Morphia

<http://mongodb.github.io/morphia/1.1/getting-started/quick-tour/>

MongoDB官方推荐的第三方Java工具排名第一位，第二位是我们都熟悉的SpringData

2016-02-24 by Think QQ:501260519

# [Getting Started](http://mongodb.github.io/morphia/1.1/getting-started/)

To help you get started quickly with Morphia follow:

* [Installation](http://mongodb.github.io/morphia/1.1/getting-started/installation-guide/)
* [Quick Tour](http://mongodb.github.io/morphia/1.1/getting-started/quick-tour/)

## [Installation](http://mongodb.github.io/morphia/1.1/getting-started/installation-guide/)

<dependencies>

<dependency>

<groupId>org.mongodb.morphia</groupId>

<artifactId>morphia</artifactId>

<version>1.0.1</version>

</dependency>

</dependencies>

## [Quick Tour](http://mongodb.github.io/morphia/1.1/getting-started/quick-tour/)

Morphia wraps the MongoDB Java driver so some level of familiarity with using the driver can be helpful. Morphia does its best to abstract much of that away but if something is confusing, please consult the Java driver [documentation](http://mongodb.github.io/mongo-java-driver/) as well.

The following code snippets come from the QuickTour.java example code that can be found with the[Morphia source](https://github.com/mongodb/morphia/blob/master/morphia/src/test/examples/java/org/mongodb/morphia/examples/QuickTour.java).

### Setting up Morphia

The following example shows how to create the initial Morphia instance. Using this instance, you can configure various aspects of how Morphia maps your entities and validates your queries.

**final** Morphia morphia = **new** Morphia();

*// tell Morphia where to find your classes*

*// can be called multiple times with different packages or classes*

morphia.mapPackage(**"org.mongodb.morphia.example"**);

*// create the Datastore connecting to the default port on the local host*

**final** Datastore datastore = morphia.createDatastore(**new** MongoClient(), **"morphia\_example"**);

**datastore.ensureIndexes();**

This snippet creates the Morphia instance we’ll be using in our simple application. The Morphia class exists to configure the Mapper to be used and to define various system-wide defaults. It is also what is used to create the Datastore we’ll be using. The Datastore takes two parameters: the MongoClient used to connect to MongoDB and the name of the database to use. With this approach we could conceivably configure Morphia once and then connect to multiple databases by creating different Datastore instances. In practice, this is likely pretty rare but it is possible.

The second line, which we skipped over, deserves a bit of consideration. In this case, we’re telling Morphia to look at every class in the package we’ve given and find every class annotated with @Entity (which we’ll cover shortly) and discover the mapping metadata we’ve put on our classes. There are several variations of mapping that can be done and they can be called multiple times with different values to properly cover all your entities wherever they might live in your application.

### Mapping Options

Once you have an instance of Morphia, you can configure various mapping options via the MappingOptionsclass. While it’s possible to specify the Mapper when creating an instance of Morphia, most users will use the default mapper. In either case, the Mapper can be fetched using the getMapper() method on theMorphia instance. The two most common elements to configure are storeEmpties and storeNulls. By default Morphia will not store empty List or Map values nor will it store null values in to MongoDB. If your application needs empty or null values to be present for whatever reason, setting these values to true will tell Morphia to save them for you. There are a few other options to configure on MappingOptions, but we’ll not be covering them here.

### Mapping Classes

There are two ways that Morphia can handle your classes: as top level entities or embedded in others. Any class annotated with @Entity is treated as a top level document stored directly in a collection. Any class with @Entity must have a field annotated with @Id to define which field to use as the \_id value in the document written to MongoDB. @Embedded indicates that the class will result in a subdocument inside another document. @Embedded classes do not require the presence of an @Id field.

