Global API Data Service Layer

Data service layers, at most basic, provide functions for:

- Querying
  - Build Query
  - Execute Query
- Transforming Data
- Possibly calling related APIs for additional data. E.g. carts --> cartItems --> product Data

Service Provider Design Considerations

- Subscriber overrides selecting additional custom fields and/or related record data impact the performance of OOTB service calls based on the amount of data processed in the Service Provider Data Transformation.
- Initially, service providers all extend the base global *ccService* class which defines some base functions for constructing queries and transforming data. These must be extended by all implementations of service providers.
  - Subscriber overrides can extend a particular service provider as these extend the base *ccService* class; e.g. *extends ccServiceProduct*
- Services are functionally grouped by the *top-level entities (*e.g. Cart, Order, Catalog).
- The naming convention follows from the entity (e.g. *ccServiceCart).*
- Services utilize our current strategy for hook methods, Map-IN:Map-OUT, namely, *global Map<String,Object> someFunction(Map<String,Object> inputData);*
  - Even if the method does not return any data initially it should always return *Map<String,Object>*.
  - Even if the method does not take any data initially it should always accept *Map<String,Object>*.

Service Provider Data Transformations

Overview

In the past, CloudCraze has returned concrete "bean"(-like) APEX classes, such as *cc_bean_Cart.* The trouble with this approach was just that subscriber-generated code had to be pushed into *extrinsic* blocks on the beans and there was an overhead of mapping the data into this block. Th e map-out approach coupled with reflective querying of the top-level entity of the APIs effectively solves that issue and allows the API to push back almost any structure.

Service Provider Data Transformation Conventions

Service providers transform data by means of key substitution on the map-out structure. Examining an sObject, the field names essentially describe a key in a map. E.g. My_Field__c => 'My Data'. These field names are converted to a more palatable format using the *cc_util_SFDCRefl ection.removeSFDCFieldnameConventions()* function. This will handle maps with lists of maps below.

- *__c, __r* and _ are removed.
- Field names are camel cased.
  - Fields that have consecutive capitals letters are the exception.
  - Example: *ccrz__My_Field__c* becomes *myField.*
- Namespaces are removed from the fieldName.
- Salesforce *Id* fields become *sfid.*
- Salesforce name fields become *sfdcName.*
- In API version 3 and above, *child-to-parent r*elationship queries will return a *Map<String, Object>* under the relationship name with a "R" appended.
  - E.g. *SELECT Id, Name, Product__r.Name, Product__r.SKU__c FROM My_Object__c* will, when subject to transformation, return a *Map<String, Object>* with the following keys:
    - *sfId:* The ID from My_Object__c
    - *sfdcName:* The Name from My_Object__c
    - *productR:* Map<String, Object>:
      - *sfdcName:* The Name from the Product__r "Parent" of My_Object__c
      - *SKU:* The SKU from the Product__r "Parent"
  - NOTE: In API versions < 3, there exists the potential for like named relationships to collide under a specific key: E.g. *SELECT Product__c, Product__r.Name, (SELECT Name FROM Product__r)* will have a collision where *Product__c* ==> "*product*", *Produ ct__r* ==> "*product*" and *Product__r* ==> "*product*".
- In API version 3 and above, *parent-to-child r*elationship queries will return a *List<Map<String, Object>>* objects under the relationship name with a "S" appended.

- E.g. *SELECT Id, Name, (SELECT Name, SKU__c FROM Product__r) FROM My_Object__c* will return a *Map<String, Object>* with the following keys:
    - *sfId:* The ID from My_Object__c
    - *sfdcName:* The Name from My_Object__c
    - *productS*: List<Map<String, Object>>:
        - *sfdcName:* The Name of a "child" product.
        - *SKU:* The SKU of the "child" product.
- NOTE: In API versions < 3, there exists the potential for like named relationships to collide under a specific key: E.g. *SELECT Product__c, Product__r.Name, (SELECT Name FROM Product__r)* will have a collision where *Product__c* ==> "*product*", *Product__r* ==> "*product*" and *Product__r* ==> "*product*".

