Salesforce Integration: Bespoke Execution using Lightning, REST, APEX, and JavaScript

In a previous post, we looked at the fantastic integration options that are available out-of-the-box with our Salesforce integration.

While the functionality that integration provides will be sufficient for the majority of of InRule customers, there are always edge cases where a different type of interaction with InRule may be appropriate. While we do not recommend implementing any of these without working with our ROAD Services group to see if there's an easier answer, these are all execution structures that we've developed prototypes for, allowing customers to integrate in non-standard ways.

Fair warning – this post is significantly more technical than the last one and is targeted towards folks that have a pretty solid understanding of Salesforce. Now that that's out of the way - let's take a look!

irServer REST Rule Execution from Lightning Web Component

If you have a custom Lightning Web Component that acts as a complex form, it's possible that you may need to execute Rules against an entity that has not yet been saved to Salesforce – the one in progress on your custom page. Since InRule's Salesforce integration operates against the entity state as it is persisted to Salesforce, this scenario wouldn't support using the native integration. However – there's nothing preventing the irServer REST Rule Execution Service from being able to handle Rule Execution requests against a Salesforce entity, as long as all the appropriate data is provided at execution time (and, if you're using the RuleHelper, those assemblies are available to irServer).

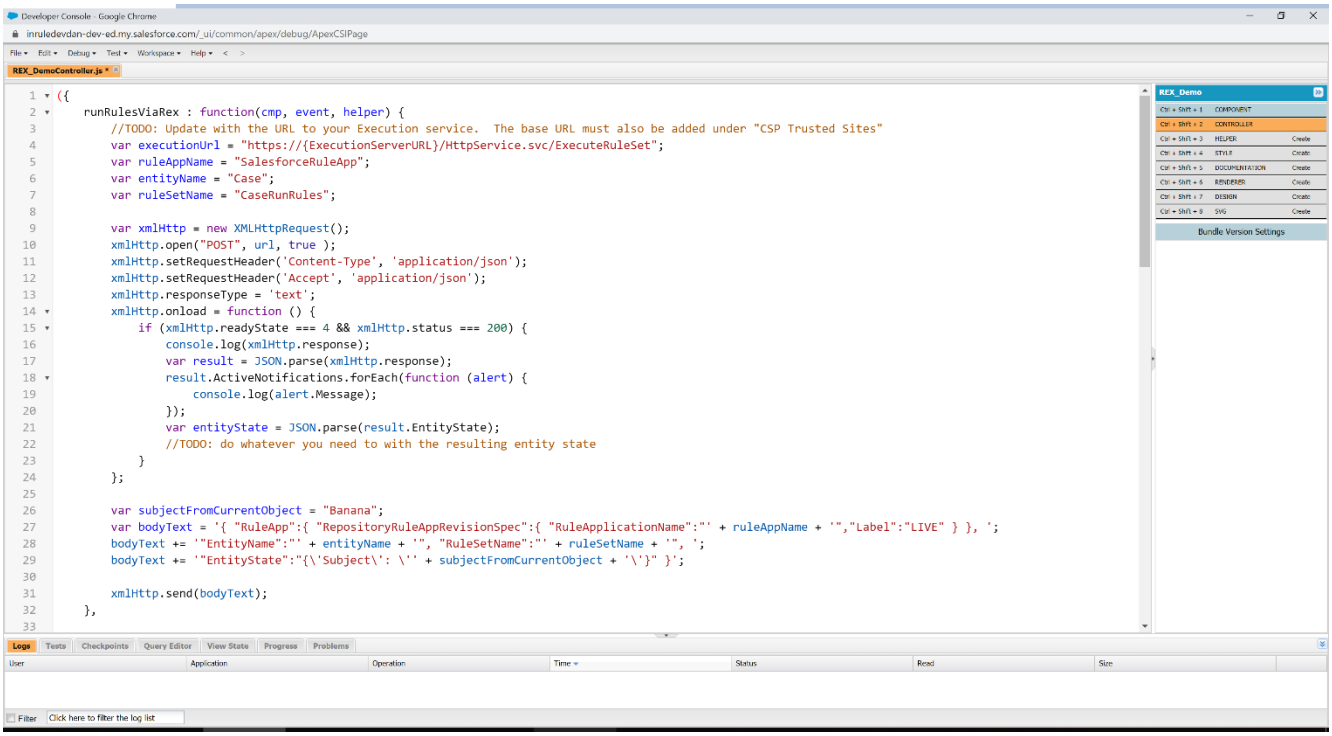
To run rules from a Lightning component, you simply need to build up the REST request in the LWC controller and send the request to the execution service, processing the response. To accomplish that, there are a number of caveats to keep in mind:

- The URL of the execution service must be configured with Salesforce under the "CSP Trusted Sites" section to allow the cross-site request to be sent.
- The URL of your Salesforce instance (or "*") must be configured in the CORS configuration of your irServer host for the request to be received.
    - For Azure-based deployments, there is a configuration page in the App Service settings in the Azure Portal.
    - For IIS-hosted services, you'll need to install the IIS CORS module from here (link) https://www.iis.net/downloads/microsoft/iis-cors-module, and then configure your execution service web.config with the following addition:

```
<system.webServer>
    <cors enabled="true">
      <add origin="*">
        <allowHeaders allowAllRequestedHeaders="true" />
      </add>
    </cors>
</system.webServer>
```

- All entity state information must be passed in the initial request, as if you were running Rules from the "Test" button on the Home tab in irAuthor rather than the Salesforce irX tab.
- Any persistence must be done within the LWC – nothing will be automatically saved after Rule execution (unless done in the Rule App using the RuleHelper assembly).

Once those are handled appropriately, here's an example of a Lightning Web Component Controller method that executes rules off a Case using the REST service.



Not too bad, right? Well, since we did it from a Lightning Web Component, why not also…

irServer REST Rule Execution from APEX Trigger or Class

That's right, we can execute using the REST service from an APEX class as well! In certain edge cases, like if you have Formula fields that rules are reliant upon and are not yet persisted in the context of the AfterUpdate trigger, you may find that running rules using the REST endpoint from APEX is an alternative option.

Because this is executing from the server-side rather than the client-side, some of the caveats are a bit different:

- The URL of the execution service must be configured with Salesforce under the "Remote Site Settings" section to allow the request to be sent.
- All entity state information must be passed in the initial request, as if you were running Rules from the "Test" button on the Home tab in irAuthor rather than the Salesforce irX tab.
- Any persistence must be done within the APEX class – nothing will be automatically saved after Rule execution (unless done in the Rule App using the RuleHelper assembly).
- If data on an entity is modified, users with the entity open will not automatically see the updated values in the UI until they refresh the page. If that is needed, you could add a Platform Event that gets published after the save operations are complete, and build a Lightning Web Component that lives on the object page and listens for that event, initiating a data refresh when it's received for the currently loaded object ID.

Again, once those are handled appropriately, here's an example of an APEX class that executes rules off a Case using the REST service:

```
public class InRuleRESHelper
{
  @future(callout=true)
  public static void RunCaseRules(Id caseId)
  {
    //TODO: Update with the URL to your Execution service.  The base URL must also be added under "Remote Site Settings"
    String executionUrl = 'https://{ExecutionServerURL}/HttpService.svc/ExecuteRuleSet';
    String ruleAppName = 'SalesforceRuleApp';
    String entityName = 'Case';
    String ruleSetName = 'CaseRunRules';
    Case c = Database.query('Select Id, Subject From Case Where Id = :caseId');

    HttpRequest req = new HttpRequest();
    //NOTE: This line assumes the Catalog credentials are configured in the execution service's settings
    String bodyText = '{"RuleApp":{"RepositoryRuleAppRevisionSpec":{"RuleApplicationName":"' + ruleAppName + '","Label":"LIVE"}},';
    bodyText += '"EntityName":"' + entityName + '","RuleSetName":"' + ruleSetName + '",';
    //TODO: This line needs to populate all fields required for Rule execution
    bodyText += '"EntityState":"{\'Subject\': \'' + c.Subject + '\'} }';
    System.debug('RES Rule Execution request will be placed with body:' + bodyText);

    req.setEndpoint(executionUrl);
    req.setMethod('POST');
    req.setHeader('Content-Type', 'application/json');
    req.setHeader('Accept', 'application/json');
    req.setBody(bodyText);
    req.setTimeout(30000); //Timeout is set high in case the RES is not active and needs to start up and compile the Rule App
    Http http = new Http();
    HTTPResponse res = http.send(req);

    JSONParser parser = JSON.createParser(res.getBody());
    RuleExecutionResponse rer = (RuleExecutionResponse)parser.readValueAs(RuleExecutionResponse.class);
    System.debug('Rule Execution Result:' +  JSON.serialize(rer));

    for (RuleExecutionResponseNotification n : rer.ActiveNotifications) {
        System.debug('Notification (' + n.NotificationType + '): ' +  n.Message);
    }

    parser = JSON.createParser(rer.EntityState);
    Case resultEntity = (Case)parser.readValueAs(Case.class);

    //TODO: Update with any fields you want to set the value on here
    String oldVal = (String)c.put('Subject', resultEntity.Subject);
    update c;
  }
  public class RuleExecutionResponse {
    public List<RuleExecutionResponseNotification> ActiveNotifications;
    public String EntityState;
  }
  public class  RuleExecutionResponseNotification {
    public String Message;
    public String NotificationType;
  }
}
```

And there you have it! Two different ways to run rules using the REST endpoint from Salesforce – one from the front-end, one from the back-end.

irJS JavaScript Rule Execution from Lightning Web Component

In the previous two examples, we used the REST execution service to perform out-of-process executions from different locations with Salesforce. What about executing rules directly within Salesforce? Impossible, I hear you say? Never!

