

Achilles



Presentation

Achilles is an **open source** advanced object mapper for **Apache Cassandra**. Among all the features:

- Advanced bean mapping (compound primary key, composite partition key, timeUUID, counter, static column ...)
- Pluggable **codec system** to define your own types
- Life cycle **interceptors** to define custom behavior before INSERT/UPDATE/DELETE/SELECT operations
- Fluent **options system** to parameter runtime statements (consistency level, retry policy, ...)
- Powerful and **type-safe DSL** to create your own queries
- Display of DML scripts & DDL statements
- Wrapper to deploy an embedded **Cassandra** server easily
- Tight integration with JUnit for productive TDD programming
- Support for Bean Validation (JSR-303)
- Support for **Lightweight Transaction** with dedicated listener interface
- Support for **Materialized View**
- Support for typed-safe **Function calls**
- Support for the new **JSON API**
- Support for multi-project compilation unit

- Flexible naming strategy & insert strategy
- Runtime **Schema Name Provider** for multi-tenant environments
- Full compatibility with Java 8 **CompletableFuture**

Warning: Achilles versions 3.x are no longer maintained, only bug-fixes are supported, please migrate to version 4.x and follow the [Migration From 3.x Guide](#)

Installation

Below is the compatibility matrix between **Achilles**, **Java Driver** and **Cassandra** versions

| Achilles version | Java Driver version | Cassandra version |
|--|---------------------|-------------------|
| 5.0.0 | 3.1.0 | 3.7 |
| 4.2.3 | 3.1.0 | 3.7 |
| 4.0.1 (limited to Cassandra 2.2.3 features) | 3.0.0-alpha5 | 2.2.3 |
| 3.2.3 (limited to Cassandra 2.1.x features) | 2.1.6 | 2.1.5 |
| 3.0.22 (limited to Cassandra 2.0.x features) | 2.1.6 | 2.0.15 |

Warning: there will be no new features for branches older than **5.0.x**. Those branches are only supported for bug fixes. New features will **not** be back-ported. Please upgrade to the latest version of **Achilles** to benefit from new features

To use **Achilles**, just add the following dependency in your **pom.xml**:

```
<dependency>
  <groupId>info.archinno</groupId>
  <artifactId>achilles-core</artifactId>
  <version>${achilles.version}</version>
</dependency>
```

Do not forget to deactivate *incremental compilation* and use *Java 8* in your **pom.xml** file

```
<build>
  <plugins>
    <plugin>
```

```

        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
            <source>1.8</source>
            <target>1.8</target>
            <useIncrementalCompilation>>false</useIncrementalCompilation>
        </configuration>
    </plugin>
</plugins>
</build>

```

Achilles 4.x requires a JDK 8 to work. It is recommended to use JDK 8 update 45 or later

For unit-testing with embedded Cassandra, add this dependency with **test** scope:

```

<dependency>
    <groupId>info.archinnov</groupId>
    <artifactId>achilles-junit</artifactId>
    <version>${achilles.version}</version>
    <scope>test</scope>
</dependency>

```

For now, **Achilles** depends on the following libraries:

1. cassandra (see matrix version above)
2. cassandra-driver-core (see matrix version above)
3. Jackson core, annotations, databind & module jaxb annotations 2.3.3
4. Google Auto Common 0.4
5. Google Auto Service 1.0-rc2
6. Java Poet 1.5.1
7. Guava 18.0
8. slf4j-api 1.7.2
9. commons-io 2.4
10. commons-lang3 3.3.2
11. commons-collections 3.2.1
12. validation-api 1.1.0.Final
13. org.eclipse.jdt.core.compiler-ecj 4.4.2

Configure Your IDE

Achilles is using code generation at compile time through annotation processors, you'll need to configure your IDE carefully. Please follow the [IDE Configuration](#) guide

5 minutes tutorial

To bootstrap quickly with **Achilles**, you can check the [5 minutes tutorial](#)

Quick Reference

To be productive quickly with **Achilles**. Most of useful examples are given in the [Quick Reference](#)

Advanced tutorial

To get a deeper look on how you can use **Achilles**, check out the [KillrChat](#) application

Documentation

All the documentation and tutorial is available in the [Wiki](#)

Versioned documentation is available at [Documentation](#)

Mailing list

For any question, bug encountered, you can use the [mailing list](#)

License

Copyright 2012-2016 DuyHai DOAN

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this application except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Changelog

Branch 5.x

- **5.0.0**
 - Bump version to 5.0.0-SNAPSHOT
 - Update FunctionCall interface because JAVA-1086 is solved
 - Write DML log in a single call to logger.debug()
 - Add CAST system function
 - Add multi-columns slice restrictions for UPDATE/DELETE
 - Add Support for JSON
 - Rename manager.query() to manager.raw()
 - Return TypedMap only if there is a function call in SELECT DSL
 - Merge Raw and Typed queries under the same package
 - Refactor TypedMap interfaces
 - Split IT modules to match Cassandra version
 - Fix #255 Allow multiples projects to run in the same JVM
 - Normalize the DSL CodeGen
 - Reorganize UPDATE/DELETE DSL CodeGen
 - Allow selection of UDT fields
 - Version DSL CodeGen classes
 - Version Validators

Branch 4.x

- **4.2.3**
 - Stop using unsafe embedded Cassandra in IT tests
 - Add missing UDT DDL logs
 - Fix #257 Handle upper-cased column name specified on @Column
 - Fix #259 Incorrect bound values for UPDATE when using TTL
- **4.2.2**
 - Move ScriptExecutor from ‘achilles-embedded’ up to ‘achilles-common’
 - Fix #256 Insufficient param for String.format in DEBUG log
 - Fix counter incr/decr generated code
 - Only return the first page of data instead of all pages
- **4.2.1**
 - Upgrade Cassandra version to 3.7
 - Upgrade to Java driver 3.1.0

- Disable unsafe Cassandra daemon
 - Fixes #254 Multi-Layer UDT fails on createBeanFromUDT()
 - Add optional input for PagingState, RetryPolicy, ConsistencyLevels and OutgoingPayload
 - Add new methods to return iterator() along with ExecutionInfo
 - Fix bug when querying with (...) tuple notation on clustering columns
 - Fix error in log message formatting
 - Add optional version of statement parameters for nullable input
 - Remove ‘password’ as reserved keyword
- **4.2.0**
 - Add @SuppressWarnings on generated code
 - SELECT DSL TypedMap API renaming
 - Upgrade to Cassandra 3.5
 - Upgrade version to 4.2.0-SNAPSHOT before release
 - Update AbstractManagerFactoryBuilder to inject SERIAL consistency level
 - Fix #245 Consistency level ordering
 - Fixes #246 Add guard before checking isXXXCompiler()
 - Fixes #248 Fix dependency injection ordering bug 2
 - Fixes #244 Fix dependency injection ordering
 - Fixes #240 Do not check for clustering column when inserting only static columns
 - Fixes #227 Support for UDF/Aggregates
 - Upgrade to Cassandra 3.4
 - Call shutdownNow() on single thread executor upon termination
 - Internal refactoring
 - Internal renaming of TypeParsingResult to FieldMetaSignature
 - Fixed debug log issue (PR #250 from bhuvanrawal/debug_log_fix)
 - Update Commons-Collections (PR #243 from cgils/master)
 - Fixes #225 Add support for function calls/UDF call in DSL API
 - Fix typo in tracing logs
 - Fix small typo on @Table javadoc
 - Provide more details for JDKOptional TypeTokens
 - Catch all exceptions during compile-time and print the complete stack trace
 - Add missing dependency on org.ow2.asm.asm 5.0.4 and driver class CassandraTypeParser
 - Make SELECT DSL also return TypedMap in addition of entity
 - Use an unsafe Cassandra Daemon for unit testing
 - Fixes issue with UDT schema not created when UDT is nested in collection

- 4.1.0
 - **BREAKING CHANGE** Rename `@Entity` to `@Table`
 - Upgrade Cassandra version to 3.3
 - Fixes #237 Support for Materialized Views
 - Add scriptTemplate support for AchillesTestResoureBuilder
 - Add JDK Optional codec
 - Onsite codec declaration override codec registry
 - Support JDK8 Date types
 - Fixes JavaDoc rendering issue on DSL methods
 - Use exact type when calling get/set on Row object
 - Only generate UDT schema associated for managed entity classes at runtime
 - Make AchillesProcessor stateful and do not regenerate classes many times
 - Update JavaPoet to version 1.5.1
 - Add support for numeric array data types
 - Fixes #222 Support for @RuntimeCodec
 - **BREAKING CHANGE** Remove support for @Codec on class, prefer @CodecRegistry
 - Code cleaning
 - Forbid to have 2 entities with the same class name
 - Implement compile time codec registry
 - Replace stream().forEach() by simple for loops
 - Fixes integration test
 - Fixes #235 Entity mapped on non existing table/keyspace cause nullpointer exception
 - Use simple for loop instead of useless stream.forEach()
 - By Default, clean data folder for embedded Cassandra
 - Fixes #230 null pointer on invalid keyspace
 - Fixes #231 Upgrade driver version to 3.0.0
 - Upgrade Cassandra version to 3.2.1

- Better error message for Schema comparison
- Clean generated source files on each rebuild
- Disable incremental compilation to fix successive ‘mvn compile’ issue
- Update Maven Surefire Plugin version to 2.13
- Update Cassandra version to 3.1.1
- Update Java Driver version to 3.0.0-rc1
- Fix Tracing on class level logger
- Add validation for nested types
- **4.0.1**
 - Support UPDATE ... IF EXISTS
 - Upgrade to Java Driver **3.0.0-alpha5**
 - Refactor AnnotationTree to support Eclipse compiler
- **4.0.0**
 - Complete rewrite

Branch 3.2.x

- **3.2.3**
 - Fixes #217 Cannot install achilles in OSGI environment due to duplicate package
BREAKING CHANGE Move `info.archinno.achilles.json.CounterSerializer` and `info.archinno.achilles.json.CounterDeSerializer` to `info.archinno.achilles.type.CounterSerializer` and `info.archinno.achilles.type.CounterDeSerializer`
- **3.2.2**
 - Fixes #219 Get the PagingState from the TypedQuery, NativeQuery

- Fixes #217 Cannot install achilles in OSGI environment due to duplicate package
 - BREAKING CHANGE** Move `info.archinnov.achilles.type.Options` to `info.archinnov.achilles.options.Options`
 - BREAKING CHANGE** Move `info.archinnov.achilles.type.OptionsBuilder` to `info.archinnov.achilles.options.OptionsBuilder`
 - Fixes #216 Interceptors do not support Entity inheritance
- **3.2.1**
 - **BREAKING CHANGE** Fixes #209 Allow UNLOGGED batches
 - * `createBatch()` is replaced by `createLoggedBatch()`
 - * `createOrderedBatch()` is replaced by `createOrderedLoggedBatch()`
 - * add `createUnloggedBatch()` and `createOrderedUnloggedBatch()`
 - Fixes #214 Achilles does not support static field only in rows
 - Merge from **3.0.20**
 - * Upgrade java driver version to 2.1.6
 - * Fixes #212 Reflection does not find entities
 - **3.2.0**
 - Update Cassandra to version 2.1.5
 - **BREAKING CHANGE** Fixes #178 `insert()` API should return `void ...`, remove `refresh()` API
 - **BREAKING CHANGE** Fixes #168, #179 and #210 Return proxy on request and not by default
 - Merge from **3.0.19**
 - * Fixes #211 Should not print bound values if `ACHILLES_DML_STATEMENT` log/DEBUG log on entity is not active
 - * Fixes #208 Add ‘close’/‘shutdown’ method for `AsyncManager`
 - * Fixes #207 Null Pointer Exception for usecase: LWP +Embedded Id
 - * PR #205 lazy creation of bound statements

Branch 3.1.x

- **3.1.7**
 - Merge from **3.0.18**
 - * Fixes #196 unable to create indexes using `CASE_SENSITIVE` naming convention

- * PR #202 Print the Hex string of the 16 first bytes of blob when ACHILLES_DML_STATEMENT is enabled
 - Adds byte length info when buffer is longer than limit
 - Enables logging of the first 16 bytes of a blob when ACHILLES_DML_STATEMENT is enabled
- * PR #201 Upgrades Achilles to work with java-driver 2.1.5
 - Replaces fromNullable with Optional.of for non nullable parameters
- * PR #200 Better thread pool lifecycle management
 - do not create default thread pool if there is a client-supplied one in config
 - shutdown the thread pool only if it is the default one
- * Fixes #197: makes embeddedcassandra configurable for the number of reads/writes threads

- **3.1.4**

- Upgrade to Java driver version 2.1.4 and Cassandra version 2.1.3
- Fixes #194 Allow <, <=, >, >= and != for conditional updates using LWT
- Merge from **3.0.17**
 - * Fixes #193 Load all counter values at once for clustered counters
 - * Fixes #187 Support BATCH statements in script for ScriptExecutor
 - * Fixes #185 Enhance ScriptExecutor to return ResultSet on execute() method
 - * Fixes #188 Add @PreDestroy on shutDown() method of PersistenceManager and document it
 - * Fixes #184 Update statements with LWT are re-prepared many times
 - * Fixes #191 CqlColumn is not mapped to the java field
 - * Merge pull request #190 from seblm/master to simplify execute batch

- **3.1.3**

- Merge from **3.0.16**
 - * Late clearing of dirty map for POST_UPDATE interceptors
 - * Fixes #145 Allow bootstrap CQL script execution for AchillesTestResource and CassandraEmbeddedServer
 - * Fixes #167 Remove the default TTL = 0 in prepared statements because some tables have default TTL value

- * Fixes #180 Expose a close() method on the PersistenceManagerFactory to close the internal thread pool properly
 - * Add withEntityClasses(Class<?> ... entityClasses) for the AchillesResourceBuilder
 - * Fixes #174 Add iterator() and iterator(int fetchSize) to the TypedQuery API
 - * Deprecate @Id and @EmbeddedId annotations
 - **DEPRECATE** @Id annotation, use @PartitionKey instead
 - **DEPRECATE** @EmbeddedId annotation, use @CompoundPrimaryKey instead
- **3.1.2**
 - Merge from **3.0.15**
 - * Migrate to Java Driver 2.1.3
 - * Pull Request #177 Use proxy instead of entity when trigger a post_update interceptor
 - * Fixes #176 Expose the Cluster object for the CassandraEmbeddedServer
 - * Fixes #175 Deprecate @Order in favor of @PartitionKey and @ClusteringColumn * **DEPRECATE**@Orderannotation, use@PartitionKeyand@ClusteringColumn‘ instead
- **3.1.1**
 - Un-ignore a test because CASSANDRA-7499 has been fixed
 - Merge from **3.0.14**
 - * Add IT for forUpdate() API
 - * Fixes #171 Refactor Options internally to use Optional everywhere
 - * Fixes #172 Lowercase LWTLocalSerial() and LWTResultListener() for OptionsBuilder
 - **DEPRECATE** OptionsBuilder.LWTLocalSerial() API, use OptionsBuilder.lwtLocalSerial() instead
 - **DEPRECATE** OptionsBuilder.LWTResultListener() API, use OptionsBuilder.lwtResultListener() instead
 - * Fixes #173 Support timestamp and LWT predicates for DELETE operations
 - * Add ifExists() and refactor the LWT predicates.Introduce ifEqualCondition()
 - **DEPRECATE** OptionsBuilder.ifConditions() API, use OptionsBuilder.ifEqualCondition() instead

- **3.1.0**
 - Remove deprecated `manager.getProxy()` API
 - Replace Log4J by LogBack because of Cassandra 2.1.2
 - Migrate Guava to 16.0 because of Cassandra 2.1.2
 - Migrate Cassandra to 2.1.2

Branch 3.0.x

- **3.0.22**
 - Fixes #217 Cannot install achilles in OSGI environment due to duplicate package
BREAKING CHANGE Move `info.archinnov.achilles.json.CounterSerializer` and `info.archinnov.achilles.json.CounterDeSerializer` to `info.archinnov.achilles.type.CounterSerializer` and `info.archinnov.achilles.type.CounterDeSerializer`
- **3.0.21**
 - Fixes #219 Get the PagingState from the TypedQuery, NativeQuery
 - Fixes #217 Cannot install achilles in OSGI environment due to duplicate package
BREAKING CHANGE Move `info.archinnov.achilles.type.Options` to `info.archinnov.achilles.options.Options`
BREAKING CHANGE Move `info.archinnov.achilles.type.OptionsBuilder` to `info.archinnov.achilles.options.OptionsBuilder`
 - Fixes #216 Interceptors do not support Entity inheritance
- **3.0.20**
 - Upgrade java driver version to 2.1.6
 - Upgrade Cassandra to version 2.0.15
 - Fixes #212 Reflection does not find entities
- **3.0.19**
 - Fixes #211 Should not print bound values if `ACHILLES_DML_STATEMENT` log/DEBUG log on entity is not active
 - Fixes #208 Add 'close'/'shutdown' method for AsyncManager
 - Fixes #207 Null Pointer Exception for usecase: LWP +Embedded Id
 - PR #205 lazy creation of bound statements
- **3.0.18**

- Upgrade to Cassandra 2.0.13
 - Fixes #196 unable to create indexes using CASE_SENSITIVE naming convention
 - PR #202 Print the Hex string of the 16 first bytes of blob when ACHILLES_DML_STATEMENT is enabled
 - * Adds byte length info when buffer is longer than limit
 - * Enables logging of the first 16 bytes of a blob when ACHILLES_DML_STATEMENT is enabled
 - PR #201 Upgrades Achilles to work with java-driver 2.1.5
 - * Replaces fromNullable with Optional.of for non nullable parameters
 - PR #200 Better thread pool lifecycle management
 - * do not create default thread pool if there is a client-supplied one in config
 - * shutdown the thread pool only if it is the default one
 - Fixes #197: makes embeddedcassandra configurable for the number of reads/writes threads
- **3.0.17**
 - Upgrade to Java driver version 2.1.4 and Cassandra version 2.0.12
 - Fixes #193 Load all counter values at once for clustered counters
 - Fixes #187 Support BATCH statements in script for ScriptExecutor
 - Fixes #185 Enhance ScriptExecutor to return ResultSet on execute() method
 - Fixes #188 Add @PreDestroy on shutDown() method of PersistenceManager and document it
 - Fixes #184 Update statements with LWT are re-prepared many times
 - Fixes #191 CqlColumn is not mapped to the java field
 - Merge pull request #190 from seblm/master to simplify execute batch
 - **3.0.16**
 - Late clearing of dirty map for POST_UPDATE interceptors
 - Fixes #145 Allow bootstrap CQL script execution for AchillesTestResource and CassandraEmbeddedServer
 - Fixes #167 Remove the default TTL = 0 in prepared statements because some tables have default TTL value
 - Fixes #180 Expose a close() method on the PersistenceManagerFactory to close the internal thread pool properly
 - Add withEntityClasses(Class<?> ... entityClasses) for the AchillesResourceBuilder

- Fixes #174 Add `iterator()` and `iterator(int fetchSize)` to the **TypedQuery** API
- Deprecate `@Id` and `@EmbeddedId` annotations
 - * **DEPRECATE** `@Id` annotation, use `@PartitionKey` instead
 - * **DEPRECATE** `@EmbeddedId` annotation, use `@CompoundPrimaryKey` instead
- **3.0.15**
 - Migrate to Java Driver 2.1.3
 - Pull Request #177 Use proxy instead of entity when trigger a `post_update` interceptor
 - Fixes #176 Expose the Cluster object for the `CassandraEmbeddedServer`
 - Fixes #175 Deprecate `@Order` in favor of `@PartitionKey` and `@ClusteringColumn` * **DEPRECATE** `@Order` annotation, use `@PartitionKey` and `@ClusteringColumn` instead
- **3.0.14**
 - Add IT for `forUpdate()` API
 - Fixes #171 Refactor Options internally to use Optional everywhere
 - Fixes #172 Lowercase `LWTLocalSerial()` and `LWTResultListener()` for `OptionsBuilder`
 - * **DEPRECATE** `OptionsBuilder.LWTLocalSerial()` API, use `OptionsBuilder.lwtLocalSerial()` instead
 - * **DEPRECATE** `OptionsBuilder.LWTResultListener()` API, use `OptionsBuilder.lwtResultListener()` instead
 - Fixes #173 Support timestamp and LWT predicates for DELETE operations
 - Add `ifExists()` and refactor the LWT predicates. Introduce `ifEqualCondition()`
 - * **DEPRECATE** `OptionsBuilder.ifConditions()` API, use `OptionsBuilder.ifEqualCondition()` instead
- **3.0.13**
 - Fixes buggy implementation of `manager.forUpdate()`
- **3.0.12**
 - Fixes #85 How do you add/remove from set using proxied object

- * Add new `manager.forUpdate()` API. More details here at [Direct Update Proxy](#)
- * **DEPRECATE** `manager.getProxy()` API
- Add ‘forceTableCreation’ system property override for embedded Cassandra
- Fixes #165 Deactivate Bean Validation on fields which are not set on proxy
- Pull Request #166 Fixing issue with Achilles counter as the counter table name is not returned in the entity metadata thus it will be created at each start up.
- Fixes #164 Class level consistency not taken into account by slice query
- Fixes #163 Add ‘key’ and ‘value’ attribute on @JSON annotation for collections & map support
- **3.0.11**
 - Migrate to Java Driver 2.1.2
 - Fixes #162 Log DML statements before executing the query
 - Fixes #160 Improve the thread pool usage by using `MoreExecutors.sameThreadExecutor()` for simple transformation
 - Fixes #159 Rename CQL3 to CQL internally
 - Fixes #161 Polish the Asynchronous API
 - Fixes #158 Rename all CAS in the API to LWT
 - * **RENAMING:** `CasCondition` -> `LWTCondition`
 - * **RENAMING:** `CASResultListener` -> `LWTResultListener`
 - * **RENAMING:** `CASResult` -> `LWTResult`
 - * **RENAMING:** `CASResultListener.onCASSuccess()` -> `LWTResultListener.onSuccess()`
 - * **RENAMING:** `CASResultListener.onCASError(...)` -> `LWTResultListener.onError(...)`
 - * **RENAMING:** `OptionsBuilder.casResultListener(...)` -> `OptionsBuilder.lwtResultListener(...)`
 - * **RENAMING:** `OptionsBuilder.casLocalSerial()` -> `OptionsBuilder.lwtLocalSerial()`
 - Fixes #157 Add shutdown hook to the internal `ExecutorService`
 - Fixes #153 Allow inserting rows in clustered table with static columns without clustering columns
 - Fixes #149 Make JSON serialization more explicit with a new @JsonType annotation

- Fixes #155 CassandraEmbeddedServer fails in maven multi-module project
- Fixes #126 NativeQuery API should also accept Statement instead of only RegularStatement
- Fixes #156 Allow @TypeTransformer on partition and clustering columns
- **3.0.10**
 - Fixes #152 Allow PersistenceManagerFactoryBuilder to pass in arbitrary parameter
 - Better defaults for the asynchronous Thread Pool
 - Add perf bench with SynchronousQueue and LBQ
 - Memory consumption improvement
- **3.0.9**
 - Add support for Asynchronous operations
- **3.0.8**
 - Upgrade to Java Driver 2.0.7 and Cassandra 2.0.11
 - Fixes #139 Avoid useless array objects creation for loggers at DEBUG/TRACE level
 - Fixes #144 FromClusterings/ToClusterings methods of SliceQuery API should take into account the clustering order
 - Merge #141 Make Achilles proxies not intercepting 'finalize method
 - Fixes #138 Wrong regexp validation for SELECT * from table
 - **RENAMING** Renaming Event enum and rename remove/removeById to delete/deleteById
 - Add new config options for JUnit Achilles Resource
- **3.0.7**
 - Upgrade to Java Driver 2.0.6 and Cassandra 2.0.10
 - Fixes #95 **Type transformer**
 - Fixes #136 Achilles is consuming too many resources
 - Fixes #42 Add a **naming strategy** to @Table
 - Fixes #111 Reorganize Maven dependencies
 - Fixes #127 NativeQuery should expose Iterator<TypedMap>
 - Fixes #112 Use Set instead of List for lifecycle interceptors to avoid duplicate
 - Fixes #129 Switch to commons-lang3
 - Fixes #128 Support **multi keyspaces**

- Fixes #133 Achilles fails with NPE in case of difference between the actual column families and the achilles mappings
 - Fixes #125 BatchedNativeQuery error with multiple statements in batch
- **3.0.6**
 - Upgrade to Java Driver **2.0.4**
 - **RENAMING** Rename AchillesCASEException to Achilles-LightweightTransactionException
 - Fixes #110 Fix JSON serialization/deserialization of Counter
 - Fixes #122 Support name & ordinal encoding for Enums
 - Fixes #108 : Slice query partition components and clustering keys should be encoded properly
 - Fixes #113 Apply **Tell, Don't Ask** to meta data
 - Pull request #121 Fixes #119 Validation of PartitionComponents fails for simple types
 - Pull request #117 Added enableSchemaUpdate and enableSchemaUpdateForTables to PersistenceManagerFactoryBean
 - Pull request #116 NPE when entity not managed for slice query
 - Pull request #115 fixes #114 where batch native queries did not accept DELETE statements
 - **3.0.5:**
 - Fixes #103 Refactor PersistenceManager & Batch hierarchy design
 - Fixes #87 @Index validation failing deploy
 - Fixes #107 DML statement not logs when dynamic logging is activated for an entity
 - Fixes #96 Use user comment as comment on table when generating DDL
 - Fixes #99 Stop using **Objenesis** for object instantiation
 - Fixes #84 Enhance native query to be added to batch manager
 - Fixes #106 CAS Result not decoding properly collection and map types
 - Fixes #100 Apply @EmptyCollectionIfNull to lazy loaded collections/map too
 - Fixes #91 Accept **RegularStatement** object instead of String type for **native query** & **typed query** API
 - Fixes #102 Make it possible to create Batch directly from PersistenceManager
 - **RENAMING** Fixes #105 Rename **persist()** to **insert()**
 - **RENAMING** Fixes #101 Normalize naming for **jsonSerialize()** and **deserializeJson()**
 - Fixes #86 PersistenceManager.typedQuery force lower case

- Fixes #90 Add new `insertOrUpdate()` method
 - Fixes #97 `SliceQuery.iterator(fetchSize)` doesn't seem to work
 - Fixes #98 NPE on proxy when invoking super
 - **RENAMING** Fixes #104 Rename parameters `OBJECT_MAPPER` to `JACKSON_MAPPER`
 - Fixes #78 Improve JavaDoc
 - Fixes #73 `TimeUUID` is handled incorrectly in `SliceQuery`
 - Fixes #46 Remove limitation on slice query with several clustering columns
 - Fixes #79 Refactor the [Slice Query API](#)
- **3.0.4:**
 - Fixes #75 Add support for [static columns](#)
 - Fixes #88 Add [SchemaBuilder](#)
 - Fixes #77 Add support for Serial consistency level for CAS operations
 - Fixes #81 Rename `BatchingPersistentManager` to `Batch`
 - Fixes #76 Enable 2 `batchingPersistenceManager` types: `Ordered` and `UnOrdered`
 - Fixes #83 Per table consistency level not used
 - Fixes #54 [Support for OSGI](#)
 - Fixes #82 [Data lossless automatic schema modification](#)
 - Fixes #66 [Trace query time from achilles pov](#)
 - Upgrade to **Cassandra 2.0.8**
 - Fixes #63: Cannot find matching property meta for the type class `java.nio.HeapByteBuffer`
 - Fixes #72 Use enum for [Configuration parameters](#) instead of plain text
 - Fixes #74 Cluster should be injected into Achilles, not built by Achilles
 - Fixes #69 Make [insert strategy](#) overridable on an entity
 - Fixes #68 Add support for `LOCAL_ONE` consistency level
 - Fixes #71 Migrates to Jackson 2.3.3
 - **3.0.3:**
 - Upgrade to **Cassandra 2.0.7** and **Java Driver 2.0.2**
 - Disable the batch statement ordering by default
 - Change default `PreparedStatement` Cache LRU size from **5000** to **10000**
 - Fixes #64 NPE in typedQuery `select.getValues()`
 - Fixes #61. Refactor implementation of insert with Insert strategies
 - Fixes #53 Add support for `@EmptyCollectionIfNull`
 - Rename `TypedQueryBuilder/NativeQueryBuilder` to `TypedQuery/NativeQuery`

- Fixes #43 Implement CAS using Options
 - Fixes #40 Add helper method to serialize Entity into Json
 - Fixes #59 **@Index** on Enum field throws AchillesBeanMappingException
 - Fixes #57 Support inheritance for **@EmbeddedId** classes
 - Fixes #58 Support list of entities classes
 - Fixes #56 Test failed: EntityMetaTest.should_tostring
 - Fixes #55 Cannot find PersistenceManagerFactory for keyspace 'achilles_test' in test
- **3.0.2:**
 - Add **FORCE_BATCH_STATEMENTS_ORDERING** parameter to allow overriding the default behavior
 - Upgrade to **Cassandra 2.0.6** and **Java Driver 2.0.1**
 - Fixes #51 Warm up proxies creation on startup
 - Fixes #49 Expose configuration for Max Prepared statement cache size
 - Fixes #52 Fix makeFieldAccessible for multi-threaded execution
 - Fixes #45 Rework on collection & map dirty check to update only added/removed/modified element
 - Improve batch log messages
 - Take into account the settings for **cluster name**, **compression** and **policies**
 - Migrate to **Guava 15.0**
 - **3.0.1:**
 - Add support for **Bean Validation (JSR-303)**
 - Fix bug on supported Java types. Add support for **byte**, **Byte** and **byte[]**
 - Add **TypedMap** for user-friendlier native query usage
 - Fix bug on **Counter** serialization
 - **3.0.0 MAJOR BREAKING CHANGES:**
 - Redesign **Counter** implementation
 - Remove **lazy loading** feature
 - Rework on **persist()** and **update()** API
 - Rename **merge()** to **update()**, **getReference()** to **getProxy()**, **unwrap()** to **removeProxy()** and **initAndUnwrap()** to **initAndRemoveProxy()**
 - Add dependency to **Objenesis** to instantiate entities without requiring default constructor to remove constraint on the mandatory default constructor for entities

