - ii. Named Principal One user will be used for all callouts, might be the owner.
- c. URL: the base url.
- d. Authentication protocol **if it is selected, that means system will take care of authentication.** No need to do callout for authentication explicitly.
- e. Authentication status should not be pending. If it is there, setup is not complete.
- f. Generate authorization header need to be checked, if any Authentication protocol is selected.



- 3. Give Access of Named Credential to Users Need in case of **Per user** setting. I have used Permission set.
- 4. Authentication Settings for External Systems **Needed in Per user setting**.
 - a. Go to my settings -> personal -> Authentication Settings for External Systems
 - b. Click new, add NC as external system definition.



It's time to write some apex code and do some fun.
 Create new class <u>AccountRestClientWithOAuth2</u> and use its methods to get and put data.

For custom Authorization and callouts:

- 1. Create remote site setting for base URL.
- 2. AccountRestClient
 - a. Call setAccessToken to get access token and set it in **platform cache and set time to live**. It is a field in seconds which can save the token expire time, based on server's details. Ex. 10 min, 5 min etc.
 - b. For each callout, check if token is there, then go for other methods

c. Else refresh -> call the setAccessToken again and get new token, save it again in cache.

Reference:

 $\frac{https://www.jitendrazaa.com/blog/salesforce/login-to-salesforce-from-salesforce-using-authentication-provider/\#more-4516$

https://developer.salesforce.com/docs/atlas.enus.apexcode.meta/apexcode/apex callouts named credentials.htm

 $\underline{https://sales force.stack exchange.com/questions/102484/use-of-named-credentials-seem-to-be-tied-to-external-data-source}$

Integration Framework

Overview: Since from UI, we can do real time callouts, because we can prevent user actions, for process based callouts, can't do it real time. It can be near real time. The framework takes callout request from any context and saves it. Then there will be some process which takes the request Queue and process it.

Classes Skeleton

```
global class ResponseWrapper{
      global List<SubResponseWrapper> jsonResponses; // bulk Responses
      global String returnCode;
global class RequestWrapper{
      global String endpoint;
      global List< SubRequestWrapper > bodies;//bulk Request
global abstract class AbstractHTTPServiceClass{
      global abstract ResponseWrapper process(String httpType, RequestWrapper rw);
}
public class SampleHTTPServiceClass extends AbstractHTTPServiceClass {
      public override ResponseWrapper process (String methodNam, RequestWrapper rw){
             if(methodName == 'getRecord'){
                    return getRecord(rw);
             }else if .....
      }
       @httpGet
      public ResponseWrapper getRecord(RequestWrapper rw){
             Http h =new Http();
             HttpRequest req = new HttpRequest();
             req.setEndpoint(rw.endpoint);
             HttpResponse\ res = h.send(req);
             return new ResponseWrapper(res.getBody());
      }
}
```

Data model

- 1. **Integration Log** to save the details of an integration callout that is to be called.
 - a. Service config Unique name of corresponding service meta data record.
 - i. Can create String like "Service.ClassName.Method" etc
 - b. Log Status New, In Progress, Success, Failed.
 - c. Retry count.
 - d. Status code
 - e. Child records Notes and Attachment
 - i. Save Request as file Request in Json converted in file
 - ii. Save Response as file Response in Json converted in file
- 2. **Service Configuration** Custom Meta data to save callout information.
 - a. Callout Service Class Name
 - b. Callout method name Method name
 - c. END-POINT Named credential name.
 - d. Relative URL service URL that gets added after End point.
 - e. Max Retry count
 - f. Request Wrapper class
 - g. Response Wrapper class
- 3. **Exception log** log exception per callout
 - a. Error
 - b. Error type
 - c. Callout Id Integration log id

Execution

- 1. Add in Queue From the Context, where Callout needs to be made,
 - a. Create Integration log by putting required Values.
 - i. Provide the meta data records' unique name to it.
 - ii. From the meta data,
 - 1. get the request Wrapper class Name,
 - 2. create JSON from wrapper.
 - 3. attach it on integration log as file.
 - iii. Mark status as In-Progress. As we cannot update it before Callout In same context.

- 2. **Process Queue** Some process which takes the set of requests and process them one by one. For example, **Batch**, **Queueable** etc
 - a. Get the set of Logs which are not successful yet.
 - b. For each Integration log that is in In-Progress.