@Entity(**"employees"**)

@Indexes(

@Index(value = **"salary"**, fields = @Field(**"salary"**))

)

**class** Employee {

@Id

**private** ObjectId id;

**private** String name;

@Reference

**private** Employee manager;

@Reference

**private** List<Employee> directReports;

@Property(**"wage"**)

**private** Double salary;

}

There are a few things here to discuss and others we’ll defer to later sections. This class is annotated using the @Entity annotation so we know that it will be a top level document. In the annotation, you’ll see"employees". By default, Morphia will use the class name as the collection name. If you pass a String instead, it will use that value for the collection name. In this case, all Employee instances will be saved in to the employees collection instead. There is a little more to this annotation but the [annotations guide](http://mongodb.github.io/morphia/1.1/guides/annotations/#entity:b8bcd3c4cba9ac16433f82561ee44461) covers those details.

The @Indexes annotation lists which indexes Morphia should create. In this instance, we’re defining an index named salary on the field salary with the default ordering of ascending. More information on indexing can found [here](http://mongodb.github.io/morphia/1.1/guides/annotations/#indexes:b8bcd3c4cba9ac16433f82561ee44461).

We’ve marked the id field to be used as our primary key (the \_id field in the document). In this instance we’re using the Java driver type of ObjectId as the ID type. The ID can be any type you’d like but is generally something like ObjectId or Long. There are two other annotations to cover but it should be pointed out now that other than transient and static fields, Morphia will attempt to copy every field to a document bound for the database.

The simplest of the two remaining annotations is @Property. This annotation is entirely optional. If you leave this annotation off, Morphia will use the Java field name as the document field name. Often times this is fine. However, some times you’ll want to change the document field name for any number of reasons. In those cases, you can use @Property and pass it the name to be used when this class is serialized out to a document to be handed off to MongoDB.

This just leaves @Reference. This annotation is telling Morphia that this field refers to other Morphia mapped entities. In this case Morphia will store what MongoDB calls a [DBRef](http://docs.mongodb.org/manual/reference/database-references/#dbrefs) which is just a collection name and key value. These referenced entities must already be saved or at least have an ID assigned or Morphia will throw an exception.

### Saving Data

For the most part, you treat your Java objects just like you normally would. When you’re ready to write an object to the database, it’s as simple as this:

**final** Employee elmer = **new** Employee(**"Elmer Fudd"**, 50000.0);

datastore.save(elmer);

Taking it one step further, lets define some relationships and save those, too.

**final** Employee daffy = **new** Employee(**"Daffy Duck"**, 40000.0);

datastore.save(daffy);

**final** Employee pepe = **new** Employee(**"Pepé Le Pew"**, 25000.0);

datastore.save(pepe);

elmer.getDirectReports().add(daffy);

elmer.getDirectReports().add(pepe);

datastore.save(elmer);

As you can see, we just need to create and save the other Employees then we can add them to the direct reports list and save. Morphia takes care of saving the keys in Elmer’s document that refer to Daffy and Pepé. Updating data in MongoDB is as simple as updating your Java objects and then callingdatastore.save() with them again. For bulk updates (e.g., everyone gets a raise!) this is not the most efficient way of doing updates. It is possible to update directly in the database without having to pull in every document, convert to Java objects, update, convert back to a document, and write back to MongoDB. But in order to show you that piece, first we need to see how to query.

### Querying

Morphia attempts to make your queries as type safe as possible. All of the details of converting your data are handled by Morphia directly and only rarely do you need to take additional action. As with everything else, Datastore is where we start:

**final** Query<Employee> query = datastore.createQuery(Employee.class);

**final** List<Employee> employees = query.asList();

This is a basic Morphia query. Here, we’re telling the Datastore to create a query that’s been typed toEmployee. In this case, we’re fetching every Employee in to a List. For very large query results, this could very well be too much to fit in to memory. For this simple example, using asList() is fine but in practicefetch() is usually the more appropriate choice. Most queries will, of course, want to filter the data in some way. There are two ways of doing this:

underpaid = datastore.createQuery(Employee.class)

.field(**"salary"**).lessThanOrEq(30000)

.asList();

The field() method here is used to filter on the named field and returns an instance of an interface with a number of methods to build a query. This approach is helpful is compile-time checking is needed. Between javac failing on missing methods and IDE auto-completion, query building can be done quite safely.

