

Хранилище кэша

Хранилища данных и кэш

Версия 8.0



Эта документация предоставляется с ограничениями на использование и защищена законами об интеллектуальной собственности. За исключением случаев, прямо разрешенных в вашем лицензионном соглашении или разрешенных законом, вы не можете использовать, копировать, воспроизводить, переводить, транслировать, изменять, лицензировать, передавать, распространять, демонстрировать, выполнять, публиковать или отображать любую часть в любой форме или посредством любые значения. Обратный инжиниринг, дизассемблирование или декомпиляция этой документации, если это не требуется по закону для взаимодействия, запрещены.

Информация, содержащаяся в данном документе, может быть изменена без предварительного уведомления и не может гарантировать отсутствие ошибок. Если вы обнаружите какие-либо ошибки, сообщите нам о них в письменной форме.

Содержание

Хранилища данных и кэш	4
Хранилище данных	4
Хранилище кэша	5
Объектная модель хранилищ	6
Доступ к хранилищам данных и кэшам из UserConnection	7
Использование кэша в EntitySchemaQuery	7
Прокси-классы хранилища данных и кэша	8
Особенности использования хранилищ данных и кэша	12

Хранилища данных и кэш



В Creatio реализована поддержка двух видов хранилищ — данные и кэш.

Назначение разделения хранилищ — логическое разделение помещаемой в хранилище информации и упрощение ее дальнейшего использования в программном коде.

Разделение хранилищ позволяет:

- Изолировать данные между рабочими пространствами и между сессиями пользователей.
- Условно классифицировать данные.
- Управлять временем жизни данных.

Физически все хранилища данных и кэша могут располагаться на абстрактном сервере хранения данных. Исключение составляют данные уровня `Request`, которые хранятся непосредственно в памяти.

Сервером хранилищ Creatio является Redis. В общем случае это может быть произвольное хранилище, доступ к которому осуществляется через унифицированные интерфейсы. Необходимо учитывать, что операции обращения к хранилищу являются ресурсоемкими, поскольку связаны с сериализацией / десериализацией данных и сетевым обменом.

Возможности работы с данными, которые предоставляют хранилища Creatio:

- Доступ к данным по ключу для чтения/записи.
- Удаление данных из хранилища по ключу.

Хранилище данных

Назначение хранилища данных — промежуточное хранение редко изменяемых (т. н. "долгосрочных") данных.

Уровни хранилища данных

Уровень	Описание
<code>Request</code>	Уровень запроса. Данные доступны в течение времени обработки текущего запроса. Соответствует значению перечисления <code>Terrasoft.Core.Store.DataLevel.Request</code> .
<code>Session</code>	Уровень сессии. Данные доступны в сессии текущего пользователя. Соответствует значению перечисления <code>Terrasoft.Core.Store.DataLevel.Session</code> .
<code>Application</code>	Уровень приложения. Данные доступны для всего приложения. Соответствует значению перечисления <code>Terrasoft.Core.Store.DataLevel.Application</code> .

Данные, помещаемые в хранилище, будут содержаться в нем до момента их явного удаления.

Ограничения времени жизни объектов:

- Для данных хранилища уровня `Request` время жизни объектов ограничено **временем выполнения запроса**.
- Для данных хранилища уровня `Session` время жизни объектов ограничено **временем существования сессии**.
- Данные хранилища уровня `Application` сохраняются на протяжении всего периода существования приложения и могут быть удалены из него только путем непосредственной **очистки внешнего хранилища**.

Хранилище кэша

Назначение хранилища кэша — хранение оперативной информации.