Try{

- i. Take the meta data record from Pre-queried Meta data list.
- ii. If retry count on log < Max retry in Meta data
 - 1. Get the service class name.
 - Get the instance of Service class using Reflection. (Type.forName)
 - 3. Call the Service Class *process* method
 - a. pass the method type from Meta data record.
 - b. get the Request Json from Attachment file, create a RequestWrappper and pass it.
 - 4. The method will do **Callout**, and returns a response.
 - 5. If the return code is 200,
 - a. update records status to Success.
 - b. Get the Response Wrapper as response, Convert into JSON and save it on log's attachment.
 - 6. Else
 - a. increase the retry count.
 - b. Change status of log to FAIL.
 - 7. Save the return code on Log record.

}catch(Exception e){

Use platform event to Create exception log record. And publish it.

- c. Once the loop gets over. We can again go for retrying the failed logs. For this, don't do step 6b. Let the log be in IN-Progress, and again trigger the same process.
- d. For some response codes, we don't want to do anything such as 404, 403. In those cases, remove them from iterating list based on status codes.

Bulk Request - Sending more than 1 log in single callout.

(Request + Response) size can be 6 MB(sync) and 12MB(Async)

In case of bulk Requests, the RequestWrapper will contain request for multiple records, and receives responses corresponding to each record in a single ResponseWrapper. Based on the responses, update individual logs.

If the endpoint takes time to process i.e., possibility of Time out (time more than 120 sec), and response is not gonna be realtime.

- **a.** The Endpoint should some response to notify that, the process will take time.
- **b.** We will expose an endpoint from our system. That Endpoint will be hit by the other system once it finishes the processing.
- **c.** We will update the required records.

Named Credentials

It specifies the URL of a callout endpoint and its required authentication parameters in one definition. It handles the authentication by itself, which is not possible in case of Remote site settings.

Fields

- 1. **URL**: Root/BASE_URL of endpoint.
- 2. **Certificate**: used for digital signature at the time of request.
- 3. **Identity Type**: Determines whether you're using one set or multiple sets of credentials to access the external system.
 - a. **Anonymous**: No identity and therefore no authentication.
 - b. Per User: Use separate credentials for each user who accesses the external system via callouts. Select this option if the external system restricts access on a per-user basis. After you grant user access through permission sets or profiles in Salesforce, users can manage their own authentication settings for external systems in their personal settings. Also set Authentication Settings for External Systems, in my settings of that user for External Data source/NC.
 - c. Named Principal: <u>Use the same set of credentials for all users who access the external system from your org</u>. Select this option if you designate one user account on the external system for all your Salesforce org users.
- 4. **Authentication protocol**: secure communication between the two systems.
 - a. Password Authentication: Username and password
 - b. **OAuth2.**0
 - i. Authentication Provider
 - ii. **Scope**: provide the scope for access token. Some common scopes are
 - 1. "api" allow current current logged in users account using API.
 - "full" Allows access to all data accessible by the logged-in user. full
 doesn't return a refresh token. You must explicitly request the
 refresh_token scope to get a refresh token.
 - 3. "refresh_token": Allows a refresh token to be returned when the requesting client is eligible to receive one.

- 4. "refresh_token, offline_access": With a refresh token, the app can interact with the user's data while the user is offline. This token is synonymous with requesting offline access
- iii. **Start Authentication Flow on Save :** To authenticate to the external system and obtain an OAuth token.
- c. JWT/ JWT Token Exchange
 - i. Issuer: who issued the JWT
 - ii. Scope: Used in JWT token Exchange
 - iii. **Token End point:** Used in JWT Token Exchange, JSON Web Token requests are sent to the provider in exchange for access tokens.
 - iv. **Per User Subject**: In case per user selected, Per user identity like user Id.
 - v. **Named Principal Subject**: In case named Principal selected. Provide some string specifies subject.
 - vi. Audiences: External service or other allowed recipients for the JWT.
 - vii. **Token Valid For:** Time for which token is valid, in Seconds, min, hours, days.
 - **viii. JWT Signing Certificate:** Certificate verifying the JWT's authenticity to external system.
- **d. AWS Signature Version 4 -** A protocol to authenticate callouts to resources in Amazon Web Services over HTTP. **The identity type must be named principal.**
 - i. AWS Access Key ID
 - ii. AWS Secret Access Key
 - iii. AWS Region: AWS region name for NC endpoint.
 - iv. AWS service : AWS utility to access.

