About Terracotta DB

Version 10.2

April 2018

Document ID: TDB-AB-102-20180417

Table of Contents

1 Introduction to Terracotta DB

Terracotta DB is a comprehensive, distributed in-memory data management solution which caters to caching and operational storage use cases, and enables transactional and analytical processing. Terracotta DB has one of the most powerful query and computation capabilities in its class, leveraging native JDK features such as Java Streams, collections, and functions.

Terracotta DB supports the following sub-systems:

1. A storage sub-system called TCStore, that caters to operational store and compute functionality. The API exposing this sub-system's functionality is the TCStore API.
2. A caching sub-system called Ehcache, that caters to caching functionality. The API exposing this sub-system's functionality is the Ehcache API.

Both sub-systems are backed by the Terracotta Server, which provides a common platform for distributed in-memory data storage with scale-out, scale-up and high availability features.

The Terracotta DB APIs

Terracotta DB offers two distinct APIs for caching and storage:

Ehcache API

The Ehcache API is an improved version of Java's de facto caching API, Ehcache. It has a powerful, streamlined, modernized caching API taking advantage of newer Java features as well as the capability to be used via the JSR-107 "JCache" API. Some of the key high level feature of this API include:

- Leverages Java generics and simplifies cache interactions
- Full compatibility with javax.cache API (JSR-107)
- Storage

    In-memory storage with optional persistence to disk and ultra-fast recovery

    Java-based Key/Value store optimized for caching workloads

    The industry's first and best Offheap storage capabilities

- Distributed Store

    Supports various scale-out and HA deployment configurations

    Flexible, fine-granular configuration of availability, consistency, and durability

TCStore API

TCStore API is an interface for distributed in-memory data storage and computation, which has powerful ties to JDK features related to streams, collections and functions.

Under the hood it is powered by a completely new and powerful storage engine which is an "Aggregate oriented, Key-Value, wide-column store" built upon a very high performance and highly scalable architecture. Some of the key high level feature of this API include:

- Flexible Data Model

  Aggregate oriented, Key-Value store

  Loose schema: modeling of data with structured and typed aggregate values within records

- Storage

  In-memory storage with optional persistence to disk and ultra-fast recovery

  Java-based Key/Value store optimized for data storage workloads

  Secondary in-memory indexes to speed-up search and compute

- Distributed Store

  Supports various scale-out and HA deployment configurations

  Flexible, fine-granular configuration of availability, consistency, and durability

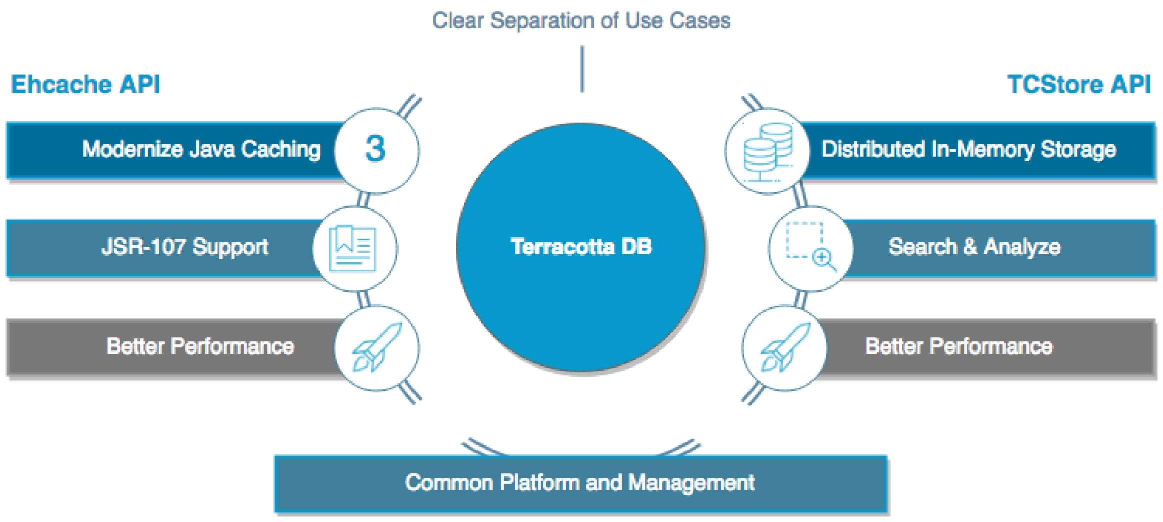
- Data Analysis

  Search and Analyze capabilities that work naturally with Java 8 technologies

  Java stream API to filter, aggregate, map data

  DSL with a library of pre-implemented lambda functions enabling server-side execution of queries

Along with the two separate APIs, Terracotta DB is built upon the next generation of Terracotta Server/Server Array. Both the Ehcache API and the TCStore API leverage the power of this new distributed computing platform with reduced complexity and enhanced performance and scalability. Similarly the shared platform provides common monitoring and management capabilities.



Ehcache versus TCStore: Why Two Different APIs?

The two different APIs help simplify development and separation of data management concerns based upon use case. In a nutshell:

Ehcache is the right API to: ...

> ... access relevant data, in large amounts, at maximum speed by simple key/value look-up
>
> ... load hot, fresh data for as short as you may need it and replace stale data by more relevant data from the System of Record
>
> ... offload work from the System of Record

because Ehcache ...

> ... continually works to keep hotter, fresher entries available at the fastest, in-memory speeds
>
> ... is optimized to for application caching needs

TCStore is the right API to: ...

> ... store data that you always expect to be there (free from eviction concerns)
>
> ... store data that you expect to perform reliable search and queries on, at in-memory speeds
>
> ... be your database of record

because TCStore ...

> ... allows data to be structured and strongly typed
>
> ... entries can be looked up by key and queried on by field

It's important to understand that there is a strict separation between the Ehcache API and the TCStore API, even when used with the same Terracotta Server:

Any information placed using the TCStore API cannot be retrieved using the Ehcache API and vice versa.

Terracotta Server provides the core distributed storage platform and is common for data placed using both the Ehcache API and the TCStore API, yet the cache data is managed separately from stored data.

2 What is Ehcache?

Features

Ehcache is the most widely-used Java-based cache. It is:

- robust,
- proven,
- full-featured,
- and integrates with other popular libraries and frameworks.

Ehcache scales from in-process caching, all the way to mixed in-process/out-of-process deployments with terabyte-sized caches.