Уровни хранилища кэша

Уровень	Описание
<code>Session</code>	Уровень сессии. Данные доступны в сессии текущего пользователя. Соответствует значению перечисления <code>Terrasoft.Core.Store.CacheLevel.Session</code> .
<code>Workspace</code>	Уровень рабочего пространства. Данные доступны для всех пользователей одного и того же рабочего пространства. Соответствует значению перечисления <code>Terrasoft.Core.Store.CacheLevel.Workspace</code> .
<code>Application</code>	Уровень приложения. Данные доступны всем пользователям приложения вне зависимости от рабочего пространства. Соответствует значению перечисления <code>Terrasoft.Core.Store.CacheLevel.Application</code> .

Для данных, хранящихся в кэше, существует **время устаревания** — граничный срок актуальности конкретного элемента кэша. Независимо от времени устаревания, все элементы удаляются из кэша при завершении времени их жизни.

Ограничения времени жизни объектов:

- Для данных уровня `Session` объекты удаляются при завершении сессии.
- Для данных уровня `Workspace` объекты удаляются при явном удалении **рабочего пространства**.
- Для данных уровня `Application` объекты сохраняются на протяжении всего периода существования приложения и могут быть удалены из него только путем непосредственной **очистки внешнего хранилища**.

Данные могут быть удалены из кэша в произвольный момент времени. В связи с этим могут возникать ситуации, когда программный код пытается получить закешированные данные, которые на момент обращения уже удалены из кэша. В этом случае вызывающему коду необходимо просто получить эти данные из постоянного хранилища и поместить их в кэш.

Объектная модель хранилищ

Для работы с хранилищем данных и кэшем в Creatio реализован ряд [классов и интерфейсов](#) пространства имен `Terrasoft.Core.Store`.

Интерфейс IBaseStore

Интерфейс определяет **базовые возможности** всех типов хранилищ и позволяет реализовать:

- Доступ к данным по ключу для чтения/записи (индексатор `this[string key]`).
- Удаление данных из хранилища по заданному ключу (метод `Remove(string key)`).
- Инициализация хранилища заданным перечнем параметров (метод `Initialize(IDictionary parameters)`).
Параметры для инициализации хранилищ Creatio вычитываются из конфигурационного файла. Перечни параметров задаются в секциях `storeDataAdapter` (для хранилища данных) и `storeCacheAdapter` (для кэша). В общем случае параметры могут задаваться произвольным образом.

Интерфейс IDataStore

Интерфейс определяет специфику работы с **хранилищами данных**. Является наследником базового интерфейса хранилищ `IBaseStore`. Дополнительно в интерфейсе определена возможность получения списка всех ключей хранилища (свойство `Keys`).

Важно. Свойство `Keys` при работе с хранилищами данных рекомендуется использовать только в исключительных случаях, когда решение задачи альтернативными способами невозможно.

Интерфейс ICacheStore

Интерфейс определяет специфику **работы с кэшем**. Является наследником базового интерфейса хранилищ `IBaseStore`. Дополнительно в интерфейсе определен метод `GetValues(IEnumerable keys)`, который возвращает словарь объектов кэша с заданными ключами. Метод позволяет оптимизировать работу с хранилищем при одновременном получении набора данных.

Класс Store

Статический класс для доступа к кэшу и хранилищам данных различных уровней. Уровни хранилища данных и кэша определены в перечислениях `Terrasoft.Core.Store.DataLevel` и `Terrasoft.Core.Store.CacheLevel` соответственно.

Статические свойства класса `Store`:

- `Data` — возвращает экземпляр провайдера хранилища данных.
- `Cache` — возвращает экземпляр провайдера кэша.

Важно. Для корректной работы хранилищ в .Net Core, необходимо заменить использование статических свойств на работу через `UserConnection` (см. ниже).

Доступ к хранилищам данных и кэшам из UserConnection

Доступ к хранилищам данных и кэшам приложения из конфигурационного кода предоставляют свойства статического класса `Store` пространства имен `Terrasoft.Core.Store`.

Альтернативным вариантом доступа к хранилищу данных и кэшу в конфигурационной логике, является доступ через экземпляр класса `UserConnection`.