5. Custom Headers and Bodies of Apex Callouts

- a. **Generate Authorization Header:** By default, Salesforce generates an authorization header and applies it to each callout that references the named credential.
- b. Allow Merge Fields in HTTP Header: HTMLENCODE can't be used on merge fields in HTTP headers.

```
// non-standard authentication
req.setHeader('X-Username', '{!$Credential.UserName}');
req.setHeader('X-Password', '{!$Credential.Password}');
// The external system expects "OAuth" as
// the prefix for the access token.
```

- req.setHeader('Authorization', 'OAuth {!\$Credential.OAuthToken}');
- c. Allow Merge Fields in HTTP Body: HTTP request bodies of callouts, you can apply the HTMLENCODE formula function to escape special characters.

 req.setBody('UserName:{!HTMLENCODE(\$Credential.Username)}')

 req.setBody('Password:{!HTMLENCODE(\$Credential.Password)}')

Authentication Providers: They authenticate users for SSO and authorize Salesforce to access protected third-party data.

Setup

Setup | Administer | Security Controls | Auth. Providers | Create New

- 1. "Consumer Key" and "Consumer Secret".
- 2. "Default Scope", it should have value as "refresh_token full". "refresh_token" and "full" should be separated by space.
- 3. Authorize Endpoint URL should be something like: https://AuthenticationProviderinstance/services/oauth2/authorize
- 4. Token Endpoint URL: https://AuthenticationProviderinstance/services/oauth2/token
- 5. Once you save "Auth. Provider" in previous step, it will provide you list of URL Copy Callback URL and edit Connected App we created in service provider Salesforce instance and paste link in callback URL field.

Streaming API

Streaming API lets you push a stream of notifications from Salesforce to client apps based on criteria that you define.

- 1. Use Change data capture
- 2. Platform events
- 3. PushTopics and subscribe using Bayeus client.

A **PushTopic** is an sObject that contains the criteria of events you want to listen. PushTopic queries support all custom objects and some of the popular standard objects, such as Account, Contact, and Opportunity.

```
PushTopic pushTopic = new PushTopic();

pushTopic.Name = 'AccountUpdates';

//The SELECT statement's field list must include Id.

//aggregate queries or semi-joins aren't supported.

pushTopic.Query = 'SELECT Id, Name, Phone FROM Account WHERE BillingCity=\'USA\'';

pushTopic.ApiVersion = 37.0;

//operation preferences, default all are true

pushTopic.NotifyForOperationCreate = true;

pushTopic.NotifyForOperationUpdate = true;

pushTopic.NotifyForOperationUndelete = true;

pushTopic.NotifyForOperationDelete = true;

//field preferences

pushTopic.NotifyForFields = 'Referenced';

insert pushTopic;
```

NotifyForFields	Description		
All	Notifications are generated for all record field changes.		
Referenced (default)	Changes to fields referenced in the SELECT and WHERE clauses are evaluated.		
Select	Changes to fields referenced in the SELECT clause are evaluated.		
Where	Changes to fields referenced in the WHERE clause are evaluated.		

Soap API

Web Services Description Language (WSDL) contains the bindings, protocols, and objects to make API calls. Salesforce provides two SOAP API WSDLs for two different use cases.

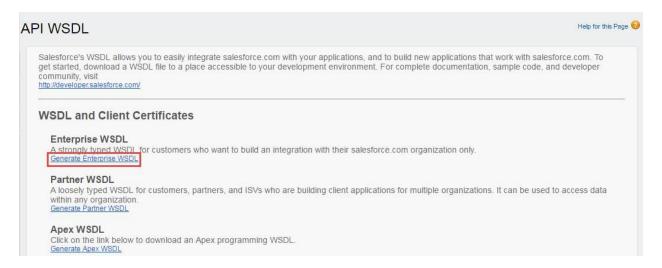
The enterprise WSDL - integration for single org

- 1. It is optimized for a single Salesforce org.
- 2. It's strongly typed, and it reflects your org's specific configuration.

The partner WSDL - integration for multiple org

- **1.** is optimized for use with many Salesforce orgs.
- 2. It's loosely typed, and it doesn't change based on an org's specific configuration.

Setup-> quick find -> api



Demo using Soap Client:

https://trailhead.salesforce.com/en/content/learn/modules/api basics/api basics soap

- 1. Install the soap client.
- 2. New soap project and upload WSDL. Once it is successful, it shows all the request that can be made to WSDL org.
- 3. Login-> request 1-> Provide username and password+Security Token
- 4. Once login is successful, you get Session ID and instance url.

- 5. For any other request, for example create, Use session Id in SessionHeader.
- 6. Hit Instance url, with the requests.