The other approach uses the filter() method which is a little more free form and succinct than field(). Here we can embed certain operators in the query string. While this is less verbose than the alternative, it does leave more things in the string to validate and potentially get wrong:

List<Employee> underpaid = datastore.createQuery(Employee.class)

.filter(**"salary <="**, 30000)

.asList();

Either query works. It comes down to a question of preference in most cases. In either approach, Morphia will validate that there is a field called salary on the Employee class. If you happen to have mapped that field such that the name in the database doesn’t match the Java field, Morphia can use either form and will validate against either name.

### Updates

Now that we can query, however simply, we can turn to in-database updates. These updates take two components: a query, and a set of update operations. In this example, we’ll find all the underpaid employees and give them raise of 10000. The first step is to create the query to find all the underpaid employees. This is one we’ve already seen:

**final** Query<Employee> underPaidQuery =datastore.createQuery(Employee.class)

.filter(**"salary <="**, 30000);

To define how we want to update the documents matched by this query, we create an UpdateOperationsinstance:

**final** UpdateOperations<Employee> updateOperations = datastore.createUpdateOperations(Employee.class)

.inc(**"salary"**, 10000);

There are many operations on this class but, in this case, we’re only updating the salary field by 10000. This corresponds to the [$inc](http://docs.mongodb.org/manual/reference/operator/update/inc/) operator. There’s one last step involved here:

**final** UpdateResults results = datastore.update(underPaidQuery, updateOperations);

This line executes the update in the database without having to pull in however many documents are matched by the query. The UpdateResults instance returned will contain various statistics about the update operation.

## Removes

After everything else, removes are really quite simple. Removing just needs a query to find and delete the documents in question and then tell the Datastore to delete them:

**final** Query<Employee> overPaidQuery = datastore.createQuery(Employee.class)

.filter(**"salary >"**, 100000);

datastore.delete(overPaidQuery);

There are a couple of variations on delete() but this is probably the most common usage. If you already have an object in hand, there is a delete that can take that reference and delete it. There is more information in the [javadoc](http://mongodb.github.io/javadoc).

# Reference Guides

* [Annotations](http://mongodb.github.io/morphia/1.1/guides/annotations/)
* [Querying](http://mongodb.github.io/morphia/1.1/guides/querying/)
* [Aggregation](http://mongodb.github.io/morphia/1.1/guides/aggregation/)
* [Indexing](http://mongodb.github.io/morphia/1.1/guides/indexing/)

## Annotations

Below is a list of all the annotations and a brief description of how to use them.

### Entity

Marks entities to be stored directly in a collection. This annotations is optional in most cases (though this is likely to change in future versions). There is no harm in including it to be more verbose, and make clear the intention for the class. The definition of this annotation looks like this:

**public** @interface Entity {

String value() **default** Mapper.IGNORED\_FIELDNAME;

CappedAt cap() **default** @CappedAt(0);

**boolean** noClassnameStored() **default** **false**;

**boolean** queryNonPrimary() **default** **false**;

String concern() **default** "";

}

|  |  |
| --- | --- |
| Parameter | Usage |
| value() | Defines the collection to use. Defaults to using the classname |
| cap() | Marks this collection as capped and sets the size to use. See the [@Capped](http://mongodb.github.io/morphia/1.1/guides/annotations/#capped)below |
| noClassnameStored() | Tells Morphia to not store the classname in the document. The default is to store the classname. |
| queryNonPrimary() | Indicates that queries against this collection can use secondaries. The default is primary only reads. |
| concern() | The WriteConcern to use when writing to this collection. The default WriteConcern depends on how the MongoClient passed to the Datastore was created. |

### Indexes

In addition to being able to declare an index on a single field you can also declare the indexes at the class level. This allows you to create more than just a single field index; it allows you to create compound indexes with multiple fields.