Этот вариант позволяет избежать использования длинных имен свойств и подключения дополнительных сборок. Для обеспечения миграции от фреймворка .NET Framework к .Net Core необходимо заменить использование статических свойств на работу через `UserConnection`.

В классе `UserConnection` реализован ряд вспомогательных свойств, позволяющих получить быстрый доступ к хранилищам данных и кэшу различных уровней:

- `ApplicationCache` возвращает ссылку на кэш уровня приложения.
- `WorkspaceCache` возвращает ссылку на кэш уровня рабочего пространства.
- `SessionCache` возвращает ссылку на кэш уровня сессии.
- `RequestData` возвращает ссылку на хранилище данных уровня запроса.
- `SessionData` возвращает ссылку на хранилище данных уровня сессии.
- `ApplicationData` возвращает ссылку на хранилище данных уровня приложения.

Пример работы с кэшем через класс `UserConnection`

```
// Ключ, с которым в кэш будет помещаться значение.
string cacheKey = "SomeKey";

// Помещение значения в кэш уровня сессии через свойство UserConnection.
UserConnection.SessionCache[cacheKey] = "SomeValue";

// Получение значения из кэша через свойство класса Store.
// В результате переменная valueFromCache будет содержать значение "SomeValue".
string valueFromCache = UserConnection.SessionCache[cacheKey] as String;
```

Использование кэша в EntitySchemaQuery

В [классе](#) `EntitySchemaQuery` реализован механизм работы с хранилищем (кэшем `Creatio` либо произвольным хранилищем, определенным пользователем). Работа с кэшем позволяет оптимизировать эффективность выполнения операций за счет обращения к закешированным результатам запроса без дополнительного обращения к базе данных. При выполнении запроса `EntitySchemaQuery` данные, полученные из базы данных, помещаются в кэш, который определяется свойством `Cache` с ключом, который задается свойством `CacheItemName`. По умолчанию в качестве кэша запроса `EntitySchemaQuery` выступает кэш `Creatio` **уровня сессии** с локальным хранением данных. В общем случае в качестве кэша запроса может выступать произвольное хранилище, которое реализует интерфейс `ICacheStore`.

Пример работы с кэшем приложения при выполнении запроса EntitySchemaQuery

```
// Создание экземпляра запроса EntitySchemaQuery с корневой схемой City.
var esqResult = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");

// Добавление в запрос колонки с наименованием города.
esqResult.AddColumn("Name");

// Определение ключа, под которым в кэше будут храниться результаты выполнения запроса.
// В качестве кэша выступает кэш уровня сессии с локальным кэшированием данных (так как не
// переопределяется свойство Cache объекта).
esqResult.CacheItemName = "EsqResultItem";

// Выполнение запроса к базе данных для получения результирующей коллекции объектов.
// После выполнения этой операции результаты запроса будут помещены в кэш. При дальнейшем обращении
// esqResult для получения коллекции объектов запроса (если сам запрос не был изменен) эти объекты
// братья из сессионного кэша.
esqResult.GetEntityCollection(UserConnection);
```

Прокси-классы хранилища данных и кэша

Доступ к хранилищам и кэшам в Creatio может быть осуществлен как напрямую (через свойства класса `Store`), так и через прокси-классы.

Прокси-классы — это специальные объекты, которые представляют собой промежуточное звено между хранилищами и вызывающим кодом. **Назначение** прокси-классов — выполнение промежуточных действий над данными перед их чтением/записью в хранилище.

Особенность прокси-классов — каждый из них является хранилищем.

Области применения прокси-классов:

- **Первоначальная настройка и конфигурирование приложения.**

В конфигурационном файле `Web.config`, который находится в корневом каталоге приложения, можно настроить использование прокси-классов для хранилищ данных и кэшей. Настройка прокси-классов для соответствующего хранилища данных или кэша осуществляется в секциях `storeDataAdapters` и `storeCacheAdapters` соответственно. В них добавляется секция `proxies`, в которой перечисляются все прокси-классы, применяемые к хранилищу.