Apex Callout using SOAP web Service

- **1.** Get the WSDL file, from resource server.
- 2. Convert wsdl to apex using WSDL2Apex. The size of any WSDL document can be at most 1MB.
- **3.** The Apex classes construct the SOAP XML, transmit the data, and parse the response XML into Apex objects. Instead of developing the logic to construct and parse the XML of the web service messages, let the Apex classes generated by WSDL2Apex internally handle all that overhead.

Expose a Class as a SOAP web Service

Source System.

- 1. Define your class as global.
- **2.** Add the **webservice** keyword and the **static** definition modifier to each method you want to expose.
- 3. The webservice keyword provides global access to the method it is added to.

```
global with sharing class MySOAPWebService {
  webservice static Account getRecord(String id) {
    // Add your code
  }
}
```

- 4. Generate this WSDL for your class and send it to target developers.
- 5. For Authentication, external applications must use either the Enterprise WSDL or the Partner WSDL for login functionality.

Target System:

- 1. Go to Apex Classes an Click button "Generate from WSDL" using WSDL.
- 2. Generate class from Webservice WSDL. It will generate 2 classes. One for Sync and other for Async calling, i.e. future based.
- 3. Generate another Class with Enerprise/partner WSDL. It will have all standard meta data methods, like describe() calls, login, logout, create, insert etc

- 4. Develop a class in Target system where you generated apex class to use **Login** method of Enterprise/partner WSDL using credentials from Source.
- 5. Then call the SOAP method developed.

classGeneratedByWSDL. MySOAPWebService instance = new classGeneratedByWSDL.
MySOAPWebService();

instance.getRecord(id);

when should you use REST vs SOAP?

- If developing public APIs, in which you don't have to control over whats going on with he consumer.
- 2. APIs that require, a lot of back-and-forth messaging, with SOAP being stateful the same type of service would require more initialization and state code. Because REST is stateless, the client context is not stored on the server between requests, giving REST services the ability to be retried independently of one another.
- 3. Services, that do not need connection all the time SOAP services require maintaining an open stateful connection with the client. REST, in contrast, enables requests that are completely independent of each other.
- 4. Json response is easy to parse and lightweight. So, response will be quick.

Which API Do I Use?

API Name	Protocol	Data Format	Communication	When to Use
REST API	REST	JSON, XML	Synchronous	
SOAP API	SOAP (WSDL)	XML	Synchronous	SOAP API to create, retrieve, update, or delete records. You can also use SOAP API to perform searches and much more.
Chatter REST API	REST	JSON, XML	Synchronous (photos are processed asynchronously)	For chatter based use cases.
User Interface API	REST	JSON	Synchronous	Build Salesforce UI for native mobile apps and custom web.
Analytics REST API	REST	JSON, XML	Synchronous	For gathering datasets from Analytics Platform
Bulk API	REST	CSV, JSON, XML	Asynchronous	Submits a batch for use cases,
Metadata API	SOAP (WSDL)	XML	Asynchronous	migrate customization changes
Streaming API	Bayeux	JSON	Asynchronous (stream of data)	For near-real-time streams of data that are based on changes in Salesforce records or custom payloads.
Apex REST API	REST	JSON, XML, Custom	Synchronous	
Apex SOAP API	SOAP (WSDL)	XML	Synchronous	
Tooling API	REST or SOAP (WSDL)	JSON, XML, Custom	Synchronous	fetch the metadata such as Apex classes, Apex triggers, custom objects, custom fields, etc. If we need to get the list of Custom Objects or Custom fields

Platform cache

- 1. Using cached data improves the performance of your app and is faster than performing SOQL queries repetitively, making multiple API calls, or computing complex calculations.
- 2. Use the cache to store static data or data that does not change often.
- 3. Salesforce evicts cached data based on a least recently used (LRU) algorithm

Types Of Cache

- 1. **Org cache** stores org-wide data that anyone in the org can use.
 - a. time-to-live Max 48 hours, default is 24 hours.
- 2. **Session cache** stores data for an individual user and is tied to that user's session. The maximum life of a session is 8 hours.

Partitions: Caching data to designated partitions ensures that the cache space is not overwritten by other apps or by less critical data.

```
Cache.Org.put('namespace.partition.key', 0);
```

Among all partitions, **there can be only one default partition**. In Default partition, we don't have to use the fully qualify the key name with the namespace and partition name when adding a key-value pair.

```
Cache.Org.put('key', 0);
```

Namespace.Partition.Key is the cache key Name format.

- 1. Namespace org namespace. Can use "local" too.
- 2. Partition partition name.
- 3. Key the key which needs to be put.