- Add lifecycle interceptors
 - Add support for **CQL paging feature** with slice query iterators
 - Add support for **parameterized query** to typed, raw typed and native queries
 - Migrate to Cassandra 2.0.3 and Java Driver 2.0.0-rc2
 - Migrate to **Java 7**
 - Improve **CassandraEmbeddedServerBuilder** to bootstrap embedded **Cassandra 2.0** server
 - Add “keyspace” parameter to **AchillesResourceBuilder**
 - Rename “achilles.ddl.force.column.family.creation” parameter to “achilles.ddl.force.table.creation”
 - Remove all **CQL** prefix from classes since there is only one implementation now
 - Split code into several Maven modules
 - Remove **Thrift** implementation
- **2.0.9:**
 - Add **reversed** attribute to **@Order** annotation
 - Fix missing validation for **find()** & **getReference()** when entity class is not managed by **Achilles**
 - Rework on CQL embedded server and introduce **CQLEmbeddedServerBuilder**
 - Fix bug when **@Column** and **@Id** annotations are put on the same field
 - Fixes #36 Schema validation fails with Cassandra 2.0.x
 - Migrates to Cassandra Driver Core 1.0.4
 - Fixes bug when ‘achilles.keyspace.name’ parameter not provided when bootstrapping with a **Cluster** and **Session** objects
 - Introduce **achilles-test** module to make Maven dependencies cleaner
 - Add simple support for secondary index
 - Create new **achilles-documentation** project to package full documentation (HTML and PDF formats) along with each release
 - **2.0.8:**
 - Provide an HTML client (AngularJS) for the Twitter Demo
 - Fixes #34 When it is impossible to call a setter on an entity, Achilles should report a better error
 - **BREAKING CHANGE** stop importing JPA jar. The number of annotations used (5) does not worth pulling JPA dependencies
 - Add custom **Jackson** serializer & deserializer on **Counter** interface to fix issue during serialization when **CounterImpl** is present
 - Rework on **ResourceBuilder** for JUnit

- **2.0.7:**
 - **BREAKING CHANGE** Rename all `xxEntityManagerxx` to `xxPersistenceManagerxx`
 - Fixes #30 Create partition row key
 - **BREAKING CHANGE** Remove join feature. Its usage is not relevant enough in real use-cases
 - Fixes #29 [All] Separate Achilles annotations and custom types in a dedicated **achilles-model** Maven module
 - Allow injection of **CQL** cluster and session object into `CQLEntityManagerFactory`
 - Fixes #27 [CQL] Cannot map compound primary key from CQL resultset back to entity for typed queries
 - Fixes #25 [CQL] Introduce new `@TimeUUID` annotation to translate into `timeuuid` in Cassandra enhancement
 - Fix #26 Lift the constraint on mandatory “entityPackages” parameters and mandatory entity class to bootstrap Achilles in **CQL**
 - Fixes #13 Redesign collections & map impl to match the one used in CQL
 - Add Eclipse cleanUp, codeTemplates and formatter for contributors
- **2.0.6:**
 - Add cleaning of embedded Cassandra data files before launching the server in JUnit rule
 - Upgrade to Hector 1.1-4 for Achilles Thrift
- **2.0.5:**
 - Minor refactor for embedded Cassandra server and test resources
 - Display bound values in DML debug messages for bound statements
 - Introduce Options to simplify setting of TTL and Timestamp
 - Force initialization of counter type when calling `initialize()`
 - Bug with bi-directional relation with a cascade persist/merge and ‘achilles.consistency.join.check’ option set to true
- **2.0.4:**
 - Migrate to Cassandra 1.2.8
 - Fix NPE when no join entity in collection/map for CQL
 - Remove Cassandra Unit library to avoid dependency on Hector for CQL version
 - Add Cassandra Embedded server. Add JUnit rule to bootstrap Cassandra embedded server and Achilles together
- **2.0.3:**

- Fix buggy implementation of `FactoryBean` for Spring integration
- Fix bug in decoding enums for Slice Queries
- **2.0.2:**
 - Internal refactoring of `PropertyMeta`
 - Support for value-less entities
 - Remove `@CompoundKey` annotation, un-used
- **2.0.1:**
 - Add Spring Integration for CQL
 - Add new parameters for **CQLEntityManagerFactory**
 - Enhance **SliceQueryBuilder** API
 - Fix bug on Thrift clustered entity validation
 - Fix small bug on CQL RawTyped queries
- **2.0.0:**
 - **Official support for CQL.** See docs for more details
 - Remove custom `Pair` type, use the one provided by Cassandra
 - Fixes bug during entities parsing when List/Set/Map have parameterized type
 - Add `removeById()` to `EntityManager` to avoid read-before-write pattern
 - **BREAKING CHANGE**, removal of `WideMap<K,V>` structure because its use cases are not consistent
 - Add DML logs to display CQL statements at runtime
 - Serialize `enum` types using `name()`
 - **BREAKING CHANGE**, removal of `@WideRow` annotation, replaced by clustered entities and slice queries for getting data (see doc for more details) .
- **1.8.2:**
 - Support immutability for `@CompoundKey` classes (Issue #5). See <https://github.com/doanduyhai/Achilles/issues/5#issuecomment-19882998> for more details on new syntax
 - **BREAKING CHANGE**, for `@CompoundKey`, enum types are serialized as String instead of Object (Issue #8)
 - **BREAKING CHANGE**, replace `@Key(order=1)` by `@Order(1)` annotation (Issue #6)
 - Throw Exception instead of having NPE when keyspace is not created in Cassandra (Issue #10)

- **BREAKING CHANGE**, replace `MultiKeyinterface` by `@CompoundKey` annotation (Issue #15)
 - It is no longer required to implements `Serializable` on all entities (Issue #11)
 - Enforce consistency in `ThriftImpl` on `persist()` by removing all row data first (Issue #14 & Issue #7)
 - Add timestamp meta data to `KeyValue<K,V>` type (Issue #9)
 - Add `initAndUnwrap()` shortcut to `EntityManager` (Issue #2)
 - Fix bug on dirty check on join collection/map
- **1.8.1:**
 - Add TTL to persist and merge operations
 - Rework of runtime consistency level
 - Fix small bug in the Column Family comment for wide rows
 - **BREAKING CHANGES** simplify `PropertyType` byte flag for Thrift persistence layer. **Need data migration**
 - Rename ‘unproxy’ to ‘unwrap’
 - Fix bug with cyclic join entities during merge and persist operations
 - Fix buggy property removal after merging
 - Ignore un-mapped properties instead of raising exception
 - Renaming in core package
 - Upgrade to Cassandra 1.2.4
- **1.8.0:**
 - Use the `org.reflections.reflections` package for entity parsing
 - Split the project into 3 modules: `core`, `thrift` and `cql`
- **1.7.3:**
 - Rework of `em.getReference()` to avoid hitting the database
 - Refactor generics
 - Migrate to JPA 2
- **1.7.2:**
 - Add commons-collections 3.2.1 to compile dependency
 - Enhance error message for entity mapping
 - Fix bug about default consistency level hard-coded to QUORUM for WideMap and Counters
 - Add `firstFirstMatching()` and `findLastMatching()` to the WideMap API
 - Fix bug. Do not load join entity if no join primary keys

- **1.7.1**: fix bug because key validation class & comparator type has changed from Cassandra 1.1 to 1.
 - **1.7**: stable release
-

Why-Achilles

Achilles development has started at the end of 2012.

At that time, there were already several frameworks to do object mapping with **Cassandra**

1. Hector Object Mapper
2. Kundera
3. Easy Cassandra
4. FireBrand
5. PlayORM
6. Asytanax
7. ...

At that time, most of them have interesting features like fluent API, simple mapping etc. but none of them offers a comprehensive feature sets necessary for a really productive development with **Cassandra**. Furthermore, advanced mapping (with composite and later with clustering primary key) were not available. No support for counters, no runtime settings for TTL, timestamp or consistency level.

I then decided to start a writing my own tool, **Achilles**.

The name itself was chosen quite lamely, I wanted a name related with the Greek mythology and since all characters related to **Cassandra** were chosen (Kundera, Hector, Helenus to name the few), I decided to go with **Achilles** but there is really no relationship with **Cassandra**, at least with regards to the Greek mythology.

So please don't ask me *what is the Achilles heel* ?, I can't really tell.

The first generation (**Achilles 1.x**) were implemented using **Hector**. It was a great effort, but the implementation and design is somehow awkward. It was very hard to rely just on **Hector** API to build up an object mapper and PersistenceManager. The code base was quite huge.

The second generation (**Achilles 2.x**) adds support for **CQL** relying on the brand new Java Driver from Datastax.

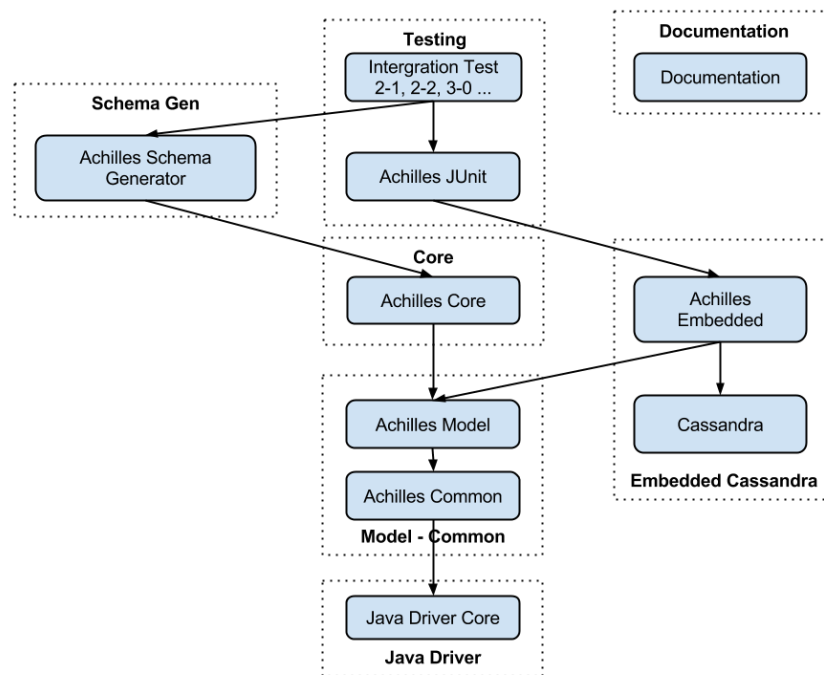
Coding was much easier, a lot of new features have been added (advanced mapping for compound primary key, clustered entity, batch mode ...)

The third generation (**Achilles 3.x**) drops the **Thrift** implementation to move on with full **CQL** and offers support for all new features in **Cassandra 2.x**

The forth generation (**Achilles 4.x**) is a complete rewrite, it uses compile-time code generation to offer greater type-safety as well as a fluent DSL query builder.

The current generation (**Achilles 5.x**) enhances source code generation and adapts the generated source code to your Cassandra version

Architecture



Above is a diagram showing all **Achilles** module dependencies:

1. **Achilles Model**: module containing only annotation and custom types.
This module is depending on the *Java Driver Core*

2. **Achilles Common:** module containing some common classes.
 3. **Achilles Core:** core module containing the framework code plus the source code for code generation. It depends on the *Achilles Common* module
 4. **Achilles Embedded:** module to start an embedded **Cassandra** server and bootstrap **Achilles**. It depends of course on the *Achilles Common* module and *Cassandra* jars
 5. **Achilles JUnit:** module providing JUnit rule to bootstrap an embedded **Cassandra** for testing. It depends on the *Achilles Embedded* and *Achilles Core* modules
 6. **Achilles Schema Generator:** stand alone module used to generate CQL schema as executable jar, to be executed in a command line. It depends on *Achilles Core*
 7. **Integration Test 2-1, 2-2, 3-0, ...:** private module containing all **Achilles** integration tests for each Cassandra version
 8. **Documentation:** module containing a zip of all version documentation (in PDF and HTML formats). This module is a stand-alone
-

IDE-Configuration

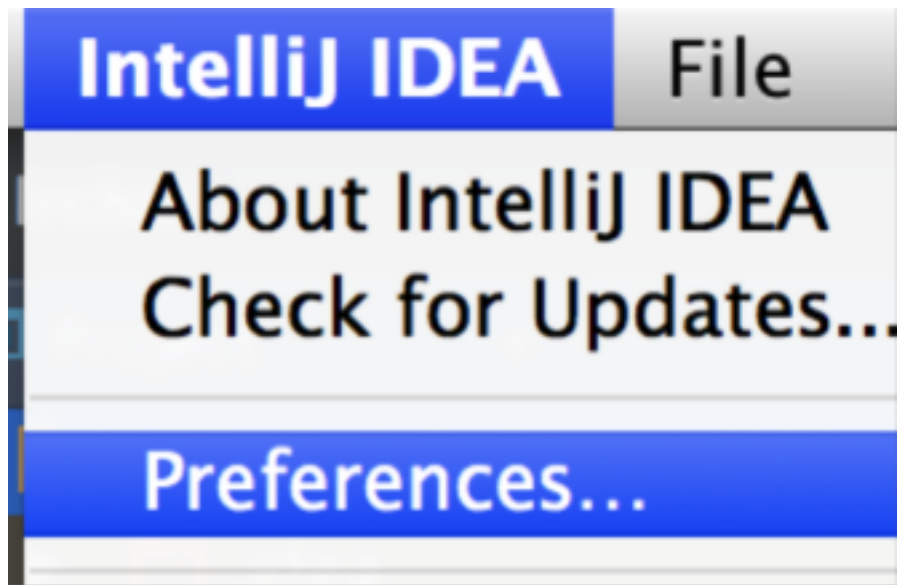
Achilles is using [Annotation Processor](#) to generate source-code at **compile time** by hooking into the compilation lifecycle of the Java compiler.

Therefore, in order to use **Achilles**, you must configure carefully your IDE.

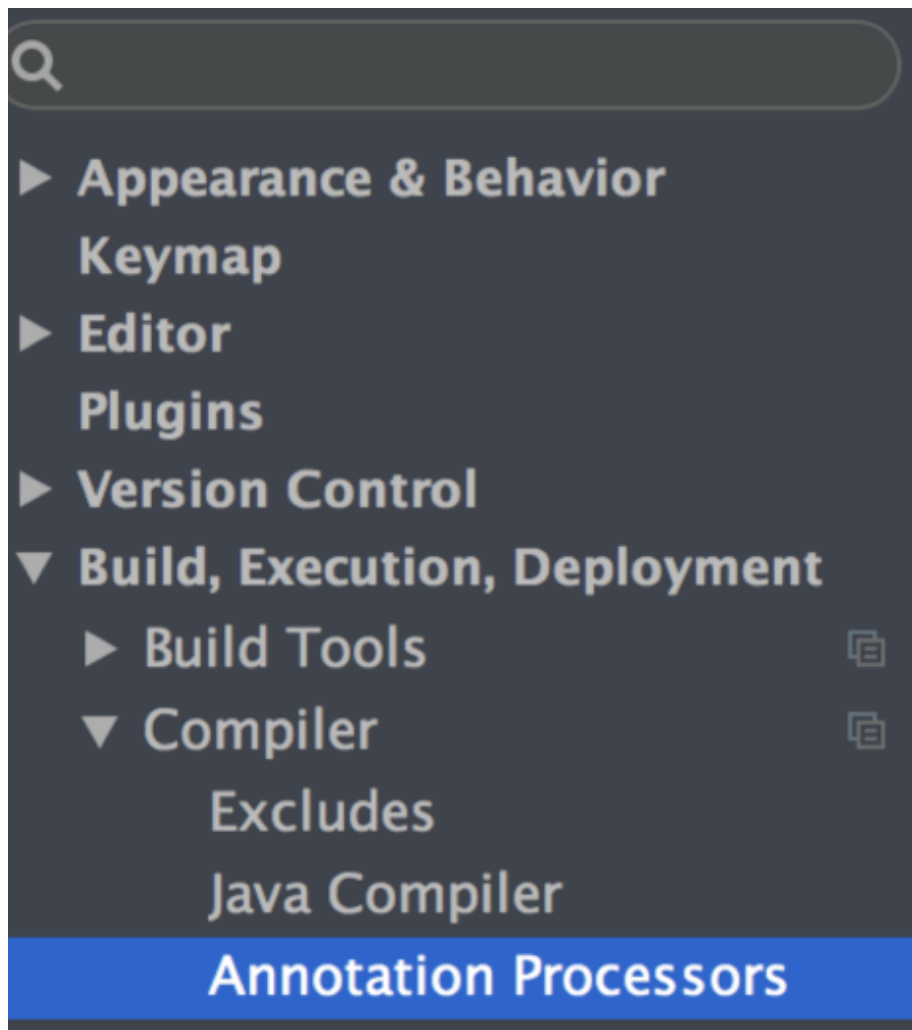
Because of the very minimalist support for annotation processors in **Eclipse**, it is recommended to use [IntelliJ](#) but **Achilles** can work with **Eclipse** too, providing some extensive configuration (see below)

Configuration for IntelliJ

In **IntelliJ** IDE, first go to the **Preferences** menu



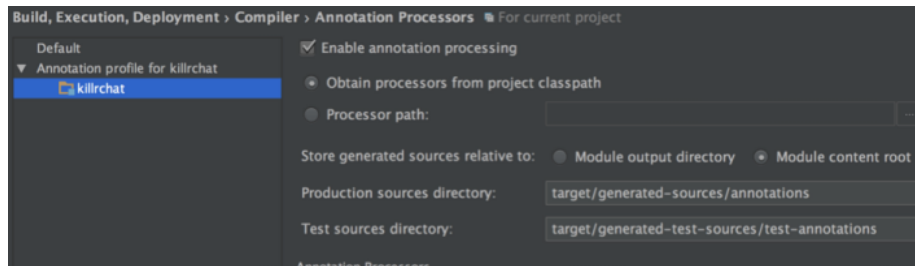
Go to **Build, Execution, Deployment** menu, expand **Compiler** and go to **Annotation Processors**



Select your project in the list on the right pane, select **Enable annotation processing**, choose

Obtain processors from project classpath, choose **Module content root** and set the following values for the directory:

- Production sources directory: **target/generated-sources/annotations**
- Test sources directory: **target/generated-test-sources/test-annotations**



It's done. Now, every time you modify your entity mapping (add a new column, update an annotation, change some types ...), you must trigger a manual rebuild of the project so that **Achilles** can re-generate the updated meta classes.

Configuration for Eclipse

Before configuring your **Eclipse**, you should ensure first that you have the `achilles-core-<version>-shaded.jar` in your **Maven** repository. For this:

- Edit your `pom.xml` and replace

```
<dependency>
  <groupId>info.archinnov</groupId>
  <artifactId>achilles-core</artifactId>
  <version>${achilles.version}</version>
</dependency>
```

by

```
<dependency>
  <groupId>info.archinnov</groupId>
  <artifactId>achilles-core</artifactId>
  <version>${achilles.version}</version>
  <classifier>shaded</classifier>
</dependency>
```

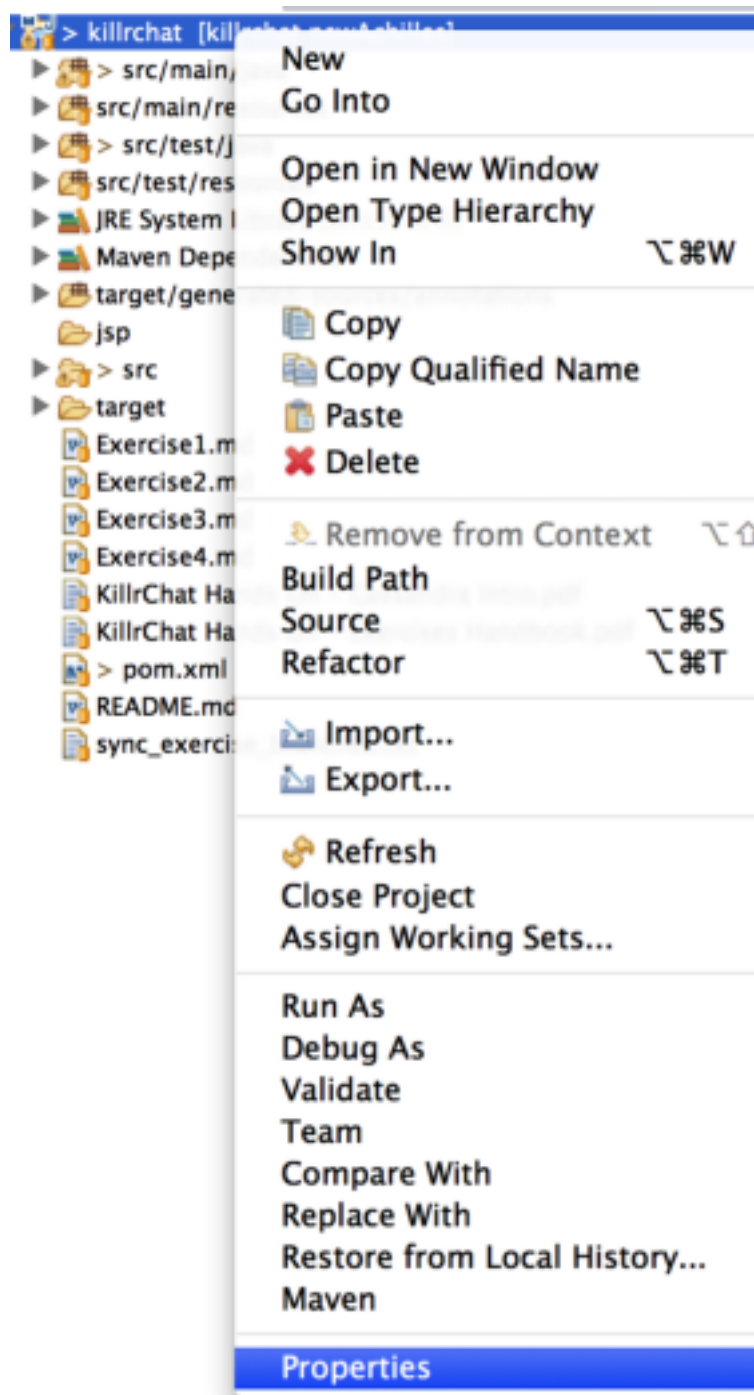
Note that we added the `<classifier>shaded</classifier>`

- Then open a shell terminal, go to your project root folder and execute `mvn dependency:resolve`
- Edit your `pom.xml` again and remove the line `<classifier>shaded</classifier>` to get back to

```
<dependency>
  <groupId>info.archinnov</groupId>
  <artifactId>achilles-core</artifactId>
  <version>${achilles.version}</version>
</dependency>
```

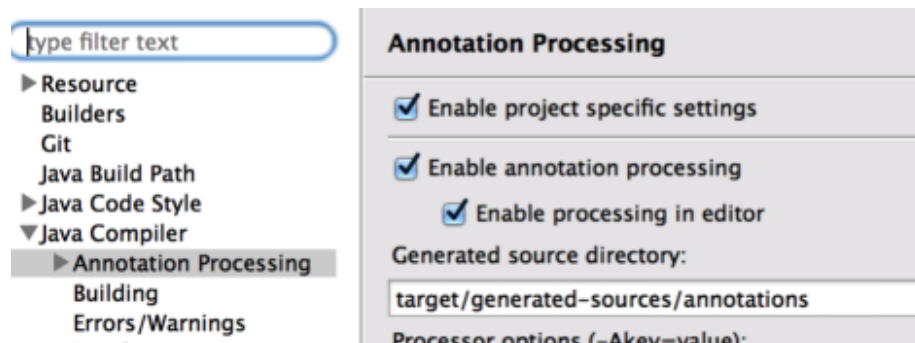
The configuration for **Eclipse** requires more manual configuration because there is no support for automatic annotation processor detection from classpath, contrary to **IntelliJ**.

First, right click on your project and select **Properties**



Expand the **Java Compiler** menu, go to **Annotation Processing** and select **Enable project specific settings**, **Enable annotation processing** (and optionally **Enable processing in editor**).

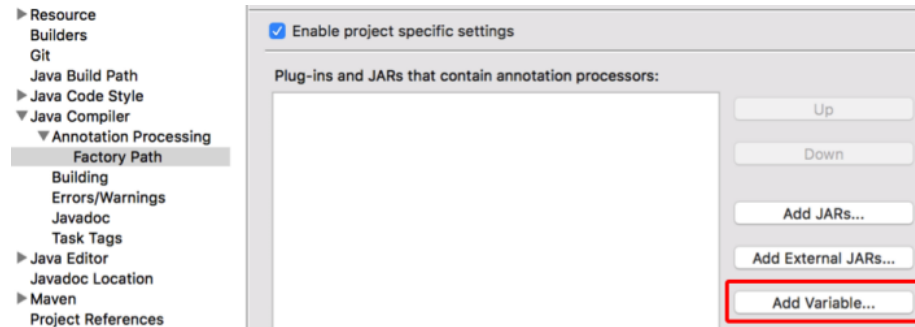
In the **Generated source directory**, put *target/generated-sources/annotations*



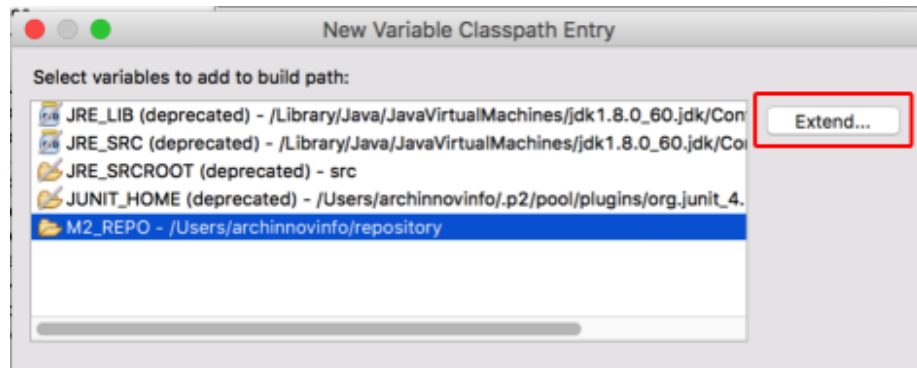
Eclipse will add a new source folder *target/generated-sources/annotations* to your project source layout.

Then, expand the **Annotation Processing** menu and click on **Factory Path**. There you should click on the button

Add Variable ...

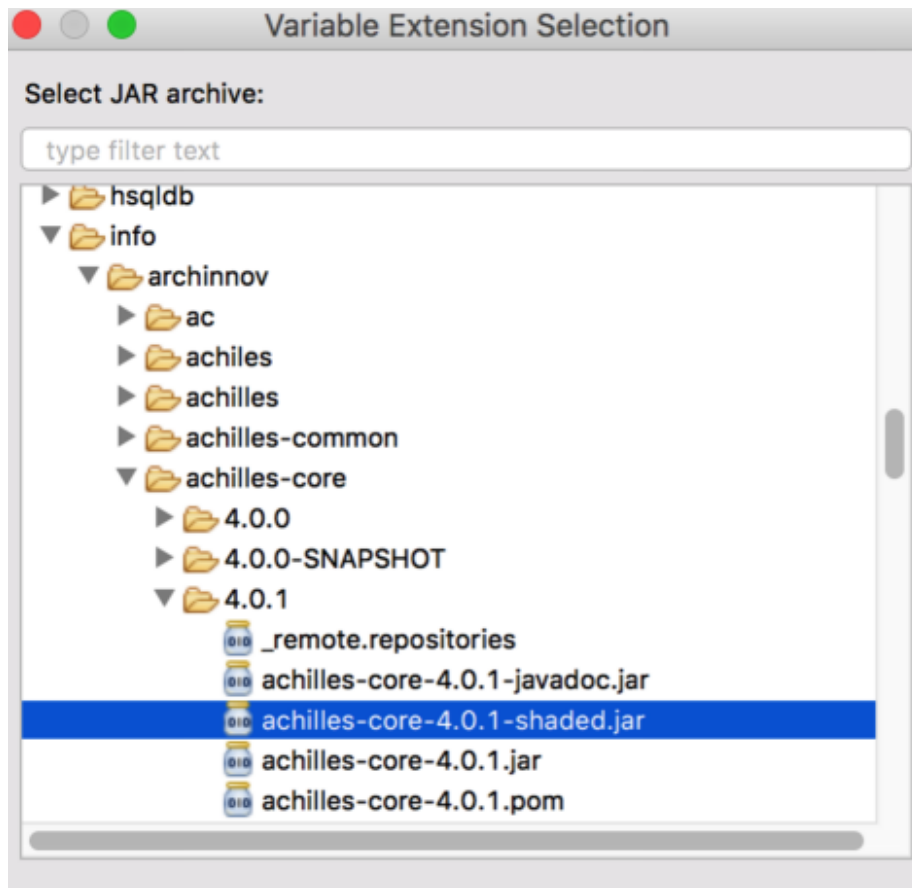


Select the **M2_REPO** variable and click on the button **Extend...**

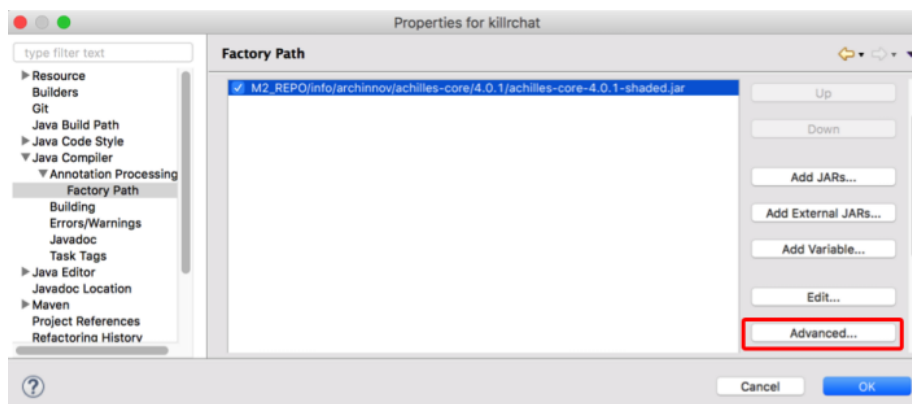


Navigate in the Maven repository folder until you find `info/archinnov/achilles-core/<version>/achilles-` and select it.