If you have some Rules that are self-contained, but are also more complex that you want to implement using hardcoded JavaScript or native Salesforce logic, you may find that using the JavaScript execution pattern fills your needs. Using our extremely fast irJS <link> execution pattern, Rule Applications (albeit with a slightly restricted functionality set) can be compiled into a JavaScript library that can be embedded within a web page and executed directly in the client browser.

Once you create your Rule App and compile it into a JavaScript file using the irJS Distribution Service, you can then upload the file to Salesforce as a Static Resource that can be referenced in Lightning Web Components. From there, the execution pattern is pretty much the same as any other JavaScript-based execution, with some Salesforce data interaction tidbits thrown in for good measure.

Because this execution patterns uses bits that already exist in Salesforce, the caveats are a bit different than the other two that we've discussed:

- The compiled irJS library file will need to be checked into Salesforce as a Static Resource. Any changes to the Rules will need to be re-compiled and re-uploaded to the Static Resource in Salesforce in order to take effect; checking into the Catalog will not impact the static resource.
- The Rule Application must be written targeting Javascript, which means that some functionality will not be able to be supported (mainly things that call out from the Rule Application during execution like SQL queries and REST requests). irX for Salesforce (which allows you to import schema) is fully supported in Javascript-targeted Rule Applications, but requests made through the RuleHelper are not.

- As long as it targets JavaScript, you can absolutely have one Rule Application used both for JavaScript execution as well as checked in and used for all your native Salesforce integration executions.
- When building the code to run the irJS Rules in the Salesforce component, you will need to manually build up an object that matches the schema in the Rule Application.
- This should go without saying, but you'll be responsible for populating the full entity state and then handling the persistence or processing after execution completes.

Not a bad list of caveats – let's take a look at the code in the Lightning Web Component that runs rules using irJS from a Static Resource:

The LWC UI bits:

```
    <ltng:require scripts="{!$Resource.IR_SfRuleApp}"/>
    <aura:attribute name="record" type="Object" />
    <aura:attribute name="simpleRecord" type="Object" />
    <aura:attribute name="recordError" type="String" />
    <force:recordData aura:id="recordEditor"
        layoutType="FULL"
        recordId="{!v.recordId}"
        targetError="{!v.recordError}"
        targetRecord="{!v.record}"
        targetFields ="{!v.simpleRecord}"
        mode="EDIT" />
```

And the LWC Controller logic:

```
    runJsRule : function(component, event, helper) {
        var entityName = component.get("v.entityType");
        var ruleName = "CaseRunRules";
        var entity = {};
        entity["Subject"] = component.get("v.simpleRecord.Subject");

        //Run Rules
        var irSession = inrule.createRuleSession();
        var irEntity = irSession.createEntity(entityName, entity);

        var callback = function(log) {
            if(!log.hasErrors){
                irSession.getActiveNotifications().forEach(function (notification) {
                    console.log("Notification (" + notification.type + "): " + notification.message);
                });

                component.set("v.simpleRecord." + propertyName, updatedSfEntity["Subject"]);
                component.find("recordEditor").saveRecord($A.getCallback(function(saveResult) {
                    if (saveResult.state === "SUCCESS" || saveResult.state === "DRAFT") {
                        console.log("Save completed successfully.");
                    } else {
                        console.log('Unable to save, state: ' + saveResult.state + ', error: ' + JSON.stringify(saveResult.error));
                    }
                }));
            }
            else{
                console.log("Error running Rules: " + log.runtimeErrors.join('\n'));
            }
        }

        console.log("Running " + ruleName + " ruleSet on " + entityName + " entity: ", entity);
        irEntity.executeRuleSet(ruleName, [], callback);
    }
})
```

Just like that, you're executing blazing-fast irJS rules in the browser, embedded within a Lightning Web Component.

With great power…

As I mentioned early on in this post, we do not recommend that folks implement these without first chatting with our ROAD Services team to see if there might be a more simple solution. However, that all these options are possible with very little customization goes to show that the fantastic level of flexibility that's built-in to InRule allows for an integration solution to fit just about any scenario you can imagine.

Think you've got one that'll stump us? Send us a message and let us know – we love a good challenge!