Creatio Academy

Back-end development

Data storage and cache

Version 8.0

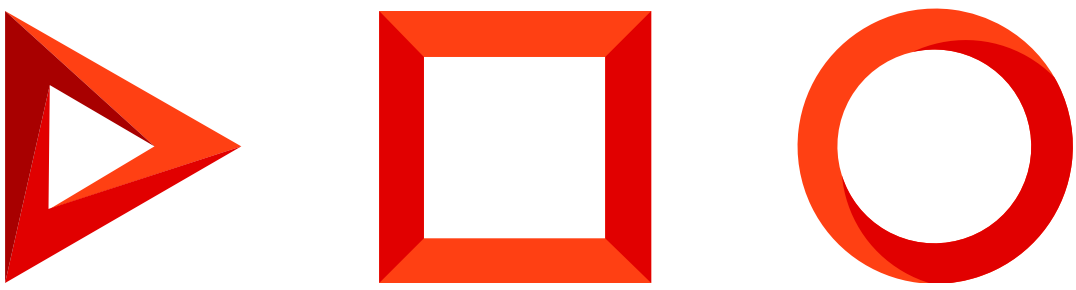

Table of Contents

Data storage and cache

Medium

The **purpose** of storages is to partition stored data logically and simplify its further use in source code.

Creatio storages let you execute the following **actions**:

- access data to read or write it via a key
- delete data from storage via a key

Storage types

Creatio supports the following storage **types**:

- data storage
- cache

Storage partitioning lets you manage the following **actions**:

- isolate data to specific workspaces and user sessions
- classify data arbitrarily
- manage data life cycle

Data storage and cache can be located physically on an arbitrary data storage server. The exception is the `Request` level data stored in memory.

Redis is a data storage server. Any storage accessed via unified interfaces can be used as a storage server. Keep in mind that storage access operations are resource-intensive since they involve data serialization/deserialization and network communication.

Data storage

The **purpose** of data storage is intermediate storage of rarely changed (i. e., long-term) data. View the data storage levels in the table below. The `Terrasoft.Core.Store.DataLevel` enumeration contains data storage levels. Learn more about the `DataLevel` enumeration in the .NET class library.

Data storage levels

Level	Description	Limits to the object life cycle
`Request`	**Request level**. Data is available while the current query is being handled.	Objects are stored while the query is being executed.
`Session`	**Session level**. Data is available in the current user session.	Objects are stored while the session exists.
`Application`	**Application level**. Data is available for the entire Creatio application.	Objects are stored while the Creatio instance exists. To delete objects from the storage, clear the external storage.

The storage retains data until data is deleted explicitly.

Cache

The **purpose** of cache is to store operational information. Cache storage comprises multiple levels. View the levels in the table below. The `Terrasoft.Core.Store.CacheLevel` enumeration contains storage levels. Learn more about the `CacheLevel` enumeration in the .NET class library.

Cache levels

Level	Description	Limits to the object life cycle
`Session`	**Session level**. Data is available in the current user session.	Objects are deleted when the session expires.
`Workspace`	**Workspace level**. Data is available to all workspace users.	Objects are deleted when you delete a workspace.
`Application`	**Application level**. Data is available to all Creatio users regardless of their workspace.	Objects are stored while the Creatio instance exists. To delete objects from the storage, clear the external storage.

Data stored in the cache has obsolescence time. The **obsolescence time** is the time limit of the cache item relevancy. Regardless of the obsolescence time, Creatio deletes all items from the cache when their life cycle expires.

You can also delete data from the cache at any time. Therefore, in some situations, the source code might attempt to retrieve cached data that has already been deleted. In this case, retrieve data from the permanent storage and put it in the cache.

The `Terrasoft.Core.Entities.EntitySchemaQuery` class implements a mechanism that works with the storage (Creatio cache or an arbitrary user-defined storage). The use of cache optimizes the efficiency of operations by accessing cached query results without an additional database query. When executing the `EntitySchemaQuery` request, Creatio adds the key (`CacheItemName` property value)and data retrieved from the database to the cache (

`Cache` property value). By default, Creatio caches `EntitySchemaQuery` queries in the session level cache that stores data locally. You can also cache queries in an arbitrary storage that has the `ICacheStore` interface implemented.

View the example that works with Creatio cache when executing an `EntitySchemaQuery` query below.