The most common case is to use the *JSON* serialization and deserialization functions to put the data into a *Map<String, Object>* format which can then be passed to the reflective utility.

ccServiceCart - Data Transformation

```
    global virtual override Map<String, Object> fetch(Map<String, Object> inputData) {


                        ... skipping ...

                        Map<String, Object> mappedProduct = (Map<String, Object>) JSON.deserializeUntyped(JSON.serialize(cart));
                        rawCartMap.put(ccAPICart.CART_OBJ, this.transformOut(mappedProduct));


                        ... skipping ...
        }

    global virtual override Map<String, Object> transformOut(Map<String, Object> rawCart) {
                return cc_util_SFDCReflection.removeSFDCFieldnameConventions(rawCart, new Set<String>()); // --> handle this better. Register other items?
    }
```

Benefits of a Bean-like Structure

Maps are great if the data is being cast into JSON structure somewhere up the chain, however, working with them in APEX controllers is difficult. In this case, the map-out data could be cast into concrete APEX bean-class. (NOTE: This is partly why the naming convention exists as it does - the transformed names line up 80% of the way with existing *cc_bean* objects...)

To cast to a concrete class, *JSON.deserialize(JSON.serialize(outputData), <YOUR_classname>.class)*

Service Provider Query Builder

Overview

The base *ccService* class defines a number of global methods that handle sections of query-related data:

- *global String objectName*

- *global virtual Map<String, Object> getFieldsMap(Map<String, Object> inputData)*
- *global virtual Map<String, Object> getSubQueryMap(Map<String, Object> inputData)*
- *global virtual Map<String, Object> getDirectQueryMap(Map<String, Object> inputData)*
- *global virtual Map<String, Object> getFilterMap(Map<String, Object> inputData)*
- *global virtual Map<String, Object> getOrderByMap(Map<String, Object> inputData)*

Most of these functions are used to emit constructs that are used to generate a query using the service's *buildQuery()* method. The intent is that the service's *runQuery()* method will use these constructs. (See the Service Provider Query Executor documentation.)

Additionally, *ccService* will handle inputData that contains *ccService.QUERYLIMIT* and *ccService.QUERYOFFSET* information for paging.

Hypothetically, a service provider for an entity that is completely separate from Salesforce/CloudCraze - i.e., RESTful data query over HTTP - should still be able to use these constructors to generate the URI for the data they're trying to get.

Service Provider "objectName" Field

Since the base *ccService* class is extend, their constructor can set the corresponding Salesforce object name. E.g. the *ccServiceProduct* would specify something like *this.objectName = 'E_Product__c';*

Service Provider "getFieldsMap"

The service provider class is expected to generate a string of the Salesforce fields names to be included in a query. Most often, extensions used the default *ccService* method since it exposes the reflective *cc_util_SFDCReflection.getFields()*, thereby allowing any subscriber or market template added code to bubble up.

Extensions could, of course, override the method and return a static string, etc. Additionally, the *getFieldsMap()* takes the inputData as a parameter and could be use that to constrain which fields make it into the query constructor.

Since this is a global method, it must return the map-out convention. The global *ccService.OBJECTFIELDS* key serves as a way to handle this data.

ccSevice - Fields

```
     global virtual Map<String, Object> getFieldsMap(Map<String,
Object> inputData) {
            return new Map<String, Object> {
                    ccService.OBJECTFIELDS => cc_util_SFDCReflection.
getFields(this.objectName, cc_util_SFDCReflection.BASESFFIELDS)
            };
    }
```

Service Provider "getSubQueryMap"

The service provider class is expected to generate a map of sub queries. The *getSubQueryMap()* takes the inputData as a parameter and can use that to constrain which sub queries actually make it into the query constructor. I.e. if a caller just wanted header data, and a *HEADERONLY* in put parameter existed, then maybe only *ProductI18N* would be included in the data that is passed back.