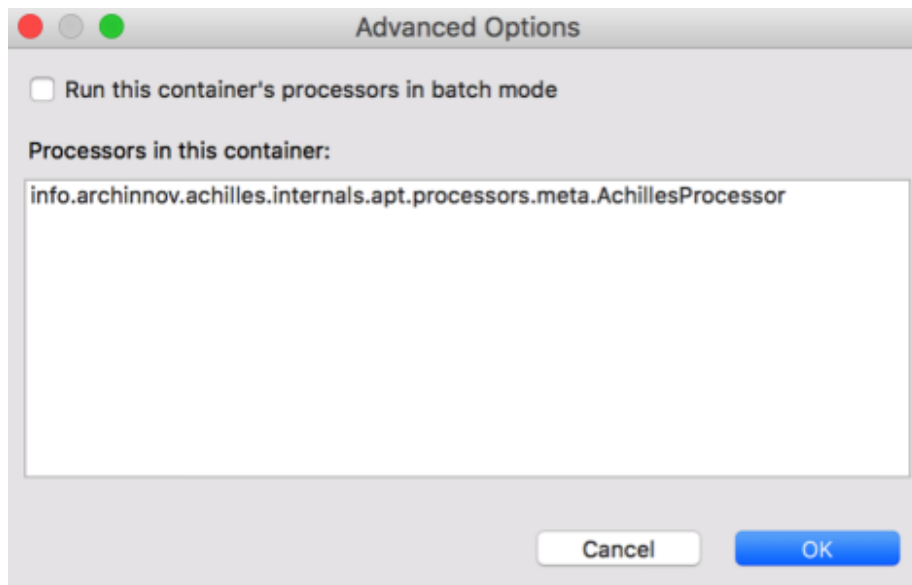
It is important to select the shaded jar and not the normal jar. In the example, the version is `achilles-core-4.0.1-shaded.jar` but you should of course select the correct version that matches the **Achilles** version in your `pom.xml`



Back to the **Factory Path** menu, click on the **Advanced...** button

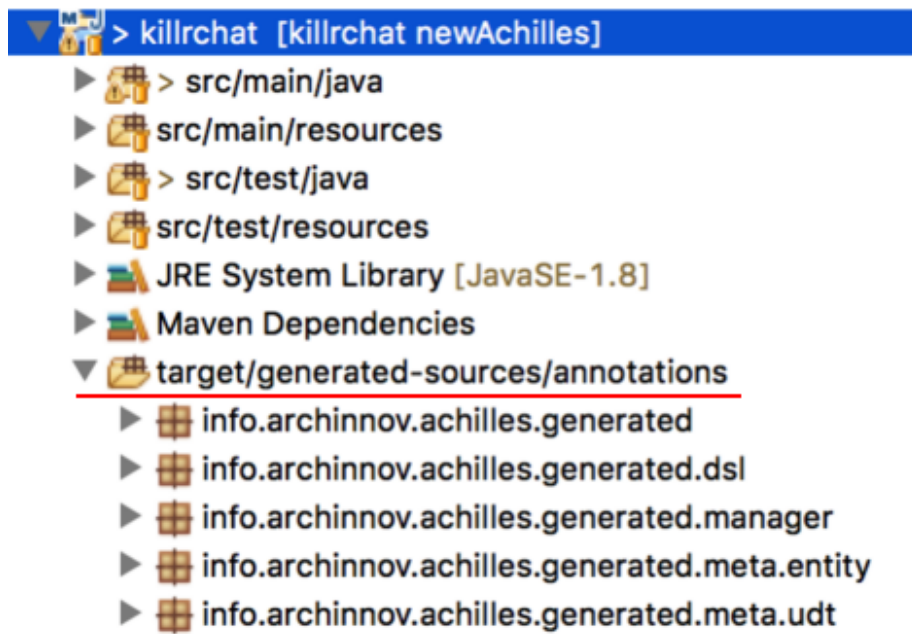


Ensure that you can see `info.archinnov.achilles.internal.spt.processors.AchillesProcessor` in the popup



Validate the changes by clicking on **Apply**. If prompted to rebuild the project, select **Yes**

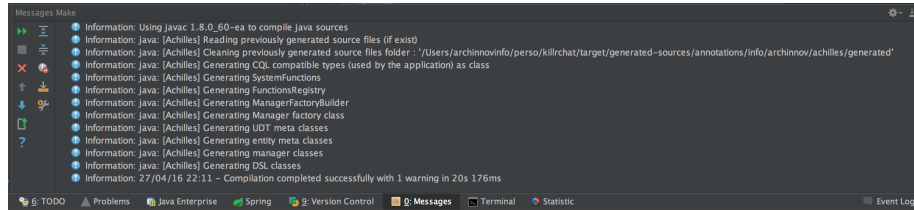
Once the project is rebuilt, you should see new generated classes in the **target/generated-sources/annotations** folder



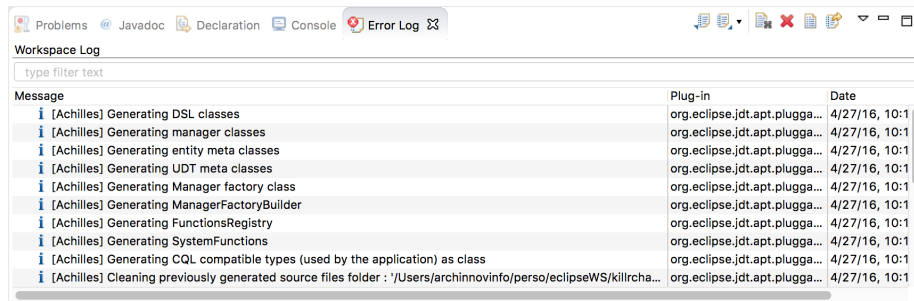
Annotation processor debug messages & issues during compilation

All info/error messages of the **Achilles** processor are displayed in the output console/compilation output console

With **IntelliJ**, processor messages are displayed in the ‘Messages window:



With **Eclipse**, processor messages are displayed in the **Error Log** window:



In some cases, if the processor does not generate the code correctly, you may need to do a clean build with Maven (this is actually the only method that is working 100% in any case). For this open a shell terminal and type `mvn clean compile` then go back to the IDE and refresh your project, you should see the generated code.

Sometimes if you’re making a mistake with the annotations on the entities or your annotations are violating some rules enforced by **Achilles**, you’ll see an explicit compilation error message in the output console/compilation output console.

For example, if you’re creating an User entity without any `@PartitionKey`, the annotation processor will issue an explicit error message:

```
@Table(keyspace = KEYSpace, table = USERS)
public class UserEntity {

    //@PartitionKey ---> ERROR BECAUSE NO PARTITION KEY DEFINED ON THE ENTITY
    private String login;

    @NotEmpty
```

```

@Column
private String pass;

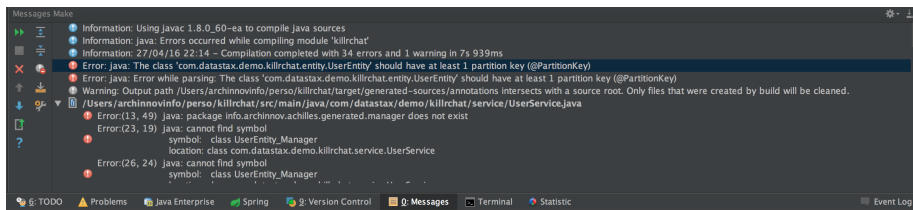
@Column
private String firstname;

@Column
private String lastname;

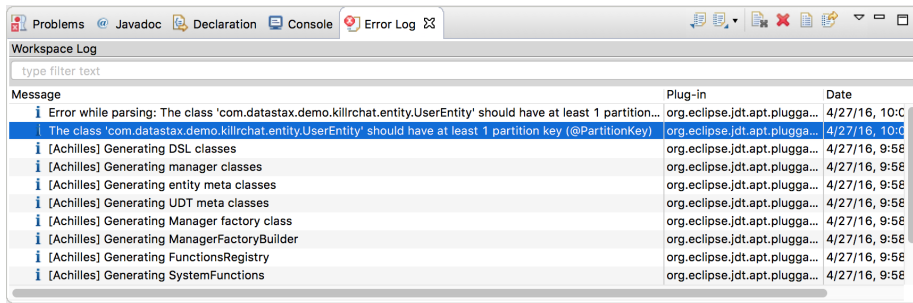
...
}

```

With IntelliJ:



With Eclipse:



Please note that if there is an error with your entity annotations, the processor will stop generating classes so that you'll get a lot of compilation errors because some infrastructure classes like `info.archinnov.achilles.generated.ManagerFactory`, `info.archinnov.achilles.generated.ManagerFactoryBuilder` or all the `info.archinnov.achilles.generated.manager.XXX_Manager` classes are not generated. This is a normal behavior.

After you fix the annotation issue, just rebuild the project and everything should be fine.

5-minutes-Tutorial

First, make sure that you have configured your IDE correctly, otherwise, read the [IDE Configuration](#) section first

Compile time configuration

Before creating any class, we should tell **Achilles** the version of **Cassandra** we're using. For this, just create an empty class/interface annotated with `@CompileTimeConfig`

```
@CompileTimeConfig(cassandraVersion = CassandraVersion.CASSANDRA_3_0_X)
public interface AchillesConfig {

}
```

Here we specify that we are using **Cassandra** version 3.0.X. Depending on the chosen version, **Achilles** will generate appropriate DSL code. You can also specify the [column mapping strategy](#), [naming strategy](#), [insert strategy](#) and the project name.

See [Configuring Achilles at compile time](#) for more details

Bean Mapping

Let's create an **User** bean

```
import info.archinnov.achilles.annotations.Column;
import info.archinnov.achilles.annotations.PartitionKey;

@Table(table="users")
public class User
{
    @PartitionKey
    private Long id;

    @Column
    private String firstname;

    @Column
    private String lastname;

    @Column(name="age_in_year")
```

```

    private Integer age;

    @Column
    private Biography bio;

    @Column
    private List<String> favoriteTags;

    @Column
    private Map<Integer,String> preferences;

    // Getters and setters ...
}

@UDT(keyspace="Test Keyspace", name = "bio_udt")
public class Biography
{
    @Column
    private String birthPlace;

    @Column
    private List<String> diplomas;

    @Column
    private String description;

    //Getters and Setters
}

```

Now compile/rebuild your project so that **Achilles** annotation processor can generate the relevant meta classes.

Bootstrap

Then you need to initialize the **Manager** instance for your **User** entity.

```

Cluster cluster = Cluster.builder()....build();
ManagerFactory managerFactory = ManagerFactoryBuilder
    .builder(cluster)
    .withDefaultKeyspaceName("Test Keyspace")
    .doForceSchemaCreation(true)
    .build();

User_Manager manager = managerFactory.forUser();

```

The `doForceSchemaCreation(true)` parameter asks **Achilles** to force the tables creation if they do not exist.

This flag should be set to **false** most of the time when going to Production. Set it to **true** during development phase to let **Achilles** do the work for you. You can track the schema creation statements by looking at

[DDL Scripts Generation](#)

Once the manager instance is defined, you can just inject it into any of your service

```
@Inject
private User_Manager manager;
...
```

Hint: **Achilles** generated classes follow a very simple convention.
Generated manager class is named as
`<your-entity-classname>_Manager`

Usage

First we create an **User** and insert it

```
User user = new User();
user.setId(1L);
user.setFirstname("DuyHai");
user.setLastname("DOAN");
user.setAge(30);

// Biography
Biography bio = new Biography();
bio.setBirthPlace("VietNam");
bio.setDiplomas(Arrays.asList("Master of Science", "Diplome d'ingenieur"));
bio.setDescription("Yet another framework developer");
user.setBio(bio);

// Favorite Tags
Set<String> tags = new HashSet<String>();
tags.add("computing");
tags.add("java");
tags.add("cassandra");
user.setFavoriteTags(tags);

// Preferences
Map<Integer,String> preferences = new HashMap<Integer,String>();
```

```

preferences.put(1, "FR");
preferences.put(2, "French");
preferences.put(3, "Paris");

user.setPreferences(preferences);

// Save user using the CRUD API
manager
    .crud()
    .insert(user)
    .execute();

```

Then we can find it by id:

```

// Find user by id using the CRUD API
User foundUser = manager
    .crud()
    .findById(1L);

// Now add anew favorite tag to this user by using the DSL API
manager
    .dsl()
    .update()
    .fromBaseTable()
    .favoriteTags().AddTo("achilles")
    .where()
    .id().Eq(1L)
    .execute();

```

Clustered entities

To use **Cassandra** native wide partitions, you can define a **clustered entity**. An entity is **clustered** when it has at least one **clustering column**

```

@Table(table="timeline")
public class Timeline
{
    @PartitionKey
    @Column("user_id")
    private Long userId;

    @ClusteringColumn

```

```

@Column("tweet_id")
private UUID tweetId;

@Column("tweet_content")
private String tweetContent;

public Timeline(Long userId, UUID tweetId, String tweetContent)
{
    this.userId = userId;
    this.tweetId = tweetId;
    this.tweetContent = tweetContent;
}

//Getters & Setters
}

```

To insert new tweets to the user timeline:

```

Long userId = user.getId();

UUID tweetId1 = ...;
UUID tweetId2 = ...;
UUID tweetId3 = ...;
UUID tweetId4 = ...;

// Insert tweets using the CRUD API
manager.crud().insert(new Timeline(userId,tweetId1, "content ...")).execute();
manager.crud().insert(new Timeline(userId,tweetId2, "content ...")).execute();
manager.crud().insert(new Timeline(userId,tweetId3, "content ...")).execute();

// Insert using the CRUD API with options (TTL)
manager
    .crud()
    .insert(new Timeline(userId,tweet4Id, "content ..."))
    .usingTimeToLive(100)
    .execute();

...
...

```

Later, you can retrieved the saved tweets using the generated **DSL API** :

```

UUID uuid2 = ...;
UUID uuid4 = ...;

```

```

// Use the DSL API to get a list of tweets
List<Timeline> foundRange = manager
    .dsl()
    .select()
    .allColumns_FromBaseTable()
    .where()
    .id().Eq(userId)
    .tweetId().Gte_And_Lt(uuid2, uuid4)
    .orderByAscending()
    .limit(10)
    .getList();

assertEquals(foundRange.size(), 2);
assertEquals(foundRange.get(0).getTweetContent(), "content ...");
assertEquals(foundRange.get(1).getTweetContent(), "content ...");

```

And that's it. To have more details on the advanced features, please check the [Documentation](#).

Quick-Reference

Let's consider the following entity for all the below examples:

```

@Table
public class User
{
    @PartitionKey
    private Long userId;

    @Column
    private String firstname;

    @Column
    private String lastname;

    public User(){ }

    public User(Long userId, String firstname, String lastname){...}

    //Getters & Setters
}

```

Inserting an entity

```
manager
    .crud()
    .insert(new User(10L, "John", "DOE"))
    .execute();
```

Inserting an entity as JSON (from Cassandra 2.2.x and after)

```
manager
    .crud()
    .insertJSON("{\"userid\": 10, \"firstname\": \"John\", \"lastname\": \"DOE\"}")
    .execute();
```

Updating a property for an entity

```
manager
    .dsl()
    .update()
    .fromBaseTable()
    .firstName().Set("Jonathan")
    .where()
    .userId().Eq(10L)
    .execute();
```

Updating a property for an entity using JSON (from Cassandra 2.2.x and after)

```
manager
    .dsl()
    .update()
    .fromBaseTable()
    .firstName().Set_FromJSON("\"Jonathan\"")
    .where()
    .userId().Eq_FromJSON("10")
    .execute();
```

Deleting an entity

Deleting an entity instance

```
User user = new User(10L, null, null);
manager
    .crud()
    .delete(user)
    .execute();
```

Deleting an entity by its id

```
manager
    .crud()
    .deleteById(10L)
    .execute();
```

Deleting a whole partition

```
manager
    .crud()
    .deleteByPartitionKeys(10L)
    .execute();
```

Deleting a property for an entity

Using the delete DSL

```
manager
    .dsl()
    .delete()
    .biography()
    .fromBaseTable()
    .where()
    .userId().Eq(10L)
    .execute();
```

Using the delete DSL with JSON (from Cassandra 2.2.x and after)

```
manager
    .dsl()
    .delete()
    .biography()
    .fromBaseTable()
    .where()
    .userId().Eq_FromJSON("10")
    .execute();
```

Using the update DSL

```
manager
    .dsl()
    .update()
    .fromBaseTable()
    .biography.Set(null)
    .where()
    .userId().Eq(10L)
    .execute();
```

Do not forget that in **CQL semantics** setting a column to **null** means deleting it

Finding entities with clustering columns

For all examples in this section, let's consider the following clustered entity representing a tweet line

```
@Table(table = "lines")
public class TweetLine
{

    @PartitionKey
    @Column("user_id")
    private Long userId;

    @ClusteringColumn(1)
    @Enumerated
    private LineType type;

    @ClusteringColumn(value = 2, asc = false) // Sort by descending order
```

```

@TimeUUID //Time uuid type in Cassandra
@Column("tweet_id")
private UUID tweetId;

@Column
private String content;

//Getters & Setters

public static enum LineType
{ USERLINE, TIMELINE, FAVORITELINE, MENTIONLINE}
}

```

Find by partition key and clustering columns

Get the last 10 tweets from timeline, starting from tweet with lastUUID

```

// Generate SELECT * FROM lines WHERE user_id = ? AND (type, tweet_id) < (?,?) AND type
List<TweetLine> tweets = manager
    .dsl()
    .select()
    .allColumns_FromBaseTable()
    .where()
    .userId().Eq(10L)
    .type_And_tweetId().type_And_tweetId_Lt_And_type_Gte(LineType.TIMELINE, lastUUID, L
    .limit(10)
    .getList();

```

Iterating through a large set of entities

Fetch all timeline tweets by batch of 100 tweets

```

Iterator<TweetLine> iterator = manager
    .dsl()
    .select()
    .allColumns_FromBaseTable()
    .where()
    .userId().Eq(10L)
    .type_And_tweetId().type_And_tweetId_Lt_And_type_Gte(LineType.TIMELINE, lastUUID, L
    .withFetchSize(100) // Fetch Size = 100 for each page
    .iterator();

while(iterator.hasNext())

```

```

{
    TweetLine timelineTweet = iterator.next();
    ...
}

```

Deleting entities with clustering columns

Deleting all timeline tweets

```

// Generate DELETE * FROM lines WHERE user_id = ? AND tpe = ?
manager.
    .dsl()
    .delete()
    .allColumns_FromBaseTable()
    .where()
    .userId().Eq(10L)
    .type().Eq(LineType.TIMELINE)
    .execute();

```

Deleting the whole partition using the [CRUD API](#)

```

// Generate DELETE * FROM lines WHERE user_id = ?
manager.
    .crud()
    .deleteByPartitionKeys(10L)
    .execute();

```

Mapping UDT

To declare a JavaBean as UDT

```

@UDT(keyspace = "...", name = "user_udt")
public class UserUDT
{
    @Column
    private Long userId;

    @Column

```

```

    private String firstname;

    @Column
    private String lastname;

    //Getters & Setters
}

```

Then you can re-use the UDT in another entity

```

@Table
public class Tweet
{
    @PartitionKey
    @TimeUUID
    private UUID id

    @Column
    private String content;

    @Column
    @Frozen
    private UserUDT author;

    //Getters & Setters
}

```

Please notice that the **@Frozen** annotation is mandatory for UDT. Unfrozen UDT is only available for Cassandra 3.6 and after

Accessing Meta Classes for Encoding/Decoding functions

Achilles annotation processor will generate, for each entity:

1. An **EntityClassName_Manager** class
2. An **EntityClassName_AchillesMeta** class

The **EntityClassName_AchillesMeta** class provides the following methods for encoding/decoding:

1. `public T createEntityFrom(Row row):` self-explanatory

2. `public ConsistencyLevel readConsistency(Optional<ConsistencyLevel> runtimeConsistency)`: retrieve read consistency from runtime value, static configuration and default consistency configuration in **Achilles**
3. `public ConsistencyLevel writeConsistency(Optional<ConsistencyLevel> runtimeConsistency)`: retrieve write consistency from runtime value, static configuration and default consistency configuration in **Achilles**
4. `public ConsistencyLevel serialConsistency(Optional<ConsistencyLevel> runtimeConsistency)`: retrieve serial consistency from runtime value, static configuration and default consistency configuration in **Achilles**
5. `public InsertStrategy insertStrategy()`: determine insert strategy using static annotation and **Achilles** global configuration
6. `public void triggerInterceptorsForEvent(Event event, T instance)`: trigger all registered interceptors for this entity type on the provided instance, given the event type

Each meta class contains a `public static` field for each property. For example, given the following entity:

```
@Table
public static User {

    @PartitionKey
    private Long userId;

    @Column
    private String firstname;

    @Column
    private String lastname;

    @Column
    private Set<String> favoriteTags;

    ...
}
```

The `User_AchillesMeta` class will expose the following `static property metas`:

1. `User_AchillesMeta.userId`
2. `User_AchillesMeta.firstname`

3. `User_AchillesMeta.lastname`
4. `User_AchillesMeta.favoriteTags`

Each property meta class will expose:

1. `public VALUETO encodeFromJava(VALUEFROM javaValue):` encode the given Java value into CQL-compatible value using the [Codec System](#)
2. `public VALUEFROM decodeFromGettable(GettableData gettableData):` decode the value of the current property from the `GettableData` object. The `GettableData` is the common interface for `com.datastax.driver.core.Row`, `com.datastax.driver.core.UDTValue` and `com.datastax.driver.core.TupleValue`

Querying Cassandra

Native query using the [RAW API](#)

```
final Statement statement = session.newSimpleStatement("SELECT firstname,lastname FROM u
List<TypedMap> rows = userManager
    .raw()
    .nativeQuery(statement, 100)
    .getList();

for(TypedMap row : rows)
{
    String firstname = row.getTyped("firstname");
    String lastname = row.getTyped("lastname");
    ...
}
```

Typed query using the [RAW API](#)

```
final Statement statement = session.newSimpleStatement("SELECT firstname,lastname FROM u

List<User> users = userManager
    .raw()
    .typedQueryForSelect(statement, 100)
    .getList();

for(User user : user)
{
```

```
    ...  
}
```

Asynchronous execution

Asynchronous for the [CRUD API](#)

```
final CompletableFuture<Empty> futureInsert = userManager  
    .crud()  
    .insert(new User(...))  
    .executeAsync();
```

```
final CompletableFuture<User> futureUser = userManager  
    .crud()  
    .findById(10L)  
    .executeAsync();
```

```
final CompletableFuture<Empty> futureDelete = userManager  
    .crud()  
    .deleteById(10L)  
    .executeAsync();
```

Note: **Empty** is a singleton enum to avoid returning a CompletableFuture of **null**

Asynchronous for the [DSL API](#)

```
final CompletableFuture<List<TweetLine>> futureTweets = tweetManager  
    .dsl()  
    .select()  
    .allColumns_FromBaseTable()  
    .where()  
    .userId().Eq(10L)  
    .type().Eq(LineType.TIMELINE)  
    .limit(30)  
    .getListAsync();
```

```

final CompletableFuture<Empty> futureUpdate = userManager
    .dsl()
    .update()
    .fromBaseTable()
    .lastName_Set("new lastname")
    .where()
    .userId().Eq(10L)
    .executeAsync();

final CompletableFuture<Empty> futureDelete = tweetManager
    .dsl()
    .delete()
    .allColumns_FromBaseTable()
    .where()
    .userId().Eq(10L)
    .type().Eq(LineType.TIMELINE)
    .executeAsync();

```

Asynchronous for the [RAW API](#)

```

final Statement statement = session.newSimpleStatement("SELECT firstname,lastname FROM u
CompletableFuture<List<TypedMap>> futureTypedMaps = userManager
    .raw()
    .nativeQuery(statement, 100)
    .getListAsync();

CompletableFuture<List<User>> futureUsers = userManager
    .raw()
    .typedQueryForSelect(statement, 100)
    .getListAsync();

```

Getting the ExecutionInfo back

For the [CRUD API](#)

```

final ExecutionInfo executionInfo = userManager
    .crud()
    .insert(new User(...))
    .executeWithStats();

```

```

final ExecutionInfo executionInfo = userManager
    .crud()
    .deleteById(10L)
    .executeWithStats();

final Tuple2<User, ExecutionInfo> resultWithExecInfo = userManager
    .crud()
    .findById(10L)
    .getWithStats();

```

For the [DSL API](#)

```

final Tuple2<List<TweetLine>, ExecutionInfo> tweetsWithStats = tweetManager
    .dsl()
    .select()
    .allColumns_FromBaseTable()
    .where()
    .userId().Eq(10L)
    .type().Eq(LineType.TIMELINE)
    .limit(30)
    .getListWithStats();

final ExecutionInfo executionInfo = userManager
    .dsl()
    .update()
    .fromBaseTable()
    .lastname_Set("new lastname")
    .where()
    .userId().Eq(10L)
    .executeWithStats();

final ExecutionInfo executionInfo = tweetManager
    .dsl()
    .delete()
    .allColumns_FromBaseTable()
    .where()
    .userId().Eq(10L)
    .type().Eq(LineType.TIMELINE)
    .executeWithStats();

```

For the [RAW API](#)

```
final Statement statement = session.newSimpleStatement("SELECT firstname,lastname FROM u
Tuple2<List<TypedMap>, ExecutionInfo> typedMapsWithStats = userManager
    .raw()
    .nativeQuery(statement, 100)
    .getListWithStats();

Tuple2<List<User>, ExecutionInfo> usersWithStats = userManager
    .raw()
    .typedQueryForSelect(statement, 100)
    .getListWithStats();
```

Working with consistency level

Defining consistency statically

```
@Table
@Consistency(read=ConsistencyLevel.ONE, write=ConsistencyLevel.QUORUM, serial = Consiste
public class User
{
    ...
}
```

Setting consistency level at runtime

```
userManager
    .crud()
    ...
    .withConsistencyLevel(ConsistencyLevel.QUORUM)
    ...

userManager
    .dsl()
    ...
    .withConsistencyLevel(ConsistencyLevel.QUORUM)
    ...
```

Working with TTL

Defining TTL statically

```
@Table
@TTL(1000)
public class User
{
    ...
}
```

Setting TTL at runtime

```
userManager
    .crud()
    .insert(...)
    ...
    .usingTimeToLive(10)
    ...

userManager
    .dsl()
    .update()
    ...
    .usingTimeToLive(10)
    ...
```

Working with Timestamp

```
userManager
    .crud()
    .insert(...)
    ...
    .usingTimestamp(new Date().getTime())
    ...

userManager
    .crud()
    .deleteById(...)
    ...
    .usingTimestamp(new Date().getTime())
    ...
```

```

userManager
    .dsl()
    .update()
    ...
    .usingTimestamp(new Date().getTime())
    ...

userManager
    .dsl()
    .delete()
    ...
    .usingTimestamp(new Date().getTime())
    ...

```

Working with Lightweight Transaction

API

```

userManager
    .crud()
    .insert(...)
    ...
    .ifNotExists()
    ...

userManager
    .crud()
    .deleteById(...)
    ...
    .ifExists()
    ...

userManager
    .dsl()
    .update()
    ...
    .ifExists()
    ...

userManager
    .dsl()
    .update()
    .fromBaseTable()

```

```

        .firstName().Set("new firstname")
        ...
        .if_Firstname().Eq("previous_firstname")
        ...

userManager
    .dsl()
    .delete()
    ...
    .ifExists()
    ...

userManager
    .dsl()
    .delete()
    ...
    .if_Firstname().Eq("previous_firstname")
    ...

```

LWT Result Listener

To have tighter control on LWT updates, inserts or deletes, **Achilles** lets you inject a listener for LWT operations result.

```

LWTResultListener lwtListener = new LWTResultListener() {

    @Override
    public void onSuccess() {
        // Do something on success
        // Default method does NOTHING
    }

    @Override
    public void onError(LWTResult lwtResult) {

        //Get type of LWT operation that fails
        LWTResult.Operation operation = lwtResult.operation();

        // Print out current values
        TypedMap currentValues = lwtResult.currentValues();
        for(Entry<String,Object> entry: currentValues.entrySet()) {
            System.out.println(String.format("%s = %s",entry.getKey(), entry.getValue()))
        }
    }
}

```

```

};

userManager
    .crud()
    .insert(new User(...))
    .ifNotExists()
    .withLWTResultListener(lwtListener)
    .execute();

//OR

userManager
    .crud()
    .insert(new User(...))
    .ifNotExists()
    .withLWTResultListener(lwtResult -> logger.error("Error : " + lwtResult))
    .execute();

```

Using counter type

```

@Table(table = "retweet_count")
public class Retweets {

    @PartitionKey
    @Column("user_id")
    private Long userId;

    @ClusteringColumn(1)
    @Enumerated
    private LineType type;

    @ClusteringColumn(value = 2, asc = false)
    @TimeUUID
    @Column("tweet_id")
    private UUID tweetId;

    @Counter
    @Column("direct_retweets")
    private Long directRetweets;

    @Counter
    @Column("total_retweets")
    private Long totlRetweets;
}

```

```

        //Getters & Setters
    }

```

Once the entity mapping is defined the **CRUD API** for counter tables is restricted to `deleteById()` and `deleteByPartitionKeys()` methods (no `insert()`).

Using materialized views (from Cassandra 3.0.X and after)

To declare a materialized view, use the

`** at MaterializedView**` annotation:

```

@MaterializedView(baseEntity = EntitySensor.class, view = "sensor_by_type")
public class ViewSensorByType {

    @PartitionKey
    @Enumerated
    private SensorType type;

    @ClusteringColumn(1)
    private Long sensorId;

    @ClusteringColumn(2)
    private Long date;

    @Column
    private Double value;

    ...
    //Getters & setters
}

@Table(table = "sensor")
public class EntitySensor {

    @PartitionKey
    private Long sensorId;

    @ClusteringColumn
    private Long date;

    @Enumerated
    @Column

```

```

    private SensorType type;

    @Column
    private Double value;

    ...
    //Getters & setters
}

```

The view should reference a base table using the attribute **baseEntity**. It should also re-use the same columns that belong to the base table primary key, possibly in a different order.

Achilles will generate only SELECT APIs for those views, UPDATE and DELETE operations are not possible.

See [Materialized View Mapping](#) for more details

Function mapping (from Cassandra 2.2.X and after)

You can declare the **signature** of your functions in a class/interface so that **Achilles** can generate type-safe API for you to be able to invoke them in the Select DSL API.