Fast and Lightweight

Fast

Ehcache's concurrency features are designed for large, high concurrency systems.

Simple

Many users of Ehcache hardly know they are using it. Sensible defaults require no initial configuration.

Small footprint

Ehcache strives to maintain a small footprint - keeping your apps as light as they can be.

Minimal dependencies

The only dependency for core use is SLF4J.

Scalable

Provides for scalability into terabytes

The largest Ehcache installations utilize multiple terabytes of data storage.

With off-heap storage, Ehcache has been tested to store 6TB of data in a single process (JVM).

Scalable to hundreds of caches

The largest Ehcache installations use hundreds of caches.

Tuned for high concurrent load on large/wide multi-CPU servers

Ehcache is specifically built and tested to run well under highly concurrent access on systems with dozens of cores. This results in an optimal balance between thread safety and performance.

Flexible

Provides multiple strategies for:

- Expiration policies
- Storage tiers (on-heap, off-heap, disk, clustered)
- Configuration of caches

Standards Based

Support of JSR-107 JCache - Java Temporary Caching API

You can use Ehcache as a JCache provider. This allows you to use JCache API calls to develop a complete application, without the need to use any Ehcache API calls.

Distributed Caching

Ehcache supports simple yet high performance distributed caching.

Enterprise Java and Applied Caching

High quality implementations for common caching scenarios and patterns.

Cacheable Commands

This is the trusty old command pattern with a twist: asynchronous behavior, fault tolerance and caching. Creates a command, caches it and then attempts to execute it.

Works with Hibernate

Ehcache is popularly used as Hibernate's second-level cache.

Transactional support through JTA

Ehcache supports JTA and is a fully XA compliant resource participating in the transaction, two-phase commit and recovery.

See the complete Transaction Module Java Documentation at www.ehcache.org.

API Documentation

The Javadoc documentation of the API can be found here:

www.ehcache.org/documentation/

Open Source Kit

There is an open-source version of Ehcache, paired with Terracotta Server open source functionality. This can be found at http://www.terracotta.org/open-source/.

Basic Terms

Cache

A cache is a collection of temporary data that either duplicates data located elsewhere or is the result of a computation. Data that is already in the cache can be repeatedly accessed with minimal costs in terms of time and resources.

Cache Entry

A cache entry consists of a key and its mapped data value within the cache.

Cache hit

When a data element is requested from cache and the element exists for the given key, it is referred to as a *cache hit* (or simply, "a hit").

Cache miss

When a data element is requested from cache and the element does not exist for the given key, it is referred to as a *cache miss* (or simply, "a miss").

Eviction

When entries are removed from the cache in order to make room for newer entries (typically when the cache has run out of data storage capacity), it is referred to as eviction.

Expiration

When entries are removed from the cache after some defined amount of time has passed, it is referred to as expiration.

Hot Data

Data that has recently been used by an application is very likely to be accessed again soon. Such data is considered *hot*. A cache may attempt to keep the hottest data most quickly available, while attempting to choose the least hot data for eviction.

System-of-Record

The system-of-record is the authoritative source of truth for the data. The cache acts as a local copy of data retrieved from or stored to the system-of-record (SOR). The SOR is often a traditional database, although it might be a specialized file system or some other reliable long-term storage. It can also be a conceptual component such as an expensive computation.

Data Freshness and Expiration

Data Freshness

Data *freshness* describes how up-to-date a copy of data (e.g. in a cache) is compared to the source version of the data (e.g. in the system-of-record (SoR). A *stale* copy is considered to be out of sync (or likely to be out of sync) with the SoR.

Databases (and other SORs) weren't built with caching outside of the database in mind, and therefore don't normally come with any default mechanism for notifying external processes when data has been updated or modified. Thus external components that have loaded data from the SoR have no direct way of ensuring that data is not stale.

Cache Entry Expiration

Ehcache can assist you with reducing the likelihood that stale data is used by your application by *expiring* cache entries after some amount of configured time. Once expired, the entry is automatically removed from the cache.

For instance, the cache could be configured to expire entries five seconds after they are put into the cache - which is a time-to-live *TTL* setting. Or to expire entries 17 seconds after the last time the entry was retrieved from the cache - which is a time-to-idle *TTI* setting.

Note: TTI is not supported for caches with clustered storage tiers.

The expiration configuration that would be most appropriate for your cache (if any) would be a mixture of a business and technical decision based upon the requirements and assumptions of your application.

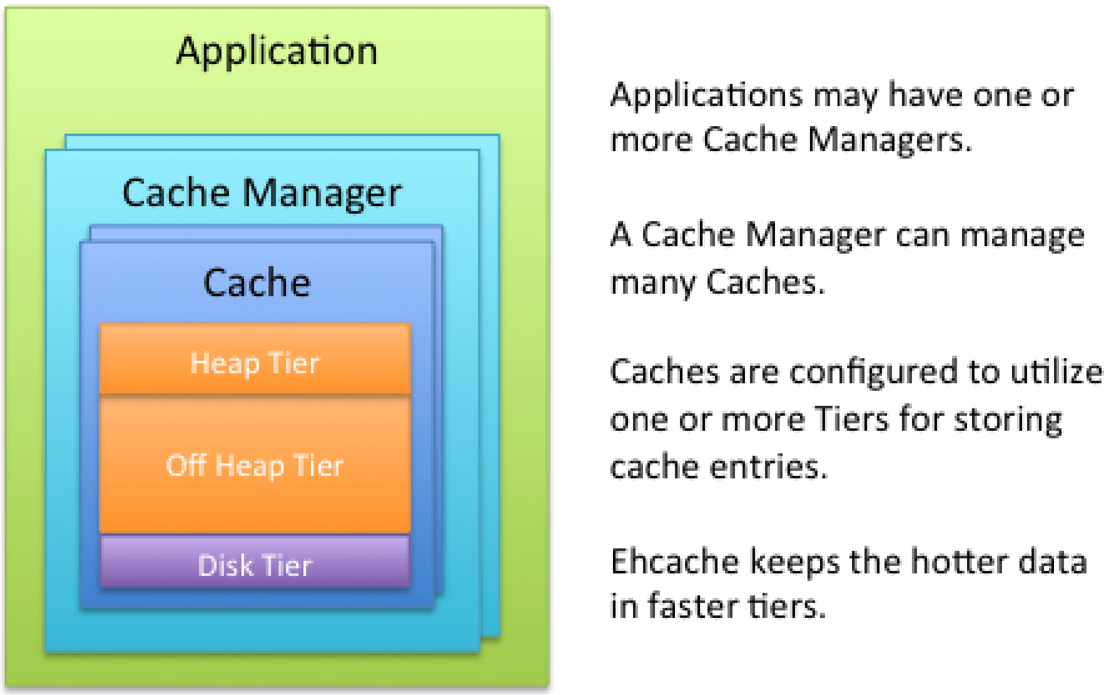
Storage Tiers

You can configure Ehcache to use various data storage areas. When a cache is configured to use more than one storage area, those areas are arranged and managed as *tiers*. They are organized in a hierarchy, with the lowest tier being called the *authority* tier and the others being part of the *caching* tier. The caching tier can itself be composed of more than one storage area. The *hottest* data is kept in the caching tier, which is typically less abundant but faster than the authority tier. All the data is kept in the authority tier, which is slower but more abundant.