Example that works with Creatio cache when executing an `EntitySchemaQuery` query

```
/* Create an EntitySchemaQuery instance that has the [City] root schema. */
var esqResult = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");

/* Add a column that contains the city name to the query. */
esqResult.AddColumn("Name");

/* Specify the key to access the query result cache.
Data is located in the session level cache that stores data locally because the Cache object pro
esqResult.CacheItemName = "EsqResultItem";

/* Execute a database query to retrieve the resulting object collection.
The query results are cached after the operation is executed. When esqResult is accessed later t
esqResult.GetEntityCollection(UserConnection);
```

Storage object model

The classes and interfaces of the `Terrasoft.Core.Store` namespace implement the mechanism that works with the data storage and cache. Learn more about the namespace in the .NET class library.

`IBaseStore` interface

The `Terrasoft.Core.Store.IBaseStore` interface defines the base features of the storage types.

The `IBaseStore` interface lets you implement the following **actions**:

- access data to read or write via a key (the `this[string key]` indexer)
- delete data from storage via a specified key (the `Remove(string key)` method)
- initialize the storage via a specified parameter list (the `Initialize(IDictionary parameters)` method) Creatio reads parameters to initialize storages from the configuration file. Set the parameter list in the `storeDataAdapter` (for data storage) and `storeCacheAdapter` (for cache) sections. You can set the parameters arbitrarily.

`IDataStore` interface

The `Terrasoft.Core.Store.IDataStore` interface defines the specifics of working with data storages. It inherits from the `IBaseStore` base storage interface. The interface lets you retrieve a list of all storage keys (the `Keys` property).

Attention. We recommend using the `Keys` property when working with data storages if it is the only way to solve the problem.

`ICacheStore` interface

The `Terrasoft.Core.Store.ICacheStore` interface defines the specifics of working with cache. It inherits from the `IBaseStore` base storage interface. The interface lets you optimize the storage workflow when retrieving a data set simultaneously. Also, it implements the `GetValues(IEnumerable keys)` method. The method returns a cache object dictionary that contains the specified keys.

`Store` class

The `Terrasoft.Core.Store.Store` static class accesses cache and data storages of various levels.

The `Store` class includes the following static **properties**:

- `Data`. Returns an instance of the data storage provider.
- `Cache`. Returns an instance of the cache provider.

Attention. To ensure the storages operate as intended in Creatio .NET Core or .NET 6, replace static properties with a connection via `UserConnection`.

Access the data storage and cache

You can access the data storage and cache in the following **ways**:

- via `UserConnection`
- via proxy classes

Access the data storage and cache via `UserConnection`

You can use the static properties of the `Store` class to access Creatio data storages and cache from the source code. Alternatively, you can access the data storage and cache via the `UserConnection` instance. This method lets you avoids using long property names and connecting additional assemblies. Replace static properties with a connection via `UserConnection` to migrate from .NET Framework to .NET Core or .NET 6.

The additional `UserConnection` class **properties** that let you access the data storage and cache on various levels quickly are as follows:

- `ApplicationCache`. Returns a link to the `Application` level cache.
- `WorkspaceCache`. Returns a link to the `Workspace` level cache.
- `SessionCache`. Returns a link to the `Session` level cache.
- `RequestData`. Returns a link to the `Request` level data storage.
- `SessionData`. Returns a link to the `Session` level data storage.

- `ApplicationData`. Returns a link to the `Application` level data storage.

View the example that works with cache via the `UserConnection` class below.

Example that works with cache via the `UserConnection` class

```
/* The key with which the value is added to the cache. */
string cacheKey = "SomeKey";

/* Add a value to the session level cache via the UserConnection property. */
UserConnection.SessionCache[cacheKey] = "SomeValue";

/* Retrieve a value from the cache via the Store class property. As a result, the valueFromCache
string valueFromCache = UserConnection.SessionCache[cacheKey] as String;
```

Access data storages and cache via proxy classes

The **proxy classes** are an intermediate element between storages and code that accesses the storages. The **purpose** of proxy classes is to execute intermediate actions with data before it is read from storage or written to storage. Each proxy class is a storage.

Use proxy classes to implement the following **actions**:

- perform the initial Creatio setup and configuration
- isolate data of the Creatio users
- execute other intermediate actions with data before adding it to the storage

To **configure the use of proxy classes for data storage and cache**:

1. Add a `proxies` section to the `storeDataAdapters` and `storeCacheAdapters` sections of the `Web.config` configuration file in the Creatio root directory.
2. List the storage proxy classes in the `proxies` section.

Creatio retrieves the settings from the configuration file and applies them to the storage type when Creatio is loaded. As such, you can create **chains of proxy classes** called consecutively. The call order of proxy classes corresponds to their order in the configuration file. The proxy class listed last in the `proxies` section is called first in the chain.

The setup of proxy class chains has the following **special features**:

- The endpoint of a proxy class chain is the cache or data storage for which the chain is defined.
- Each proxy class works with either data storage or cache. Define the storage type in either the `ICacheStoreProxy.CacheStore` or `IDataStoreProxy.DataStore` property. The property can link to another proxy class, a storage, or cache, but this is unknown to the proxy class. A proxy class can be a storage with which other proxy classes work.