ccSeviceCart - SubQueries

```
    global virtual override Map<String, Object> getSubQueryMap
(Map<String, Object> inputData) {
                return ccServiceCart.SUBQUERYMAP;
    }
    private static final Map<String, Object> SUBQUERYMAP = new
Map<String, Object>{
            // A subQuery off of a cart.
        'E_CartItems__r' =>'(SELECT AbsoluteDiscount__c,Cart__c,
CartItemType__c,Category__c,Coupon__c,Comments__c,DisplayProduct__c,
Is_Subscription_Selected__c,ItemLabel__c,Name,OriginalQuantity__c,
ParentCartItem__c,ParentProduct__c,PercentDiscount__c,Price__c,
Product__c,ProductType__c,Quantity__c,StoreId__c,SubAmount__c,
Subscription_Duration__c,Subscription_Frequency__c,UnitOfMeasure__c
FROM E_CartItems__r)'
    };
```

- The keys of the map should be the *relationship name,* including the *__r.*
- Items should define their own bind variables. If these map to an input item, then those should be used from the API.
- Items should define their own boolean conditions - i.e. AND, OR
- The map could be empty, indicating that no sub queries are desired/expected.

Service Provider "getDirectQueryMap"

In some cases, it is advantageous for the service method to call a query directly, for example, to support a large dependent data set (e.g. a Cart with many Cart Items). In these cases, the service class describes queries that will be passed directly to a ccServiceDAO provider and queried directly. Just like the service provider specifies a map of sub-queries to call, the service provider can describe a map of direct queries to use. These are bubbled up to the service class via the getDirectQueryMap().

The getDirectQueryMap*()* takes the inputData as a parameter and could be used to constrain the query constructor. E.g. if a caller just wanted header data, and a *HEADERONLY* input parameter existed, then maybe only *ProductI18N* would be included in the data that is passed back.

- The keys of the map should be the *relationship name,* including the __r, **or** object name.
    - The relationship name is used when a sub query deemed too large to run as a sub query, is moved to a direct query.
    - NOTE: The *ccAPI.SIZING ccAPI.SZ_REL* should still control the method in which these queries are run/aggregated into the return data.
- Items should define their own bind variables. If these map to an input item, then those should be used from the API.
- Items should define their own boolean conditions - i.e. AND, OR
- The map could be empty, indicating that no direct queries are desired/expected.

ccSeviceCart - Direct Queries

```
private static final Map<String, Object> DIRECTQUERYMAP = new
Map<String, Object>{
        'E_CartItems__r'=>'SELECT ALL MY FIELDS FROM E_CartItem__c
WHERE Cart__c IN :cartIds',
        'E_CartItemPricingTier__c'=>'SELECT ALL MY FIELDS FROM
E_CartItemPricingTier__c WHERE CartItem__c in :cartItemIds'
};

global virtual override Map<String, Object> getDirectQueryMap
(Map<String, Object> inputData) {
                return DIRECTQUERYMAP;
}
```

```
... CONTROLLING WHICH QUERIES ARE RUN FROM SIZING BLOCKS ...


/*
 Check which directQueries we should run.
 If the call to handleReSizes returns null, then we do all the directQ's
 If the call to it returns data, we check if the map contains the
directQ's we want to run.
 If the map doesn't contain, for example, E_CartItem__r, then we won't
run that guy.


 Assuming the sizing looks like:
        { "SIZING":
                {
                        "cart": {
                                "sz_rel": ["E_CartItems__r"]
                        }
                }
        }
*/

DIRECTQUERYMAP.put(ccAPI.SIZING, ccUtil.defv(inputData, ccAPI.SIZING,
new Map<String, Object>()));
Map<String, Object> ldQMap = handleRelSizes(DIRECTQUERYMAP);
DIRECTQUERYMAP.remove(ccAPI.SIZING);

Boolean isCIRUN=true;
if(ldQMap != null){
        if(!ldQMap.containsKey('E_CartItems__r')){
                isCIRUN=false;
        }
}
// Can zero out the LDQMAP. (Free memory)
ldQMap=null;


if(isCIRUN){
        // Query for CartItems using a direct query.
        Map<String,Object> ciInput=new Map<String,Object>{
                // Want to get the query they supply.
                ccService.QUERYSTRING=>(String) this.getDirectQueryMap
(inputData).get('E_CartItems__r'),
                'cartIds'=>cisByCartId.keySet()
        };

        // This is a method to handle bulk queries for large data.
Iterate over the return results.
        for (E_CartItem__c item : (List<E_CartItem__c>) ((ccServiceDao)
this.initSvcDAO(ciInput).get(ccService.SVCDAO)).doQuery()  ) {
                // DO STUFF IN THE QUERY...
        }
}
```