При загрузке приложения настройки считываются из конфигурационного файла и применяются к соответствующему виду хранилища. Таким образом можно выстраивать **цепочки прокси-классов**, которые будут последовательно выполняться. Порядок прокси-классов в цепочке выполнения соответствует их порядку в конфигурационном файле. При этом первым в цепочке выполнения выступает прокси-класс, перечисленный последним в секции `proxies`, то есть выполнение осуществляется "снизу вверх".

Особенности настройки цепочки прокси-классов:

- Конечной точкой применения цепочки прокси-классов является конкретный кэш или хранилище

данных, для которого эта цепочка определяется.

- Каждый прокси-класс работает или с хранилищем данных, или с хранилищем кэша. Вид хранилища определяется с помощью свойств `ICacheStoreProxy.CacheStore` или `IDataStoreProxy.DataStore`. Однако, на что именно ссылается это свойство (на другой прокси-класс, на конечное хранилище или кэш), для прокси-класса неизвестно. При этом сам прокси-класс может выступать в качестве хранилища, с которым будут работать другие прокси-классы.

Пример настройки прокси-классов

```
<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>
```

В соответствии с настройками, приведенными в примере, цепочка выполнения прокси-классов, например, хранилища данных, будет следующей:

`RequestDataProxyName3 → RequestDataProxyName2 → RequestDataProxyName1 → RequestDataAdapterClassName` (конечное хранилище данных уровня запроса).

• Разграничение данных различных пользователей приложения.

С помощью прокси-классов решается задача **изоляции данных** между пользователями. Наиболее простым решением этой задачи является трансформация ключей значений перед помещением их в хранилище (например, путем добавления к ключу дополнительного префикса, специфичного для конкретного пользователя). Использование таких прокси-классов обеспечивает уникальность ключей хранилища. Это позволяет избежать потери и искажения данных при одновременной попытке разных пользователей записать в хранилище различные значения с одинаковым ключом. Пример работы с прокси-классами трансформации ключей [приведен ниже](#). В общем случае с помощью прокси-классов можно реализовать сложную логику.

• Выполнение других промежуточных действий с данными перед помещением их в хранилище.

В прокси-классах можно реализовывать логику выполнения любых произвольных действий с данными перед помещением их в хранилище или получения их из него. Вынесение логики обработки

данных в прокси-класс позволяет избежать дублирования кода, что, в свою очередь, облегчает его модификацию и сопровождение.

Базовые интерфейсы прокси-классов

Чтобы класс можно было **использовать в качестве прокси-класса** для работы с хранилищами, он должен реализовывать один или оба интерфейса пространства имен `Terrasoft.Core.Store`:

- `IDataStoreProxy` — интерфейс прокси-классов хранилища данных.
- `ICacheStoreProxy` — интерфейс прокси-классов кэша.

Каждый из этих интерфейсов имеет одно свойство — ссылку на то хранилище (или кэш), с которым работает данный прокси-класс. Для интерфейса `IDataStoreProxy` это свойство `DataStore`, а для интерфейса `ICacheStoreProxy` — свойство `CacheStore`.

Прокси-классы трансформации ключей

Прокси-классы, которые **реализуют логику трансформации ключей значений**, помещаемых в хранилища.

- Класс `KeyTransformerProxy` — абстрактный класс, который является базовым классом для всех прокси-классов, преобразующих ключи кэша. Реализует методы и свойства интерфейса `ICacheStoreProxy`. Чтобы избежать дублирования логики, при создании пользовательских прокси-классов для трансформации ключей рекомендуется наследоваться от этого класса.
- Класс `PrefixKeyTransformerProxy` — прокси-класс, преобразующий ключи кэша путем добавления к ним заданного префикса.