For this, use the **@FunctionRegistry** annotation:

For more details, see [Functions Mapping](#)

```

@FunctionRegistry
public interface MyFunctionRegistry {

    Long convertToLong(String longValue);

}

```

Please note that you'll need to declare your user-defined function by yourself with **Cassandra**, **Achilles** only takes care of the function signature for the code generation, not the function declaration.

Simple object mapping

You can use the **Manager** object for simple object mapping

```

// Execution of custom query
Row row = session.execute(...).one();

```

```
User user = userManager.mapFromRow(row);
```

Getting native Session and Cluster object

You can retrieve the native **Session** and **Cluster** object from the **Manager**

```
Session session = userManager.getNativeSession();  
  
Cluster cluster = userManager.getNativeCluster();
```

Generating bound statements/query string from the APIs

Generating `com.datastax.driver.core.BoundStatement`

```
BoundStatement bs = userManager  
    .crud()  
    ...  
    .generateAndGetBoundStatement();  
  
BoundStatement bs = userManager  
    .dsl()  
    ...  
    .generateAndGetBoundStatement();
```

Generating **query string**

```
String statement = userManager  
    .crud()  
    ...  
    .getStatementAsString();  
  
String statement = userManager  
    .dsl()  
    ...  
    .getStatementAsString();
```

Extract bound values from the APIs

Extract **raw** bound values

```
List<Object> boundValues = userManager
    .crud()
    ...
    .getBoundValues();

List<Object> boundValues = userManager
    .dsl()
    ...
    .getBoundValues();
```

Extract **encoded** bound values. The encoding relies on **Achilles** [Codec System](#)

```
List<Object> encodedBoundValues = userManager
    .crud()
    ...
    .getEncodedBoundValues();

List<Object> encodedBoundValues = userManager
    .dsl()
    ...
    .getEncodedBoundValues();
```

Injecting schema name at runtime

Normally you define the keyspace/table name statically using the **@Table** annotation.

However, in a multi-tenant environment, the keyspace/table name is not known ahead of time but only during runtime. For this, **Achilles** defines an interface **SchemaNameProvider**:

```
public interface SchemaNameProvider {

    /**
     * Provide keyspace name for entity class
     */
    <T> String keyspaceFor(Class<T> entityClass);
```

```

    /**
     * Provide table name for entity class
     */
    <T> String tableNameFor(Class<T> entityClass);
}

```

You can implement this interface and inject the schema name provider at runtime.

Both **CRUD API** and

DSL API accept dynamic binding of schema name:

```

final SchemaNameProvider dynamicProvider = ...;

userManager
    .crud()
    ...
    .withSchemaNameProvider(dynamicProvider)
    .execute();

userManager
    .dsl()
    .select()
    ...
    .from(dynamicProvider)
    .where()
    ...

userManager
    .dsl()
    .update()
    .from(dynamicProvider)
    ...
    .where()
    ...

userManager
    .dsl()
    .delete()
    ...
    .from(dynamicProvider)
    ...
    .where()
    ...

```

Generating DDL scripts

Using DML logs

Sometime it is nice to let **Achilles** generate for you the CREATE TABLE script. To do that:

- Set up unit test using [Achilles JUnit test resource](#)
- Activate the DDL logging by setting **DEBUG** level on the logger `ACHILLES_DDL_SCRIPT`

```
<logger name="ACHILLES_DDL_SCRIPT">  
  <level value="DEBUG" />  
</logger>
```

Using the SchemaGenerator

Achilles provides a module **achilles-schema-generator** to help you generate CQL schema scripts for your entities. More details [here](#)

Generating DML statements

To debug **Achilles** behavior, you can enable DML statements logging by setting **DEBUG** level on the logger `ACHILLES_DML_STATEMENT`

```
<logger name="ACHILLES_DML_STATEMENT">  
  <level value="DEBUG" />  
</logger>
```

Advanced-Tutorial:-KillrChat

For an example of an advanced usage of **Achilles**, you can check out the [KillrChat](#) application.

It is using almost all **Achilles** features

Unit-testing

Test Resources

For TDD lovers, **Achilles** comes along with a **JUnit** rule to start an embedded Cassandra server in memory and bootstrap the framework.

The rule extends JUnit `ExternalResource` and exposes the following methods:

```
public class AchillesTestResource
{
    public Session getNativeSession();

    public ScriptExecutor getScriptExecutor();

    public ManagerFactory getManagerFactory();
}
```

To obtain a resource, you should use the `AchillesTestResourceBuilder` which exposes the following methods

1. `withScript(scriptLocation)`: provide a CQL script file, reachable in the **classpath**, to be executed upon the Embedded Cassandra server startup. This method is useful to create the appropriate schema for unit testing
2. `createAndUseKeyspace(String keyspaceName)`: create the given keyspace (`SimpleStrategy`) upon server startup and connect to it.
3. `entityClassesToTruncate(Class<?>...entityClassesToTruncate)`: truncate tables used by given entity classes
4. `tablesToTruncate(String...tablesToTruncate)`: truncate given tables
5. `truncateBeforeTest()`: self-explanatory
6. `truncateAfterTest()`: self-explanatory
7. `truncateBeforeAndAfterTest()`: self-explanatory. **DEFAULT** value
8. `public ManagerFactory build(BiFunction<Cluster, StatementsCache, ManagerFactory> managerFactoryBuilder)`: provide a bi-lambda function to build the manager factory given the `com.datastax.driver.core.Cluster` object and **Achilles** shared `StatementsCache`

The `StatementsCache` is provided to avoid re-creating a new instance of `StatementsCache` for each test class and re-preparing the same statements.

Usage

Below is a sample code to demonstrate how to use **Achilles** rules

```
public class MyTest
{
    @Rule
    public AchillesTestResource<ManagerFactory> resource = AchillesTestResourceBuilder
        .forJUnit()
        .withScript("script1.cql")
        .withScript("script2.cql")
        .tablesToTruncate("users", "tweet_lines") // entityClassesToTruncate(User.class, Tweet.class)
        .createAndUseKeyspace("unit_test")
        .
        .
        .build((cluster, statementsCache) -> ManagerFactoryBuilder
            .builder(cluster)
            .doForceSchemaCreation(true)
            .withStatementCache(statementsCache) //MANDATORY
            .withDefaultKeyspaceName("achilles_embedded")
            .build()
        );

    private User_Manager userManager = resource.getManagerFactory().forUser();

    @Test
    public void should_test_something() throws Exception
    {
        ...
        userManager
            .crud()
            .insert(...)
        ...
    }
}
```

In the log file, if the logger `ACHILLES_DML_STATEMENT` is activated, we can see:

```
DEBUG ACHILLES_DML_STATEMENT - TRUNCATE users
```

```

DEBUG ACHILLES_DML_STATEMENT - Query ID b3bc7f3c-d232-4d43-8014-96d08b90804c : [INSERT INTO
DEBUG ACHILLES_DML_STATEMENT -   Java bound values : [271273866878694400, doanduyhai, DuyHa
DEBUG ACHILLES_DML_STATEMENT -   Encoded bound values : [271273866878694400, doanduyhai, Duy
DEBUG ACHILLES_DML_STATEMENT -   ResultSet[ exhausted: true, Columns[]]
DEBUG ACHILLES_DML_STATEMENT - Query ID b3bc7f3c-d232-4d43-8014-96d08b90804c results :

DEBUG ACHILLES_DML_STATEMENT - TRUNCATE users

```

Please notice the `TRUNCATE users` query issued **before** and **after** the test for clean up.

Native session

By calling `getNativeSession()` upon the resource, you can access directly the `com.datastax.driver.core.Session` object of the native Java driver. It may be useful for triggering manual CQL statement to assert the test results

Script executor

For some tests, we may insert test fixtures before executing the test itself. It can be done programmatically using the native `session` object but it can become quickly cumbersome if you have many data to insert.

For those scenarios, **Achilles** exposes a `ScriptExecutor` than will handle CQL script execution for you.

The script executor also support BATCH statements in your CQL scripts

Bootstrap from `AchillesTestResource` Example:

```

@Rule
public AchillesTestResource resource = ...

private Session session = resource.getNativeSession();

private ScriptExecutor scriptExecutor = resource.getScriptExecutor();

@Test
public void should_create_user() throws Exception {
    //Given

```

```

        scriptExecutor.executeScript("insert_user.cql");

        //When
        ...

        //Then
        ...
    }

```

In the above example, the script executor takes a string as input, pointing to a CQL script file on the classpath.

As per Maven convention, it is recommended to put all your test CQL script into the `src/test/resources` folder

Standalone instance You can also create a **stand-alone ScriptExecutor instance**. The constructor of the `ScriptExecutor` class only requires a `com.datastax.driver.core.Session` object

```

final Session session = ...;

final ScriptExecutor executor = new ScriptExecutor(session);

executor.executeScript("...");

```

Script Templates You can also make script templates and inject bound values at runtime. Let's say we have the following `insert_user.cql` script:

```

INSERT INTO users(id, login, firstname, lastname)
VALUES(${id}, ${login}, ${firstname}, ${lastname});

```

All the values are parameterized so that at runtime, we can call this template and inject the values:

```

@Test
public void should_create_user() throws Exception {
    //Given
    Map<String, Object> values = new HashMap<>();
    values.put("id", RandomUtils.nextLong(0L, Long.MAX_VALUE));
    values.put("login", "doanduyhai");
    values.put("firstname", "DuyHai");
    values.put("lastname", "DOAN");

    //Inject the values into the template to generate the final CQL script

```

```

        scriptExecutor.executeScriptTemplate("insert_user.cql", values);

        //When
        ...

        //Then
        ...
    }

```

The script executor exposes other useful methods to let you execute arbitrary CQL statement as plain text or as Statement object:

```

/**
 * Execute plain text CQL statement and return a native ResultSet
 */
public ResultSet execute(String statement);

/**
 * Execute a CQL statement and return a native ResultSet
 */
public ResultSet execute(Statement statement);

/**
 * Execute asynchronously a plain text CQL statement and return a future of native ResultSet
 */
public AchillesFuture<ResultSet> executeAsync(String statement);

/**
 * Execute asynchronously a CQL statement and return a future of native ResultSet
 */
public AchillesFuture<ResultSet> executeAsync(Statement statement)

```

CQL-embedded-cassandra-server

Achilles exposes a simple builder to start an embedded **Cassandra** server and optionally bootstrap the framework.

Such feature is already available for JUnit testing using the [AchillesTestResourceBuilder](#) discussed previously.

However there is still a need to start an embedded Cassandra outside of testing context and more importantly, to connect to several keyspaces.

To use the embedded Cassandra server, you must pull the following dependency:

```
<dependency>
  <groupId>info.archinno</groupId>
  <artifactId>achilles-embedded</artifactId>
  <version>${achilles.version}</version>
</dependency>
```

The builder API:

```
CassandraEmbeddedServerBuilder
    .builder()
    .withClusterName("Test Cluster")
    .withCommitLogFolder("/home/user/cassandra/commitlog")
    .withConcurrentReads(16)
    .withConcurrentWrites(16)
    .withCQLPort(9042)
    .withDataFolder("/home/user/cassandra/data")
    .withDurableWrite(true)
    .withHintsFolder("/home/user/cassandra/hints")
    .withKeyspaceName("achilles_test")
    .withParams(new TypedMap(...))
    .withSavedCachesFolder("/home/user/cassandra/saved_caches")
    .withScript("src/test/resources/startup_script.cql")
    .withScriptTemplate("src/test/resources/startup_script_template.cql", values)
    .withStoragePort(7990)
    .withStorageSSLPort(7999)
    .withThriftPort(9160)
    .cleanDataFilesAtStartup(true)
    .buildNativeSession() || buildNativeCluster() || buildServer();
```

Most of the parameters are optional, below are the default values:

- clusterName: Achilles Embedded Cassandra Cluster
- keyspaceName: achilles_embedded
- dataFolder: target/cassandra_embedded/data
- commitLogFolder: target/cassandra_embedded/commitlog
- savedCachesFolder: target/cassandra_embedded/saved_caches
- cleanDataFilesAtStartup: true
- concurrentReads: number of concurrent reads
- concurrentWrites: number of concurrent writes
- cqlPort: randomized at runtime
- thriftPort: randomized at runtime

- `storagePort`: randomized at runtime
- `storageSSLPort`: randomized at runtime
- `durableWrite`: `true`
- `script`: startup CQL script to be executed.
You can specify many scripts by calling the method `withScript()` multiple times
- `script template`: startup CQL script template to be executed given provided values.
You can specify many script templates by calling the method `withScriptTemplate()` multiple times

The builder can return:

1. either a `com.datastax.driver.core.Cluster` object
2. or a `com.datastax.driver.core.Session` object
3. or an **Achilles** `CassandraEmbeddedServerBuilder` object

There will be **at most one embedded Cassandra server** started per JVM (more precisely, per classloader) event if many `CassandraEmbeddedServer` are built. But because the **keyspace name** is provided each time, the builder will create **as many keyspaces as specified**.

Achilles-Versioning

Until now, versioning for **Achilles** has been quite chaotic. From now on, the strategy for version change is defined below.

Versioning rule

Achilles version is defined using 3 numbers: **major. medium. minor**

- **major**: a change in major version is triggered by
 - a migration to a new **Cassandra** major version (**2.1** → **3.0** for example)
 - an important breaking API change
- **medium**: a change in medium version is triggered by

- an introduction of a new feature/API
- a removal of deprecated API/feature
- a migration to new **Cassandra medium** version (**2.0.x** → **2.1** for example)
- **minor**: a change in minor version is triggered by
 - bug fixes
 - internal code refactoring without impact on the API or client behavior (except for deprecated features, see below)
 - migration to new **Cassandra minor** version (**2.0.10** → **2.0.11** for example)
 - migration to new Java Driver version

API deprecation rule

Before being removed, an API/feature/type/annotation will be marked as **@Deprecated** and will co-exist with the new/replacement API/feature/type/annotation for **at least** 3 minor versions.

If you're using an API/feature/type/annotation marked as **deprecated**, expect it to be removed:

- on the next major version
- on the next medium version
- on the next 4th minor versions counting from the current

Any deprecation will be properly documented in the [Changelog](#) and the Java Doc redirects to the appropriate new API/feature/type/annotation

Compile-Time-Config

When using **Achilles** you should be aware of the fact that the framework invoked at **compile time** and at **runtime**.

At **compile time** **Achilles** will parse your source code to generate additional source code. To configure **Achilles** at compile time, you can use the **@CompileTimeConfig** annotation on a class or an interface

The interface/class can (and should) be **empty**. What interests us here is the attributes set on the annotation

```

@CompileTimeConfig(cassandraVersion = CassandraVersion.CASSANDRA_3_0_X,
    columnMappingStrategy = ColumnMappingStrategy.IMPLICIT,
    namingStrategy = NamingStrategy.SNAKE_CASE,
    insertStrategy = InsertStrategy.ALL_FIELDS,
    projectName = "Project1"
public interface AchillesCompileTimeConfig {

}

```

What you can specify as configuration at compile time are:

- the cassandra version. Default = **CassandraVersion.CASSANDRA_3_0_X**
- the **column mapping strategy**. Default = **ColumnMappingStrategy.EXPLICIT**
- the **naming strategy**. Default = **NamingStrategy.LOWER_CASE**
- the **insert strategy**. Default = **InsertStrategy.ALL_FIELDS**
- the project name. Default = “”

Please refer to the appropriate section for each of those parameters.

Specifying the **Cassandra** version will help **Achilles** unlock and generate new features that are only supported by this version and after.

For example, materialized views are not available before **CassandraVersion.CASSANDRA_3_0_X**

Below is the Cassandra version feature matrix:

| Cassandra version | Supported features | Related JIRA(s) |
|-------------------|-----------------------|-----------------|
| 2.1.X | All existing features | N/A |

| Cassandra version | Supported features | Related JIRA(s) |
|-------------------|--|--|
| 2.2.X | <ul style="list-style-type: none"> • JSON Syntax • User Defined Functions • User Defined Aggregates | <ul style="list-style-type: none"> • CASSANDRA-7970 • CASSANDRA-7395, CASSANDRA-7526, CASSANDRA-7562, CASSANDRA-7740, CASSANDRA-7781, CASSANDRA-7929, CASSANDRA-7924, CASSANDRA-7812, CASSANDRA-8063, CASSANDRA-7813, CASSANDRA-7708 • CASSANDRA-8053 |

| Cassandra version | Supported features | Related JIRA(s) |
|-------------------|---|--|
| 3.0.X | <ul style="list-style-type: none"> • Materialized Views • Support for IN restrictions on any partition key component or clustering key as well as support for EQ and IN multicolumn restrictions has been added to UPDATE and DELETE statement • Support for single-column and multi-column slice restrictions (>, >=, <= and <) has been added to DELETE statements | <ul style="list-style-type: none"> • CASSANDRA-6477 |
| 3.1 | Nothing | N/A |
| 3.2 | Add support for type casting in selection clause | CASSANDRA-10310 |

And below is the impact of each new feature on **Achilles**-generated code:

Cassandra Feature

Achilles generated code

JSON Syntax

- `manager.crud().insertJson(String json)`
- `manager.dsl()....where().xxx().Eq_FromJSON()`
- `manager.dsl().select().allColumnsAsJSON_FromBaseTable()....where()....getJSON()`
- `manager.dsl().update()....xxx().Set_FromJSON()`
- `manager.dsl().update()....if_xxx().Eq_FromJSON()`
- `manager.dsl().delete()....if_xxx().Eq_FromJSON()`

User Defined Function/User Defined Aggregates

- @FunctionRegistry is allowed
- *info.archinnov.achilles.generated.function.FunctionsRegistry* generated class

Materialized Views

- @MaterializedView is allowed

Support for IN restrictions on clustering columns for UPDATE/DELETE

- `manager.dsl().update()...where()...xxx().IN(...)`
- `manager.dsl().delete()...where()...xxx().IN(...)`

Support for multi-column slice restrictions (>, >=, <= and <) for DELETE

- `manager.dsl().delete()...where()...xxx().Gt(...)`
- `manager.dsl().delete()...where()...xxx().Gt_And_Lt(...)`

Support for type casting in selection clause

- `manager.dsl().select().function(SystemFunctions.castAsxxx()...)`

Last but not least, the **projectName()** attribute is crucial for those who are using **Achilles** for multiple projects.

For more details see [Multi-Project support](#)

Configuration-Parameters

Parameters

Achilles comes with some configuration parameters in the class **info.archinnov.achilles.configuration.Co**

Entity Management

- **MANAGED_ENTITIES** (OPTIONAL): list of entities to be managed by Achilles.

Example: *my.project.entity,another.project.entity*

DDL

- **FORCE_TABLE_CREATION** (OPTIONAL): create missing column families for entities if they are not found. **Default = 'false'**.

If set to **false** and no table is found for any entity, **Achilles** will raise an **AchillesInvalidTableException**

Native Session

- **NATIVE_SESSION** (OPTIONAL): provide a pre-configured Java driver session instead of letting **Achilles** build it

Keyspace name

- **KEYSPACE_NAME** (OPTIONAL): provide the keyspace name to connect to.
If not provided, **Achilles** will use the keyspace name defined on each entity using the **keyspace** attribute of the **@Table** annotation.

If you do not provide the keyspace name in configuration and your entity does not define any keyspace name statically on the **@Table** annotation, **Achilles** will raise an exception at runtime

JSON Serialization

- **JACKSON_MAPPER_FACTORY** (OPTIONAL): an implementation of the *info.archinnov.achilles.json.JacksonMapperFactory* interface to build custom Jackson **ObjectMapper** based on entity class
- **JACKSON_MAPPER** (OPTIONAL): default Jackson **ObjectMapper** to use for serializing entities

If both **JACKSON_MAPPER_FACTORY** and **JACKSON_MAPPER** parameters are provided, **Achilles** will ignore the **JACKSON_MAPPER** parameter and use **JACKSON_MAPPER_FACTORY**

If none is provided, **Achilles** will use a default Jackson **ObjectMapper** with the following configuration:

1. MapperFeature.SORT_PROPERTIES_ALPHABETICALLY = true
2. SerializationInclusion = JsonInclude.Include.NON_NULL
3. DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES = false
4. AnnotationIntrospector pair : primary = JacksonAnnotationIntrospector, secondary = JaxbAnnotationIntrospector

Consistency Level

- **CONSISTENCY_LEVEL_READ_DEFAULT** (OPTIONAL): default read consistency level for all entities
- **CONSISTENCY_LEVEL_WRITE_DEFAULT** (OPTIONAL): default write consistency level for all entities
- **CONSISTENCY_LEVEL_SERIAL_DEFAULT** (OPTIONAL): default serial consistency level for all entities
- **CONSISTENCY_LEVEL_READ_MAP** (OPTIONAL): map(String,String) of read consistency levels for tables

Example:

```
"table1" -> "ONE"  
"table2" -> "QUORUM"  
...
```

- **CONSISTENCY_LEVEL_WRITE_MAP** (OPTIONAL): map(String,String) of write consistency levels for tables

Example:

```
"table1" -> "ALL"  
"table2" -> "EACH_QUORUM"  
...
```

- **CONSISTENCY_LEVEL_SERIAL_MAP** (OPTIONAL): map(String,String) of serial consistency levels for tables

Example:

```
"table1" -> "SERIAL"  
"table2" -> "LOCAL_SERIAL"  
...
```

Events Interceptors

- **EVENT_INTERCEPTORS** (OPTIONAL): list of events interceptors.

Bean Validation

- **BEAN_VALIDATION_ENABLE** (OPTIONAL): whether to enable Bean Validation.
- **POST_LOAD_BEAN_VALIDATION_ENABLE** (OPTIONAL): whether to enable Bean Validation for **POST_LOAD** events. Please note that this flag is taken into account **only if** **BEAN_VALIDATION_ENABLE** is true
- **BEAN_VALIDATION_VALIDATOR** (OPTIONAL): custom validator to be used.

If no validator is provided, **Achilles** will get the default validator provided by the default Validation provider.

If Bean Validation is enabled at runtime but no default Validation provider can be found, an exception will be raised and the bootstrap is aborted

Prepared Statements Cache

- **PREPARED_STATEMENTS_CACHE_SIZE** (OPTIONAL): define the LRU cache size for prepared statements cache.

By default, common operations like **insert**, **find** and **delete** are prepared before-hand for each entity class.

For **update** and all operations generated by the **DSL API**, since the updated fields and timestamp value are only known at runtime,

Achilles will prepare the statements only on the fly and save them into a **Guava LRU cache**.

The default size is 10000 entries. Once the limit is reached, oldest prepared statements are evicted, causing **Achilles** to re-prepare them and get warnings from the Java Driver.

You can get details on the LRU cache state by putting the logger `info.archinnov.achilles.internals.cache.StatementsCache` on **DEBUG**

- **STATEMENTS_CACHE** (OPTIONAL): provide an instance of the class `info.archinnov.achilles.internals.cache.StatementsCache` to store all prepared statements. This option is useful for unit testing to avoid re-preparing many times the same prepared statements

Insert Strategy

- **GLOBAL_INSERT_STRATEGY** (OPTIONAL): define the global insert strategy for all entities. Choose between `ConfigurationParameters.InsertStrategy.ALL_FIELDS` and `ConfigurationParameters.InsertStrategy.NOT_NULL_FIELDS`. Default value is `ConfigurationParameters.InsertStrategy.ALL_FIELDS`.

For more details, please check [Insert Strategy](#)

Naming Strategy

- **GLOBAL_NAMING_STRATEGY** (OPTIONAL): define the global naming strategy to be applied to keyspace, table and column names. If no strategy is defined, **Achilles** will fallback to `NamingStrategy.LOWER_CASE`

For more details, please check [Naming Strategy](#)

Bean Factory

- **DEFAULT_BEAN_FACTORY** (OPTIONAL): inject the default bean factory to instantiate new entities and UDT classes. The implementation class should implement the interface `info.archinnov.achilles.type.factory.BeanFactory`. The default implementation is straightforward:

```
@Override
public <T> T newInstance(Class<T> clazz) {
    try{
        return clazz.newInstance();
    } catch (InstantiationException | IllegalAccessException e) {
        ....
    }
}
```

Schema Name Provider

- **SCHEMA_NAME_PROVIDER** (OPTIONAL): define a schema name provider to bind dynamically an entity to a keyspace/table name at runtime. This feature is useful mostly in a multi-tenant context. The given provider should implement the `info.archinnov.achilles.type.SchemaNameProvider` interface

```

public interface SchemaNameProvider {
    /**
     * Provide keyspace name for entity class
     */
    <T> String keyspaceFor(Class<T> entityClass);

    /**
     * Provide table name for entity class
     */
    <T> String tableNameFor(Class<T> entityClass);
}

```

More details on [Dynamic Schema Name](#)

Runtime Codec

- **RUNTIME_CODECS** (OPTIONAL): inject a list of stateful codecs to be used by Achilles. Those codecs are instantiated at runtime.

See [Runtime Codecs](#) for more details

Asynchronous Operations

- **EXECUTOR_SERVICE** (OPTIONAL): define the executor service (thread pool) to be used by **Achilles** for its internal asynchronous operations
- **DEFAULT_EXECUTOR_SERVICE_MIN_THREAD** (OPTIONAL): define the minimum thread count for the executor service used by **Achilles** for its internal asynchronous operations.
- **DEFAULT_EXECUTOR_SERVICE_MAX_THREAD** (OPTIONAL): define the maximum thread count for the executor service used by **Achilles** for its internal asynchronous operations.
- **DEFAULT_EXECUTOR_SERVICE_THREAD_KEEPAIVE** (OPTIONAL): define the duration in seconds during which a thread is kept alive before being destroyed, on the executor service used by **Achilles** for its internal asynchronous operations.
- **DEFAULT_EXECUTOR_SERVICE_QUEUE_SIZE** (OPTIONAL): define the size of the **LinkedBlockingQueue** used by the executor service used by **Achilles** for its internal asynchronous operations
- **DEFAULT_EXECUTOR_SERVICE_THREAD_FACTORY** (OPTIONAL): define the thread factory used by **Achilles** for its internal asynchronous operations.

The default thread pool will be configured as follow:

```
new ThreadPoolExecutor(DEFAULT_EXECUTOR_SERVICE_MIN_THREAD, DEFAULT_EXECUTOR_SERVICE_MAX_THREAD,
    DEFAULT_EXECUTOR_SERVICE_THREAD_KEEPALIVE, TimeUnit.SECONDS,
    new LinkedBlockingQueue<Runnable>(DEFAULT_EXECUTOR_SERVICE_QUEUE_SIZE),
    DEFAULT_EXECUTOR_SERVICE_THREAD_FACTORY)
```

To close properly the thread pool on application removal, **Achilles** exposes the `PersistenceManagerFactory.shutdown()` method.

This method is annotated with `javax.annotation.PreDestroy` so that in a managed container, it will be invoked automatically.

Otherwise you can always manually call the `shutdown()` method.

For more details, please check [Asynchronous Operations](#)

Configuration

To configure **Achilles** with the above parameters, you need to provide a Map of key/value as constructor argument to the `PersistenceManagerFactory` class.

Example:

```
Map<ConfigurationParameters, Object> configMap = new HashMap<ConfigurationParameters, Object>();

Cluster cluster = Cluster.builder().contactPoints("localhost").withPort(9042).build();

configMap.put(FORCE_TABLE_CREATION, true);

ManagerFactory factory = ManagerFactoryBuilder
    .build(cluster, configMap); // providing the cluster object is MANDATORY
```

Alternatively, you can use the builder API from the `ManagerFactoryBuilder`:

```
ManagerFactory factory = ManagerFactoryBuilder
    .builder(cqlCluster)
    .withDefaultReadConsistency(ConsistencyLevel.QUORUM)
    .withDefaultWriteConsistency(ConsistencyLevel.QUORUM)
    .withKeyspaceName("my_keyspace")
```

```
.withExecutorServiceMinThreadCount(5)
.withExecutorServiceMaxThreadCount(10)
.doForceTableCreation(false)
.build();
```

Achilles-Custom-Types

CassandraVersion This is an enum used to declare the **Cassandra** version you're using for compile to [configure Achilles at compile time](#).

```
@CompileTimeConfig(cassandraVersion = CassandraVersion.CASSANDRA_3_0_X)
public interface AchillesConfig {

}
```

Depending on the chosen version, **Achilles** will generate appropriate DSL code.

Codec Define a codec to transform a source type into a target type.

The source type can be any Java type. The target type should be a Java type supported by Cassandra

```
public interface Codec<FROM, TO> {

    Class<FROM> sourceType();

    Class<TO> targetType();

    TO encode(FROM fromJava) throws AchillesTranscodingException;

    FROM decode(TO fromCassandra) throws AchillesTranscodingException;

}
```

Example of **LongToString** codec:

```
public class LongToString implements Codec<Long,String> {
    @Override
```

```

    public Class<Long> sourceType() {
        return Long.class;
    }

    @Override
    public Class<String> targetType() {
        return String.class;
    }

    @Override
    public String encode(Long fromJava) throws AchillesTranscodingException {
        return fromJava.toString();
    }

    @Override
    public Long decode(String fromCassandra) throws AchillesTranscodingException {
        return Long.parseLong(fromCassandra);
    }
}

```

CodecSignature Utility class to identify uniquely a codec. A codec signature is composed of:

- **sourceType**
- **targetType**
- optionally, **codecName** if provided

See [Runtime Codec](#) for more details

Column Mapping Strategy You can specify the column mapping strategy at compile time using the `@CompileTimeConfig` annotation:

```

@CompileTimeConfig(cassandraVersion = CassandraVersion.CASSANDRA_3_0_X, columnMappingStrategy = ColumnMappingStrategy.EXPLICIT)
public interface AchillesConfig {

}

```

The default behavior is `ColumnMappingStrategy.EXPLICIT`

See [Column Mapping Strategy](#) for more details

Encoding Define the encoding type for enum values. 2 choices are available:

1. `info.archinnov.achilles.annotations.Enumerated.Encoding.NAME`
encode enum using their `name()`
2. `info.archinnov.achilles.annotations.Enumerated.Encoding.ORDINAL`
encode enum using their `ordinal()`

More details on [Enum type](#)

Empty It is just a marker enum for asynchronous operations. Sometimes an operation does not return any meaningful result (like `deleteById(...)` or `insert(...)`).