A subscriber wishing to modify the direct query should override the getDirectQueryMap() method and call super.getDirectQueryMap() to modify the contents. Most content will be required for the product to work out of the box, so modifications are urged to be additive in nature.

Service Provider "getFilterMap"

The service provider class is expected to generate a map of filter criteria. The *getSubFilterMap()* takes the inputData as a parameter and can use that to constrain which filter clauses actually make it into the query constructor. I.e. if a caller just wanted to query for a singular product, instead of a list, then only the singular case would be used.

ccSeviceCart - Filter

```
     global virtual override Map<String, Object> getFilterMap
(Map<String, Object> inputData) {
            Map<String, Object> localizedFilterMap = new Map<String,
Object>(ccServiceCart.FILTERMAP);

            if (inputData.get(ccApiCart.CART_ID) != null) { // If
cartId, then drop the singular query from the map.
                    localizedFilterMap.remove(ccApiCart.CART_ENCID);
                } else if (inputData.get(ccApiCart.CART_ENCID) != null)
{ // If a list, then drop the singular query from the map.
                        localizedFilterMap.remove(ccApiCart.CART_ID);
                }
                if (inputData.get(ccApiCart.BYSTOREFRONT) == null) {
                        localizedFilterMap.remove(ccApiCart.
BYSTOREFRONT);
                }
                if (inputData.get(ccAPICart.BYOWNER) == null) {
                        localizedFilterMap.remove(ccApiCart.
BYOWNER);
                }

                if (inputData.get(ccAPICart.CARTSTATUS) == null) {
                        localizedFilterMap.remove(ccApiCart.
CARTSTATUS);
                }
                if (inputData.get(ccAPICart.CARTTYPE) == null) {
                        localizedFilterMap.remove(ccApiCart.
CARTTYPE);
                }
                return localizedFilterMap;
    }

        private static final Map<String, Object> FILTERMAP = new
```

```
Map<String, Object>{
        ccAPICart.CART_ID => ' Id = :' + ccAPICart.CART_ID + ' ',
        ccAPICart.CART_ENCID => ' EncryptedId__c in :' + ccAPICart.
CART_ENCID + ' ',
        'isDeleted' => ' AND IsDeleted = false ',
        ccAPICart.CARTTYPE => ' AND CartType__c = :' + ccAPICart.
CARTTYPE, // TODO - handle wishlist carts as well.
        ccAPICart.BYSTOREFRONT => ' AND Storefront__c =:' + ccAPICart.
BYSTOREFRONT,
        ccAPICart.BYOWNER => ' AND OwnerId =:' + ccAPICart.BYOWNER,
        ccAPICart.CARTSTATUS => ' AND CartStatus__c = :' + ccAPICart.
CARTSTATUS
    };
```

- The keys of the map should match the input parameters defined in the API.
- Filter items should define their own bind variables. If these map to an input item, then those should be used from the API.
- Filter items should define their own boolean conditions - i.e. AND, OR
- The map could be empty.

Service Provider "getOrderByMap"

This is just a map that describes an ordering in the query. Again, it takes the map-in convention so an input parameter could change the manner in which the data is organized.

- The keys of the map should be descriptive. If ordering was expect to change based on input, then it should match the input parameter /key.
- The map could be empty.

Service Provider LIMIT and OFFSET Provisions

By simply supplying the *inputData* with

- *ccService.QUERYLIMIT*: Integer input
- *ccService.QUERYOFFSET*: Integer input

The *buildQuery* function will use those in constructing the query.