Data stores supported by Ehcache include:

- **On-Heap Store** – Utilizes Java's on-heap RAM memory to store cache entries. This tier utilizes the same heap memory as your Java application, all of which must be scanned by the JVM garbage collector. The more heap space your JVM utilizes the more your application performance will be impacted by garbage collection pauses. This store is extremely fast, but is typically your most limited storage resource.

- **Off-Heap Store** – Limited in size only by available RAM. Not subject to Java garbage collection (GC). Is quite fast, yet slower than the On-Heap Store because data must be moved to and from the JVM heap as it is stored and re-accessed.
- **Disk Store** – Utilizes a disk (file system) to store cache entries. This type of storage resource is typically very abundant but much slower than the RAM-based stores. As for all applications using disk storage, it is recommended to use a fast and dedicated disk to optimize the throughput.
- **Clustered Storage** – This data store is a cache on a remote server. The remote server may optionally have a failover server providing improved high availability. Since clustered storage comes with performance penalties due to such factors as network latency as well as for establishing client/server consistency, this tier, by nature, is slower than local off-heap storage.



Applications may have one or more Cache Managers.

A Cache Manager can manage many Caches.

Caches are configured to utilize one or more Tiers for storing cache entries.

Ehcache keeps the hotter data in faster tiers.

Topology Types

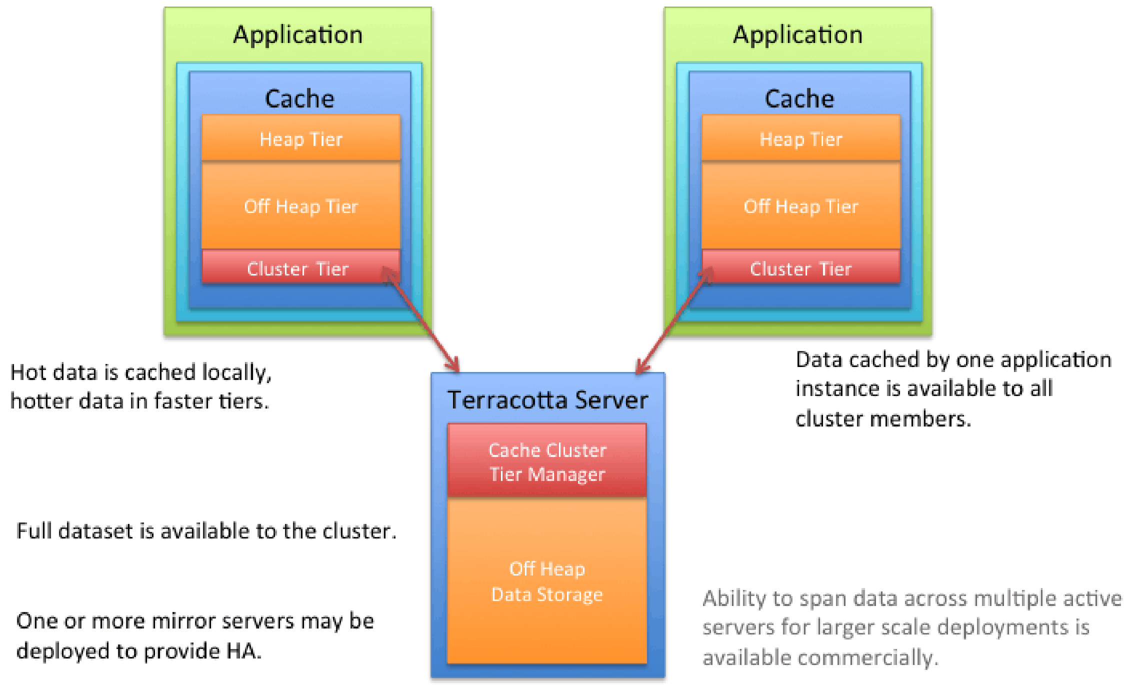
The following describes the basic types of caching topologies:

Standalone

The data set is held in the application node. Any other application nodes are independent with no communication between them. If a standalone topology is used where there are multiple application nodes running the same application, then their caches are completely independent.

Distributed / Clustered

The data is held in a remote server (or array of servers) with a subset of hot data held in each application node. This topology offers a selection of consistency options. A distributed topology is the recommended approach in a clustered or scaled-out application environment. It provides the best combination of performance, availability, and scalability.


Application
Cache
Heap Tier
Off Heap Tier
Cluster Tier
Application
Cache
Heap Tier
Off Heap Tier
Cluster Tier
Hot data is cached locally, hotter data in faster tiers.
Terracotta Server
Cache Cluster Tier Manager
Off Heap Data Storage
Data cached by one application instance is available to all cluster members.
Full dataset is available to the cluster.
One or more mirror servers may be deployed to provide HA.
Ability to span data across multiple active servers for larger scale deployments is available commercially.

It is common for many production applications to be deployed in clusters of multiple instances for availability and scalability. However, without a distributed cache, application clusters exhibit a number of undesirable behaviors, such as:

- **Cache Drift** - If each application instance maintains its own cache, updates made to one cache will not appear in the other instances. A distributed cache ensures that all of the cache instances are kept in sync with each other.
- **Database Bottlenecks** - In a single-instance application, a cache effectively shields a database from the overhead of redundant queries. However, in a distributed application environment, each instance must load and keep its own cache fresh. The overhead of loading and refreshing multiple caches leads to database bottlenecks as more application instances are added. A distributed cache eliminates the per-instance overhead of loading and refreshing multiple caches from a database.