Still we need to be able to ensure that the asynchronous operation has been executed correctly by calling `get()`.

If the execution has been successful, the singleton **Empty** is returned, otherwise an exception is raised

For more details, please check [Asynchronous Operations](#)

InsertStrategy Define the insert strategy on an entity. Right now only 2 values are possible:

1. `info.archinnov.achilles.type.InsertStrategy.NOT_NULL_FIELDS`
2. `info.archinnov.achilles.type.InsertStrategy.ALL_FIELDS`

Upon call to `insert()`, depending on the chosen strategy **Achilles** will

1. insert all fields on the entity, even if they are null
2. insert only **non null** fields of the entity

Check here for more details on the [Insert strategy](#)

Interceptor This is an interface defining the contract for a lifecycle interceptor. For more details, please check [Interceptors](#)

```
public interface Interceptor<T>
{
    boolean acceptEntity(Class<?> entityClass);

    public void onEvent(T entity);

    public List<Event> events();
}
```

LWTResultListener For all LWT operations, **Cassandra** returns an [applied] boolean column telling whether the operation has been successful or not. If a LWT update operation failed, **Cassandra** also returns the current values that differ from the ones used by the LWT update.

To intercepts the LWT operation result and current values, you can register a **LWTResultListener** using the **Options** API (see above)

The signature of the **LWTResultListener** is:

```
public interface LWTResultListener {

    default void onSuccess() {
        // Do nothing
    }

    void onError(LWTResult lwtResult);
}
```

The **LWTResult** type is defined as:

```
public class LWTResult {
    private final Operation operation;
    private final TypedMap currentValues;

    public LWTResult(Operation operation, TypedMap currentValues) {
        this.operation = operation;
        this.currentValues = currentValues;
    }

    ...

    public static enum Operation {INSERT, UPDATE}
}
```

For more details on LWT operations, please refer to [Lightweight Transaction](#)

NamingStrategy Define the naming strategy for the keyspace, table and column name. There are 3 possible strategies:

1. **SNAKE_CASE**: transform all schema name using [snake case](#)
2. **CASE_SENSITIVE**: enclose the name between double quotes (") for escaping the case

3. **LOWER__CASE**: transform the name to lower case

Check here for more details on the [Naming strategy](#)

SchemaNameProvider This is an interface to be implemented to provide dynamic binding of keyspace/table name at runtime.

The interface is

```
public interface SchemaNameProvider {  
  
    /**  
     * Provide keyspace name for entity class  
     */  
    <T> String keyspaceFor(Class<T> entityClass);  
  
    /**  
     * Provide table name for entity class  
     */  
    <T> String tableNameFor(Class<T> entityClass);  
}
```

The provider can be used with [CRUD API](#) and [DSL API](#)

Tuples Since **Cassandra** supports tuple type and the **Java driver** only exposes a `com.datastax.driver.core.TupleValue`, **Achilles** introduces **Tuple1**, **Tuple2**, ..., **Tuple10** types. They all extends the **Tuple** interface and exposes the following methods:

```
//Get all tuple values, in the correct order  
public List<Object> values();  
  
//Return the 1st component of the tuple  
public A _1();  
  
//Return the 2nd component of the tuple  
public B _2();  
  
...  
  
//Return the 10th component of the tuple  
public J _10();
```

Internally, **Achilles** will perform automatic conversion between those tuple types to/from `com.datastax.driver.core.TupleValue` so that you can use them directly in your entities

```
@Table
public class Entity {

    @Column
    private Tuple3<String, List<String>, Long> tuple3;

    ...
}
```

Note: in the above example, there is no need to add **@Frozen** on the **List<String>** because all tuples are frozen by default.

TypedMap The native query API used to return a `Map<String, Object>` as result. It is not very user-friendly because it forces you do manual type casting. Example:

```
Map<String, Object> columns = manager.nativeQuery("SELECT * FROM users WHERE userId = 10");

String name = (String)columns.get("name"); // BAD !
Long age = (Long)columns.get("age"); // BAD !
```

TypedMap is just an extension of `Map<String, Object>` offering two extras methods for convenience:

```
@SuppressWarnings("unchecked")
public <T> T getTyped(String key) {
    T value = null;
    if (super.containsKey(key) && super.get(key) != null) {
        value = (T) super.get(key);
        return value;
    }
    return value;
}

public <T> T getTypedOr(String key, T defaultValue) {
```

```

        if (super.containsKey(key)) {
            return getTyped(key);
        } else {
            return defaultValue;
        }
    }
}

```

Of course there is no magic, the dirty casting you no longer do, `getTyped()` will do it for you. The target type is passed at runtime while calling the method. `getTypedOr()` lets you provide a fall-back value.

Example of usage:

```

TypedMap columns = manager.nativeQuery("SELECT * FROM users WHERE userId = 10").first();

// Explicit type (String) is passed to method invocation
String name = columns.<String>getTyped("name");

// No need to provide explicit type. The compiler will infer type in this case
Long age = columns.get("age");

```

Achilles-Annotations

Below is a list of all **Achilles** annotations.

@ClusteringColumn This annotation indicates which field is a clustering column.

Put this annotation on several fields to have many clustering columns.

The attribute **value** indicates the ordering of this clustering column.

If you want to store the data in reversed order, set the attribute **asc** to **false**

The attribute **asc** indicates the sorting order. By default **asc = true**

Unlike Java indexes, the ordering starts at 1. It is a design choice since it is more natural for human being to start counting at 1

```

@Table
public class MyEntity
{
    @PartitionKey
    private Long userId;
}

```

```

    @ClusteringColumn(value = 1, asc = false)
    private Date date;

    @ClusteringColumn(2)
    private String type;

    ...
}

```

@Codec Define a custom codec system to encode any Java type into Cassandra-compatible types.

The provided Codec class should implement the **Codec** interface.

Let's consider the following codec transforming a **Long** to a **String**

```

public class LongToString implements Codec<Long,String> {
    @Override
    public Class<Long> sourceType() {
        return Long.class;
    }

    @Override
    public Class<String> targetType() {
        return String.class;
    }

    @Override
    public String encode(Long fromJava) throws AchillesTranscodingException {
        return fromJava.toString();
    }

    @Override
    public Long decode(String fromCassandra) throws AchillesTranscodingException {
        return Long.parseLong(fromCassandra);
    }
}

```

Example of **simple Long** type to **String** type transformation

```

@Column
@Codec(LongToString.class)

```

```

private Long longToString;

@Column
private List<@Codec(LongToString.class) Long> listOfLongToString;

@Column
private Map<Integer, @Codec(LongToString.class) Long> mapOfLongToStringValue;

```

@CodecRegistry Marker annotation to be used on configuration class for compile-time code generation. The type (class, abstract class or interface) having this annotation will expose a list of codecs to be used by **Achilles** during source code parsing.

Ex:

```

@CodecRegistry
public [class | abstract class | interface] MyCodecRegistry {

    //Source type = int, target type = String (according to IntToStringCodec codec)
    @Codec(IntToStringCodec.class)
    private int intToString;

    //Source type = MyOwnType, target type = String (according to MyOwnTypeToStringCodec)
    @Codec(MyOwnTypeToStringCodec.class)
    private MyOwnType myOwnTypeToString;

    //Source type = AnotherBean, target type = String (because of {@literal @}JSON)
    @JSON
    private AnotherBean beanToJSON;

    //Source type = MyEnum, target type = int (because of Encoding.ORDINAL)
    @Enumerated(Encoding.ORDINAL)
    private MyEnum enumToOrdinal;
}

```

See [Codec Registry](#) for more details

@Column Indicates a field to be mapped by **Achilles**. When the value attribute of **@Column** is given, the field will be mapped to this name. Otherwise the field name will be used.

Example:

```
@Column("age_in_years")
private Long age;
```

Remark: the custom `value` set on a column will override any [Naming Strategy](#) defined on the table/globally.

The case of the column name defined with `@Column` will be respected by **Achilles** and if the column name has mixed upper & lower cases, **Achilles** will enclose it between double quotes when creating the table.

Example:

```
@Column("itemsCount")
private Long itemsCount;
```

The generated DDL script would be:

```
CREATE TABLE ...
(
    ...

    "itemsCount" bigint,

    ...
)
```

Similarly, all generated CQL statements will enquote the column name.

@CompileTimeConfig You can specify configuration parameters at **compile time** using the **CompileTimeConfig** as below:

```
@CompileTimeConfig(cassandraVersion = CassandraVersion.CASSANDRA_3_0_X)
public interface AchillesConfig {

}
```

Depending on the chosen version, **Achilles** will generate appropriate DSL code. You can also specify the [column mapping strategy](#), [naming strategy](#), [insert strategy](#) and the project name.

See [Configuring Achilles at compile time](#) for more details.

@Computed This annotation defines a computed column e.g. a column generated by a function call.

For example to define the column `writetime(value)`:

```
@Table
public class Entity {

    ...

    @Column
    private String value;

    @Computed(function = "writetime", alias="writetime_value", targetColumns = {"value"})
    private Long valueWriteTime;
}
```

This annotation defines 4 attributes:

1. **function**: name of the CQL function to apply. It can be a pre-defined function like `writetime()` or an **user defined function**
2. **alias**: alias for this column. Necessary for disambiguation in case the same function is applied on different columns
3. **targetColumns**: list of **CQL columns** (and not Java field name) to apply the function to. If the function accepts multiple parameters, the order of the provided columns does matter
4. **cqlClass**: the CQL-compatible java class that represents the output type of the function

```
@Table
public class Entity {

    ...

    @Column("first_name")
    private String firstname;

    @Computed(function = "writetime", alias="writetime_firstname", targetColumns = {"first_name"})
    private Long valueWriteTime;
}
```

In the above example, **targetColumns** should point to *first_name*, which is the CQL column, and not *firstname* which represents the Java field name

@Consistency This annotation should be used on an entity

You need to specify the *read*, *write* and *serial* attribute to define the corresponding consistency level.

Example:

```
@Table
@Consistency(read=ConsistencyLevel.ONE, write=ConsistencyLevel.QUORUM, serial=ConsistencyLevel.SERIAL)
public class Entity
{
    ...
}
```

@EmptyCollectionIfNull For collections and maps, **Cassandra** does not distinguish between empty collection/map and null collection/map.

Therefore if you save an empty list, it is equivalent to setting **null** to this list, thus **deleting** it. When reading back the list, **Cassandra** will return a null value so **Achilles** will map back to a null list.

To avoid having to check for null, you can add the **@EmptyCollectionIfNull** annotation on a **collection or map**, **Achilles** will then map **null** value to an empty instance of the collection/map.

@Enumerated Define the encoding for enum values. There are 2 defined encoding types: **Encoding.NAME** and **Encoding.ORDINAL**, which use respectively the enum name & ordinal for encoding.

Example:

```
@Column(name = "pricing")
@Enumerated(Encoding.ORDINAL)
private Pricing pricing;

@Column(name = "pricings")
private List<@Enumerated(Encoding.ORDINAL) Pricing> pricings;

@Column(name = "pricing_per_country")
private Map<@Enumerated(Encoding.ORDINAL) Country, @Enumerated(Encoding.ORDINAL) Pricing> pricing_per_country;
```

More details on [Enum type](#)

@Frozen Define a frozen UDT or nested frozen UDT/collection

```
@Table
public class Entity {

    @Column
    @Frozen
    private MyUDT myUdt;

    @Column
    private List<@Frozen MyUDT> udtList;

    @Column
    private List<@Frozen Map<Integer, String>> nestedCollection;

}
```

@FunctionRegistry Marks a class as a function registry and let **Achilles** manage it

```
@FunctionRegistry
public interface MyFunctions {

    Integer sumOf(int val1, int val2);

    Long toLong(Date javaDate);

}
```

Note: it is possible to declare several function registries in your source code, just annotate them with **@FunctionRegistry**

Warning: it is not possible to declare 2 different functions with the same name and signature in the same keypace

Achilles will raise a compilation error when encountering such case. Ex:

```
@FunctionRegistry
public interface MyFunctionRegistry {

    String toString(long value);

    String toString(int value); // OK because parameter type is different

    String toString(long value); // KO because same signature as the first function

}
```

Remark 1: functions return types cannot be primitive, use boxed types instead

Remark 2: **Achilles'** codec system also applies for function parameters and return type

```
@FunctionRegistry
public interface FunctionsWithCodecSystemRegistry {

    // CQL function signature = listtojson(consistencylevels list<text>), returns text
    String listToJson(List<@Enumerated ConsistencyLevel> consistencyLevels);

    // CQL function signature = getinvalue(input text), returns text
    @Codec(IntToString.class) String getIntValue(String input);

}
```

Remark 3: functions name and parameters' name are lower-cased by Cassandra automatically

The **@FunctionRegistry** annotation accepts a **keyspace** attribute to specify in which **keyspace** the declared functions belong to. If not specified the currently logged keyspace will be used.

See [Functions Mapping](#) for more details

@Index This annotation defines a **secondary index** on a regular column

Additionally you can define the name of the index using the **name** attribute on the annotation.

Example

```
@Table
public class User {

    @PartitionKey
    private Long id;

    //Simple index
    @Column
    @Index
    private String countryCode;

    //Simple index with custom name
    @Column("custom_name")
    @Index(name = "country_code_idx")
}
```

```

    private String customName;

    //Index on collection
    @Column
    @Index
    private List<String> indexedList;

    //Full index on collection because of the usage of @Frozen
    @Column
    @Frozen
    @Index
    private List<String> indexedFullList;

    //Index on map key
    @Column("indexed_map_key")
    private Map<@Index String, Long> indexOnMapKey;

    //Index on map entry
    @Column("index_map_entry")
    @Index
    private Map<String, Long> indexOnMapEntry;

    //Custom index
    @Column
    @Index(indexClassName = "com.compagny.index.SecondaryIndex", indexOptions = "{key1
    private String custom;

    ...
}

```

The above code will generate the following CQL script:

```

CREATE INDEX IF NOT EXISTS ON my_entity(countryCode);

CREATE INDEX country_code_idx IF NOT EXISTS ON my_entity(custom_name);

CREATE INDEX IF NOT EXISTS ON my_entity(indexedlist);

CREATE INDEX IF NOT EXISTS ON my_entity(FULL(indexedfulllist));

CREATE INDEX IF NOT EXISTS ON my_entity(KEYS(indexed_map_key));

CREATE INDEX IF NOT EXISTS ON my_entity(ENTRY(index_map_entry));

```

```
CREATE CUSTOM my_entity_custom_index INDEX IF NOT EXISTS
ON my_entity(custom)
USING 'com.compagny.index.SecondaryIndex'
WITH OPTIONS = {'key1': 'value1', 'key2': 'value2'};
```

The **@Index** annotation exposes the following attributes:

1. **name**: index name. If not provided, will be “fieldname_index”
2. **indexClassName**: class name of custom index. This class should be present on the Cassandra server
3. **indexOptions**: index option string. Use the JSON map style {'key1': 'property1', 'key2': 'property2', ... }

@JSON Tell **Achilles** to serialize an un-supported data type into JSON string.
Example:

```
//Simple value encoding
@Column
@JSON
private MyPOJO pojoAsJson;

//Collection value encoding
@Column
private List<@JSON MyPOJO> myPojosAsJson;

//Map key encoding
@Column
private Map<@JSON MyPOJO, String> mapKeyEncoding;

//Map value encoding
@Column
private Map<String, @JSON MyPOJO> mapValueEncoding;

// Tuple value encoding
@Column
private Tuple3<String, @JSON MyPOJO, Integer> tupleValueEncoding;
```

@MaterializedView Marks a class as a **materialized view** and let **Achilles** manage it

```
@MaterializedView(baseEntity = UserEntity.class)
public class UserByCountryView {...}
```

This annotation defines 3 attributes:

- **baseEntity** (MANDATORY): the entity class from which this materialized view is derived
- **keyspace** (OPTIONAL): the name of the keyspace in which this materialized view belongs to
- **view** (OPTIONAL): the name of the materialized view. Defaults to the short class name if not provided

See [Materialized View Mapping](#) for more details

@PartitionKey This annotation indicates which field is a partition key for the table

Put this annotation on several fields to have a composite partition key.

The attribute **value** indicates the ordering of this partition component in a composite partition key.

The attribute **value** defaults to 1.

Unlike Java indexes, the ordering starts at 1. It is a design choice since it is more natural for human being to start counting at 1

```
private static class Entity
{
    @PartitionKey
    private Long id;

    @ClusteringColumn
    private UUID date;
    ...
}

private static class CompositePartitionEntity
{
    @PartitionKey(1)
    private Long id;

    @PartitionKey(2)
    private String type;

    @ClusteringColumn
    private UUID date;
    ...
}
```

In the above example, `id` and `type` are part of the composite partition key. `date` is the clustering key.

Remark: all fields annotated with `@PartitionKey` should be consecutive with respect to their ordering.
Failing this condition will raise an exception during compilation

@RuntimeCodec Transform a custom Java type into one of native types supported by the Java driver. Normally you'll use the `@Codec` annotation and provide a codec class but if your codec class is stateful or its construction needs some external dependencies and cannot be instantiated using the default no-args constructor, you can register the codec using this annotation and build it at runtime before injecting it into **Achilles**

Ex:

```
/Compile time
@Column
@RuntimeCodec(cqlClass = String.class)
private MyBean bean;

//Runtime
final Cluster cluster = .... // Create Java driver cluster object
final Codec<MyBean, String> statefulCodec = new .... // Create your codec with initialize
final CodecSignature<MyBean, String> codecSignature = new CodecSignature(MyBean.class, String.class)

ManagerFactory factory = ManagerFactoryBuilder
    .builder(cluster)
    ...
    .withRuntimeCodec(codecSignature, statefulCodec)
    .build();
```

A codec is looked up and identified uniquely at runtime using the following information:

- **sourceType**
- **targetType** (see `cqlClass` below)
- optionally, **codecName** (see below) if provided

The `@RuntimeCodec` annotation has 2 attributes:

1. **cqlClass** (MANDATORY): specify the target CQL type for the runtime codec. It is necessary to provide this type at compile time so that **Achilles** can generate appropriate meta data

2. **codecName** (OPTIONAL): useful to distinguish 2 different codecs having the same **sourceType** and **targetType**

@Static Defines a static column

```
@Table
public class Entity {

    @PartitionKey
    private Long id;

    @Column
    @Static
    private String staticVal;

    @ClusteringColumn
    private Date date;

    ...
}
```

The **@Static** annotation can only be used in an entity having **@ClusteringColumn** otherwise **Achilles** will raise an exception

@Strategy Define the insert and naming strategy on an entity.

For the insert strategy, 2 values are possible: `info.archinnov.achilles.type.InsertStrategy.NOT_NULL_FIELDS` and `info.archinnov.achilles.type.InsertStrategy.ALL_FIELDS`

Upon call to `insert()`, depending on the chosen strategy **Achilles** will

1. insert all fields on the entity, even if they are null
2. insert only **non null** fields of the entity

```
@Table
@Strategy(insert = InsertStrategy.ALL_FIELDS)
public class MyBean
{
    @PartitionKey
    private Long id;

    ...
}
```

Check here for more details on the [Insert strategy](#)

For the naming strategy, 3 values are possible: `info.archinnov.achilles.type.NamingStrategy.SNAKE_CASE`, `info.archinnov.achilles.type.NamingStrategy.LOWER_CASE` and `info.archinnov.achilles.type.NamingStrategy.CASE_SENSITIVE`

- **SNAKE_CASE**: transform all schema name using [snake case](#)
- **CASE_SENSITIVE**: enclose the name between double quotes (") for escaping the case
- **LOWER_CASE**: transform the name to lower case

If not set, the strategy defaults to `info.archinnov.achilles.type.NamingStrategy.LOWER_CASE`

```
@Table(keyspace="myKeyspace", table="myTable")
@Strategy(naming = NamingStrategy.SNAKE_CASE)
// final keyspace name will be transformed to 'myKeyspace' and table name to 'myTable'
// they are defined statically by @Table
public class Entity
{
    @PartitionKey
    private Long id;

    @Column
    private String firstName; //column name transformed to 'first_name'

    @Column(name = "misc")
    private String customInfo; //column name will be 'misc' because name defined on @Column
    ...
}
```

Check here for more details on the [Naming strategy](#)

@Table Indicates that an entity is candidate for persistence. By default

Achilles creates a table whose name is the **short class name** of the entity.

If you want to define a specific keyspace name, set the **keyspace** attribute.

If you want to define a specific table name, set the **table** attribute.

Example:

```
@Table(keyspace = "back_end", table="users")
public class User
{
    ...
    ...
}
```

Please note that Cassandra limits the table name to 48 characters max (because of limitations in Windows for file path max lengths)

@TimeUUID This annotation tells **Achilles** to map a Java UUID field to Cassandra timeuuid type.

Example:

```
@Table
public class Entity
{
    @PartitionKey
    private Long id;

    @TimeUUID
    @Column
    private UUID date;
    ...
}
```

This is especially useful in **CQL** to map to timeuuid type so you can use **Timeuuid functions** like `dateOf()/now()/minTimeuuid()/maxTimeuuid()` or `unixTimestampOf()` on native queries

@TTL This annotation should be used on an entity to define the default time to live in seconds

```
@Table
@TTL(3600) //1h time to live
public class Entity
{
    ...
}
```

@UDT Put on a JavaBean, this annotation indicates that this class is an UDT

```
@UDT(keyspace = "my_ks", name="user_udt")
public class UserUDT {
```

```

    @Column
    private String firstname;

    @Column
    private String lastname;

    ...
}

```

Then you can re-use the UDT class in another entity

```

@Table
public class Tweet {

    @PartitionKey
    private UUID tweetId;

    @Column
    private String content;

    @Column
    @Frozen
    private UserUDT author;
}

```

It is necessary to use the **@Frozen** annotation on the UDT field

Supported-Java-Types

Below are a list of Java types that are supported natively by **Achilles** (without requiring any codec declaration) and their corresponding **CQL** type

| Java Type | CQL Type |
|-----------|----------|
| byte | tinyint |
| Byte | tinyint |
| short | smallint |
| Short | smallint |
| boolean | boolean |

| Java Type | CQL Type |
|--|---------------------------|
| Boolean | boolean |
| int | int |
| Integer | int |
| java.math.BigInteger | varint |
| long | bigint |
| Long | bigint |
| double | double |
| Double | double |
| java.math.BigDecimal | decimal |
| float | float |
| Float | float |
| byte[] | blob |
| Byte[] | blob |
| ByteBuffer | blob |
| double[] | frozen<list<double>> |
| float[] | frozen<list<float>> |
| int[] | frozen<list<int>> |
| long[] | frozen<list<bigint>> |
| java.util.Date | timestamp |
| com.datastax.driver.core.LocalDate | date |
| java.time.Instant | timestamp |
| java.time.LocalDate | date |
| java.time.LocalDateTime | time |
| java.time.ZonedDateTime | tuple<timestamp, varchar> |
| java.net.InetAddress | inet |
| String | text |
| java.util.UUID | uuid |
| info.archinnov.achilles.type.tuples.Tuple1 ... Tuple10 | tuple<...> |
| java.util.List<X> | list<X> |
| java.util.Set<X> | set<X> |
| java.util.Map<X,Y> | map<X,Y> |

| Java Type | CQL Type |
|-----------------------|----------|
| java.util.Optional<X> | X |

Table-Mapping

For bean mapping, only **field-based access type** is supported by **Achilles**. Furthermore, there is no default mapping for fields. If you want a field to be mapped, you must annotate it with **@Column** / **@PartitionKey** / **@ClusteringColumn** / **@Computed**. All fields that are not annotated by either annotations is considered transient by **Achilles**

Common rules

Achilles maps an entity into a **Cassandra** table. Each row represents an instance of the entity and each column represents a field.

For clustered entities, each tuple of (partition key, clustering column 1, ... clustering column N) represent a logical row although it translates into distinct physical columns by the underlying storage engine.

Entity matching

- **Achilles** entities must conform to *JavaBean convention* with respect to accessors.
- A Java bean is candidate to be an **Achilles** entity if it is annotated with **@Table**
- All entities must have at least one field annotated with **@PartitionKey**
- All entities must have a public constructor so that initialization block will be invoked when instantiated by **Achilles**

Table name

- If the attribute *table* is provided by the **@Table** annotation, it will be used as table name in **Cassandra** for the entity
- Otherwise, the table name is derived from the entity class name (not fully qualified name) by replacing all “.” characters by “_”

In any case, the provided or derived table name should match the regexp pattern
: `[a-zA-Z0-9_]{1,48}`

The limitation to 48 characters comes from **Cassandra** restriction
on table name size

Examples:

```
@Table
public class MyEntity
{
    @PartitionKey
    private Long id;
    ...
}
```

Inferred table name = **myentity**

```
@Table(table = "another_entity")
public class AnotherEntity
{
    @PartitionKey
    private Long id;
    ...
}
```

Inferred table name = **another__entity**

Field mapping

A field is managed by **Achilles** if it is annotated with `@Column` / `@PartitionKey`
/ `@ClusteringColumn` / `@Computed`.

This is the **explicit** column mapping default behavior. You also can opt-in for
an **implicit** column mapping behavior. See

[Column Mapping Strategy](#) for more details on this

Column name

- By default, the column name in **Cassandra** is the same as the entity field
name.
- The column name in **Cassandra** can be overridden by providing the *value*
attribute on the `@Column` annotation.

On `@PartitionKey` / `@ClusteringColumn` fields, you can override the column name by adding a `@Column` annotation.

Examples:

```
...
@Column("tweet_id")
@ClusteringColumn
private UUID tweetId;
...
```

Column name in **Cassandra** = **name**

```
...
@Column(name="age_in_year")
private Integer age;
...
```

Column name in **Cassandra** = **age_in__year**

Un-mapped fields

It is possible to have values stored in **Cassandra** that is not mapped to any field/property of the entity. In this case **Achilles** will simply ignore them

Column Mapping Strategy

The default strategy for field/column mapping is to use the `@PartitionKey`, `@ClusteringColumn` and `@Column` annotations. This is the **explicit** mapping strategy. However, you can choose an **implicit** mapping strategy so that **Achilles** will pick all fields of your Java bean as **Cassandra** columns, except those annotated by `@Transient`.

In any case, the `@PartitionKey` and `@ClusteringColumn` annotations are still required to define the primary key of your table.

You can specify the column mapping strategy at compile time using the `@CompileTimeConfig` annotation:

```

@CompileTimeConfig(cassandraVersion = CassandraVersion.CASSANDRA_3_0_X, columnMappingStrategy = ColumnMappingStrategy.EXPLICIT)
public interface AchillesConfig {

}

```

The default behavior is `ColumnMappingStrategy.EXPLICIT`

See [Configuring Achilles at compile time](#) for more details

Static columns

Since **Cassandra** 2.0.6 it is possible to define **static columns**. A static column is a column which is related to the partition key(s).

There is only one distinct value of static column per partition key.

Consequently it is only possible to define static column within an entity having at least one `@ClusteringColumn` annotation

To define a static column, use the `@Static` annotation

```

...

@Column
@Static
private Integer age;
...

```

Collection and Map support

With **CQL**, collection and maps are supported natively so there is nothing special to do for **Achilles**

Example:

```

...
@Column
private List<String> addresses;
...

```

In the **CQL** semantics, a **null** collection/map or an **empty** collection/map is equivalent. This is not the case in Java.

To avoid null check in client code, you can add the `@EmptyCollectionIfNull` annotation on the collection/map. This way when deserializing values from **Cassandra**, **Achilles** will instantiate an empty collection/map instead of **null**

Enum type

Let's consider the following enums

```
public enum Country {
    USA, CANADA, FRANCE, UNITED_KINGDOM, GERMANY ...
}

public enum Pricing {
    PREMIUM, VALUE, ECONOMIC ...
}
```

Achilles does support enum types using the **@Enumerated** annotation. By default enums are serialized as String using the **name()** method.

Consequently, the ordering for enum types is based on their name, and not their natural ordering in Java (based on declaration order).

```
@Column(name = "pricing")
@Enumerated // or @Enumerated(Encoding.NAME)
private Pricing pricing;
```

This is an important detail and can be a gotcha if you don't pay attention when doing slice queries on entities having an enum as clustering column.

Achilles also lets you serialize the enums using their ordinal. In this case, you must change the value to **Encoding.ORDINAL**

```
@Column(name = "pricing")
@Enumerated(Encoding.ORDINAL)
private Pricing pricing;
```

The **@Enumerated** annotation also works on collections & maps.

```
@Column(name = "pricings")
private List<@Enumerated(Encoding.NAME) Pricing> pricings;

@Column(name = "pricing_per_country")
private Map<@Enumerated(Encoding.ORDINAL) Country, @Enumerated(Encoding.NAME) PricingType> pricingPerCountry;
```

Please note that when **Achilles** encounters an enum without the **@Enumerated** annotation (on simple value, collections or maps), it will encode it using the name. This is the **default behavior**

Counter type

Use the **@Counter** annotation on a column of type **Long** to indicate it is a counter column. Please note that if an entity contains a counter type, all other columns (normal and static) should also be of counter type. However, **clustering columns** can be of type different than counter.

Example:

```
...
@Column
@Counter
private Long tweetsCount;
...
```

UDT support

Achilles supports **Cassandra UDT (User Defined Type)** mapping. To mark a JavaBean as an UDT, just use the **@UDT** annotation:

```
@UDT(keyspace = "my_ks", name="user_udt")
public class UserUDT {

    @Column
    private String firstname;

    @Column
    private String lastname;

    ...
}
```

Then you can re-use the UDT class in another entity

```
@Table
public class Tweet {

    @PartitionKey
    private UUID tweetId;

    @Column
    private String content;
```

```

    @Column
    @Frozen
    private UserUDT author;
}

```

It is necessary to use the **@Frozen** annotation on the UDT field

Tuple types

Achilles also supports **CQL** tuples using the custom **Tuple** types.

```

@Table
public class Entity {

    ...

    @Column
    private Tuple3<Integer, String, List<String>> tuple3;

}

```

There are 10 classes for tuple types: **Tuple1**, **Tuple2**, ..., **Tuple10**. See [Tuples](#) for more details

Note: it is not necessary to add **@Frozen** on tuple columns because they are **by default already frozen**.