Service Provider "buildQuery"

Services construct queries by calling the *buildQuery()* method. This is an aggregator which combs through various getters for fields, subqueries, filter criteria, and order criteria. The method accepts the inputData map since it will pass this through to the getters. *ccService* implements a working base query generator.

ccService - buildQuery

```
global class ccServiceFoo extends ccrz.ccServiceProduct {
    /*
    Aggregates the SOQL query string sub functions.
    We can use this one function instead of the "getFieldsMap",
"getFilterMap", "getSubQueryMap", etc.
    */
    global override Map<String, Object> buildQuery(Map<String, Object>
inputData) {
        Map<String, Object> retData = super.buildQuery(inputData);
        ccrz.ccLog.log(LoggingLevel.DEBUG,'M:X:ccServiceFoo:buildQuery:
original:',retData);

        // SELECT
        String qs = 'SELECT ';
        // 1. getFieldsMap: Get the object fields.
        qs += (String) this.getFieldsMap(inputData).get('objectFields');

        // 2. getSubQueryMap: Apply sub queries, if there are any.
        if (!this.getSubQueryMap(inputData).isEmpty()) {
            qs += ',';
            for(Object subQuery : this.getSubQueryMap(inputData).
values()) {
                qs += (String) subQuery + ',';
            }
            qs = qs.substring(0, qs.lastIndexOf(',')); // get's rid of
comma.
        }
        // FROM
        qs += ' FROM ' + this.objectName;

        // WHERE
        // 3. getFilterMap: Apply filters
        if (!this.getFilterMap(inputData).isEmpty()) {
            qs += ' WHERE ';
            for (Object filterConditions : this.getFilterMap(inputData).
values()) {
                qs += (String) filterConditions;
            }
        }
        // ORDER BY
        // 4. getOrderByMap: Apply Order By Filters.
        if (!this.getOrderByMap(inputData).isEmpty()) {
            qs += ' ORDER BY ';
            for (Object orderConditions : this.getOrderByMap(inputData).
values()) {
                qs += (String) orderConditions;
            }
        }
        // LIMIT
        // Apply a limit
        if(inputData.containsKey(ccrz.ccService.QUERYLIMIT)){
            qs+=' LIMIT '+Integer.valueOf(inputData.get(ccrz.ccService.
QUERYLIMIT));
        }else{
            //default limit
            qs+=' LIMIT 1000 ';
        }
        // OFFSET (optional)
        // Apply an offset
        if(inputData.containsKey(ccrz.ccService.QUERYOFFSET)){
            qs+=' OFFSET '+Integer.valueOf(inputData.get(ccrz.ccService.
QUERYOFFSET));
        }

        retData.put(ccrz.ccService.QUERYSTRING,qs);
        ccrz.ccLog.log(LoggingLevel.DEBUG,'M:X:ccServiceFoo:buildQuery:
overridden:',retData);
        return retData;
    }
}
```

- The query returned follows the familiar map-out convention and uses the key *ccService.QUERYSTRING* as a handle.
- Bind fields are defined in the subQuery and Filters. The caller of the *buildQuery* method should have them available as local variables.
- The function now also checks the for LIMIT and OFFSET keys in the input data.

Service Provider Query Executor

Overview

Service implementations supply the query generation part - i.e. accumulate the fields, subqueries, and filter conditions based on input data via the *buildQuery()* method - then pass the constructed query to a function, *runQuery()*, which uses a class, ccServiceDAO, which handles mapping of bind variables and then running the query.

Instantiation

Services classes are responsible for instantiating *ccServiceDAO* classes - the base *ccService* class defines a global method, *initSvcDAO*, which instantiates a *ccServiceDAO* and returns the class under the key *ccService.SVCDAO*. This is done because all global functions use a Map-in /Map-out convention.

Overriding

The *ccServiceDAO* is global and can be extended. Service classes may return the default version or versions that handle queries differently. Note that if you are running queries which reference *ccrz* data from outside *ccrz* packaged code, correct namespaces must be applied in your queries.