Пример работы с кэшем уровня сессии через прокси-класс `PrefixKeyTransformerProxy`

```
// Ключ, с которым значение будет помещаться в кэш через прокси.
string key = "Key";

// Префикс, который будет добавляться к ключу значения прокси-классом.
string prefix = "customPrefix";

// Создание прокси-класса, который будет использоваться для записи значений в кэш уровня сессии
ICacheStore proxyCache = new PrefixKeyTransformerProxy(prefix, Store.Cache[CacheLevel.Session]);

// Запись значения с ключом key в кэш через прокси-класс.
// Фактически это значение записывается в глобальный кэш уровня сессии с ключом prefix + key.
proxyCache[key] = "CachedValue";

// Получение значения по ключу key через прокси.
var valueFromProxyCache = (string)proxyCache[key];

// Получение значения по ключу prefix + key непосредственно из кэша уровня сессии.
var valueFromGlobalCache = (string>UserConnection.SessionCache[prefix + key];
```

В итоге переменные `valueFromProxyCache` и `valueFromGlobalCache` будут содержать одинаковое значение `CachedValue`.

- Класс `DataStoreKeyTransformerProxy` — прокси-класс, преобразующий ключи хранилища данных путем добавления к ним заданного префикса.

Пример работы с хранилищем данных через прокси-класс `DataStoreKeyTransformerProxy`

```
// Ключ, с которым значение будет помещаться в хранилище через прокси.
string key = "Key";

// Префикс, который будет добавляться к ключу значения прокси-классом.
string prefix = "customPrefix";

// Создание прокси-класса, который будет использоваться для записи значений в хранилище уровня
IDataStore proxyStorage = new DataStoreKeyTransformerProxy(prefix) { DataStore = Store.Data[D

// Запись значения с ключом key в хранилище через прокси-класс.
// Фактически это значение записывается в глобальное хранилище уровня сессии с ключом prefix
proxyStorage[key] = "StoredValue";

// Получение значения по ключу key через прокси.
var valueFromProxyStorage = (string)proxyStorage[key];

// Получение значения по ключу prefix + key непосредственно из хранилища уровня сессии.
var valueFromGlobalStorage = (string)UserConnection.SessionData[prefix + key];
```

В итоге переменные `valueFromProxyStorage` и `valueFromGlobalStorage` будут содержать одинаковое значение `StoredValue`.

Локальное кэширование данных

На базе прокси-классов в Creatio реализован механизм **локального кэширования данных**.

Основная цель кэширования данных — снижение нагрузки на сервер хранения данных и времени выполнения запросов при работе с редко изменяемыми данными.

Логику механизма локального кэширования реализует внутренний прокси-класс `LocalCachingProxy`. Этот прокси-класс выполняет кэширование данных на текущем узле веб-фермы. Класс выполняет мониторинг времени жизни кэшированных объектов и получает данные из глобального кэша только в том случае, если кэшированные данные не актуальны.

Для применения механизма локального кэширования на практике используются **методы расширения** для интерфейса `ICacheStore` из статического класса `CacheStoreUtilities`:

- `WithLocalCaching()` — переопределенный метод, который возвращает экземпляр класса `LocalCachingProxy`, выполняющего локальное кэширование.

- `WithLocalCachingOnly(string)` — метод, выполняющий локальное кэширование данных заданной группы элементов с мониторингом их актуальности.
- `ExpireGroup(string)` — метод устанавливает признак устаревания для заданной группы элементов. При вызове этого метода все элементы заданной группы становятся неактуальными и не возвращаются при запросе данных.