Nested collections, UDT or tuples do not need the **@Frozen** annotation either

Computed columns

It is possible to map the result of a function application to a column. For this, uses **@Computed**:

```

@Table
public class Entity {

    ...

    @Column
    private String value;

    @Computed(function = "writetime", alias="writetime_value", targetColumns = {"value"})

```

```

        private Long valueWriteTime;
    }

```

Please note that field mapped to computed values are only retrieve for SELECT query. They will be ignored for INSERT/UPDATE/DELETE.

For more details, see [@Computed](#)

Consistency Level

- Consistency levels can be defined on the entity using the **@Consistency** annotation. This applies to all fields
- Consistency levels can be overridden at runtime

Example:

```

@Table(table = "user_tweets")
@Consistency(read = QUORUM, write = QUORUM, serial = SERIAL)
public class UseEntity {

    @PartitionKey
    private Long id;

    @Column
    private String name;

    ...
}

```

In the above example:

- the entity is read using **QUORUM** and inserted using **QUORUM** consistency levels
- for LightWeight Transaction operations, **Achilles** will use the **SERIAL** consistency level

Value-less Entity

Achilles does support entity with only **@PartitionKey** / **@ClusteringColumn** fields. They are called **value-less** entities. A common use case for value-less entities is indexing data.

Time UUID

To map a Java UUID type to **Cassandra** `timeuuid` type, you need to add the `@TimeUUID` annotation on the target field.

Secondary Index

To add a secondary index on a column, use the `@Index` annotation. For more information on the usage, check [@Index](#)

Inheritance strategy

Achilles does support mapping inheritance. The strategy is **one table per class**. This is the most natural and straightforward strategy for inheritance.

Examples:

Case 1:

```
@Table(table = "parent")
public class ParentEntity {
    @PartitionKey
    protected Long id;

    @Column
    protected String name;
}

@Table(table = "child")
public class ChildEntity extends ParentEntity {
    @Column
    private String type;
}
```

With the above example, **Achilles** will create the following tables:

```
CREATE TABLE parent(
    id bigint PRIMARY KEY,
    name text
);

CREATE TABLE child(
    id bigint PRIMARY KEY,
```

```

        name text,
        type text
    );

```

Case 2:

```

public class ParentEntity {
    @PartitionKey
    private Long id;
}

@Table(table = "child1")
public class Child1Entity extends ParentEntity {
    @Column
    private String name;
}

@Table(table = "child2")
public class Child2Entity extends ParentEntity {
    @Column
    private String type;
}

```

For this case, **Achilles** will create the following tables:

```

CREATE TABLE child1(
    id bigint PRIMARY KEY,
    name text
);

CREATE TABLE child2(
    id bigint PRIMARY KEY,
    type text
);

```

Please note that all fields in parent classes should have **protected** or **package protected** visibility otherwise **Achilles** cannot access them and parse their annotations

Naming Strategy

You can define a strategy to transform the keyspace name, table name and column name defined on your entities. Available strategies are:

- **SNAKE_CASE**: transform all schema name using [snake case](#)
- **CASE_SENSITIVE**: enclose the name between double quotes (") for escaping the case
- **LOWER_CASE**: transform the name to lower case

The naming strategy can be defined at 2 places, by their ascending order of priority:

1. globally using the attribute **namingStrategy()** on the *****@CompileTimeConfig***** annotation: this strategy will apply to all entities
2. locally on each class using the **@Strategy** annotation

However, all those strategies are overridden if you set the name on the column manually using the **name** attribute of the **@Column** annotation

Naming Strategy priority

Priority (ascending order)

Description

1 (lowest priority)

Global naming strategy defined at compile time on **@CompileTimeConfig**

2

Locally on each entity using the **@Strategy** annotation

3 (highest priority)

Defined using the *value* attribute on the **@Column** annotation

Example:

```
@Table(keyspace="myKeyspace", table="myTable")
@Strategy(naming = NamingStrategy.SNAKE_CASE)
// final keyspace name will be 'myKeyspace' and table name to 'myTable' because defined on @Table
public class MyBean
{
    @PartitionKey
    private Long id;

    @Column
    private String firstName; //column name transformed to 'first_name'

    @Column(name = "misc")
    private String customInfo; //column name will be 'misc' because name defined on @Column
    ...
}
```

The case of the column name defined with `@Column` will be respected by **Achilles** and if the column name has mixed upper & lower cases, **Achilles** will enclose it between double quotes when creating the table.

Example:

```
@Column("itemsCount")
private Long itemsCount;
```

The generated DDL script would be:

```
CREATE TABLE ...
(
    ...

    "itemsCount" bigint,
    ...
)
```

Similarly, all generated CQL statements will enquote the column name.

Insert-Strategy

Rationale

The `Manager.crud().insert()` operation generates INSERT statements with all fields of the entity.

If some fields are not set (e.g. having `null` value), then **Achilles** just set the column to `null` in **Cassandra**

This sounds reasonable but has a huge impact on performance.

Indeed for the CQL semantics, **setting a column to null means deleting it** thus creating a `tombstone` column.

Let's say you have an **User** entity with around 10 fields representing user details. It is obvious that on user account creation not all of them are provided, probably only login/name/password fields are filled.

When inserting this user in **Cassandra** you'll create 3 columns for the login/name/password fields and around 7 **tombstones**. Later on during compaction **Cassandra** will need to clean up those 7 **tombstones**.

Why don't we in the first place not insert those fields that are **null** thus not creating useless **tombstones** ?

The insert strategies below are the answer for this issue.

Insert all fields

This is the default behavior for **Achilles**. Although creating a lot of **tombstones**, this strategy still has a huge advantage with regard to the data consistency.

Let's suppose you have the following sequence of code:

```
manager
    .crud()
    .insert(new User(10,'johndoe','John DOE','iamjohndoe!',32))
    .execute();

...
manager
    .crud()
    .insert(new User(10,'johndoe','John DOE','iamjohndoe!'))
    .execute();
```

What we would expect is that the 2nd **insert()** operation will wipe all the data from the first **insert()** and that's the case with the default **insert all fields** strategy.

Indeed the second **insert()** will generate a **INSERT INTO user(id,login,name,password,age,...) VALUES(10,'johndoe','John DOE','iamjohndoe!',null,null,...)** statement which erases the previous value for the fields **age** and it saves our day.

Insert not null fields

This strategy only insert not null fields of an entity. If we take the previous example:

```
manager
    .crud()
    .insert(new User(10,'johndoe','John DOE','iamjohndoe!',32))
```

```

        .execute();

...
manager
    .crud()
    .insert(new User(10, 'johndoe', 'John DOE', 'iamjohndoe!'))
    .execute();

```

The second `insert()` now will generate a `INSERT INTO user(id,login,name,password) VALUES(10,'johndoe','John DOE','iamjohndoe!')` statement, erasing existing values for id/login/name/password but letting the old age column intact.

If we fetch all data from **Cassandra** for this user, we will end up with inconsistent data.

To avoid that you'll need to issue a `delete()` operation before inserting again.

Configuration

Insert strategy can be customized for each entity or globally using the `@CompileTimeConfig` annotation. To choose between one or other strategy, annotate your entity with `@Strategy(insert = ...)`. There are 2 possible values

- `info.archinnov.achilles.type.InsertStrategy.ALL_FIELDS`
 - Pros: data consistency
 - Cons: may generate a lot of tombstones
- `info.archinnov.achilles.type.InsertStrategy.NOT_NULL_FIELDS`
 - Pros: does not create useless `tombstone`
 - Cons: may introduce data inconsistency when overwriting existing partition with new value

Insert Strategy priority

Priority (ascending order)

Description

1 (lowest priority)

Global naming strategy defined at compile time on `@CompileTimeConfig`

2

Locally on each entity using the `@Strategy` annotation

3 (highest priority)

at runtime, using the *.withInsertStrateg()* method on the *manager.crud().insert()* DSL

Materialized-View-Mapping

Achilles supports the new **Cassandra 3.0** materialized view feature. To mark an entity as

a materialized view, you can use the **@MaterializedView** annotation.

```
@Table(table = "user")
public class UserEntity {
    ...
}
@MaterializedView(baseEntity = UserEntity.class, view = "user_by_country")
public class UserByCountryView {
}
```

The **@MaterializedView** annotation defines 3 attributes:

- **baseEntity** (MANDATORY): the entity class from which this materialized view is derived
- **keyspace** (OPTIONAL): the name of the keyspace in which this materialized view belongs to
- **view** (OPTIONAL): the name of the materialized view. Defaults to the short class name if not provided

A materialized view must comply to some rules:

- the materialized view can declare a subset or all of the base entity fields
- each field in the materialized view should match the name, source type and target type of the same field in the base entity
- all base entity **primary key** fields should be in the materialized view **primary key**
- all collection and UDT types of the base entity should be duplicated to the materialized view
- a materialized view is allowed to have among its primary key fields maximum 1 field which is not a primary key from the base table

Failing one of these rules will trigger a compilation error.

Furthermore, since only read operations are allowed on a materialized view, the generated manager for the view has a restricted API:

- `crud().findById(...)`
- `dsl().select(...)`
- `raw().typedQueryForSelect(...)`
- `raw().nativeQuery(...)`

For `raw().nativeQuery(...)`, a runtime check is performed on the provided statement to ensure it is a **SELECT** statement

Functions-Mapping

Achilles supports the new **Cassandra 3.0 UDF** (*User Defined Function*) and **UDA** (*User Defined Aggregate*) features. **Achilles** does not make difference between **UDF** and **UDA** and consider both of them as mere *function calls*.

Function declaration

Before being able to declare an **UDF** / **UDA** you need to create a **function registry**. For this, create a class/abstract class/interface and annotates it with **@FunctionRegistry**

```
@FunctionRegistry
public interface MyFunctionRegistry {
    ...
}
```

Once created, you can register your functions using the registry class:

```
@FunctionRegistry
public interface MyFunctionRegistry {

    Long string_to_long(String input);

    String long_to_string(Long input);
}
```

Remark: because of a limitation of the JDK annotation processing API, it is not allowed to declare function return types using primitives. Use their corresponding boxed types instead

The types used in the method signatures can be any types defined in [Supported Java Types](#). Furthermore the functions declaration can leverage [Achilles Codec System](#) for parameter types and return type too!

```
@FunctionRegistry
public interface FunctionsWithCodecSystemRegistry {

    // CQL function signature = list_to_json(consistencylevels list<text>), returns text
    String list_to_json(List<@Enumerated ConsistencyLevel> consistencyLevels);

    // CQL function signature = get_int_value(input text), returns text
    @Codec(IntToString.class) String get_int_value(String input);

}
```

In **Cassandra**, a function declaration and usage is scoped to a **keyspace**. Therefore the **@FunctionRegistry** annotation exposes a **keyspace** attribute for you to specify in which **keyspace** the declared functions should belong to.

```
@FunctionRegistry(keyspace = "production")
public interface ProductionFunctionRegistry {

    ...

}
```

It is possible to declare many functions in the same registry class as well as having many function registry classes, you just need to annotate them with **@FunctionRegistry**. However duplicate function declaration is not allowed, **Achilles** will issue a compilation error if it finds 2 functions with the **same name, same signature** (parameter types and return type) being declared in the **same keyspace** no matter if they were declared in the same registry class or not.

Generated type classes

In order to guarantee **type-safe function calls** in its API, **Achilles** will generate a class hierarchy of all [Supported Java Types](#) as well as all types defined on all user entities. Those types are parsed at compile time by the processor and the corresponding classes are generated inside the package **info.archinnov.achilles.generated.function**

An example of generated type classes:

```

// Native supported types
Array_Byte_Type
Array_Primitive_byte_Type
...
BigDecimal_Type
BigInteger_Type
Boolean_Type
...
ZonedDateTime_Type

// Non native types present in user entities, views & function registries
List_String_Type
Optional_String_Type
Tuple2_Integer_List_String_Type
...

```

The naming convention for those type classes is quite simple, they all end with **_Type** suffix. The type name is normalized by replacing any **<** or **>** by **_** . For example if the type **Map<Integer, List<String>>** is used in an entity, the type class **Map_Integer_List_String_Type** will be generated

We need to generate all these type classes (not to be confused with **type class** in **Scala**) to generate type safe function call API.

Type safe function calls

At compile time **Achilles** will parse all the methods and their signature to generate 2 classes:

- **info.archinnov.achilles.generated.function.SystemFunctions:**
this class contains all the pre-defined system functions in **Cassandra** like **now()**, **ttl()**, **writetime()** ...
- **info.archinnov.achilles.generated.function.FunctionsRegistry:**
this class contains the compiled version of all the functions declared in all registries (classes annotated with **@FunctionRegistry**)

Those classes will be useful for the Select **DSL API**. Example:

```

manager
  .dsl()
  .select()
  .id()
  .function(SystemFunctions.toUnixTimestamp(SimpleEntity_AchillesMeta.COLUMNS.DATE), "write")
  .function(SystemFunctions.writetime(SimpleEntity_AchillesMeta.COLUMNS.VALUE), "write")

```

```

        .fromBaseTable()
    ...

```

The `SystemFunctions.toUnixTimestamp(Date_Type input)` signature accepts a `Date_Type` type class as input. To be able to apply this function we have to provide a column that has the same type. Fortunately, for each entity **Achilles** generates a class `XXX_AchillesMeta.COLUMNS` which list all annotated fields of the entity `XXX` and the type of those columns are one of the generated type class.

Please note that it is possible to have nested function calls as long as the input type class and returned type class match:

```

manager
    .dsl()
    .select()
    .id()
    .function(SystemFunctions.max(SystemFunctions.writetime(SimpleEntity_AchillesMeta.COLUMNS.id(), SystemFunctions.toUnixTimestamp(SimpleEntity_AchillesMeta.COLUMNS.id())))
    .fromBaseTable()
    ...

```

You can also call **UDF** / **UDA** using the class `info.archinnov.achilles.generated.function.FunctionsRegistry`:

```

manager
    .dsl()
    .select()
    .id()
    .function(FunctionsRegistry.list_to_json(SimpleEntity_AchillesMeta.COLUMNS.id(), SimpleEntity_AchillesMeta.COLUMNS.id()))
    .fromBaseTable()
    ...

```

The `function(FunctionCall functionCall, String alias)` method requires a 2nd parameter which is the alias for this function call. With **Cassandra** it is not mandatory to define an alias, you can always retrieve the function call value on a row using the canonical name `function_name(column_name *)` but it is much more easier and error-proof to require an explicit alias for every function call.

Please note that when you're using a function call at runtime through the SELECT DSL API, **Achilles** can only return a **TypedMap** because it cannot map the value of this function call from the CQL row.

If you want to map the result of a function call to a field of an entity, use the `@ComputedColumn` annotation instead.

Example:

```

TypedMap row = manager
    .dsl()
    .select()
    .id()
    .function(FunctionsRegistry.list_to_json(SimpleEntity_AchillesMeta.COLUMNS.CONSI
    .fromBaseTable()
    .where()
    .id().Eq(id)
    .getTypedMap();

// Retrieve the function call result using the alias
String json = row.getTyped("consistency_list_to_json");

```

Codec-System

Definition and usage

Sometimes it is useful to be able to transform a Java type into a native Java type supported by Cassandra.

An example could be to convert **Joda DateTime** into `java.util.Date`.

To support such usage, **Achilles** lets you define your own transformation using **@Codec** on each field of your entity.

The definition of the **@Codec** annotation is:

```

public @interface Codec {

    /**
     * Codec Implementation class. The provided Codec class should implement the {@link
     */
    Class<? extends info.archinnov.achilles.type.codec.Codec> value();
}

```

To define a type transformer, you need to provide a codec class. Your codec class should implement the **Codec** interface.

Example:

Let's consider the following codec transforming a **Long** to a **String**

```

public class LongToString implements Codec<Long,String> {
    @Override
    public Class<Long> sourceType() {
        return Long.class;
    }

    @Override
    public Class<String> targetType() {
        return String.class;
    }

    @Override
    public String encode(Long fromJava) throws AchillesTranscodingException {
        return fromJava.toString();
    }

    @Override
    public Long decode(String fromCassandra) throws AchillesTranscodingException {
        return Long.parseLong(fromCassandra);
    }
}

```

Example of **simple Long** type to **String** type transformation

```

@Column
@Codec(LongToString.class)
private Long longToString;

```

Example of **List<Long>** to **List<String>** transformation

```

@Column
private List<@Codec(LongToString.class) Long> listOfLong;

```

Example of **Set<Long>** to **Set<String>** transformation

```

@Column
private Set<@Codec(LongToString.class) Long> setOfLong;

```

Example of key Map transformation: **Map<Long,Double>** to **Map<String,Double>**

```

@Column
private Map<@Codec(LongToString.class) Long, Double> mapKeyTransformation;

```

Example of value Map transformation: `Map<Integer,Long>` to `Map<Integer,String>`

```
@Column
private Map<Integer,@Codec(LongToString.class) Long> mapValueTransformation;
```

You can also set the `@Codec` annotation on the target class. Example:

```
// Codec definition
public class BeanToStringCodec implements Codec<MyBean, String> {

    @Override
    public Class<MyBean> sourceType() { return MyBean.class };

    @Override
    public Class<String> targetType() { return String.class };

    @Override
    public String encode(MyBean fromJava) throws AchillesTranscodingException {...};

    @Override
    public MyBean decode(String fromCassandra) throws AchillesTranscodingException {...}
}

// Custom bean with @Codec annotation
@Codec(BeanToStringCodec.class)
public class MyBean {

    ...

}

//Usage

@Table
public class MyEntity {

    ...

    @Column
    private MyBean myBean; //No need to add @Codec because already defined on class MyB
}
```

Codec declaration and type

There are 2 ways to declare your codec:

- directly on-site using the `@Codec` annotation put on the field, as shown above
- inside a class, abstract class or interface annotated by `@CodecRegistry`

If a codec is both declared in a `@CodecRegistry` and on-site. The on-site definition will have higher priority. Example:

```
@CodecRegistry
public interface MyCodecRegistry {

    @Enumerated(Encoding.NAME)
    MyEnum myEnum;
}

@Table
public class MyEntity {

    @Enumerated(Encoding.ORDINAL) // has higher priority than @Enumerated(Encoding.NAME)
    @Column
    private MyEnum myEnum;
}
```

There are 2 types of codec:

- simple codecs using the `@Codec` as shown above
- runtime codecs which are instantiated and injected into **Achilles** at runtime

Codec Registry

Marker annotation to be used on configuration class for compile-time code generation. The type (class, abstract class or interface) having this annotation will expose a list of codecs to be used by **Achilles** during source code parsing.

Ex:

```
@CodecRegistry
public [class | abstract class | interface] MyCodecRegistry {
```

```

    //Source type = int, target type = String (according to IntToStringCodec codec)
    @Codec(IntToStringCodec.class)
    private int intToString;

    //Source type = MyOwnType, target type = String (according to MyOwnTypeToStringCodec)
    @Codec(MyOwnTypeToStringCodec.class)
    private MyOwnType myOwnTypeToString;

    //Source type = AnotherBean, target type = String (because of {@literal @}JSON)
    @JSON
    private AnotherBean beanToJSON;

    //Source type = MyEnum, target type = int (because of Encoding.ORDINAL)
    @Enumerated(Encoding.ORDINAL)
    private MyEnum enumToOrdinal;
}

```

It is possible to declare several codec registries in your source code, just annotate them with **@CodecRegistry**

Warning: it is not possible to declare 2 different codecs for the same source type for all registered codec registries. **Achilles** will raise a compilation error when encountering such case

Ex:

```

@CodecRegistry
public class MyCodecRegistry {

    @Codec(MyOwnTypeToStringCodec.class)
    private MyOwnType myOwnTypeToString;

    // ERROR, not possible to have a 2nd codec for the same source type MyOwnType
    @Codec(MyOwnTypeToBytesCodec.class)
    private MyOwnType myOwnTypeToBytes;
}

```

Runtime Codec

Transform a custom Java type into one of native types supported by the Java driver. Normally you'll use the **@Codec** annotation and provide a codec class but if your codec class is stateful or its construction needs some external

dependencies and cannot be instantiated using the default no-args constructor, you can register the codec using this annotation and build it at runtime before injecting it into **Achilles**

Ex:

```
//Compile time
@Column
@RuntimeCodec(cqlClass = String.class)
private MyBean bean;

//Runtime
final Cluster cluster = .... // Create Java driver cluster object
final Codec<MyBean, String> statefulCodec = new .... // Create your codec with initialize
final CodecSignature<MyBean, String> codecSignature = new CodecSignature(MyBean.class, S

ManagerFactory factory = ManagerFactoryBuilder
                        .builder(cluster)
                        ...
                        .withRuntimeCodec(codecSignature, codec)
                        .build();
```

A codec is looked up and identified uniquely at runtime using the following information:

- **sourceType**
- **targetType** (see **cqlClass** below)
- optionally, **codecName** (see below) if provided

The **@RuntimeCodec** annotation has 2 attributes:

1. **cqlClass** (MANDATORY): specify the target CQL type for the runtime codec. It is necessary to provide this type at compile time so that **Achilles** can generate appropriate meta data
2. **codecName** (OPTIONAL): useful to distinguish 2 different codecs having the same **sourceType** and **targetType**

DDL-Scripts-Generation

If you want to create manually all the column families instead of letting **Achilles** do it for you, you can active the log level of **ACHILLES_DDL_SCRIPT** to **DEBUG**. The creation script will be displayed

Sample **log4j.xml** config file

```

<logger name="ACHILLES_DDL_SCRIPT">
  <level value="DEBUG" />
</logger>

```

Example of entity:

```

@Table
public class UserEntity
{

    @PartitionKey
    private Long id;

    @Column
    private String name;

    @Column
    private String label;

    @Column(name = "age_in_years")
    private Long age;

    @Column
    private List<String> friends;

    @Column
    private Set<String> followers;

    @Column
    private Map<Integer, String> preferences;

    @Column
    private Counter version;
}

```

Logs output:

```

CREATE TABLE UserEntity(
  age_in_years bigint,
  name text,
  label text,
  id bigint,
  friends list,
  followers set,
  preferences map,

```

```
PRIMARY KEY(id)
);
```

Schema-Generator

Achilles provides a command-line schema generator as well as an class to generate CQL schema script.

Usage

Command-line

First you need to compile your entity classes with **Achilles** and generate a **.jar** file. Then retrieve the **achilles-schema-generator--shaded.jar** file from your Maven repository.

In a shell, type:

```
java -cp <jar_containing_your_entity_classes>:achilles-schema-generator-<version>-shaded.jar
```

The generator will scan your **<jar_containing_your_entity_classes>** file for the generated meta classes and generate the CQL schema into the **<target_cql_script>** file using the provided **<keyspace_name>**.

Please note that if the keyspace name is defined statically on the entity using **@Table**, it will override the given **<keyspace_name>**

As a jar dependency

You can also use the **SchemaGenerator** in a project by pulling the Maven dependency

```
<dependency>
  <groupId>info.archinno</groupId>
  <artifactId>achilles-schema-generator</artifactId>
  <version>${achilles.version}</version>
</dependency>
```

Usage:

```
// Create a schema String
String cqlSchema = SchemaGenerator.builder()
    .withKeyspace("default_keyspace_name")
    .generateCustomTypes(true) //default = true
    .generateIndices(true) //default = true
    .generate();

/**
 *
 * Generate to an instance of java.lang.Appendable
 *
 */
SchemaGenerator.builder()
    .withKeyspace("default_keyspace_name")
    .generateCustomTypes(true) //default = true
    .generateIndices(true) //default = true
    .generateTo(appendable);

/**
 *
 * Generate to an instance of java.io.File
 *
 */
SchemaGenerator.builder()
    .withKeyspace("default_keyspace_name")
    .generateCustomTypes(true) //default = true
    .generateIndices(true) //default = true
    .generateTo(file);

/**
 *
 * Generate to an instance of java.nio.file.Path
 *
 */
SchemaGenerator.builder()
    .withKeyspace("default_keyspace_name")
    .generateCustomTypes(true) //default = true
    .generateIndices(true) //default = true
    .generateTo(path);
```

Statements-Logging-and-Tracing

Global statements logging

It is possible to display CQL statements in the log. For this you need to activate the `ACHILLES_DML_STATEMENT` logger.

Sample **log4j.xml** config file

```
<logger name="ACHILLES_DML_STATEMENT">
  <level value="DEBUG" />
</logger>
```

Expected log messages:

```
DEBUG ACHILLES_DML_STATEMENT - Query ID 3873d787-c379-4bec-85c0-907a96b362ba : [SELECT id,da
DEBUG ACHILLES_DML_STATEMENT -   Java bound values : [5107957575091721216, Thu Oct 01 02:00:
DEBUG ACHILLES_DML_STATEMENT -   Encoded bound values : [5107957575091721216, Thu Oct 01 02:
DEBUG ACHILLES_DML_STATEMENT - ResultSet[ exhausted: false, Columns[id(bigint), date(timesta
DEBUG ACHILLES_DML_STATEMENT - Query ID 3873d787-c379-4bec-85c0-907a96b362ba results :
      id: 5107957575091721216, date: Thu Oct 01 02:00:00 CEST 2015, consistencylist: [QUORUM,
```

Achilles displays in the log:

1. the query string
2. the bound values in raw Java
3. the encoded bound values using the [Codec System](#)
4. the returned first 10 rows if it is a SELECT statement

For each **sent** query, **Achilles** generates a **Query ID** so that you can correlate the query string with the returned results.

Dynamic statements logging

It is also possible to activate **DML statements display** only on specific entities, in this case, set the log level of those entities to **DEBUG**

```
<logger name="com.project.test.MyEntity">
  <level value="DEBUG" />
</logger>
```

The **DML statements display** is dynamic at runtime so if your logging framework allow dynamic update of log level (**Logback** does it) then it is possible to activate statement logging at runtime for a short period of time for debugging purposes.

Dynamic queries tracing

Sometime, showing statements is not enough to diagnose performance issue. We then need to activate request tracing. But how to do without changing the code ?

Achilles comes to the rescue and allow **dynamic tracing activation**. Again all you need to do is changing the log level of your entity to **TRACE**

```
<logger name="com.project.test.MyEntity">
  <level value="TRACE" />
</logger>
```

Example of output of a Lightweight Transaction operation:

```
TRACE Query tracing at host localhost/127.0.0.1:9465 with achieved consistency level null
TRACE *****
TRACE Description | Sou
TRACE Preparing 55b436a0-09b2-11e4-8838-4f290a9552fc | /12
TRACE Parsing SELECT preferred_ip FROM system.peers WHERE peer='127.0.0.1' | /12
TRACE Preparing statement | /12
TRACE Executing single-partition query on peers | /12
TRACE Acquiring sstable references | /12
TRACE Merging memtable tombstones | /12
TRACE Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones | /12
TRACE Merging data from memtables and 0 sstables | /12
TRACE Read 0 live and 0 tombstoned cells | /12
TRACE Sending message to /127.0.0.1 | /12
TRACE Message received from /127.0.0.1 | /12
TRACE Parsing SELECT * FROM system.paxos WHERE row_key = 0x000000000000000a AND cf_id = c06b | /12
TRACE Preparing statement | /12
TRACE Executing single-partition query on paxos | /12
TRACE Acquiring sstable references | /12
TRACE Merging memtable tombstones | /12
TRACE Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones | /12
TRACE Promising ballot 55b436a0-09b2-11e4-8838-4f290a9552fc | /12
TRACE Parsing UPDATE paxos USING TIMESTAMP 1405162227466000 AND TTL 864000 SET in_progress_b | /12
TRACE Preparing statement | /12
TRACE Acquiring switchLock read lock | /12
TRACE Appending to commitlog | /12
```

```

TRACE Adding to paxos memtable | /12
TRACE Sending message to /127.0.0.1 | /12
TRACE Message received from /127.0.0.1 | /12
TRACE Processing response from /127.0.0.1 | /12
TRACE Reading existing values for CAS precondition | /12
TRACE Executing single-partition query on entity_with_enum | /12
TRACE Acquiring sstable references | /12
TRACE Merging memtable tombstones | /12
TRACE Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones | /12
TRACE CAS precondition is met; proposing client-requested updates for 55b436a0-09b2-11e4-883 | /12
TRACE Sending message to /127.0.0.1 | /12
TRACE Message received from /127.0.0.1 | /12
TRACE Parsing SELECT * FROM system.paxos WHERE row_key = 0x0000000000000000a AND cf_id = c06b | /12
...

```

Generated-Classes

Using compile-time code generation through [Annotation Processor](#), **Achilles** can

1. inspect the source code at compile time
2. perform validation on the source code at compile time
3. have access to all generics parameters at compile time
4. generate meta classes at compile time
5. generate type-safe DSL builder at compile time

Meta classes

For each entity class annotated with **@Table**, **Achilles** will generate a corresponding

```
info.archinnov.achilles.generated.meta.entity.<entity_class_name>_AchillesMeta
class
```

This meta class exposes the following useful methods:

1. `public T createEntityFrom(Row row):` self-explanatory
2. `public ConsistencyLevel readConsistency(Optional<ConsistencyLevel> runtimeConsistency):` retrieve read consistency from runtime value, static configuration and default consistency configuration in **Achilles**

3. `public ConsistencyLevel writeConsistency(Optional<ConsistencyLevel> runtimeConsistency)`: retrieve write consistency from runtime value, static configuration and default consistency configuration in **Achilles**
4. `public ConsistencyLevel serialConsistency(Optional<ConsistencyLevel> runtimeConsistency)`: retrieve serial consistency from runtime value, static configuration and default consistency configuration in **Achilles**
5. `public InsertStrategy insertStrategy()`: determine insert strategy using static annotation and **Achilles** global configuration
6. `public void triggerInterceptorsForEvent(Event event, T instance)` : trigger all registered interceptors for this entity type on the provided instance, given the event type

Each meta class contains a `public static` field for each property. For example, given the following entity:

```
@Table
public static User {

    @PartitionKey
    private Long userId;

    @Column
    private String firstname;

    @Column
    private String lastname;

    @Column
    private Set<String> favoriteTags;

    ...
}
```

The `User_AchillesMeta` class will expose the following static property meta:

1. `User_AchillesMeta.userId`
2. `User_AchillesMeta.firstname`
3. `User_AchillesMeta.lastname`
4. `User_AchillesMeta.favoriteTags`

Each property meta class will expose:

1. `public VALUETO encodeFromJava(VALUEFROM javaValue):` encode the given Java value into CQL-compatible value using the **[Codec System]**
2. `public VALUEFROM decodeFromGettable(GettableData gettableData):` decode the value of the current property from the **GettableData** object. The **GettableData** is the common interface for `com.datastax.driver.core.Row`, `com.datastax.driver.core.UDTValue` and `com.datastax.driver.core.TupleValue`

For type-safe function call (from **Cassandra_2_2_X** and after), **Achilles** also generates a **COLUMNS** field inside the `info.archinnov.achilles.generated.meta.entity.<entity_class_name>` class. This field will expose all the entity fields with the correct mirror type useful for type-safe function calls.