View the example that configures proxy classes below.

Example that configures proxy classes

```
<storeDataAdapters>
    <storeAdapter levelName="Request" type="RequestDataAdapterClassName">
        <proxies>
            <proxy name="RequestDataProxyName1" type="RequestDataProxyClassName1" />
            <proxy name="RequestDataProxyName2" type="RequestDataProxyClassName2" />
            <proxy name="RequestDataProxyName3" type="RequestDataProxyClassName3" />
        </proxies>
    </storeAdapter>
</storeDataAdapters>

<storeCacheAdapters>
    <storeAdapter levelName="Session" type="SessionCacheAdapterClassName">
        <proxies>
            <proxy name="SessionCacheProxyName1" type="SessionCacheProxyClassName1" />
            <proxy name="SessionCacheProxyName2" type="SessionCacheProxyClassName2" />
        </proxies>
    </storeAdapter>
</storeCacheAdapters>
```

The call chain of data storage proxy classes is as follows: `RequestDataProxyName3` → `RequestDataProxyName2` → `RequestDataProxyName1` → `RequestDataAdapterClassName` (the final data storage on the `Request` level).

Proxy classes let you isolate data to specific users. The easiest way to do this is to transform value keys before adding them to the repository, for example, by adding a user prefix to the key. Such proxy classes ensure the storage keys are unique. This avoids data loss and corruption when different users write values using the same key simultaneously.

Proxy classes let you implement the mechanism that executes arbitrary actions with data before data is added to or retrieved from the storage. Since the proxy class implements data handling logic, you do not need to duplicate code, which makes the code easier to modify and maintain.

Base proxy class interfaces

- `Terrasoft.Core.Store.IDataStoreProxy`. Interface for proxy classes of the data storage.
- `Terrasoft.Core.Store.ICacheStoreProxy`. interface for proxy classes of the cache.

Implement one or both interfaces to use a class as a proxy class for working with storage.

Each interface has one property that contains a link to the storage or cache with which the current proxy class works. It is the `DataStore` property for the `IDataStoreProxy` interface and `CacheStore` property for the `ICacheStoreProxy` interface.

Proxy classes that transform keys

The mechanism that transforms storage value keys is implemented in the following **proxy classes**:

- `Terrasoft.Core.Store.KeyTransformerProxy`. Abstract base class of all proxy classes that transform cache

keys. Implements methods and properties of the `ICacheStoreProxy` interface. Inherit from the `KeyTransformerProxy` class when you create custom proxy classes to avoid duplicate logic.

- `Terrasoft.Core.Store.PrefixKeyTransformerProxy`. Proxy class that transforms cache keys by adding a specified prefix.

  View the example that works with the session level cache via the `PrefixKeyTransformerProxy` proxy class below.

  Example that works with the session level cache via the `PrefixKeyTransformerProxy` proxy cla...

  ```
  /* The key with which to add the value to the cache via the proxy class. */
  string key = "SomeKey";

  /* The prefix to add to the key value using the proxy class. */
  string prefix = "customPrefix";

  /* Create a proxy class that writes values to the session level cache. */
  ICacheStore proxyCache = new PrefixKeyTransformerProxy(prefix, Store.Cache[CacheLevel.Session

  /* Write values that have the "SomeKey" key to the cache via the proxy class. Essentially, wr
  proxyCache[key] = "CachedValue";

  /* Retrieve values that have the "key" key via the proxy class. */
  var valueFromProxyCache = (string)proxyCache[SomeKey];

  /* Retrieve values that have the "prefix + key" key directly from session level cache. */
  var valueFromGlobalCache = (string)UserConnection.SessionCache[prefix + key];
  ```

  As a result, the `valueFromProxyCache` and `valueFromGlobalCache` variables will contain the same `CachedValue` value.

- `Terrasoft.Core.Store.DataStoreKeyTransformerProxy`. Proxy class that transforms data storage keys by adding a specified prefix.

  View the example that works with the data storage via the `DataStoreKeyTransformerProxy` proxy class below.

  Example that works with the data storage via the `DataStoreKeyTransformerProxy` proxy class

  ```
  /* The key with which to add the value to the storage via the proxy class. */
  string key = "SomeKey";

  /* The prefix to add to the key value using the proxy class. */
  string prefix = "customPrefix";

  /* Create a proxy class that writes values to the session level data storage. */
  IDataStore proxyStorage = new DataStoreKeyTransformerProxy(prefix) { DataStore = Store.Data[D
  ```

```
/* Write values that have the "SomeKey" key to the storage via the proxy class. Essentially,
proxyStorage[key] = "StoredValue";

/* Retrieve values that have the "SomeKey" key via the proxy class. */
var valueFromProxyStorage = (string)proxyStorage[key];

/* Retrieve values that have the "prefix + key" key directly from session level storage. */
var valueFromGlobalStorage = (string)UserConnection.SessionData[prefix + key];
```

As a result, the `valueFromProxyStorage` and `valueFromGlobalStorage` variables will contain the same `StoredValue` value.

