## Array Store

* Provide an interface to load and edit in-memory array, and handle related events.
* Array store is immutable.

### Array Store configuration

CONFIGURATION – properties to configure ArrayStore.

* Data – array associated with the store.
* errorHandler - Specifies the function that is executed when the store throws an error.
* Key – key prop(s) to access data item uniquely.
* onLoading, onLoaded – function for data load to store.
* onInserted, onInserting – function for individual data item addition.
* onUpdated, onUpdating – function for individual data item updation.
* onRemoved, onRemoving – function for individal data item removal.
* onModified, onModifying – function for dataitem add, update or removed.

### Array Store methods

METHOD – methods to control array store.

* byKey(key) – get an data item with specific key.
* clear() – clear all data associated with array store.
* createQuery() – create Q for underlying array.
* Insert(values) – add a data item to store.
* key() – get key property.
* keyOf(obj) – get key value of a data item
* load() – start loading data.
* off(eventName),
* off(eN, handler),
* on(eN, handler),
* on(events),
* push(changes) – push the changes
* remove(key) – remove a data item from store.
* totalCount(options) – get total count of items.
* Update(key, value) – update the data item with a specific key.

### Options in detail

onInserted(value, key)

* Type – function
* Param – value (obj), key(obj, string, number)

onInserting(value)

* Type – fn
* Param – value(obj)

onLoading(loadOptions) – how to load the data in datasource component

* loadOptions.filter: An object that specifies filter conditions for the data.
* loadOptions.sort: An array of objects that specifies sorting options for the data.
* loadOptions.select: An array of strings that specifies which fields to include in the loaded data.
* loadOptions.skip: A number that specifies the number of records to skip when loading data.
* loadOptions.take: A number that specifies the number of records to load.
* loadOptions.userData: Custom data passed to the load operation.

onLoaded(result) – once the data items are loaded to the store.

onModified() and onModifying()

* Type: fn

onPush(changes)

* Type: fn
* Param: Array<any>

onRemoving(key) and onRemoved(key)

* Tyep: fn
* Param: object, string, num

onUpdating(key, value) and onUpdated(k, v)

* Type: fn
* Param: key-obj, str, num; v – obj

### Methods in detail

byKey(key)

* Key – obj, string, num
* Ret – promise (jquery or native)

clear() – clear array associated with it \_array = []

createQuery() – create query for underlying array

* Ret – the Query object
* NOTE: Query object is used to perform query operation on underlying store like, **filtering**, **sorting**, **grouping**, and **selecting** **specific** **fields.**

Insert(value) – adds a data item to the store

* Param – value (obj)
* Ret – promise (jquery)

Key() – return key property (obj, string, num)

keyOf(obj) – returns the key portion of given object. The obj need not be from ArrayStore items.

load() and load(loadOptions) – Used to load / retrieve data from underlying \_array.

* Ret – promise (jquery) => .done’s callback contains result as param.

Off(eventName) and off(eventName, eventHandler) – used to detach event attached using on methods.

* Params – eventName (string), eventHandler (func)
* Ret – ArrayStore instance

On(eventName, eventHandler) – subscribe to an event. (param and ret - same as above).

On(events) – subscribe to events.

remove(key)- removes a data item with a specific key from the store.

* Param – key (obj, string, num)
* Return – promise

totalCount(loadOptions) – helps in getting count of items from load() method.

* Param – this loadOptions is passed to load method
* Ret – promise, when on resolved get in then’s callback param.

Update(key, value) – updates a data item with specified key

* Params – key (string, obj, num), value (obj)
* Ret – promise

## Custom Store

* CustomStore enables you to impl. Your own **data acess logic** for consuming data from **any source**.
* CustomStore impl. Depends on where u are processing data, on client or server.
* For client data processing => u must impl. Load method to load data from data source.
* Motivation befind custom store – CustomStore is more flexible when interacting with data source like remote server. You can implement custom logic for load, insert, delete, update. It can be used when real time updates are crutial.

### Custom Store configuration

* byKey – custom impl. For getting data item by key from store.
* cacheRawData: Specifies raw data should be cached? (loadMode: “raw” must).
* errorHandler – when store throws error this handler is invoked.
* Insert – custom impl. For inserting data item to store.
* Key – specified what part of data item play(s) part of key.
* Laod – custom impl. For load(options) method.
* loadMode – specify how data is treated returned by load method.
* onInserting, onInserted, onModifiying, onModified, onRemoving, onRemoved, onLoading, onLoaded, onPush, onUpdated, onUpdating.
* Remove – custom impl. For remove(key) method.
* totalCount – custoim impl for totalCount(options)
* update – custom impl for update(key, value)
* useDefaultSearch – true if loadMode = “raw” and false if it is “processed”.