Without Sharing DAO

The base CloudCraze package does include an extended query runner that is marked *without sharing.* It is named *ccServiceDAOWO*. This class defines two methods which are necessary for the *without sharing* mechanism to function correctly.

- doQuery
- doSearch

In all other respects, this is the similar to the standard *ccServiceDAO* class.

To use the class, override the *ccService.initSVCDAO* method and specify the *ccServiceDAOWO* as the return.

Without Sharing Override

```
// Override and return a WO sharing DAO.
public virtual override Map<String, Object> initSvcDAO(Map<String,
Object> inputData){
        return new Map<String, Object>{ccService.SVCDAO=>new
ccServiceDAOWO(inputData)};
}
```

Binding Variables

In APEX, a caller needs a localized instantiation of a variable that is referenced in the query string in order to bind a variable. If no local variable is found in the scope, the system looks for a class variable to bind. Consider the following example:

Bind Variables.

```
String myAccountName = 'Foo';
String queryString = 'SELECT Id FROM Account WHERE Name = :
myAccountName';

List<SObject> results = Database.Query(String.escapeSingleQuotes
(queryString));
```

Since the block that is going to run the query doesn't have access to all the bind variable in a local scope, *ccServiceDao* uses the class variable to build the bind. This is accomplished by doing the following:

1. The *inputData* is a *Map<String, Objects>* that holds the the query string and all the various modifiers used to construct the query.

   ```
   {
           'queryString': 'SELECT Id FROM Account WHERE Name = :
   myAccountName',
           'myAccountName': 'foo'
   }
   ```

2. These keys are all examined, all keys, excluding queryString, are bound to member variables of type Object in the constructor.
3. Finally, the *queryString* is rewritten to substitute the *inputData* keys to the correct member bind variable (e.g. *bind01)*

Service DAO - Query Runner Example of Class Binding

```
public with sharing class ccServiceDao {

    private Integer bindIdx = 0;

    private Object bind01;
    private Object bind02;
    private Object bind03;
```

```
        public String modifiedQueryString; // The new Query string with
bind variables replaced.
        private Map<String, String> variableToBind = new Map<String,
String>();

        private String assignBindVar(Object bindValue){
        bindIdx++;
        if (1 == bindIdx) {
            bind01 = bindValue;
            return 'bind01';
        } else if (2 == bindIdx ) {
            bind02 = bindValue;
            return 'bind02';
        } else if (3 == bindIdx ) {
            bind03 = bindValue;
            return 'bind03';
                }
        }

        public ccServiceDao(Map<String, Object> inputData) {
                modifiedQueryString = (String) inputData.get(ccService.
QUERYSTRING);
                // iterate over the input and create a mapping of
keyName (bind var) to bind
                // check that the key name doesn't already exist.
                for (String key : inputData.keySet()) {
                        if (key != ccService.QUERYSTRING &&
variableToBind.get(key) == null ) {
                                // Sets the obect to the bind var.This
is the object AND puts the data to the name.
                                variableToBind.put(key, assignBindVar
(inputData.get(key)));
                        }
                }
                //System.debug('***** variableMap: ' + variableToBind);

                for(String key : variableToBind.keySet() ) { // Blow
down the list and replace the keys.
                        modifiedQueryString=modifiedQueryString.replace
(':' + key, ':' + variableToBind.get(key));
                }
                //System.debug('***** mqs (after): ' +
modifiedQueryString);
        }
```

Using the ccServiceDAO

Usage from the *ccService* class is simple:

1. The service implementors should construct queries in ways to reference bind variables in the subquery or the filter clauses by key.

```
        private static final Map<String, Object> FILTERMAP = new
Map<String, Object>{
        ccAPIProduct.PRODUCTID => ' Id = :' + ccAPIProduct.
PRODUCTID,
        ccAPIProduct.PRODUCTIDLIST => ' Id in :' + ccAPIProduct.
PRODUCTIDLIST
        }
```