3 What is TCStore?

Core Concepts of TCStore

TCStore organizes its data into collections, so-called datasets.

Each `Dataset` is comprised of zero or more records.

Each `Record` has a key, unique within the dataset, and zero or more cells.

Each `Cell` has a name, unique within the record; a declared type; and a non-null value.

While records within a dataset must have a uniform key type, records are not required to have uniform content — each record may be comprised of cells having different names and/or different types.

Each record and each cell is self-describing and is understood by the storage engine.

TCStore Data Storage Model - typed data

```
+--------+  +------------------------------------------------------------+
|        |  |  +--------------------+ +------------+ +-----------------+  |
|  Key   |  |  |  "name" : "Alex"   | | "age" : 19 | | "pic" : byte[] |  |
|        |  |  +--------------------+ +------------+ +-----------------+  |
+--------+  +------------------------------------------------------------+
```

Since records and cells are self-describing, records and cells can be manipulated efficiently:

- retrieved: no need to retrieve the entire `Record`, when all you want is a single `Cell`;
- mutated: cell values can be directly changed such as incrementing an `Integer` cell;
- searched: nearly, every cell type is searchable. Indexes may be added to improve search patterns;
- computed: computation, mutative or not, to be executed against the records of a dataset;

Using popular/industry definitions, TCStore is an "Aggregate oriented, Key-Value, wide-column NoSQL store". As noted above, the individual records stored within TCStore contain cells with type information enabling the store to make use of the data it holds. However, like other NoSQL stores, TCStore is schema-less in its core design, allowing individual records to contain identical sets of cells, a subset of common cells, or a completely different sets of cells.

As such, and like other NoSQL stores, TCStore is not intended for usage patterns that are traditional to tabular data or RDBMSs. Data contained within TCStore are not and cannot be directly relational, and care should be taken to use modeling techniques (such as de-normalization of data) other than those commonly used with RDBMSs.

Type System

Fundamental to TCStore is the type system used in the data model.

The supported data types are:

Type	Description/Mapping to	Associated with ...
`BOOL`	A boolean value (either `true` or `false`), mapping to `java.lang.Boolean`	cells of type `Cell<Boolean>` cell definitions of type `BoolCellDefinition` and `CellDefinition<Boolean>`
`BYTES`	An array of bytes, signed 8-bit each, mapping to `byte[]`	cells of type `Cell<byte[]>` cell definitions of type `BytesCellDefinition` and `CellDefinition<byte[]>`
`CHAR`	A single UTF-16 character, 16-bit unsigned, mapping to `java.lang.Character`	cells of type `Cell<Character>` cell definitions of type `CharCellDefinition` and `CellDefinition<Character>`
`DOUBLE`	A 64-bit floating point value, mapping to `java.lang.Double`	cells of type `Cell<Double>` cell definitions of type `DoubleCellDefinition and CellDefinition<Double>`
`INT`	A signed 32-bit integer value, mapping to `java.lang.Integer`	cells of type `Cell<Integer>` cell definitions of type IntCellDefinition and `CellDefinition<Integer>`
`LONG`	A signed 64-bit integer value, mapping to `java.lang.Long`	cells of type `Cell<Long>` cell definitions of type `LongCellDefinition` and `CellDefinition<Long>`
`STRING`	A variable length sequence of `CHAR`, mapping to java.lang.String	cells of type `Cell<String>` cell definitions of type `StringCellDefinition` and `CellDefinition<String>`

The key of a `Record` may be an instance of any of the above types except `BYTES`. The value of a `Cell` may be an instance of any one of the above types.

Datasets

A `Dataset` is a collection of `Record` instances. Each `Record` instance is uniquely identified by a key within the `Dataset`. The key type is declared when the `Dataset` is created. Aside from the `Record` key type, a `Dataset` has no predefined schema.

Records

A `Record` is a key plus an unordered set of "name to (typed) value" pairs representing a natural aggregate of your domain model. Each `Record` within a `Dataset` can hold completely different sets of cells, as there is no schema to conform to. `Record` instances held within a given `Dataset` are immutable. Changing one or multiple values on a `Record` creates a new instance of that Record which replaces the old instance.

`Record` represents the only atomically alterable type in TCStore. You can mutate as many `Cell`s of a given `Record` as you wish as an atomic action, but not across multiple `Record` instances. This includes adding and removing cells at the same time as changing the values of existing cells.

Cell Definitions and Values

A `Record` contains zero or more `Cell` instances, each derived from a `CellDefinition`. A `CellDefinition` is a "type/name" pair (e.g. `String firstName`). From a `CellDefinition` you can create a `Cell` (e.g. `firstName = "Alex"`, where `"Alex"` is of type `String`) to store in a `Record`. The name of the `Cell` is the name from the `CellDefinition` used to create the cell; the value of the `Cell` is of the type specified in the `CellDefinition` used to create the cell.

`Cell` instances cannot contain `null` values. However, the API will let you test a `Record` for the absence of a cell.

Note: The `Cell` instances within a `Record` are unordered.

Creating a `CellDefinition` Instance

There are multiple ways of providing a `CellDefinition` describing cells used in a dataset. The following example shows various ways of specifying a `CellDefinition` for a cell holding a `String` value.

```
StringCellDefinition NAME =
    CellDefinition.defineString("nameCell");                // <1>
CellDefinition<String> ALT_NAME =
    CellDefinition.defineString("altNameCell");             // <2>
CellDefinition<String> BASIC_NAME =
    CellDefinition.define("basicNameCell", Type.STRING);    // <3>
```

1. A `CellDefinition` supporting a value type of `String` can be using the `CellDefinition.defineString()` method.
2. A `StringCellDefinition` is also a `CellDefinition<String>`.
3. In addition to the `CellDefinition.defineString()` method, the `CellDefinition.define()` may be used to create a `CellDefinition` for a `String`.

Cell definitions for other supported types (see "Type system" on page 18) are handled similarly.

The following are the `CellDefinition` instances used in several examples shown in this document:

Counter Demo Cells

```
LongCellDefinition counterCell = CellDefinition.defineLong("longCounter");
BoolCellDefinition stoppedCell = CellDefinition.defineBool("stopped");
StringCellDefinition stoppedByCell = CellDefinition.defineString("stoppedBy");
```

In this group, three cells are defined

counterCell holds the value of the counter

stoppedCell indicates if the counter has been stopped or not

stoppedByCell holds the name of the stopping user if the counter is stopped

Creating a `Cell` Instance

Creating a `Cell` instance is typically done from a `CellDefinition` instance using the `CellDefinition.newCell()` method.