Local data caching

The **local data caching** mechanism is based on proxy classes. The **purpose** of data caching is to reduce the load on the data storage server and execution time of requests when working with rarely changed data.

The `LocalCachingProxy` internal proxy class implements the local caching mechanism. The class caches data on the current node of the web farm. Learn more in Wikipedia.

The class inspects the lifetime of cached objects and retrieves data from the global cache if the cached data is obsolete.

To cache data locally, use the following **extended methods** of the `ICacheStore` interface in the `CacheStoreUtilities` static class:

- `WithLocalCaching()`. Overloaded method that returns the `LocalCachingProxy` class instance.
- `WithLocalCachingOnly(string)`. Caches data of the specified element group locally and monitors data relevance.
- `ExpireGroup(string)`. Sets the obsolescence flag for a specified element group. When you call this method, all elements of the specified group become irrelevant and are not returned upon data query.

View the example that works with the workspace cache via local caching below.

Example that works with the workspace cache via local caching

```
/* Create a proxy class that caches data locally. The elements that are written to the cache via
ICacheStore cacheStore1 = Store.Cache[CacheLevel.Workspace].WithLocalCaching("Group1");

/* Add an element to cache via the proxy class. */
cacheStore1["SomeKey1"] = "Value1";

/* Create a proxy class that caches data locally. The elements that are written to the cache via
ICacheStore cacheStore2 = Store.Cache[CacheLevel.Workspace].WithLocalCaching("Group1");
cacheStore2["SomeKey2"] = "Value2";

/* Set the obsolescence flag for the elements in the Group1 group. Items that have SomeKey1 and
cacheStore2.ExpireGroup("Group1");
```

```
/* Attempt to retrieve values that have SomeKey1 and SomeKey2 keys from the cache after the elem
var cachedValue1 = cacheStore1["SomeKey1"];
var cachedValue2 = cacheStore1["SomeKey2"];
```

Special features of using data storage and cache

To increase the efficiency of working with data storage and cache, keep in mind the following **special features**:

- Only serializable objects can be added to storage. This is due to the specifics of working with the Creatio core storage. The object is pre-serialized when data is saved to storage, and deserialized when it is retrieved. The exception is the `Request` level data storage.
- Avoid redundant queries to the repository in the source code because storage access operations are resource intensive.

  View the examples that work with storages below.

  Optimal code (option 1)

  ```
  /* Write an object from the storage to an intermediate variable. */
  object value = UserConnection.SessionData["SomeKey"];

  /* Check the value of the intermediate variable. */
  if (value != null) {
      /* Return the value. */
      return (string)value;
  }
  ```

  Optimal code (option 2)

  ```
  /* Use the GetValue() extended method. */
  return UserConnection.SessionData.GetValue<string>("SomeKey");
  ```

  Non-optimal code

  ```
  /* Address network and deserialize data. */
  if (UserConnection.SessionData["SomeKey"] != null) {
      /* Address network and deserialize data once again. */
      return (string)UserConnection.SessionData["SomeKey"];
  }
  ```

Optimal code

```
/* Delete data from the session level storage without previous verification. */
UserConnection.SessionData.Remove("SomeKey");
```

Non-optimal code

```
/* Address network and deserialize data. */
if (UserConnection.SessionData["SomeKey"] != null) {
    /* Address network and deserialize data once again. */
    UserConnection.SessionData.Remove("SomeKey");
}
```

- Creatio executes any status changes of the object retrieved from the data storage or cache in memory locally and does not commit them to the storage automatically. To ensure the changes appear in the repository, write the changed object to the repository explicitly.

  View the example that adds data to the storage below.

  Example that adds data to the storage

```
/* Retrieve the value dictionary from the session data storage via the "SomeDictionary" key.
Dictionary<string, string> dic = (Dictionary<string, string>)UserConnection.SessionData["Some

/* Change the values of a dictionary item. Changes are not committed to the storage. */
dic["Key"] = "ChangedValue";

/* Add an item to the dictionary. Changes are not committed to the storage. */
dic.Add("NewKey", "NewValue");

/* Commit a dictionary to the data storage via the "SomeDictionary" key. The changes are comm
UserConnection.SessionData["SomeDictionary"] = dic;
```