Пример работы с кэшем рабочего пространства с использованием локального кэширова..

```
// Создание первого прокси-класса, который выполняет локальное кэширование данных. Все элементы,
// записываемые в кэш через этот прокси, принадлежат к группе контроля актуальности Group1.
ICacheStore cacheStore1 = Store.Cache[CacheLevel.Workspace].WithLocalCaching("Group1");

// Добавление элемента в кэш через прокси-класс.
cacheStore1["Key1"] = "Value1";

// Создание второго прокси-класса, который выполняет локальное кэширование данных. Все элементы,
// записываемые в кэш через этот прокси, также принадлежат к группе контроля актуальности Group1.
ICacheStore cacheStore2 = Store.Cache[CacheLevel.Workspace].WithLocalCaching("Group1");
cacheStore2["Key2"] = "Value2";

// Для всех элементов группы Group1 устанавливается признак устаревания. Устаревшими считаются э
// с ключами Key1 и Key2, так как они принадлежат к одной группе контроля актуальности Group1, н
// то, что добавлены в кэш через разные прокси.
cacheStore2.ExpireGroup("Group1");

// Попытка получения значений из кэша по ключам Key1 и Key2 после того, как эти элементы были п
// устаревшие. В результате переменные cachedValue1 и cachedValue2 будут содержать значение null
var cachedValue1 = cacheStore1["Key1"];
var cachedValue2 = cacheStore1["Key2"];
```

Особенности использования хранилищ данных и кэша

Для повышения эффективности работы с хранилищами данных и кэшами в конфигурационной логике и в логике реализации бизнес-процессов необходимо учитывать особенности их использования.

- Все объекты, помещаемые в хранилища, должны быть **сериализуемыми**. Это обусловлено спецификой работы с хранилищами ядра приложения. При сохранении данных в хранилища (за исключением хранилища данных уровня `Request`) объект предварительно **сериализуется**, а при получении — **десериализуется**.
- Обращение к хранилищу является относительно ресурсоемкой операцией. В связи с этим в коде необходимо избегать излишних обращений к хранилищу. В примерах 1 и 2 приводятся корректные и некорректные варианты работы с кэшем.

Пример 1

Неоптимальный код

```
// Выполняется сетевое обращение и десериализация данных.
if (UserConnection.SessionData["SomeKey"] != null)
{
    // Повторно выполняется сетевое обращение и десериализация данных.
    return (string)UserConnection.SessionData["SomeKey"];
}
```

Оптимальный код (вариант 1)

```
// Получение объекта из хранилища в промежуточную переменную.
object value = UserConnection.SessionData["SomeKey"];

// Проверка значения промежуточной переменной.
if (value != null)
{
    // Возврат значения.
    return (string)value;
}
```

Оптимальный код (вариант 2)

```
// Использование расширяющего метода GetValue().
return UserConnection.SessionData.GetValue<string>("SomeKey");
```

Пример 2**Неоптимальный код**

```
// Выполняется сетевое обращение и десериализация данных.
if (UserConnection.SessionData["SomeKey"] != null)
{
    // Повторно выполняется сетевое обращение и десериализация данных.
    UserConnection.SessionData.Remove("SomeKey");
}
```

Оптимальный код

```
// Удаление выполняется сразу, без предварительной проверки.  
UserConnection.SessionData.Remove("SomeKey");
```

- Любые изменения состояния объекта, полученного из хранилища данных или кэша, происходят локально в памяти и не фиксируются в хранилище автоматически. Поэтому, чтобы эти изменения отобразились в хранилище, измененный объект необходимо явно в него записать.

Пример записи данных в хранилище

```
// Получение словаря значений из хранилища данных сессии по ключу "SomeDictionary".  
Dictionary<string, string> dic = (Dictionary<string, string>)UserConnection.SessionData["SomeDictionary"];  
  
// Изменение значения элемента словаря. Изменения в хранилище не зафиксировались.  
dic["Key"] = "ChangedValue";  
  
// Добавление нового элемента в словарь. Изменения в хранилище не зафиксировались.  
dic.Add("NewKey", "NewValue");  
  
// В хранилище данных по ключу "SomeDictionary" записывается словарь. Теперь все внесенные из  
// зафиксированы в хранилище.  
UserConnection.SessionData["SomeDictionary"] = dic;
```