For applications using well-structured data, statically declaring the cell definitions and using `newCell` can aid code clarity.

For applications with more fluid requirements where data is not well structured, it is possible to create the needed `CellDefinition` instances implicitly when creating `Cell` instances.

```
Cell<String> nameCell = NAME.newCell("Alex");                     // <1>
Cell<String> mollyCell = nameCell.definition().newCell("Molly");  // <2>
Cell<String> aliasCell = Cell.cell("aliasCell", "Tattoo");        // <3>
CellSet cells = CellSet.of(                                       // <4>
    cell("inlineNameCell", "Alex"),
    cell("inlineWeightCell", 118.5D),
    cell("inlineCountCell", 2L),
    cell("inlineCategoryCell", '®'),
    cell("inlineAgeCell", 42),
    cell("inlineEmployedCell", true),
    cell("inlineVaultCell",
        new byte[] {(byte) 0xCA,(byte) 0xFE,(byte) 0xBA,(byte) 0xBE}));
```

1. Creates a new instance of a nameCell having the value "Alex". When using pre-defined cell definitions, this is the most common way to create a new `Cell` instance, the `CellDefinition.newCell` method. In this example, `NAME` is a statically-declared `CellDefinition`.

2. A new `Cell` instance can also be created from an existing `Cell` through the original cell's `CellDefinition`. A `Cell` instance reveals its `CellDefinition` through the `Cell.definition()` method.

3. For applications with more fluid data needs, a `CellDefinition` may be implicitly defined in conjunction with creating a cell using the `Cell.cell()` method. When the `Cell.cell()` method is used, the value type for the `CellDefinition` is determined from the value supplied. The supported value types are limited to the supported types identified in the type system. In this example, a the `CellDefinition` created is equivalent to `CellDefinition.define("aliasCell", Type.String)`. The `Cell` instance

is equivalent to that created by `CellDefinition.define("aliasCell", Type.String).newCell("Tattoo")`.

4. When paired with a static import for `com.terracottatech.store.Cell.cell`, creation of `Cell` instances inline becomes somewhat "cleaner". In this example, several cells, with their implicit cell definitions, inline with the creation of the `CellSet` to contain them.

   Note: Caution is advised when using `Cell.cell`. A `Record` (or CellSet) can contain no more than one `Cell` of a given name regardless of type.

Note: Note that neither `CellSet` nor `Record` guarantees an order of the `Cell` instances contained within. If cell ordering is required, the ordering must be applied by application code.

API Usages

For convenience, the software kit contains several demo programs that elaborate on how to use the TCStore API.

The following sections discuss aspects of the TCStore API using code of these demo programs.

Please mind:

The TCStore API in general and in particular the following code snippets are based on Java 8 concepts and constructs. Strong familiarity with Java 8 is required to fully understand these examples.

To make good use of the TCStore API, you will also need knowledge of Java 8 Lambda Expressions, Java 8 Streams and Collectors and topics such as patterns, parallelization and performance.

Lifecycle

In order to use a `Dataset` you need to create one or use one that you had created previously. One client node in the cluster is responsible for creating the `Dataset`.

Clustered Dataset

A Terracotta Server can be used to host a `Dataset` that can be shared among multiple clients. To do this, you must create a clustered `DatasetManager` instance identifying the server to host the dataset.

```
DatasetManager datasetManager =
    DatasetManager.clustered(connectionURI).build();             // <1>
DatasetConfiguration configuration =
    datasetManager.datasetConfiguration()
        .offheap(offHeapResource).build();                       // <2>
    Dataset<String> counterSet = datasetManager.createDataset(
         "counters", Type.STRING, configuration))                // <3>
```

1. First, a `DatasetManager` instance is built. For a clustered dataset, the URI identifying the Terracotta Server must be specified.
2. The dataset configuration for a clustered `Dataset` must identify the name of an off-heap storage resource to be used on the server. The name specified here must match a name provided in a `service/offheap-resources/resource` element in the server's XML configuration.
3. The `Dataset` is created by using the `DatasetManager.createDataset(String, Type<K>, DatasetConfiguration)` method.

Note: Creation of `DatasetManager` and `Dataset` instances is often done using a Java 7 *try-with-resources* statement. Both `DatasetManager` and `Dataset` extend `java.lang.AutoCloseable`. Each should be closed when no longer needed to permit resources to be reclaimed.

For purposes of clarity, this detail is omitted from the samples in this document.

Data Access and Data Model

Once you've obtained a reference to a `Dataset`, you can read, add, remove and mutate data that it holds. That data is held in the form of `Record` instances. The following section demonstrates the basic create/replace/update/delete (CRUD) operations on a dataset.

Basic CRUD Operations

```
DatasetWriterReader<String> counterAccess =
    counterSet.writerReader();                                    // <1>
String someCounterKey = "someCounter";
boolean added = counterAccess.add(                                // <2>
    someCounterKey, counterCell.newCell(0L),
        stoppedCell.newCell(false));
if (added) {
  System.out.println("No record with the key: " + someCounterKey
      + " existed. The new one was added");
}
Optional<Record<String>> someCounterRec =
    counterAccess.get(someCounterKey);                            // <3>
Long longCounterVal = someCounterRec.flatMap(r ->
    r.get(counterCell)).orElse(0L);                               // <4>
System.out.println("someCounter is now: " + longCounterVal);
counterAccess.update(someCounterKey, write(counterCell, 10L));    // <5>
someCounterRec = counterAccess.get(someCounterKey);
System.out.println("someCounter is now: "
    + someCounterRec.flatMap(r -> r.get(counterCell)).orElse(0L));
Optional<Record<String>> deletedRecord =
    counterAccess.on(someCounterKey).delete();                    // <6>
System.out.println("Deleted record with key: "
    + deletedRecord.map(Record::getKey).orElse("<none>"));
```

1. Define a `Dataset` access object, in this case a `DatasetWriterReader` instance, over the dataset.

2. `DatasetWriterReader.add` lets you add a new `Record` for a given key in the `Dataset` but only if no Record already exists for the key provided. Should a Record already exist, `false` is returned and no changes are made to the Dataset.