For more details, see [Functions Mapping](#)

Manager classes

In addition to meta classes, **Achilles** also generates for each entity a `info.archinnov.achilles.generated.manager.<entity_class_name>_Manager` class that exposes the following methods:

1. `crud()`: exposes the [CRUD API](#)
2. `dsl()`: exposes the [DSL API](#)
3. `raw()`: exposes the [RAW API](#)

Manager Factory class

To be type-safe, **Achilles** also generates one `info.archinnov.achilles.generated.ManagerFactory` class that exposes as many `public <entity_class_name>_Manager for<entity_class_name>()` methods as there are different entity classes.

Compilation Unit

Please note that all entity classes and related UDT classes should be in the same **compilation unit**, e.g. the source code of **all** entity classes and related **UDT** classes should be compiled together in one go.

If your project import some external entity classes shipped in a **.jar** file, **Achilles** does not have access to its

source code, thus cannot generate all the meta and manager classes for those entities.

Read [multi-project support](#) if you need this feature

Thread-safety-and-state

By design, **Achilles ManagerFactory** and **Manager** are stateless. They only contain references to meta data and pre-built prepared statements. For common operations the **Manager** delegates the job to underlying implementation classes.

Thus they are **thread-safe** and can be injected as a singleton in any of your DAO/Services.

Example:

```
...
@Inject
private User_Manager userManager;
...
...
```

Manager-API

Below is a list of APIs supported by **Achilles Manager**:

CRUD

This API performs simple INSERT/DELETE operations on the entity. It exposes the following methods:

- `findById(...)`: find an entity providing the complete primary key
- `insert(ENTITY instance)`: insert an instance of the entity
- `delete(ENTITY instance)`: delete the entity
- `deleteById(...)`: delete the entity with the provided complete primary key

- `deleteByPartitionKeys(...)`: delete the entity with the provided complete partition key component(s)

For each of those operations, you can specify runtime values for

1. Consistency level/Serial consistency level
2. Retry policy
3. Fetch size
4. Outgoing payload
5. Paging state (useful only for SELECT)
6. `ifExists()` for INSERT, `ifNotExists()` for DELETE
7. tracing ON/OFF
8. `ResultSet` async listener
9. Row async listener
10. Lightweight Transaction result listener for INSERT/DELETE

The `ResultSet` async listener is just a function `Function<ResultSet, ResultSet>` to spy on the returned raw `com.datastax.driver.core.ResultSet` object. You are not allowed to call methods that will **consume** this resultset like `one()`, `all()`, `iterator()`, `fetchMoreResults()`

The Row async listener is a function `Function<Row, Row>` and lets you manipulate the raw `com.datastax.driver.core.ResultSet` object. This time, it is possible to read values from the row because it is already fetched

LightWeight Transaction result listener should implement the interface [LWTResultListener](#)

DSL

The DSL API lets you create type-safe SELECT, UPDATE and DELETE queries for your entity.

Select DSL The Select DSL look like:

```
User user = manager
    .dsl()
    .select()
    .id()
```

```

.firstname()
.lastname()
.fromBaseTable()
.where()
.userId().Eq(id)
.getOne();

```

The DSL exposes the following final methods:

- `ENTITY getOne()`: get the first found entity
- `CompletableFuture<ENTITY> getOneAsync()`: get the first found entity asynchronously
- `Tuple2<ENTITY, ExecutionInfo> getOneWithStats()`: get the first found entity with execution info
- `CompletableFuture<Tuple2<ENTITY, ExecutionInfo>> getOneAsyncWithStats()`: get the first found entity with execution info asynchronously
- `List<ENTITY> getList()`: get found entities
- `CompletableFuture<List<ENTITY>> getListAsync()`: get found entities asynchronously
- `Tuple2<List<ENTITY>, ExecutionInfo> getListWithStats()`: get found entities with execution info
- `CompletableFuture<Tuple2<List<ENTITY>, ExecutionInfo>> getListAsyncWithStats()`: get found entities with execution info asynchronously
- `Iterator<ENTITY> iterator()`: return an iterator for found entities
- `TypedMap getTypedMap()`: get the first found CQL row as an instance of `TypedMap`
- `Tuple2<TypedMap, ExecutionInfo> getTypedMapWithStatus()`: get the first found CQL row with execution info as an instance of `TypedMap`
- `CompletableFuture<TypedMap> getTypedMapAsync()`: get the first found CQL row asynchronously as an instance of `TypedMap`
- `CompletableFuture<Tuple2<TypedMap, ExecutionInfo>> getTypedMapAsyncWithStats()`: get the first found CQL row with execution info asynchronously an instance of `TypedMap`
- `List<TypedMap> getTypedMaps()`: get the found CQL rows as a list of `TypedMap`

- `List<Tuple2<TypedMap, ExecutionInfo>> getTypedMapsWithStats():` get the found CQL rows with execution info as a list of `TypedMap`
- `CompletableFuture<List<TypedMap>> getTypedMapsAsync():` get the found CQL rows asynchronously as a list of `TypedMap`
- `CompletableFuture<Tuple2<List<TypedMap>, ExecutionInfo>> getTypedMapsAsyncWithStats():` get the found CQL rows with execution info asynchronously as a list of `TypedMap`
- `Iterator<TypedMap> typedMapIterator():` return an iterator for found CQL rows as `TypedMap`

It is not mandatory to provide values for the **WHERE** clause, you can call the method

`without_WHERE_Clause()` to retrieve values

Please note that as soon as you invoke a function call (with `.function()`, **Achilles** can only return the results as `TypedMap` and not as entities) because it cannot map the result of the function call into a field of your entity.

If you need to map the result of a function call into a field of your entity, use the `@Computed` annotation instead.

Similar to the **CRUD API**, you can define runtime values for consistency levels, retry policy, ...

Update DSL The Update DSL look like:

```
manager
    .dsl()
    .update()
    .fromBaseTable()
    .firstname().Set("new firstname")
    .where()
    .userId().Eq(id)
    .execute();
```

The DSL exposes the following final methods:

- `execute():` execute the update
- `CompletableFuture<Empty> executeAsync():` execute the update asynchronously
- `ExecutionInfo executeWithStats():` execute the update and return the execution info

- `CompletableFuture<ExecutionInfo>> executeAsyncWithStats()`: execute the update and return the execution info asynchronously

Similar to the [CRUD API](#), you can define runtime values for consistency levels, retry policy, ...

Delete DSL The Delete DSL look like:

```
manager
    .dsl()
    .delete()
    .firstname()
    .lastname()
    .fromBaseTable()
    .where()
    .userId().Eq(id)
    .execute();
```

The DSL exposes the following final methods:

- `execute()`: execute the delete
- `CompletableFuture<Empty> executeAsync()`: execute the delete asynchronously
- `ExecutionInfo executeWithStats()`: execute the delete and return the execution info
- `CompletableFuture<ExecutionInfo>> executeAsyncWithStats()`: execute the delete and return the execution info asynchronously

Similar to the [CRUD API](#), you can define runtime values for consistency levels, retry policy, ...

Raw Queries

The RAW queries API lets you inject any instance of:

- `com.datastax.driver.core.RegularStatement`
- `com.datastax.driver.core.PreparedStatement`
- `com.datastax.driver.core.BoundStatement`

and execute the query for you.

Typed Query The Typed Query API execute the statement and map the returned `com.datastax.driver.core.Row` object(s) back to entity instance. Thus this API can only be used for **SELECT** statements.

```
Statement statement = ...;

User instance = userManager
    .raw()
    .typedQueryForSelect(statement)
    .getOne();
```

This API exposes the following final methods:

- `ENTITY getOne()`: get the first found entity
- `CompletableFuture<ENTITY> getOneAsync()`: get the first found entity asynchronously
- `Tuple2<ENTITY, ExecutionInfo> getOneWithStats()`: get the first found entity with execution info
- `CompletableFuture<Tuple2<ENTITY, ExecutionInfo>> getOneAsyncWithStats()`: get the first found entity with execution info asynchronously
- `List<ENTITY> getList()`: get found entities
- `CompletableFuture<List<ENTITY>> getListAsync()`: get found entities asynchronously
- `Tuple2<List<ENTITY>, ExecutionInfo> getListWithStats()`: get found entities with execution info
- `CompletableFuture<Tuple2<List<ENTITY>, ExecutionInfo>> getListAsyncWithStats()`: get found entities with execution info asynchronously
- `Iterator<ENTITY> iterator()`: return an iterator for found entities

Similar to the **CRUD API**, you can define runtime values for consistency levels, retry policy, ...

Native Query The Native Query API execute the statement and map the returned `com.datastax.driver.core.Row` object(s) back to instance(s) of **TypedMap**.

```
Statement statement = ...;
final TypedMap found = userManager
    .raw()
    .nativeQuery(statement)
    .getOne();
```

This API exposes the following final methods:

- `TypedMap getOne()`: get the first found row
- `CompletableFuture<TypedMap> getOneAsync()`: get the first found row asynchronously
- `Tuple2<TypedMap, ExecutionInfo> getOneWithStats()`: get the first found row with execution info
- `CompletableFuture<Tuple2<TypedMap, ExecutionInfo>> getOneAsyncWithStats()`: get the first found row with execution info asynchronously
- `List<TypedMap> getList()`: get found entities
- `CompletableFuture<List<TypedMap>> getListAsync()`: get found rows asynchronously
- `Tuple2<List<TypedMap>, ExecutionInfo> getListWithStats()`: get found rows with execution info
- `CompletableFuture<Tuple2<List<TypedMap>, ExecutionInfo>> getListAsyncWithStats()`: get found rows with execution info asynchronously
- `Iterator<ENTITY> iterator()`: return an iterator for found rows
- `execute()`: execute the statement
- `CompletableFuture<Empty> executeAsync()`: execute the statement asynchronously
- `ExecutionInfo executeWithStats()`: execute the statement and return the execution info
- `CompletableFuture<ExecutionInfo>> executeAsyncWithStats()`: execute the statement and return the execution info asynchronously

Similar to the [CRUD API](#), you can define runtime values for consistency levels, retry policy, ...

Dynamic Schema Name

In some multi-tenant environment, the keyspace/table name cannot be known ahead of time but only during runtime.

For this purpose, **Achilles** introduces a [SchemaNameProvider](#) interface to let people bind keyspace/table names dynamically at runtime.

This provider can be used with [CRUD API](#) and [DSL API](#):

```
final SchemaNameProvider dynamicProvider = ...;

userManager
    .crud()
    ...
    .withSchemaNameProvider(dynamicProvider)
    .execute();

userManager
    .dsl()
    .select()
    ...
    .from(dynamicProvider)
    .where()
    ...

userManager
    .dsl()
    .update()
    .from(dynamicProvider)
    ...
    .where()
    ...

userManager
    .dsl()
    .delete()
    ...
    .from(dynamicProvider)
    ...
    .where()
    ...
```

Raw Statement Generation

The **Manager** instance also exposes the following methods to generate raw `com.datastax.driver.core.BoundStatement` and bound values. They are available for all both [CRUD API](#) and [DSL API](#)

- `BoundStatement generateAndGetBoundStatement():`
 - generate the prepared statement or get it from the statement cache if it already exists
 - extract values from entity or from the given API
 - bind values to the prepared statement
 - return the `com.datastax.driver.core.BoundStatement` instance
- `String getStatementAsString():` self-explanatory
- `List<Object> getBoundValues():` extract raw Java values from entity or from the given API
- `List<Object> getEncodedBoundValues():` similar as `getBoundValues()` but encode the values using the [Codec System](#)

Other Methods

Apart from the 3 API ([CRUD](#), [DSL](#) and [Query](#)), the **Manager** class also exposes some utility methods:

- `public ENTITY mapFromRow(Row row):` map a given instance of `com.datastax.driver.core.Row` to an instance of entity
- `public Session getNativeSession():` return the native `com.datastax.driver.core.Session` object used by this **Manager**
- `public Cluster getNativeCluster():` return the native `com.datastax.driver.core.Cluster` object used by this **Manager**

Asynchronous-Operations

All the [CRUD API](#), [DSL API](#) and [RAW API](#) support asynchronous execution

Asynchronous CRUD

Insert

```
final CompletableFuture<Empty> future = userManager
    .crud()
    .insert(...)
    .executeAsync();

final CompletableFuture<ExecutionInfo> future = userManager
    .crud()
    .insert(...)
    .executeAsyncWithStats();
```

Select

```
final CompletableFuture<User> future = userManager
    .crud()
    .findById(...)
    .getOneAsync();

final CompletableFuture<Tuple2<User, ExecutionInfo>> future = userManager
    .crud()
    .findById(...)
    .getOneAsyncWithStats();
```

Delete

```
final CompletableFuture<Empty> future = userManager
    .crud()
    .deleteById(...)
    .executeAsync();

final CompletableFuture<ExecutionInfo> future = userManager
    .crud()
    .deleteById(...)
    .executeAsyncWithStats();

final CompletableFuture<Empty> future = userManager
    .crud()
    .deleteByPartitionKeys(...)
    .executeAsync();
```

```

final CompletableFuture<ExecutionInfo> future = userManager
    .crud()
    .deleteByPartitionKeys(...)
    .executeAsyncWithStats();

```

Asynchronous DSL

Select

```

final CompletableFuture<User> future = userManager
    .dsl()
    .select()
    ...
    .getOneAsync();

final CompletableFuture<Tuple2<User, ExecutionInfo>> future = userManager
    .dsl()
    .select()
    ...
    .getOneAsyncWithStats();

final CompletableFuture<List<User>> futures = userManager
    .dsl()
    .select()
    ...
    .getListAsync();

final CompletableFuture<Tuple2<List<User>, ExecutionInfo>> futures = userManager
    .dsl()
    .select()
    ...
    .getListAsyncWithStats();

```

Update

```

final CompletableFuture<Empty> future = userManager
    .dsl()
    .update()
    ...
    .executeAsync();

```

```

final CompletableFuture<ExecutionInfo> future = userManager
    .dsl()
    .update()
    ...
    .executeAsyncWithStats();

```

Delete

```

final CompletableFuture<Empty> future = userManager
    .dsl()
    .delete()
    ...
    .executeAsync();

final CompletableFuture<ExecutionInfo> future = userManager
    .dsl()
    .delete()
    ...
    .executeAsyncWithStats();

```

Asynchronous Query

Typed Query

```

final CompletableFuture<User> future = userManager
    .raw()
    .typedQueryForSelect(...)
    ...
    .getOneAsync();

final CompletableFuture<Tuple2<User, ExecutionInfo>> future = userManager
    .raw()
    .typedQueryForSelect(...)
    ...
    .getOneAsyncWithStats();

final CompletableFuture<List<User>> futures = userManager
    .raw()
    .typedQueryForSelect(...)
    ...
    .getListAsync();

```

```

final CompletableFuture<Tuple2<List<User>, ExecutionInfo>> futures = userManager
    .raw()
    .typedQueryForSelect(...)
    ...
    .getListAsyncWithStats();

```

Native Query

```

final CompletableFuture<TypedMap> future = userManager
    .raw()
    .nativeQuery(...)
    ...
    .getOneAsync();

final CompletableFuture<Tuple2<TypedMap, ExecutionInfo>> future = userManager
    .raw()
    .nativeQuery(...)
    ...
    .getOneAsyncWithStats();

final CompletableFuture<List<TypedMap>> futures = userManager
    .raw()
    .nativeQuery(...)
    ...
    .getListAsync();

final CompletableFuture<Tuple2<List<TypedMap>, ExecutionInfo>> futures = userManager
    .raw()
    .nativeQuery(...)
    ...
    .getListAsyncWithStats();

final CompletableFuture<Empty> future = userManager
    .raw()
    .nativeQuery(...)
    ...
    .executeAsync();

final CompletableFuture<ExecutionInfo> future = userManager
    .raw()
    .nativeQuery(...)
    ...
    .executeAsyncWithStats();

```

The Empty enum

It is just a marker enum. Sometimes an operation does not return any meaningful result (like `delete()` or `update()`).

Still we need to be able to ensure that the asynchronous operation has been executed correctly by calling `get()`.

If the execution has been successful, the singleton `Empty` is returned, otherwise an exception is raised

Exception Handling

All exceptions that could be raised during the blocking call to `get()` will return the following runtime exception

- `DriverException`
- `DriverInternalError`
- `AchillesLightWeightTransactionException`
- `AchillesBeanValidationException`
- `AchillesInvalidTableException`
- `AchillesStaleObjectStateException`
- `AchillesException`

Thread Pooling

By default a Thread Pool executor with a **LinkedBlockingQueue** of size 1000 will be used for all asynchronous operations.

The default configuration of this thread pool is given below

```
new ThreadPoolExecutor(5, 20, 60, TimeUnit.SECONDS,  
new LinkedBlockingQueue<Runnable>(1000),  
new DefaultExecutorThreadFactory());
```

The default implementation of the **ThreadFactory** is

```
public class DefaultExecutorThreadFactory implements ThreadFactory {  
  
    private static final Logger logger = getLogger("achilles-default-executor");  
  
    private final AtomicInteger threadNumber = new AtomicInteger(0);  
    private Thread.UncaughtExceptionHandler uncaughtExceptionHandler = new Thread.UncaughtExceptionHandler() {  
        @Override  
        public void uncaughtException(Thread t, Throwable e) {
```

```

        logger.error("Uncaught asynchronous exception : "+e.getMessage(), e);
    }
};

@Override
public Thread newThread(Runnable r) {
    Thread thread = new Thread(r);
    thread.setName("achilles-default-executor-" + threadNumber.incrementAndGet());
    thread.setDaemon(true);
    thread.setUncaughtExceptionHandler(uncaughtExceptionHandler);
    return thread;
}
}

```

Of course, you can provide your own thread pool or configure the default thread pool with your custom parameters.

Please refer to [Thread Pool Config](#)

To close properly the thread pool on application removal, **Achilles** exposes the `ManagerFactory.shutdown()` method.

This method is annotated with `javax.annotation.PreDestroy` so that in a managed container, it will be invoked automatically.

Otherwise you can always manually call the `shutdown()` method.

Consistency-Level

Static definition

Consistency levels can be defined statically with entity mapping using the `@Consistency` annotation.

Please refer to [Achilles Annotations](#) for more details.

Runtime setting

For all the [CRUD API](#) and [DSL API](#), **Achilles** allow you to inject Consistency level:

```

manager
    .crud() | .dsl()
    ...
    .withConsistencyLevel(...)

```

`.execute()` | `getOne()` | `getList()` ...

Settings priority

Consistency levels can be defined at different places. Below is a summary of all type of consistency levels and their respective priority

Priority (in ascending order).

Description

0

Hard-coded defaults **LOCAL_ONE** and **LOCAL_SERIAL**

1

Default consistency level defined in **Achilles** configuration (if any)

2

Default consistency level defined on the *com.datastax.driver.core.Cluster* object

3

Default read/write/serial **consistency level map** defined in **Achilles** configuration (if any)

4

Static **@Consistency** annotation on class

5

Defined at runtime using *withConsistency(...)* method

Quick-Reference

Let's consider the following entity for all the below examples:

```
@Table
public class User
{
    @PartitionKey
    private Long userId;
```

```

@Column
private String firstname;

@Column
private String lastname;

public User(){

public User(Long userId, String firstname, String lastname){...}

//Getters & Setters
}

```

Inserting an entity

```

manager
    .crud()
    .insert(new User(10L,"John","DOE"))
    .execute();

```

Inserting an entity as JSON (from Cassandra 2.2.x and after)

```

manager
    .crud()
    .insertJSON("{\"userid\": 10, \"firstname\": \"John\", \"lastname\": \"DOE\"}")
    .execute();

```

Updating a property for an entity

```

manager
    .dsl()
    .update()
    .fromBaseTable()
    .firstName().Set("Jonathan")
    .where()

```

```
.userId().Eq(10L)
.execute();
```

Updating a property for an entity using JSON (from Cassandra 2.2.x and after)

```
manager
    .dsl()
    .update()
    .fromBaseTable()
    .firstName().Set_FromJSON("\"Jonathan\"")
    .where()
    .userId().Eq_FromJSON("10")
    .execute();
```

Deleting an entity

Deleting an entity instance

```
User user = new User(10L, null, null);
manager
    .crud()
    .delete(user)
    .execute();
```

Deleting an entity by its id

```
manager
    .crud()
    .deleteById(10L)
    .execute();
```

Deleting a whole partition

```
manager
    .crud()
```

```
.deleteByPartitionKeys(10L)
.execute();
```

Deleting a property for an entity

Using the delete DSL

```
manager
    .dsl()
    .delete()
    .biography()
    .fromBaseTable()
    .where()
    .userId().Eq(10L)
    .execute();
```

Using the delete DSL with JSON (from Cassandra 2.2.x and after)

```
manager
    .dsl()
    .delete()
    .biography()
    .fromBaseTable()
    .where()
    .userId().Eq_FromJSON("10")
    .execute();
```

Using the update DSL

```
manager
    .dsl()
    .update()
    .fromBaseTable()
    .biography.Set(null)
    .where()
    .userId().Eq(10L)
    .execute();
```

Do not forget that in **CQL semantics** setting a column to **null** means deleting it

Finding entities with clustering columns

For all examples in this section, let's consider the following clustered entity representing a tweet line

```
@Table(table = "lines")
public class TweetLine
{
    @PartitionKey
    @Column("user_id")
    private Long userId;

    @ClusteringColumn(1)
    @Enumerated
    private LineType type;

    @ClusteringColumn(value = 2, asc = false) // Sort by descending order
    @TimeUUID //Time uuid type in Cassandra
    @Column("tweet_id")
    private UUID tweetId;

    @Column
    private String content;

    //Getters & Setters

    public static enum LineType
    { USERLINE, TIMELINE, FAVORITELINE, MENTIONLINE}
}
```

Find by partition key and clustering columns

Get the last 10 tweets from timeline, starting from tweet with lastUUID

```
// Generate SELECT * FROM lines WHERE user_id = ? AND (type, tweet_id) < (?,?) AND type = TIMELINE
List<TweetLine> tweets = manager
    .dsl()
    .select()
    .allColumns_FromBaseTable()
    .where()
    .userId().Eq(10L)
    .type_And_tweetId().type_And_tweetId_Lt_And_type_Gte(LineType.TIMELINE, lastUUID, lastUUID)
    .limit(10)
    .getList();
```

Iterating through a large set of entities

Fetch all timeline tweets by batch of 100 tweets

```
Iterator<TweetLine> iterator = manager
    .dsl()
    .select()
    .allColumns_FromBaseTable()
    .where()
    .userId().Eq(10L)
    .type_And_tweetId().type_And_tweetId_Lt_And_type_Gte(LineType.TIMELINE, lastUUID, L)
    .withFetchSize(100) // Fetch Size = 100 for each page
    .iterator();

while(iterator.hasNext())
{
    TweetLine timelineTweet = iterator.next();
    ...
}
```

Deleting entities with clustering columns

Deleting all timeline tweets

```
// Generate DELETE * FROM lines WHERE user_id = ? AND tpe = ?
manager.
    .dsl()
    .delete()
    .allColumns_FromBaseTable()
    .where()
    .userId().Eq(10L)
    .type().Eq(LineType.TIMELINE)
    .execute();
```

Deleting the whole partition using the **CRUD API**

```
// Generate DELETE * FROM lines WHERE user_id = ?
manager.
    .crud()
    .deleteByPartitionKeys(10L)
```

```
.execute();
```

Mapping UDT

To declare a JavaBean as UDT

```
@UDT(keyspace = "...", name = "user_udt")
public class UserUDT
{
    @Column
    private Long userId;

    @Column
    private String firstname;

    @Column
    private String lastname;

    //Getters & Setters
}
```

Then you can re-use the UDT in another entity

```
@Table
public class Tweet
{
    @PartitionKey
    @TimeUUID
    private UUID id

    @Column
    private String content;

    @Column
    @Frozen
    private UserUDT author;

    //Getters & Setters
}
```

Please notice that the **@Frozen** annotation is mandatory for UDT. Unfrozen UDT is only available for Cassandra 3.6 and after

Accessing Meta Classes for Encoding/Decoding functions

Achilles annotation processor will generate, for each entity:

1. An **EntityClassName_Manager** class
2. An **EntityClassName_AchillesMeta** class

The **EntityClassName_AchillesMeta** class provides the following methods for encoding/decoding:

1. `public T createEntityFrom(Row row):` self-explanatory
2. `public ConsistencyLevel readConsistency(Optional<ConsistencyLevel> runtimeConsistency):` retrieve read consistency from runtime value, static configuration and default consistency configuration in **Achilles**
3. `public ConsistencyLevel writeConsistency(Optional<ConsistencyLevel> runtimeConsistency):` retrieve write consistency from runtime value, static configuration and default consistency configuration in **Achilles**
4. `public ConsistencyLevel serialConsistency(Optional<ConsistencyLevel> runtimeConsistency):` retrieve serial consistency from runtime value, static configuration and default consistency configuration in **Achilles**
5. `public InsertStrategy insertStrategy():` determine insert strategy using static annotation and **Achilles** global configuration
6. `public void triggerInterceptorsForEvent(Event event, T instance) :` trigger all registered interceptors for this entity type on the provided instance, given the event type

Each meta class contains a `public static` field for each property. For example, given the following entity:

```
@Table
public static User {

    @PartitionKey
    private Long userId;

    @Column
    private String firstname;

    @Column
    private String lastname;
```

```

        @Column
        private Set<String> favoriteTags;

        ...
    }

```

The `User_AchillesMeta` class will expose the following **static property metas**:

1. `User_AchillesMeta.userId`
2. `User_AchillesMeta.firstname`
3. `User_AchillesMeta.lastname`
4. `User_AchillesMeta.favoriteTags`

Each property meta class will expose:

1. `public VALUETO encodeFromJava(VALUEFROM javaValue)`: encode the given Java value into CQL-compatible value using the [Codec System](#)
2. `public VALUEFROM decodeFromGettable(GettableData gettableData)`: decode the value of the current property from the **GettableData** object. The **GettableData** is the common interface for `com.datastax.driver.core.Row`, `com.datastax.driver.core.UDTValue` and `com.datastax.driver.core.TupleValue`

Querying Cassandra

Native query using the [RAW API](#)

```

final Statement statement = session.newSimpleStatement("SELECT firstname,lastname FROM u
List<TypedMap> rows = userManager
    .raw()
    .nativeQuery(statement, 100)
    .getList();

for(TypedMap row : rows)
{
    String firstname = row.getTyped("firstname");
    String lastname = row.getTyped("lastname");
    ...
}

```

Typed query using the [RAW API](#)

```
final Statement statement = session.newSimpleStatement("SELECT firstname,lastname FROM u

List<User> users = userManager
    .raw()
    .typedQueryForSelect(statement, 100)
    .getList();

for(User user : user)
{
    ...
}
```

Asynchronous execution

Asynchronous for the [CRUD API](#)

```
final CompletableFuture<Empty> futureInsert = userManager
    .crud()
    .insert(new User(...))
    .executeAsync();

final CompletableFuture<User> futureUser = userManager
    .crud()
    .findById(10L)
    .executeAsync();

final CompletableFuture<Empty> futureDelete = userManager
    .crud()
    .deleteById(10L)
    .executeAsync();
```

Note: **Empty** is a singleton enum to avoid returning a CompletableFuture of **null**

Asynchronous for the DSL API

```
final CompletableFuture<List<TweetLine>> futureTweets = tweetManager
    .dsl()
    .select()
    .allColumns_FromBaseTable()
    .where()
    .userId().Eq(10L)
    .type().Eq(LineType.TIMELINE)
    .limit(30)
    .getListAsync();

final CompletableFuture<Empty> futureUpdate = userManager
    .dsl()
    .update()
    .fromBaseTable()
    .lastname_Set("new lastname")
    .where()
    .userId().Eq(10L)
    .executeAsync();

final CompletableFuture<Empty> futureDelete = tweetManager
    .dsl()
    .delete()
    .allColumns_FromBaseTable()
    .where()
    .userId().Eq(10L)
    .type().Eq(LineType.TIMELINE)
    .executeAsync();
```

Asynchronous for the RAW API

```
final Statement statement = session.newSimpleStatement("SELECT firstname,lastname FROM u
CompletableFuture<List<TypedMap>> futureTypedMaps = userManager
    .raw()
    .nativeQuery(statement, 100)
    .getListAsync();

CompletableFuture<List<User>> futureUsers = userManager
    .raw()
    .typedQueryForSelect(statement, 100)
```

```
.getListAsync();
```

Getting the ExecutionInfo back

For the [CRUD API](#)

```
final ExecutionInfo executionInfo = userManager
    .crud()
    .insert(new User(...))
    .executeWithStats();

final ExecutionInfo executionInfo = userManager
    .crud()
    .deleteById(10L)
    .executeWithStats();

final Tuple2<User, ExecutionInfo> resultWithExecInfo = userManager
    .crud()
    .findById(10L)
    .getWithStats();
```

For the [DSL API](#)

```
final Tuple2<List<TweetLine>, ExecutionInfo> tweetsWithStats = tweetManager
    .dsl()
    .select()
    .allColumns_FromBaseTable()
    .where()
    .userId().Eq(10L)
    .type().Eq(LineType.TIMELINE)
    .limit(30)
    .getListWithStats();

final ExecutionInfo executionInfo = userManager
    .dsl()
    .update()
    .fromBaseTable()
    .lastname_Set("new lastname")
```

```

        .where()
        .userId().Eq(10L)
        .executeWithStats();

final ExecutionInfo executionInfo = tweetManager
    .dsl()
    .delete()
    .allColumns_FromBaseTable()
    .where()
    .userId().Eq(10L)
    .type().Eq(LineType.TIMELINE)
    .executeWithStats();

```

For the **RAW API**

```

final Statement statement = session.newSimpleStatement("SELECT firstname,lastname FROM u
Tuple2<List<TypedMap>, ExecutionInfo> typedMapsWithStats = userManager
    .raw()
    .nativeQuery(statement, 100)
    .getListWithStats();

Tuple2<List<User>, ExecutionInfo> usersWithStats = userManager
    .raw()
    .typedQueryForSelect(statement, 100)
    .getListWithStats();

```

Working with consistency level

Defining consistency statically

```

@Table
@Consistency(read=ConsistencyLevel.ONE, write=ConsistencyLevel.QUORUM, serial = Consiste
public class User
{
    ...
}

```

Setting consistency level at runtime

```
userManager
    .crud()
    ...
    .withConsistencyLevel(ConsistencyLevel.QUORUM)
    ...

userManager
    .dsl()
    ...
    .withConsistencyLevel(ConsistencyLevel.QUORUM)
    ...
```

Working with TTL

Defining TTL statically

```
@Table
@TTL(1000)
public class User
{
    ...
}
```

Setting TTL at runtime

```
userManager
    .crud()
    .insert(...)
    ...
    .usingTimeToLive(10)
    ...

userManager
    .dsl()
    .update()
    ...
    .usingTimeToLive(10)
    ...
```

Working with Timestamp

```
userManager
  .crud()
  .insert(...)
  ...
  .usingTimestamp(new Date().getTime())
  ...
```

```
userManager
  .crud()
  .deleteById(...)
  ...
  .usingTimestamp(new Date().getTime())
  ...
```

```
userManager
  .dsl()
  .update()
  ...
  .usingTimestamp(new Date().getTime())
  ...
```

```
userManager
  .dsl()
  .delete()
  ...
  .usingTimestamp(new Date().getTime())
  ...
```

Working with Lightweight Transaction

API

```
userManager
  .crud()
  .insert(...)
  ...
  .ifNotExists()
  ...
```

```
userManager
  .crud()
  .deleteById(...)
```

```

...
    .ifExists()
...

userManager
    .dsl()
    .update()
    ...
    .ifExists()
    ...

userManager
    .dsl()
    .update()
    .fromBaseTable()
    .firstName().Set("new firstname")
    ...
    .if_Firstname().Eq("previous_firstname")
    ...

userManager
    .dsl()
    .delete()
    ...
    .ifExists()
    ...

userManager
    .dsl()
    .delete()
    ...
    .if_Firstname().Eq("previous_firstname")
    ...