2. Add any other computed bind variables (e.g. *locale* is usually converted to *fullLocale* and *langLocale*) to the inputData
3. Ensure that the input data holds the proper key names.
4. The *runQuery* service implementor method should call a *ccServiceDAO* constructor and then that classes *doQuery()* method.

```
    global virtual Map<String, Object> runQuery(Map<String,
Object> inputData) {
        ...
        try {
                        Map<String, Object> initRes=this.initSvcDAO
(inputData);
                        ccServiceDao queryRunner=(ccServiceDao)
initRes.get(ccService.SVCDAO);
                        inputData.put(ccService.QUERYRESULTS,
queryRunner.doQuery());


        } catch (Exception e) {
                        ...
        }
        return inputData;
    }
```

5. Process returned results by the *QUERYRESULTS* key.

Ad-Hoc Usage:

1. Create a Map that defines the:
    a. Query hashed under the *ccService.QUERYSTRING* key ...and...
    b. any bind parameters the caller wanted to inject.
2. Use a service method to init the DAO or create a ccService class and call the default implementation.

Ad Hoc Query Running

```
// Query for attachments, for example:

Map<String,Object> groupInput=new Map<String,Object>{
        ccService.QUERYSTRING=>'SELECT Id,Name,ParentId FROM Attachment
WHERE ParentId in :mediaIds',
        'mediaIds'=>allMediaIds
};


List<Attachment> mediaAttachments=(List<Attachment>) ((ccServiceDao)
this.initSvcDAO(groupInput).get(ccService.SVCDAO)).doQuery();
```

Iterative Querying

When data sets become very large in APEX, it is necessary to iterate over the dataset using a for-loop-query. This is possible using the *runQuery (buildQuery())* convention inside service classes. Using this approach, time and heap become the biggest limiters on the number of items a service can process.

Iterative Querying Example

```
for(E_Cart__c item : (List<E_Cart__c>) this.runQuery(this.buildQuery
(inputData)).get(ccService.QUERYRESULTS)){
        // Do things with carts...
}
```

Service Layer Platform Cache

Overview

Some service class implementations utilize platform cache to increase performance of fetch calls. The use of platform cache is built into the OOTB fetch call implementation. Any service override of the ccrz.ccService.fetch method will not be able to take advantage of the CloudCraze platform cache. Not all OOTB service classes take advantage of platform cache. Refer to the specific service class documentation.

Cache Related Parameters

The following constants are defined on ccrz.ccApi and utilized by the cache methods on ccrz.ccService:

- ccrz.ccApi.CACHE_KEYMAP
- ccrz.ccApi.CACHE_TTL
- ccrz.ccApi.CACHE_DATA
- ccrz.ccApi.CACHE_KEY
- ccrz.ccApi.CACHE_INP

Cache-related Methods

prepareCacheKey

The service provider method prepareCacheKey is responsible for constructing a Map<String,Object>, which is used to generate the key for the cached data. The returned map should contain the fields necessary to ensure that repeated calls to ccrz.ccService.fetch() with the parameters would always return the same set of data. The implementation should include all keys passed to the fetch method as well as any context-related data necessary for the fetch. For example, if the parameter ccrz.ccAPIProduct.PRODUCTSTOREFRONT is not passed to ccrz.ccServiceProduct.fetch(), then the parameter is implied to be ccrz.cc_CallContext.storefront.

A key point in understanding how the cache key is generated is that it considers all inputs. For example, calls to ccrz.ccApiCategory.fetch() with different sizing parameters will result in different cache keys, and therefore, cache entries, being created.

ccrz.ccService.prepareCacheKey is called by OOTB fetch implementations after all implied parameters have been resolved. The exact point where this occurs will vary by the specific ccrz.ccService class. The signature of ccrz.ccService.prepareCache is:

```
global virtual Map<String,Object> prepareCacheKey(Map<String,Object>
inputData)
```

The key constructed from the returned map is used to first see if data already exists in platform cache. If the data exists then that data is returned (no queries are performed). The ccrz.ccService.prepReturn is always called for the retrieved data.

The key is also used, if the data is not in platform cache, to store any retrieved data in platform cache.