3. `DatasetWriterReader.get` lets you retrieve a `Record`, wrapped in an `Optional`, from the `Dataset` using the key that was used to add the record in the dataset. If a record with the specified key does not exist in the dataset then `Optional.empty()` is returned.

4. Since an `Optional` is returned from `get`, `Optional.flatMap` may be used to extract information from the `Record`. In this case, the value of the *counterCell* is extracted. If the record does not contain the *counterCell*, zero is returned by the `Optional.orElse` method

5. Mutates the `Record`. In this example, the `counterCell` value is updated. `UpdateOperation.write` method is a helper method provided to update individual cells in a record. One thing to note here is that this form of `update` method does not return anything.

6. Deletes the `Record`. If a record with the given key is available in the dataset, it is removed and returned, wrapped in an `Optional`. If there is no record with the given key, then an empty `Optional` is returned.

Complex Mutative Operations

The example that follows shows a complex mutative operation and a conditional delete operation. This example operates on a dataset containing ten records having keys `counter0` through `counter9` each of which has a *counterCell* and a *stoppedCell*. Another cell, *stoppedByCell*, is defined but is not present in any of the records.

Single Record Update/Delete

```
String advancedCounterKey = "counter9";
Optional<String> token = counterAccess
    .on(advancedCounterKey)                                  // <1>
    .update(UpdateOperation.custom(                          // <2>
        record -> {                                          // <3>
          if (!record.get(stoppedCell).orElse(false)         // <4>
              && record.get(counterCell).orElse(0L) > 5) {
            CellSet newCells = new CellSet(record);          // <5>
            newCells.set(stoppedCell.newCell(true));
            newCells.set(stoppedByCell.newCell("Albin"));
            newCells.remove(counterCell);
            return newCells;
          } else {
            return record;                                   // <6>
          }
        }))
    .map(Tuple::second)                                      // <7>
    .map(r -> r.get(stoppedCell).orElse(false)
        ? r.get(stoppedByCell).orElse("<unknown>") : "<not_stopped>");
deletedRecord = counterAccess.on("counter0")
                   .iff(stoppedCell.isFalse()).delete();     // <8>
```

1. Selecting the `counter9` record ...

2. Update the record ...

3. Using a *custom*, i.e. non-DSL lambda. DSL is discussed in the following section *"Query and Compute Capabilities"*.
4. Gates the update so the record is only mutated if *stoppedCell* is `false` and *counterCell* is greater than five.
5. The *custom* update creates a `CellSet` copied from the existing record and then modifies it by setting the *stoppedCell* and *stoppedByCell*, and removing the *counterCell*. Note that the *stoppedByCell* gets added to the record by this update.
6. If the record does not meet the selection criterion (*stoppedCell == false && counterCell > 5*), the original record is returned.
7. Maps the output of the `update` operation (a `Tuple` containing the old and new records) to select only the new record (the second of the tuple) then maps the new record to obtain the "stopped by" value if the record is actually flagged as stopped.
8. Delete the record with the key `counter0` if, and only if, *stoppedCell* is `false`.

Query and Compute Capabilities

The following example shows a simple computation over the `Record` instances in a `Dataset`: an average of the *counterCell* values. This example uses a Java stream on the `Dataset` and a pipeline using Java lambda expressions.

Simple `Stream<Record<K>>` *Using Lambdas*

```
OptionalDouble avg;
try (final Stream<Record<String>> recordStream =
    counterAccess.records()) {                                 // <1>
  avg = recordStream
      .filter(record -> !record.get(stoppedCell).orElse(false))  // <2>
      .mapToLong(record -> record.get(counterCell).orElse(0L))   // <3>
      .average();                                              // <4>
}
```

1. Retrieves a `Stream<Record<K>>` to operate on. Note the use of the *try-with-resources* statement. Streams obtained from a `Dataset` should be closed when no longer needed.
2. Filters the `Stream` for all `Record<K>` having the stoppedCell value as *false*, which are the counters that are not stopped.
3. Maps the `Stream<Record<K>>` to a `LongStream` of the values of the `counterCell`.
4. Calculates the average of all of these values using the Java 8 `LongStream.average` method.

A key aspect of using a Java stream is that no elements (in this case, `Record` instances) get processed until a terminal operation is invoked on the `Stream`, in this example, the `.average()` operation. As part of the terminal operation processing, TCStore tries to resolve the best possible way to execute the query.

While the example above is completely functional using TCStore, it isn't optimal as the example Java lambda expressions are not introspectable.

In a distributed environment, TCStore must move data (at least the `Cell<Boolean>` for all `stoppedCell` instances to filter on and then all matching `Cell<Long>` for `counterCell` instances) over the network to the client node to evaluate each and every lambda in the pipeline.

> **Note:** As with other Java `Stream` instances, a `Stream` instance obtained from a `Dataset` can be consumed exactly one time.

In the next example below, we re-express the query from the previous example *"Simple Stream<Record<K>> Using Lambdas"*, this time using TCStore's fluent Domain Specific Language (DSL) for querying and computing. Using the DSL, TCStore is capable of understanding the actual query and/or computation being requested and optimizing it for an execution in a distributed environment. (For information on how to make use of existing cell indexes, see the following section *"Cell Indexes"*):

Simple Stream<Record<K>> using TCStore API DSL

```
try (final Stream<Record<String>> recordStream =
    counterAccess.records()) {                               // <1>
    avg = recordStream
      .filter(stoppedCell.isFalse())                         // <2>
      .mapToLong(counterCell.longValueOr(0L))                // <3>
      .average();                                            // <4>
}
```

1. As above, retrieving a `Stream<Record<K>>`;
2. Filters on the `stoppedCell` being `false`;
3. Retrieves the values of `counterCell` as a `long`;
4. And finally, as above, averages them.

The DSL makes the actual query more readable to everyone and the example above is more self-describing than the initial implementation above it.