```

LWT Result Listener

To have tighter control on LWT updates, inserts or deletes, **Achilles** lets you inject a listener for LWT operations result.

```

LWTResultListener lwtListener = new LWTResultListener() {

    @Override
    public void onSuccess() {
        // Do something on success
    }
}

```

```

        // Default method does NOTHING
    }

    @Override
    public void onError(LWTResult lwtResult) {

        //Get type of LWT operation that fails
        LWTResult.Operation operation = lwtResult.operation();

        // Print out current values
        TypedMap currentValues = lwtResult.currentValues();
        for(Entry<String,Object> entry: currentValues.entrySet()) {
            System.out.println(String.format("%s = %s",entry.getKey(), entry.getValue()))
        }
    }
};

userManager
    .crud()
    .insert(new User(...))
    .ifNotExists()
    .withLWTResultListener(lwtListener)
    .execute();

//OR

userManager
    .crud()
    .insert(new User(...))
    .ifNotExists()
    .withLWTResultListener(lwtResult -> logger.error("Error : " + lwtResult))
    .execute();

```

Using counter type

```

@Table(table = "retweet_count")
public class Retweets {

    @PartitionKey
    @Column("user_id")
    private Long userId;

    @ClusteringColumn(1)
    @Enumerated

```

```

    private LineType type;

    @ClusteringColumn(value = 2, asc = false)
    @TimeUUID
    @Column("tweet_id")
    private UUID tweetId;

    @Counter
    @Column("direct_retweets")
    private Long directRetweets;

    @Counter
    @Column("total_retweets")
    private Long totlRetweets;

    //Getters & Setters
}

```

Once the entity mapping is defined the **CRUD API** for counter tables is restricted to `deleteById()` and `deleteByPartitionKeys()` methods (no `insert()`).

Using materialized views (from Cassandra 3.0.X and after)

To declare a materialized view, use the

`** at MaterializedView**` annotation:

```

@MaterializedView(baseEntity = EntitySensor.class, view = "sensor_by_type")
public class ViewSensorByType {

    @PartitionKey
    @Enumerated
    private SensorType type;

    @ClusteringColumn(1)
    private Long sensorId;

    @ClusteringColumn(2)
    private Long date;

    @Column
    private Double value;
}

```

```

    ...
    //Getters & setters
}

@Table(table = "sensor")
public class EntitySensor {

    @PartitionKey
    private Long sensorId;

    @ClusteringColumn
    private Long date;

    @Enumerated
    @Column
    private SensorType type;

    @Column
    private Double value;

    ...
    //Getters & setters
}

```

The view should reference a base table using the attribute **baseEntity**. It should also re-use the same columns that belong to the base table primary key, possibly in a different order.

Achilles will generate only SELECT APIs for those views, UPDATE and DELETE operations are not possible.

See [Materialized View Mapping](#) for more details

Function mapping (from Cassandra 2.2.X and after)

You can declare the **signature** of your functions in a class/interface so that **Achilles** can generate type-safe API for you to be able to invoke them in the Select DSL API.

For this, use the **@FunctionRegistry** annotation:

For more details, see [Functions Mapping](#)

```

@FunctionRegistry
public interface MyFunctionRegistry {

```

```

    Long convertToLong(String longValue);
}

```

Please note that you'll need to declare your user-defined function by yourself with **Cassandra**, **Achilles** only takes care of the function signature for the code generation, not the function declaration.

Simple object mapping

You can use the **Manager** object for simple object mapping

```

// Execution of custom query
Row row = session.execute(...).one();

User user = userManager.mapFromRow(row);

```

Getting native Session and Cluster object

You can retrieve the native **Session** and **Cluster** object from the **Manager**

```

Session session = userManager.getNativeSession();

Cluster cluster = userManager.getNativeCluster();

```

Generating bound statements/query string from the APIs

Generating `com.datastax.driver.core.BoundStatement`

```

BoundStatement bs = userManager
    .crud()
    ...
    .generateAndGetBoundStatement();

BoundStatement bs = userManager
    .dsl()
    ...
    .generateAndGetBoundStatement();

```

Generating **query string**

```
String statement = userManager
    .crud()
    ...
    .getStatementAsString();

String statement = userManager
    .dsl()
    ...
    .getStatementAsString();
```

Extract bound values from the APIs

Extract **raw bound values**

```
List<Object> boundValues = userManager
    .crud()
    ...
    .getBoundValues();

List<Object> boundValues = userManager
    .dsl()
    ...
    .getBoundValues();
```

Extract **encoded** bound values. The encoding relies on **Achilles [Codec System](#)**

```
List<Object> encodedBoundValues = userManager
    .crud()
    ...
    .getEncodedBoundValues();

List<Object> encodedBoundValues = userManager
    .dsl()
    ...
    .getEncodedBoundValues();
```

Injecting schema name at runtime

Normally you define the keyspace/table name statically using the **@Table** annotation.

However, in a multi-tenant environment, the keyspace/table name is not known ahead of time but only during runtime. For this, **Achilles** defines an interface **SchemaNameProvider**:

```
public interface SchemaNameProvider {

    /**
     * Provide keyspace name for entity class
     */
    <T> String keyspaceFor(Class<T> entityClass);

    /**
     * Provide table name for entity class
     */
    <T> String tableNameFor(Class<T> entityClass);
}
```

You can implement this interface and inject the schema name provider at runtime.

Both **CRUD API** and

DSL API accept dynamic binding of schema name:

```
final SchemaNameProvider dynamicProvider = ...;

userManager
    .crud()
    ...
    .withSchemaNameProvider(dynamicProvider)
    .execute();

userManager
    .dsl()
    .select()
    ...
    .from(dynamicProvider)
    .where()
    ...

userManager
    .dsl()
    .update()
```

```

        .from(dynamicProvider)
        ...
        .where()
        ...

userManager
    .dsl()
    .delete()
    ...
    .from(dynamicProvider)
    ...
    .where()
    ...

```

Generating DDL scripts

Using DML logs

Sometime it is nice to let **Achilles** generate for you the CREATE TABLE script. To do that:

- Set up unit test using [Achilles JUnit test resource](#)
- Activate the DDL logging by setting **DEBUG** level on the logger ACHILLES_DDL_SCRIPT

```

<logger name="ACHILLES_DDL_SCRIPT">
    <level value="DEBUG" />
</logger>

```

Using the SchemaGenerator

Achilles provides a module **achilles-schema-generator** to help you generate CQL schema scripts for your entities. More details [here](#)

Generating DML statements

To debug **Achilles** behavior, you can enable DML statements logging by setting **DEBUG** level on the logger ACHILLES_DML_STATEMENT

```

<logger name="ACHILLES_DML_STATEMENT">
    <level value="DEBUG" />
</logger>

```

Lightweight-Transaction

Achilles offers a complete support for Lightweight Transaction (**LWT**) feature

LWT for insertions

To insert an entity if it does not exist (using the *IF NOT EXISTS* clause in **CQL**):

```
manager
    .crud()
    .insert(new MyEntity(...))
    .ifNotExists()
    .execute();
```

LWT for deletions

To delete an entity if it already exists (using the *IF EXISTS* clause in **CQL**):

On the **CRUD API**

```
manager
    .crud()
    .deleteById(...)
    .ifExists()
    .execute();
```

On the **DSL API**

```
manager
    .dsl()
    .delete()
    ...
    .ifExists()
    .execute();
```

LWT for conditional updates or deletions

```
@Table(table = "user")
public class User {
    @PartitionKey
    private Long id;

    @Column
    private String login;

    @Column
    private String name;

    @Column
    private Integer credits;
}

...

// Conditional updates
manager
    .dsl()
    .update()
    .fromBaseTable()
    .where()
    ...
    .ifExists()
    .execute();

manager
    .dsl()
    .update()
    .fromBaseTable()
    ...
    .credits().Set(...) //set new credits
    ...
    .where()
    .if_Credits().Eq(...) //check for previous credits value
    .execute();

manager
    .dsl()
    .update()
    .fromBaseTable()
    ...
```

```

        .credits().Set(...) //set new credits
        .where()
        ...
        .if_Credits().Gte(...) //check for previous credits value >=
        .execute();

// Conditional delete
manager
    .dsl()
    .delete()
    ....
    .where()
    ...
    .ifExists()
    .execute();

manager
    .dsl()
    .delete()
    ....
    .where()
    ...
    .if_Credits().Eq(...) //check for previous credits value
    .execute();

```

Please note that non equal conditions are only available since **Cassandra 2.1.2**. Those conditions are not supported for prior versions.

LWT Result Listener

To have tighter control on LWT updates or inserts, **Achilles** lets you inject a listener for LWT operations result

```

LWTResultListener lwtListener = new LWTResultListener() {

    default void onSuccess() {
        // Do something on success
        // Default behavior is an no-op
    }

    public void onError(LWTResult lwtResult) {

```

```

        //Get type of LWT operation that fails
        LWTResult.Operation operation = lwtResult.operation();

        // Print out current values
        TypedMap currentValues = lwtResult.currentValues();
        for(Entry<String,Object> entry: currentValues.entrySet()) {
            System.out.println(String.format("%s = %s",entry.getKey(), entry.getValue()))
        }
    }
};

manager
    .dsl()
    .update()
    ...
    .withLwtResultListener(lwtListener)
    .execute();

```

Let's take a concrete example. Suppose you have inserted in the `user` table the following data:

```
INSERT INTO user(id,login,name) VALUES(10,'johndoe','John DOE');
```

Now you try to update with `OptionsBuilder.lwtEqualCondition("login","jdoe")`

```
UPDATE user SET name = 'Johnny DOE' WHERE id=10 IF login='jdoe';
```

```

[applied] | login
-----+-----
      False | johndoe

```

Had you registered a `LWTResultListener`, the returned `TypedMap` of `LWTResult.currentValues()` would contain:

- key: “[applied]”, value: false (boolean)
- key: “login”, value: “johndoe” (String)

Remark: it is possible to register multiple LWT listeners

LWT Serial Consistency

It is also possible to define Serial consistency level at runtime for LWT operations

```
manager
    .dsl()
    .update()
    ...
    .withSerialConsistency(...)
    .execute();
```

Batch-Mode

Support for Batch

Achilles does not offer a direct support for **CQL Batch**. However it is extremely easy

to generate a `com.datastax.driver.core.BoundStatement` using the [Raw Statement Generation](#)

and add it to a `com.datastax.driver.core.Batch` instance yourself.

```
final BatchStatement batch = new BatchStatement();

BoundStatement insert = manager
    .crud()
    .insert(...)
    .generateAndGetBoundStatement();

BoundStatement update = manager
    .dsl()
    .update()
    ...
    .generateAndGetBoundStatement();

batch.add(insert);
batch.add(update);

manager.getNativeSession().execute(batch);
```

Et voila !

JSON-Serialization

Default configuration

It is possible to tell **Achilles** to serialize any unsupported data type into an JSON string using the **@JSON** annotation.

```
//Simple value encoding
@Column
@JSON
private MyPOJO pojoAsJson;

//Collection value encoding
@Column
private List<@JSON MyPOJO> myPojosAsJson;

//Map key encoding
@Column
private Map<@JSON MyPOJO, String> mapKeyEncoding;

//Map value encoding
@Column
private Map<Integer, @JSONMyPOJO> mapValueEncoding;
```

JSON serialization is way faster than the old plain Object serialization since only data are serialized, not class structure.

By default, **Achilles** sets up an internal **Object Mapper** with the following feature config:

1. MapperFeature.SORT_PROPERTIES_ALPHABETICALLY = true
2. SerializationInclusion = JsonInclude.Include.NON_NULL
3. DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES = false
4. AnnotationIntrospector pair : primary = JacksonAnnotationIntrospector, secondary = JaxbAnnotationIntrospector

Jackson will serialize all your entities even if they do not have any **Jackson** annotations. You can also use **JAXB** annotations.

Custom Object Mapper

It is possible to inject a pre-configured **Jackson Object Mapper** as configuration parameter to bootstrap the `ManagerFactory` class using the **OBJECT_MAPPER** parameter.

```
Map<ConfigurationParameters, Object> configMap = new HashMap<>();
configMap.put(JACKSON_MAPPER, preConfiguredObjectMapper);

...
```

Custom Jackson Mapper Factory

Last but not least, it is possible to further custom JSON serialization using the `ObjectMapperFactory` interface and setting the **JACKSON_MAPPER_FACTORY** parameter:

```
public interface JacksonMapperFactory
{
    public <T> ObjectMapper getMapper(Class<T> type);
}

Map<ConfigurationParameters, Object> configMap = new HashMap<>();
configMap.put(JACKSON_MAPPER_FACTORY, customJacksonMapperFactoryImpl);

...
```

When both **JACKSON_MAPPER_FACTORY** and **JACKSON_MAPPER** params are provided for configuration, **Achilles** will ignore the **JACKSON_MAPPER** param and only use the **JACKSON_MAPPER_FACTORY** one

Interceptors

Introduction

To have tighter control on the persistence lifecycle of an entity, you can rely on **Interceptors**. Each interceptor is called upon a *lifecycle event* and modifies the entity state.

Below is the list of all possible *lifecycle events*:

1. PRE_INSERT
2. POST_INSERT
3. PRE_DELETE
4. POST_DELETE
5. POST_LOAD

Usage

To use interceptors, you must provide your own implementation of the interface `Interceptor<T>`:

```
public interface Interceptor<T> {  
  
    boolean acceptEntity(Class<?> entityClass);  
  
    public void onEvent(T entity, Event event);  
  
    public List<Event> events();  
}  
  
public enum Event  
{  
    PRE_INSERT, POST_INSERT, PRE_DELETE, POST_DELETE, POST_LOAD;  
}
```

- The `boolean acceptEntity(Class<?> entityClass)` method should be implemented to filter interceptors.
- The `onEvent(T entity)` method should be implemented to perform functional logic.
The whole raw entity is provided as method argument as well as the current event
- The `List<Event> events()` method defines a list of events this interceptor should be triggered for

Example:

```
public class UserInterceptor extends Interceptor<User>  
{  
  
    public boolean acceptEntity(Class<?> entityClass) {
```

```

        if(entityClass != null) return User.class.equals(entityClass);
        return false;
    }

    public void onEvent(User entity) {
        if(entity.getBiography() == null) {
            entity.setBiography("TO DO");
        }
    }

    public List<Event> events() {
        return Arrays.asList(PRE_INSERT, PRE_UPDATE);
    }
}

```

The above `UserInterceptor` will update the biography to “TO DO” if it is not set before updating/saving the entity.

Configuration

To register your interceptors with **Achilles**, you should pass them as parameter to the configuration parameter map:

```

...
List<Interceptor<?>> interceptors = Arrays.asList(userInterceptor);
configMap.put(EVENT_INTERCEPTORS, interceptors);
...

```

Gotchas

There are some gotchas when using interceptors:

Null-check. Indeed when you craft your own CQL query using the **RAW API**, you may not select all fields to be retrieved.

Consequently the entity will have some null fields when the **POST_LOAD** interceptors are applied.

For safety and to avoid the dreadful `NullPointerException`, it is strongly recommended to perform null checks in the code of your interceptors.

This recommendation also applies to `crud().findById()` method since some fields may be null in **Cassandra**

Primary key It is absolutely a bad idea to modify the primary key in your interceptors.

In most cases there is no sensible reason to do so.

To avoid `NullPointerException`, Achilles will enforce null check (but not value check) on the primary key after each `PRE_INSERT`/`PRE_DELETE` interceptor invocation.

Bean-Validation

Configuration

Achilles allows you to apply Bean Validation (JSR-303) to entities before save, update and load actions.

To activate bean validation, all you need to do is:

- set **`BEAN_VALIDATION_ENABLE`** parameter to **`true`**
- annotate your entities with proper validation annotations
- add a bean validation implementation as dependency of your project.
You can use the reference implementation (**Hibernate Validator**)

A special pre-mutate (INSERT/UPDATE) **Interceptor** is built when bean validation is enabled.

This interceptor will be the **last interceptor** to be triggered if more interceptors are registered for an entity.

This way, you can perform functional logic on the entity before invoking bean validation.

The interceptor is bound to the `PRE_INSERT` and `PRE_UPDATE` events of the entity lifecycle.

Achilles will apply validation on entity having constraints on field, property or class (custom validation).

Right now **Achilles** does not validate method arguments or return values.

On validation error, an **`AchillesBeanValidationException`** will be thrown. You can setup a *try-catch* block around this exception for validation error handling.

POST_LOAD Validation

By default, **Achilles** does not invoke bean validator for the **POST_LOAD** event. To enable **POST_LOAD** bean validation you must set the **POST_LOAD_BEAN_VALIDATION_ENABLE** parameter to true. In this case **Achilles** will register a special **POST_LOAD** interceptor to perform bean validation. Please notice that this interceptor will be the **first** to be invoked.

Please note that the **POST_LOAD_BEAN_VALIDATION_ENABLE** parameter requires the **BEAN_VALIDATION_ENABLE** parameter to be true otherwise it has no effect

Validation group

The default bean validation interceptor will not be invoked for grouped constraints. If you want to use them, you can provide a custom bean validation interceptor.

Below is an example of such interceptor:

```
public class Entity {
    @PartitionKey
    private Long id;

    @Column
    @NotNull(groups = CustomValidationGroup.class)
    private String name;
}

public static class CustomValidationInterceptor implements Interceptor<Entity> {

    private final Validator validator;

    public CustomValidationInterceptor(Validator validator) {
        this.validator = validator;
    }

    @Override
    public boolean accepEntity(Class<?> entityClass) {
        if(entityClass != null) return Entity.class.equals(entityClass);
        return false;
    }
}
```

```

@Override
public void onEvent(Entity entity) {
    Set<ConstraintViolation<Entity>> violations = validator.validate(entity, CustomVa
    ...
}

@Override
public List<Event> events() {
    return asList(PRE_INSERT, PRE_UPDATE);
}
}

```

Extending validation to others lifecycle events

It is possible to extend bean validation to other lifecycle events than `PRE_INSERT` and `PRE_UPDATE`. You just need to provide custom bean validation interceptors. Follow the same approach as above

Multi-Project-Support

Normally, all entity classes and related UDT classes/function registries should be in the same **compilation unit**. If you compile different projects wit **Achilles**, you'll get different versions of `info.archinnov.achilles.generated.ManagerFactory` instance, one for each project.

Now if you import the source code of one project into another, there will be 2 instances of **ManagerFactory** in the classpath and the latest generated instance will shadow the one generated by the imported project.

To solve this issue, just give a different name for each of your project using the `projectName()` attribute of the `@CompileTimeConfig` annotation, **Achilles** will then generate:

- `ManagerFactoryBuilder_For_<project_name>`
- `ManagerFactory_For_<project_name>`

As an example, let's say you have **Project1** and **Project2**.

- Project1
 - UserEntity

- CredentialsEntity
- @CompileTimeConfig(projectName = "Project1")
- Project2
 - ClickStreamEntity
 - ActionEntity
 - @CompileTimeConfig(projectName = "Project2")

To bootstrap **Achilles** for **Project1**:

```
Cluster cluster = Cluster.builder()...build();
ManagerFactory_For_Project1 project1ManagerFactory = ManagerFactoryBuilder_For_Project1
    .builder(cluster)
    .withDefaultKeyspaceName("project1")
    .build();

UserEntity_Manager userManager = project1ManagerFactory.forUserEntity();
CredentialsEntity_Manager credentialsManager = project1ManagerFactory.forCredentialsEntity();
```

To bootstrap **Achilles** for **Project2**:

```
Cluster cluster = Cluster.builder()...build();
ManagerFactory_For_Project2 project2ManagerFactory = ManagerFactoryBuilder_For_Project2
    .builder(cluster)
    .withDefaultKeyspaceName("project2")
    .build();

ClickStream_Manager clickStreamManager = project2ManagerFactory.forClickStreamEntity();
ActionEntity_Manager actionManager = project2ManagerFactory.forActionEntity();
```

With this feature, you can import source code of **Project1** into **Project2** and use them together even if they come from different **compilation units**

Migration-Guide-4-to-5

If you are migrating from **Achilles 4.x** to **Achilles 5.x**, below are the changes and how to transition:

SELECT DSL API

- `where().xxx_Eq(...)`: replaced by `where().xxx().Eq(...)`
- `where().xxx_Gt(...)`: replaced by `where().xxx().Gt(...)` and similarly for all other inequality relations
- you can now select sub-fields of UDTf with `select().udt().allColumns()` or `select().udt().sub-column()`

UPDATE DSL API

- `update().fromBaseTable().xxx_Set(...)`: replaced by `update().fromBaseTable().xxx().Set(...)`
- `update().fromBaseTable().list_AppendTo(...)`: replaced by `update().fromBaseTable().list().AppendTo(...)` and similarly for all other operations on list
- `update().fromBaseTable().set_Add(...)`: replaced by `update().fromBaseTable().set().Add(...)` and similarly for all other operations on set
- `update().fromBaseTable().map_PutTo(...)`: replaced by `update().fromBaseTable().map().PutTo(...)` and similarly for all other operations on map
- `where().xxx_Eq(...)`: replaced by `where().xxx().Eq(...)`
- you can now use the **IN** clause for all clustering columns

DELETE DSL API

- `where().xxx_Eq(...)`: replaced by `where().xxx().Eq(...)` and similarly for all other inequality relations
- you can now use the **IN** clause for all clustering columns

Query DSL

- `manager.query()` has been renamed to `manager.raw()`

Migration-Guide-3-to-4

If you are migrating from **Achilles 3.x** to **Achilles 4.x**, below are the changes and how to transition:

Annotation Changes

- **@Entity**: renamed to **@Table**
- **@CompoundPrimaryKey**: removed, now you can use directly **@PartitionKey** and **@ClusteringColumn** in the entity without having to declare a separated class for the compound primary key
- **@TypeTransformer**: removed, use **@Codec** or **@RuntimeCodec** instead. See [Codec System](#) for more details

Custom Types Changes

- **Counter**: removed, use **@Counter** annotation on a **Long** field instead
- **CounterBuilder**: removed. Now use the **DSL API** to increment/decrement counters
- **Options** and **OptionsBuilder**: removed. Now all options are exposed as methods in [CRUD API](#) and [DSL API](#)
- **IndexCondition**: removed. Will be replaced by a **Index** API later
- **AchillesFuture<T>**: removed and replaced by **CompletableFuture<T>**
- **EntityWithPagingState<T>**: removed, replaced by `com.datastax.driver.core.ExecutionInfo`
- **TypedMapsWithPagingState**: removed.

Dirty Checking And Proxy The feature has been removed because it added more complexity. The flexible [DSL API](#) can cover most of related use-cases

PersistenceManager The old **PersistenceManager** is now replaced by compile-time generated Manager where X is the entity class

PersistenceManager API

- **insert(...)**: replaced by `crud().insert(...)`
- **update(...)**: removed. Feature covered by `dsl().update()`
- **forUpdate(...)**: removed. Feature covered by `dsl().update()`
- **delete(...)**: replaced by `crud().delete(...)`
- **deleteById(...)**: replaced by `crud().deleteById(...)` and `crud().deleteByPartitionKeys(...)`
- **find(...)**: replaced by `crud().findById(...)`
- **initialize(...)**: removed. No more needed because proxy is no longer supported
- **removeProxy(...)**: removed. No more needed because proxy is no longer supported

- **initAndRemoveProxy(...)**: removed. No more needed because proxy is no longer supported
- **serializeToJson(...)**: removed. If needed, use `XXX_AchillesMeta.encode()`

Slice Query API The **Slice Query** API is replaced by the [DSL API](#)

Query API The **Typed Query** API is replaced by `query().typeQueryForSelect(...)`.

See [RAW API](#)

The **Native Query** API is replaced by `query().nativeQuery(...)`. See

[RAW API](#)

The **Indexed Query** API has been removed and will be replaced by a `indexQuery()` API later

Counter API All the **Counter** API and related methods have been removed.

Now just use `dsl().update()` to handle

counter increment/decrement. Counter deletion is achieved using

`crud().deleteById()` or

`crud().deleteByPartitionKeys()` for static counters

Consistency Level The `@Consistency` annotation can now only be used on

a class. Runtime ad-hoc consistency level setting is handled

by the appropriate method for each existing API

Runtime Options Options at runtime are now handled seamlessly by the

appropriate methods for each existing API.

LightWeight Transaction LightWeight Transaction operations are handled

seamlessly by the appropriate methods (`ifNotExists()`,

`ifExists()`, `if_XXX_Eq(...)` ...) on each existing API

Batch Support The Batch mode has been removed. Now **Achilles** relies on

native Java driver **BatchStatement** to handle

batches

Dynamic Schema Update This feature is no longer supported. Schema

changes are **sensitive** operations in production and should be handled

by human operator, not automatically by a software.

Bootstrapping Achilles 3.x:

```
Cluster cluster = Cluster.builder()...build();
PersistenceManagerFactory persistenceManagerFactory = PersistenceManagerFactoryBuilder
    .builder(cluster)
    .withEntityPackages("my.package1,my.package2")
    .withConnectionContactPoints("localhost")
    .withCQLPort(9041)
    .withKeyspaceName("Test Keyspace")
    .forceTableCreation(true).build();
```

Achilles 4.x:

```
Cluster cluster = Cluster.builder()...build();
ManagerFactory managerFactory = ManagerFactoryBuilder
    .builder(cluster)
    .withDefaultKeyspaceName("Test Keyspace")
    .forceTableCreation(true)
    .build();

User_Manager manager = managerFactory.forUser();
```

JUnit Support The old JUnit bootstrap has been changed:

Achilles 3.x:

```
@Rule
public AchillesResource resource = AchillesResourceBuilder
    .withEntityPackages("com.project.entity")
    .withKeyspaceName("myKeyspace")
    .tablesToTruncate("table1","anotherTable")
    .build();
```

Achilles 4.x:

```
@Rule
public AchillesTestResource resource = AchillesTestResourceBuilder
    .forJUnit()
    .withScript("script1.cql")
    .withScript("script2.cql")
    .tablesToTruncate("users", "tweet_lines") // entityClassesToTruncate(User.class, Tweet.class)
```

```

        .createAndUseKeyspace("unit_test")
        .
        ...
        .build((cluster, statementsCache) -> ManagerFactoryBuilder
            .builder(cluster)
            .doForceSchemaCreation(true)
            .withStatementCache(statementsCache) //MANDATORY
            .withDefaultKeyspaceName("achilles_embedded")
            .build()
        );

```

FAQs

1. Why can't I see the class **EntityManagerFactory** ? It does not exist!
 Please **clean**, **recompile** and **refresh** your source code. If you encounter a compilation error, activate debug to access the root cause. It can be done in command line with `mvn clean compile -X`. The `-X` option activate verbose log messages. You may want to redirect the output to a temporary file for analysis later:
`mvn clean compile -X > /tmp/logs.txt`
2. I often have compilation error after updating my entities
 You need to **clean**, **recompile** and **refresh** your source code so that the changes you made triggers new source code generation
3. Any support for property indexing & text search ?
 Technically possible by putting Lucence in the loop but lots of work to do. You're welcomed to help.
 For those who have some money, Datastax offers a custom version of **Apache Solr** with Cassandra under the hood, which boost the performance. Check it out, it's quite nice
4. If I spot a bug in **Achilles**, how can I have it fixed ?
 First, you can report the bug on the [mailing list](#) and in GitHub by creating an issue
 Generally, detected bugs are fixed quite fast. If you want it to go faster, just fix it yourself and propose a pull-request.

5. How can I have new features in **Achilles** ?

Same answer as for point 2. Create a new issue or preferably a pull-request. I will be happy to merge it into the master branch

6. How can I contribute to **Achilles** ?

The most important contribution you can do for **Achilles** is using it extensively and report bugs. Right now the framework is quite mature and most of obvious bug have been spotted & fixed. However there are still corner cases we might have missed.

If you want to propose a pull-request, please respect the following points:

- (a) Each new feature/piece of code added to the code base should be at least unit-tested
- (b) Add a new integration test (IT) whenever possible for the new code