### Custom Store methods - specified methods used to access data associated with custom store

* byKey(key) – gets a data item with a specific key
* clearRawDataCache (key) – delete data from the cache (cacheRawData must be true).
* Insert(value) – adds a data item to store
* Key() – get key property of custom store
* keyOf(obj) – gets obj’s key portion
* load() or laod(option) – starts loading data.
* Off and on
* Push(changes) – pushes data changes to store and notified the DataSource.
* Remove(key) – remove a data item with a specific key
* totalCount(option) – gets total no of items the laod function returns
* update(key, value) – updates a data item with a specific key

### CustomStore Option in detail (important method only)

byKey

cacheRawData – true if you are caching raw data (only meaning ful when loadMode is raw).

* Type : bool (true)

Insert(value) – custom impl to insert an data item to to store.

* Type: fn
* Param: value (obj)
* Ret – promise

Key

load() and load(options) – specifies custom impl for load(options) method.

* Type: fn
* Param – option (obj)
* Ret – promise or array

loadMode – specifies how data returned by load method is treated.

* Type: string
* Default – “processed”
* Accepted values – “raw”| “processed”

onInserted(value, key), onInserting(value)

onLoaded(result), onLoading(loadOptions)

onModified(), on Modifying()

onPush(changes) – executed before the changes are pushed to the store.

onRemoved(key), onRemoving(key)

onUpdated(key, value), onUpdating(key, value)

remove(key) – specify custom impl of remove(key) method.

* Type: fn
* Param – obj, string, num
* Ret - promise

totalCount – specifies custom impl of totalCount(loadOptions)

* Type: fn
* Param: loadOption (obj)
* Ret – promise

Update – specifies custom impl of update(key, value)

useDefaultSearch – specifies the store combines search and filter expr?

* Type - bool
* Default value – undefined

### CustomStore methods in detail (only important methods)

byKey(key)

clearRawDataCache() – cacheRawData must be **true.**

Insert(value)

Key()

Load(options)

Off, on methods

push(changes) – pushes data changes to the store and notifies to DataSource

remove(key)

totalCounts(options)

update(key, value)

## Load Option object

* load option object specifies setting according to which server processes data.
* This object is more often passed to load method

### Options

* **customQueryParams** – object for storing additional setting that is to be send to server (relevant to OdataStore only).
* **expand** – array of string that represent **names of navigation** properties to be loaded simultaneously with the OdataStore.
* **Filter** – defines filtering parameters. Possible values: =, <>, >, >=, <, <=, startswith, endswith contains, notcontains

E.g.,

‘datafield’, ‘=’, 3],

[ "!", [ "dataField", "=", 3 ] ],

[ [ "dataField", "=", 10 ], "and", ["anotherDataField", "<", 3 ], “or",["anotherDataField", ">", 11 ]]]

url = <https://services.odata.org/V4/OData/OData.svc/Products>

url?$filter=ID eq 3

url? $filter=not(dataField eq 3)

url?$filter=(dataField eq 10) and ((anotherDataField lt 3) or (anotherDataField gt 11))

* **Group –** specify to load data from data source in group.Defines grouping level to be applied to data. (arrray, obj, string)
  + **Selector** – field name to group by
  + **Desc** – order in which group by
  + **isExpanded** –whether to return the group’s item or null.
  + **groupInterval** – numeric value (in length) if group field is numeric. A string in case of date type.
* **groupSummary** – adds summary to each group. Value: "sum", "avg", "min", "max" or "count".
* **parentId** – id of rows that can be expanded.
* **requireGroupCount** – whether top level group count is required.
* **requireTotalCount** – whether total count of data object is required.
* **searchExpr** (name)**, searchOperation** (contains)**, searchValue** (prajval) - used for simple search or filter operation.
* **select –** used to specify specific column or field to load.
* **skip –** specify no. of data items to be skipped.
* **sort –** defines sorting strategy (on which field and how to sort)
* **take** – no of data object to be loaded.
* **totalSummary** – specifies on which field what summary operation to perform. { selector: "field", summaryType: "sum" }.
* **userData** – additional setting to be send to server.

## DataSource API

* DataSource is an object that proivde an API for processing data from an underlying store.
* NOTES:
  + If created a DataSource instance outside an UI component then it must be disposed when no longer needed,
  + If created inside an UI compoent, it is disposed automatically.
  + DataSource is not made to be used b/w multiple component simultaneously. Instead maintain a common store object for those components.
* DataSource is immutable, cannot change configuration at runtime.

### DataSource options

**customQueryParams** – passed to Odata service with load query. (only in consideration while working with ODataStore).

**expand** – specifies the navigation property that is to be loaded with Odata entity. (only in consideration while working with ODataStore).