If we refactor the *"Single Record Update/Delete"* example from above using DSL to avoid moving the data to the client node, then this is what it would look like:

DSL-based counter update

```
                                                                   // <1>
import static com.terracottatech.store.UpdateOperation.allOf;
import static com.terracottatech.store.UpdateOperation.remove;
import static com.terracottatech.store.UpdateOperation.write;
...
      String dslCounterKey = "counter8";
      token = counterAccess
          .on(dslCounterKey)
          .iff(stoppedCell.isFalse()
              .and(counterCell.valueOr(0L).isGreaterThan(5L)))      // <2>
          .update(
              allOf(write(stoppedCell, true),                      // <3>
                  write(stoppedByCell, "Albin"),
                  remove(counterCell)))
          .map(Tuple::second)
          .map(r -> r.get(stoppedCell).orElse(false)
              ? r.get(stoppedByCell).orElse("<unknown>") :
                  "<not stopped>");                                // <4>
```

1. The use of static imports for the DSL helper methods is recommended to make the DSL-based code more readable.
2. The `iff` operation (if-and-only-if) is used to enable the following update operation only if the specified condition is `true`. In this case, *stoppedCell* is `false` and *counterCell* is greater than five.
3. This `update` operation specifies a collection of mutations to perform:

   (1) adds or updates the *stoppedCell* value to `true`,

   (2) adds or updates the *stoppedByCell* to `Albin`, and

   (3) removes *counterCell*.

   As with previous example, the `Record` is updated by all of these mutations in one atomic operation.
4. Maps the output of the update operation (a `Tuple` containing the old and new records) to select only the new record (the second of the tuple) then maps the new record to obtain the *stoppedByCell* value if the record is actually flagged as stopped.

The `delete` shown in the example *"Single Record Update/Delete"* is already expressed in proper DSL form and is not repeated in the example above.

Similar to the example *"DSL-based Counter Update"* above, a bulk operation is performed using a Java `Stream` over the records in the dataset through the `com.terracottatech.store.DatasetReader.records` method. With this pattern, bulk update is performed using TCStore:

Bulk Update

```
import static com.terracottatech.store.UpdateOperation.allOf;
import static com.terracottatech.store.UpdateOperation.remove;
import static com.terracottatech.store.UpdateOperation.write;
...
        try (final Stream<Record<String>> recordStream =
          counterAccess.records()) {                        // <1>
          recordStream
              .filter(stoppedCell.isFalse())                // <2>
              .forEach(counterAccess.functions()            // <3>
                  .update(
                      allOf(write(stoppedCell, true),       // <4>
                          write(stoppedByCell, "Albin"),
                          remove(counterCell))));
        }
```

1. Retrieve a `Stream<Record<String>>` through which updates will be performed. As with the previous `Stream`-based examples, note the use of the *try-with-resources* statement to enforce closure of the stream when operations are complete.
2. Filters the `Record` instances in the stream dropping those for which *stoppedCell* is `true`. Only the non-stopped records get past.
3. `forEach` is the *terminal operation* for the stream. `forEach` takes a `java.util.function.Consumer`.

In this example, a special consumer is used: one obtained from the `com.terracottatech.store.DatasetWriterReader.functions` method.

A consumer formed from `DatasetWriterReader.functions` can be used to update or delete records in the dataset from which the consumer was obtained.

4. As with some previous "Update" examples, an update operation with two write mutations and one remove mutation is specified. The update is applied to each selected record atomically - each record in its own atomic update.

Cell Indexes

To improve the performance of operations using streams obtained from `DatasetReader.records()`, cell indexes may be created for a `Dataset`. A cell index is defined against a `CellDefinition` instance. Entries are created in the index for all distinct `Cell` values for cells in the `Dataset` typed by the indexed `CellDefinition`. Each index entry associates that `Cell` value with the keys of all the `Record` instances containing that `Cell` value. When a stream pipeline refers to an indexed `CellDefinition`, particularly in a `filter` operation `Predicate` expressed using the TCStore DSL, iteration over the records in the dataset may be driven using the associated cell index.

A cell index may only be defined on a `CellDefinition` if its data type is `BOOL`, `CHAR`, `DOUBLE`, `INT`, `LONG`, or `STRING`.

An index may NOT be defined for a `CellDefinition` of type `BYTES`.

There are two ways to define an index:

(1) through the `DatasetConfigurationBuilder.withIndex` method during dataset creation or

(2) through the `Indexing` instance obtained from the dataset.

Defining an Index During `Dataset` *Creation*

```
StringCellDefinition LAST_NAME = CellDefinition.defineString("lastName");
DoubleCellDefinition NICENESS = CellDefinition.defineDouble("niceness");
...
DatasetManager datasetManager =
    DatasetManager.clustered(connectionURI).build();
DatasetConfiguration configuration =
    datasetManager.datasetConfiguration()             // <1>
    .offheap("offheap")
    .index(Person.LAST_NAME, IndexSettings.btree())   // <2>
    .index(Person.NICENESS, IndexSettings.btree())    // <3>
    .build();
Dataset<String> persons =
     datasetManager.createDataset(
        "people", Type.STRING, configuration)         // <4>
```

1. Create a `DatasetConfiguration` ...
2. ... specifying indexes for the `LAST_NAME` ...
3. and `NICENESS` cell definitions.
4. Create a `Dataset` using the configuration with the indexes.

Adding an Index to a `Dataset` *After Creation*

```
Indexing indexing = counterSet.getIndexing();           // <1>
Operation<Index<Boolean>> indexOp =
    indexing.createIndex(stoppedCell, IndexSettings.btree());    // <2>
try {
  indexOp.get();   // <3>
} catch (InterruptedException | ExecutionException e) {
  throw new AssertionError(e);
}
```

1. Get the `Indexing` instance for the `Dataset`. The `Indexing` instance for a `Dataset` may be used to add or delete a dataset's indexes.
2. Define an index specifying a `CellDefinition` identifying the cells whose values make up the index keys.
3. Runtime index creation is *asynchronous*; call the `get` method to await completion of the indexing operation.

Advanced Topics

As TCStore API is a storage API targeted for distributed datasets, the picture wouldn't be complete without addressing ...

- Durability
- Atomicity guarantees
- Asynchronicity

Durability Guarantees

TCStore API will always aim for maximum durability. The exact meaning of this depends on the configuration of the store itself:

- For non-persistent, non-replicated stores, it means the data made it to the server.
- For persistent, non-replicated stores, it means the data made it to the disk of the server.
- For persistent, replicated stores, it means the data made it to the disk of the servers and all replicas.

Atomicity

Beside the previously discussed read and write settings, TCStore API makes a `Record` the atomic unit of work.

So when you build a `Stream<Record<?>>` where you filter and then mutate (a bulk update), it will make sure you never mutate a `Record<?>` that doesn't pass that `Predicate` you used to filter, even if some other writer mutated the given `Record<?>` concurrently.

Also, if you mutate two `Cell` instances of one `Record`, depending on the write and read settings either both new or old values would be observed, but a reader would never be able observe one old and one new value for that given write.