Store and Retrieve Data in Org Cache

```
// Get partition

Cache.OrgPartition orgPart = Cache.Org.getPartition('local.CurrencyCache');

// Add cache value to the partition. Usually, the value is obtained from a

// callout, but hardcoding it in this example for simplicity.

orgPart.put('DollarToEuroRate', '0.91');

// Retrieve cache value from the partition

String cachedRate = (String)orgPart.get('DollarToEuroRate');

if (cachedRate != null) {

// Display this exchange rate

} else {

// We have a cache miss, so fetch the value from the source.

// Call an API to get the exchange rate.

}
```

Store and Retrieve Data in Session Cache

```
// Get partition

Cache.SessionPartition sessionPart = Cache.Session.getPartition('local.CurrencyCache');

// Add cache value to the partition

sessionPart.put('FavoriteCurrency', 'JPY');

// Retrieve cache value from the partition

String cachedRate = (String)sessionPart.get('FavoriteCurrency');
```

```
remove() :- to delete an item from cache.
```

MAX_TTL_SECS: A constant in Org and Session class represents maximum amount of time, in seconds, to keep the cached value in the session cache.

Access cached values stored in the platform cache from a Visualforce page by using the **\$Cache.Session** or **\$Cache.Org** global variables.

\$Cache.Session.namespace.partition.key \$cache.org. namespace.partition.key

Encryption-Decryption - Classic encryption

1. EncodingUtil

- String base64Decode(String inputString)
- String base64Encode(Blob inputBlob)
- Blob convertFromHex(String inputString)
- String convertToHex(Blog inputBlob)
- String urlDecode(String inputString, String encodingScheme)
- String urlEncode(String inputString, String encodingScheme)
 Note: for URL encode and decode second parameter will be encoding scheme i.e.,
 UTF-8. All methods are static so you can call by class name.

2. Crypto

Provides methods for creating digests, message authentication codes, and signatures, as well as encrypting and decrypting information. These functions are using AES algorithm to generate the private key and these functions are using AES128, AES256, AES192 algorithms to encrypt and decrypt information. The AES algorithm requires initialization vector to encrypt and decrypt data. **AES stands for Advanced Encryption Standard algorithm**

Platform encryption - Shield Platform Encryption

Classic encryption uses 128-bit (AES) keys. Shield Encryption uses a stronger 256-bit AES key.

tenant secrets are used to derive your encryption keys. They work with the Salesforce-generated master secret, but your tenant secret is specific to your org.

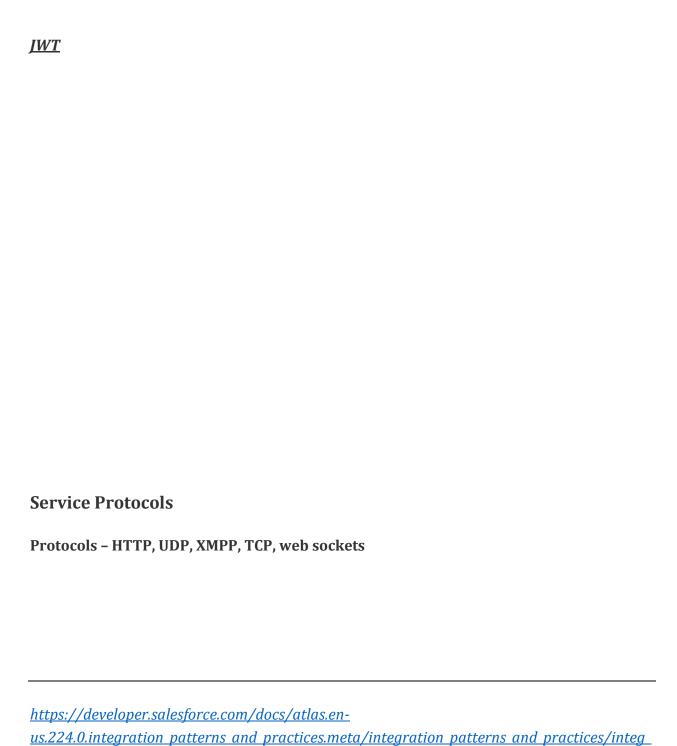
Setup

- 1. Provide access to **Manage Encryption Keys** (System permission) to user.
- 2. Go to Key Management and Generate Tenant Secret. Export it.
- 3. Go to **Encryption Policy**, choose **Encrypt fields** and select required ones.
- 4. Destroy the key. Data will be encrypted now.
- 5. Upload the key to see the data again.

Outbound messaging In workflows

Outbound messaging allows you to specify that changes to fields within Salesforce can cause messages with field values to be sent to designated external servers.

It Uses SOAP API.



pat intro overview.htm