**OData entity** – refers to a **single data record** or an **object** that can be identified and manipulated using an unique identifier. Entities are fundamental building blocks of an Odata Service and they represent the that can be queried, created, updated or deleted.

**Filter** – specify the data filtering condition.

**group** – specify data grouping properties.

**langParams** – specify params for lang specific sorting and filtering.

**map** (fn) – specify an item mapping function.

**onChanged** (fn) – executed after data is **loaded**.

**onLoadError** (fn) – executed when error while loading.

**onLoadingChanged** (fn) – executed when data loading status change.

**pageSize** – specify max no of datan items per page. (only applied when paginate: true).

**paginate** – whether load data items by pages or all at once.

**postProcessing** (fn) - allows you to perform custom processing on the data after it has been loaded from the underlying data store but before it is used by the data source.

**pushAggregationTimeout** – in milliseconds, changes are aggregated before changes are pushed to DataSource.

**requireTotalCount**

**reshapeOnPush** – whether to reapply **sorting**, **filtering**, **grouping** and other data processing operation after receving push.

**searchExpr** – specify field to search.

**searchOperation** – specify opertion to be applied while searching in data source

**seachValue** – value to which search expression is compared.

**Select** – fields to be selected from data objects.

**Sort** – specify data sorting properties.

**Store** – configure the store underying the DataSource.

Note:

* not specifying store option, DataSource will maintain a CustomStore internally and you can configure this custom store by providing customstore options in options of DataSource.
* And if you define customstore properties the it will override the store option.

### DataSource methods

Describe methods to control DataSource.

* Cancel(opnId) – cancels the load opn with specified identifier.
* Dispose() – dispose all resources allocated to the DS instance.
* filter() and filter(filterExpr) – gets and sets filter property’s value.
* group() and group(grpExpr) – gets and sets group property’s value.
* isLastPage() – checks if current page items count is less than pageSize.
* isLoaded() – checks whether data is loaded in the DataSource.
* isLoading() – checks whether data is being loading into the dataSource.
* Items() – gets an array of data items on current page.
* Key() – gets the value of the underlying store’s key property.
* Load() – starts loading data.
* loadOptions() – get an object of current data processing setting.
* Off(), on(), …
* pageIndex() and pageIndex(newIndex) – sets and get page index. (on next load method call).
* pageSize() and pageSize(size) – gets and sets page size.
* Paginate() and paginate(value) – gets and sets paginate property value.
* Reload() – clear DS current itmes and called laod() method.
* requireTotalCount() and requireTotalCount(value) – gets and sets requireTotalCount property’s value.
* searchExpr () and searchExpr(value) – gets and sets searchExpr property value.
* searchOpertion and searchOpertion(value) – gets and sets searchOpns property’s value.
* searchValue and searchValue (value) – gets and sets searchValue property’s value.
* select() and select(value) – gets and sets select property value.
* sort() and sort(value) – gets and sets sort property value.
* store()- gets an instance of underlying store.
* totalCount() - Gets the number of data items in the store after the last load() operation without paging.

### DataSource Events

* changed – data loaded
* loadError – loading fail
* loadingChanged – data loading status is changed

## LocalStore API

LocalStore provide an interface for loading and editing data from HTML web storage (window.localstorage). unlike ArrayStore, LocalStore donot store data in in-memory rather it store it on disk.

We need to specify **name** under which it will save our data in local storage.

localStore is immutable, cannot change configuration at runtime.

### LocalStore Confiuration

**data** – array of items.

**errorHandler (fn)** – execute when store throws an error.

**flushInterval** – milliseconds, specifies the time at which data changes and the moment when these changes are saved in local storage.

**Immediate** – whether changes must be reflected to store immdiately.

**Key** – specify the key property(s) to access data item uniquely.

**name** – acts as key to manipulate in local storage. dx-data-localStore- is prefix to [name].

**onInserting(value)** and **onInserted(value, key)** – donot wait for insertion in local storage.

**onLoading(loadOptions)** and **onLoaded(result)** – before and after data is loaded to the store.

**onPush(changes)** – executed before changes are push to store. (Push method donot update the remove data soruce here local storage)

**onRemoving(key)** and **onRemoving(key)** – before and after a data item is removed.

**onUpdating(key, value)** and **onUpdated(key, value)** – before and after a data item is updated.

**onModifying ()** and **onModified()** – before and after data is inserted, updated or removed.

### LocalStore methods

Describes methods that control local store.