The underlying principle is that a `Record` instances are effectively immutable.

Asynchronicity

As mentioned previously, TCStore API is targeted at distributed deployments. As such, it is inherently asynchronous. Although all the API examples we used in this document are synchronous, TCStore exposes an asynchronous API for you to use:

Asynchronous Operations

```
AsyncDatasetWriterReader<String> asyncAccess =
    counterAccess.async();                                        // <1>
Operation<Boolean> addOp =
    asyncAccess.add("counter10", counterCell.newCell(10L));       // <2>
Operation<Optional<Record<String>>> getOp =
    addOp.thenCompose((b) -> asyncAccess.get("counter10"));       // <3>
Operation<Void> acceptOp = getOp.thenAccept(or -> or.ifPresent(   // <4>
    r -> System.out.println("The record with key " + r.getKey() +
         " was added")));
try {
  acceptOp.get();      // <5>
} catch (InterruptedException | ExecutionException e) {
  e.printStackTrace();
}
```

1. Retrieves an asynchronous accessor to the `Dataset` in the form of `AsyncDatasetWriterReader`.
2. Schedule a *write a new* `Record`, identified by the key counter10 with an initial value of `counterCell` as 10. This returns an `Operation` instance which can then be waited upon (as a `java.util.concurrent.Future`) or combined with other operations (as a `java.util.concurrent.CompletionStage`).
3. Chain a get, to be performed once the write is complete, to the write operation.

   This does NOT affect the write nor does it combine the write and the get into an atomic unit.

   The result of this combination is also an `Operation` instance.
4. To the get `Operation` instance, chain a `Consumer` that processes the `Optional<Record<String>>` returned from the get. The result of this chain construction returns an `Operation<Void>`. `Consumer` returns no result.
5. As with *addOp* and *getOp*, the *acceptOp*`Operation` is a handle to an operation scheduled for background completion. This operation sequence will complete on its own but, to await completion, one of the `Future.get` methods must be called.

The default API is synchronous, as it is probably more obvious for everyone to get started with. But the power of the asynchronous API is exposed as well.

Note: When using the asynchronous API, operation sequencing is not maintained. If, in the example above, the get was not chained to the write, the get could be executed before the write even though the `Operation` instances were created in the opposite order. Mind, that two asynchronous operations are independent.

Frequently Asked Questions

What About my Caching Use Cases?

This part of the documentation covers the data storage API that is available as TCStore API.

Caching use cases are covered by the Ehcache API. This API covers caching use cases exclusively, and has a JSR-107-compliant interface.

To understand which API matches your particular use case we recommend for you to read *"Introduction to Terracotta DB > TCStore API Versus Ehcache: Why Two Different APIs?"*.

The Ehcache API and the TCStore API can operate on the same Terracotta Server Array instance simultaneously. However, their data sets will be segregated. I.e. you cannot "put" with the Ehcache API and expect to "get" the same entry with the TCStore API.

How Do I Store my Java Objects with the TCStore API?

The base TCStore API does not handle conversion of Java objects into records with typed cells.

In order to store objects, a common practice may be to store them serialized (in a byte[] cell) and supplement that with additional cells on the same record that contain the extracted fields (or any other values) that may be of interest for search and compute.

No Strong Schemas, Really?

Really! And this is a very powerful thing!

If you aren't convinced, spend some time searching the web for articles about "schema-on-write" vs. "schema-on-read". Modern systems that have sophisticated needs and must deal with multiple and/or quickly evolving data sources highly favor schema-on-read semantics - thus most NoSQL offerings take this approach.

Use cases such as those in IoT ("Internet of Things") spaces highly favor schema-on-read (weak or no enforced schema). As an example, different brands/models of sensors may provide data for the same thing (e.g. "humidity") in various different data types and formats yet the flood of data needs to be consumed and stored quickly.

TCStore API's "loose schema" approach - provides a powerful hybrid of schema-on-write and schema-on-read.

While core TCStore API functionality allows for completely schema-less usage, some use cases may prefer having some enforcement of some schema-like requirements. As such, some higher level constraints could be placed upon `Dataset` instances to enforce some loose `Dataset`-wide schemas, even though the underlying store would allow anything.

TCStore API does not include supplemental APIs for such functionality out of the box, however end-users could utilize patterns such as Decorators upon a `Dataset` to apply enforcement of schema (schema-on-write).

In deciding what functionality TCStore API may come to include with respect to enforcing loose-schemas, the Terracotta team is interested in hearing about your use cases. This is because there are many possible rule sets/semantics for loose-schema enforcement.

For example, possible meaning of a loose-schema definition could be:

A) schema defines the minimal set of cells a record must contain before storage

B) schema defines the maximal set of cells a record can contain before storage

C) schema defines the absolute set of cells that a record must contain before storage

D) schema defines the minimal set of cells a record must contain for it to be retrieved from storage

E) schema defines the absolute set of cells a record must contain for it to be retrieved from storage

F) etc ...

4 What is the Terracotta Server?

The Terracotta Server provides the distributed data platform for Terracotta products. A cluster of Terracotta Servers configured to work together is referred to as a Terracotta Server Array (TSA). A Terracotta Server Array can vary from a single server, to a basic two-server tandem for High Availability (HA), to a multi-server array providing configurable scale, high performance, and deep failover coverage.

The main features of the Terracotta Server include:

- **Distributed In-memory Data Management**

  Manages 10-100x more data in memory than data grids
- **Scalability Without Complexity**

  Simple configuration and deployment option for scaling-up and/or scale-out to meet growing demand and facilitate capacity planning
- **High Availability**

  Instant failover for continuous uptime and services
- **Configurable Health Monitoring**

  Terracotta health checker monitors client and server health
- **Persistent Application State**

  Automatic permanent storage of all current shared in-memory data with ultra-fast recovery upon server restarts
- **Automatic Node Reconnection**

  Temporarily disconnected server instances and clients rejoin the cluster without operator intervention

5 What is the Terracotta Management Console?

The Terracotta Management Console (TMC) is a web-based administration and monitoring application with many capabilities and advantages, including the following:

- Feature-rich and easy-to-use interface
- Remote management capabilities requiring only a web browser and network connection
- Visualize cluster topologies, monitor health, and manage Terracotta Servers
- Aggregates performance and usage statistics from multiple Terracotta nodes
- Cross-platform deployment
- Flexible deployment model, which can plug into both development environments and secure production architectures