Inputs

The passed map will contain all keys passed to ccrz.ccService.fetch() as well as any additional implied parameters. For example, for ccServiceProduct, the call will contain ccrz.ccAPIProduct.PRODUCTSTOREFRONT. Refer to the specific service class for a list of the parameters passed.

Outputs

The returned map should contain a single entry, ccrz.ccApi.CACHE_KEYMAP. This map should uniquely identify the request such that repeated calls to ccrz.ccService.fetch() would always return the same set of results.

Returning null or an empty Map<String,Object> from this method causes the underlying service class to not use platform cache for that call.

Example

```
global virtual override Map<String,Object> prepareCacheKey(Map<String,
Object> inputData){
        //Let the base service setup the core data
        Map<String,Object> superRet = super.prepareCacheKey(inputData);

        Map<String,Object> keyMap = (Map<String,Object>)superRet.get
(ccrz.ccApi.CACHE_KEYMAP);

        //Check that we don't get an empty map or null
        if(ccrz.ccUtil.isNotEmpty(keyMap)){
                //Add a new entry to the map.  This entry makes the
cache key unique only for
                //the current day.  Even if an underlying cache entry
is still valid across midnight GMT
                //from the perspective of the key generation this
request would try to access the cache
                //using a different underlying key.  Hence, this
effectively clears the cache for these
                //entries at midnight GMT.
                keyMap.put('aDateKey',Datetime.now().dayOfYearGmt());
        }
        return keyMap;
}
```

prepareCacheData

The prepareCacheData method can be used by implementers to manipulate data before that data is put into the cache.

The signature ccrz.ccService.prepareCacheData is:

```
global virtual Map<String,Object> prepareCacheData(Map<String,Object>
inputData)
```

Default, OOTB implementations of this method simply return the input data.

If the method returns null or an empty map then the data will not be cached.

Inputs

ccrz.ccApi.CACHE_DATA

A Map<String,Object> which is the data to be cached.

ccrz.ccApi.CACHE_TTL

An Integer which is the time to live (in seconds) for the cache data. This value specifies the amount of time for which the cached data remains available in the cache. For example, if `600` is passed, then the data will live within the cache for 10 minutes. After 10 minutes, the data can no longer be retrieved from the cache and must be requeried.

ccrz.ccApi.CACHE_KEY

A String which is the key used for the cached entry. This key is constructed from the return of the ccrz.ccService.prepareCacheKey method.

ccrz.ccApi.CACHE_INP

A Map<String,Object> containing the input data for the fetch method. This map may contain data above and beyond what was originally passed into the ccrz.ccService.fetch method. For example, for ccrz.ccServiceProduct, the implied parameters are added to the input data map passed to the fetch call.

Outputs

ccrz.ccApi.CACHE_DATA

A Map<String,Object> which is the data to be cached.

ccrz.ccApi.CACHE_TTL

An Integer which is the time to live (in seconds) for the cache data. This value specifies the amount of time for which the cached data remains available in the cache. For example, if `600` is passed, then the data will live within the cache for 10 minutes. After 10 minutes, the data can no longer be retrieved from the cache and must be re-queried.

ccrz.ccApi.CACHE_KEY

A String which is the key used for the cached entry. This key is constructed from the return of the ccrz.ccService.prepareCacheKey method.

Example

```
global virtual override Map<String,Object> prepareCacheData(Map<String,
Object> inputData){
        //Do not cache data for the storefront XYZStore
        if('XYZStore'.equals(ccrz.cc_CallContext.storefront)){
                return null;
        }

        //For ABCStore, set the TTL to half of the configured value
        if('ABCStore'.equals(ccrz.cc_CallContext.storefront)){
                Integer myTTL = (Integer)inputData.get(ccrz.ccApi.
CACHE_TTL);
                if(null != myTTL){
                        Map<String,Object> retMap = new Map<String,
Object>(inputData);
                        retMap.put(ccrz.ccApi.CACHE_TTL,myTTL/2);
                        return retMap;
                }
        }
        return inputData;
}
```