**byKey(key)** – get data item by specific key.

**clear()** – removes data from local storage.

**createQuery()** – create and returns a Query object underlying array.

**Insert(value)** – add data item to store.

**Key() –** get key option’s value.

**keyOf(obj)** – gets key portion from passed object.

**load()** and **load(options)** – starts loading data

const loadOptions = {

**filter**: ['name', '=', 'John Doe'],

**sort**: [{ field: 'name', desc: false }],

**group**: [{ field: 'category', desc: false }],

**select**: ['id', 'name', 'category'],

**skip**: 0,

**take**: 10,

**requireTotalCount**: true

};

off() and on() methods

push() – push changes to \_array. It donot modified to localstorage.

remove(key) – remove a data with a specific key from the store.

totalCount(options) – gets the total count of items the load() function returns.

update(key, value) – update a data item with specific key.

## Query object

In DevExtreme, the load function and loadOptions are typically used with data sources such as ArrayStore, CustomStore, and others to handle data loading operations. The Query object, on the other hand, provides a more flexible and powerful way to work with data on the client side. Here’s a detailed comparison of their purposes and use cases:

### load Function and loadOptions

**Purpose**: The load function is used to retrieve data from a data source. The loadOptions parameter specifies the criteria for data loading, such as filters, sorting, pagination, and grouping.

**Typical Usage**: These are typically used when you have a predefined data source and want to load data based on specific criteria. The data source can be a remote server, local array, or any custom source.

### Query Object

**Purpose**: The Query object is a utility for performing complex data manipulations directly on the client side. It provides methods for filtering, sorting, grouping, aggregating, and selecting data.

**Typical Usage**: The Query object is used when you need to perform advanced data operations without relying on the data source’s built-in capabilities. It is particularly useful for scenarios where you want to chain multiple operations and transform the data dynamically.

### Why Use Query?

**Flexibility**: The Query object allows chaining multiple operations, providing a fluent API for complex data transformations.

**Client-Side Processing**: It is designed for client-side data manipulation, which can be more efficient when working with local data.

**Advanced Operations**: It supports advanced operations like grouping, aggregating, and selecting specific fields, which might be more cumbersome to implement using loadOptions.

### Query methods

**aggregate(seed, step, finalize)** – helps to have custom summaries for data items. This method is used to perform complex aggregation that are not covered in summary fns like, sum, count, avg, etc.

seed = initial value,

step = (agg, item) => {modify agg and return agg}

const finalize = agg => {return modification on agg}

**aggregate(step)** – omits seed and finalize, it’s a short cut. For 1’st step the item value is first item.

**avg()**- only applies to numeric value. Cals avg of all values.

**avg(getter)** – cals the avg of all values got using getter. Usually getter will be used as name of field that is numeric type.

**count()** – cals no of data items.

**enumerate()** – executes the query. This is asynchronous alternative for toArray() method.

**filter(criteria)** – filters the data item using a filter expr.

**filter(predicate)** – filter the data items using custom filter function. Fn returns t or f.

**groupBy(getter)** – group data items by getter

groupBy(“category”

groupBy(item => item.price > 50 ? “High”: “low”)

groupBy([“category”, “subCategory”])

**max() and max(getter) –** max() applies for numeric values only.

First, call the select(getter) method to select the object field that provides numeric values if the Query is associated with an array of objects, or use the max(getter) method instead of max().

**Min() and min(getter)** - min() applies for numeric values only.

First, call the select(getter) method to select the object field that provides numeric values if the Query is associated with an array of objects, or use the min(getter) method instead of min().

**select(getter)** – select individual field from the data items.

select(‘field’)

select(item => ({  
 id: item.id,

name: item.firstName + “ “ + item.lastName,

ageGroup: item.age > 30 ? “old” : “young”,  
}))

select([‘name’, item => item.firstName + “ “ + item.lastName, item => item.age > 30 ? “old” : “young”,])

**slice(skip, take)** – get `take` no of items from `skip` index.

**sortBy(getter)** – sort data items by the specified getter in asc order.

sortBy(‘age’)

sortBy((a, b) => a.name.localCompare(b.name, 'en', { sensitivity: 'base' })) // used when two fields are involved, OR sort on nested properties, OR custom comparator non standard way of comparision

sortBy(['age', (a, b) => a.name.localeCompare(b.name)])

**sortBy(getter, desc)** - Sorts data items by the specified getter in the specified sorting order.

**sum()** – cals the sum of all values.

**sum(getter)**  - cals the sum of all values found using a getter.

NOTE: In the sum method of DevExtreme's Query object, the getter parameter is not an array because the purpose of the sum method is to calculate the sum of a single numeric property in the data set. Unlike the select or sortBy methods, which may accept multiple criteria for transformation or sorting, the sum method focuses on a single numeric property for calculation.

**thenBy( getter)**  - sorts data item by one more getter in ascending order. This method can only follow the sortBy(getter), sortBy(getter, desc), thenBy(getter, desc), or another thenBy(getter) method call.

**thenBy(getter, desc)**  - sorts data item by one more getter in specified sorting order.

**toArray()**  - gets data items associated with the **Query**. A synchronous approach to the enumerate() method.