**public** @interface Indexes {

Index[] value();

}

To see the next few annotations in context, please refer

to [TestIndexCollections.java](https://github.com/mongodb/morphia/blob/master/morphia/src/test/java/org/mongodb/morphia/TestIndexCollections.java) or [TestIndexed.java](https://github.com/mongodb/morphia/blob/master/morphia/src/test/java/org/mongodb/morphia/indexes/TestIndexed.java) in the Morphia source.

### Index

**public** @interface Index {

String value() **default** "";

String name() **default** "";

**boolean** unique() **default** **false**;

**boolean** dropDups() **default** **false**;

**boolean** background() **default** **false**;

**boolean** sparse() **default** **false**;

**boolean** disableValidation() **default** **false**;

**int** expireAfterSeconds() **default** -1;

Field[] fields() **default** {};

IndexOptions options() **default** @IndexOptions();

}

There are two pieces to this annotation that are mutually exclusive. The first group of parameters are considered legacy. They are safe to use but are unlikely to survive past the 1.x series. These options and more have been conglomerated in the @IndexOptions annotation.

|  |  |
| --- | --- |
| Parameter | Usage |
| value() | List of fields (prepended with “-” for desc; defaults to asc). |
| name() | The name of the index |
| unique() | Requires values in the index to be unique |
| dropDups() | Drop any duplicate values during the creation of a unique index. |
| background() | Create this index in the background. There are some [considerations](http://docs.mongodb.org/manual/tutorial/build-indexes-in-the-background/#considerations) to keep in mind. |
| sparse() | Create a [sparse](http://docs.mongodb.org/manual/core/index-sparse/) index |
| disableValidation() | By default, Morphia will validate field names being index. This disables those checks. |
| expireAfterSeconds() | Creates a [TTL Index](http://docs.mongodb.org/manual/core/index-ttl/) on a date field. |
| fields() | This is the new way to define which fields to index. [Details](http://mongodb.github.io/morphia/1.1/guides/annotations/#Field) can be found below. |
| options() | This is the new way to define index options. [Details](http://mongodb.github.io/morphia/1.1/guides/annotations/#IndexOptions) can be found below. |

### Field

**public** @interface Field {

String value();

IndexType type() **default** IndexType.ASC;

**int** weight() **default** -1;

}

|  |  |
| --- | --- |
| Parameter | Usage |
| value() | The field to include in the index |
| type() | The type of index to create. This includes the following values: ASC, DESC, GEO2D,GEO2DSPHERE, TEXT |
| weight() | When defining a text index, this is the weight to apply |

### IndexOptions

**public** @interface IndexOptions {

String name() **default** "";

**boolean** unique() **default** **false**;

**boolean** dropDups() **default** **false**;

**boolean** background() **default** **false**;

**boolean** sparse() **default** **false**;

**boolean** disableValidation() **default** **false**;

**int** expireAfterSeconds() **default** -1;

String language() **default** "";

String languageOverride() **default** "";

}

|  |  |
| --- | --- |
| Parameter | Usage |
| name() | The name of the index |
| unique() | Requires values in the index to be unique |
| dropDups() | Drop any duplicate values during the creation of a unique index. |
| background() | Create this index in the background. There are some [considerations](http://docs.mongodb.org/manual/tutorial/build-indexes-in-the-background/#considerations) to keep in mind. |
| sparse() | Create a [sparse](http://docs.mongodb.org/manual/core/index-sparse/) index |
| expireAfterSeconds() | Creates a [TTL Index](http://docs.mongodb.org/manual/core/index-ttl/) on a date field. |
| disableValidation() | By default, Morphia will validate field names being index. This disables those checks. |
| language() | Default language for the index. |
| languageOverride() | The field in the document to use to override the default language. |

### Indexed

Applied to a Java field, marks the field to be indexed by MongoDB.

**public** @interface Indexed {

IndexDirection value() **default** IndexDirection.ASC;

String name() **default** "";

**boolean** unique() **default** **false**;

**boolean** dropDups() **default** **false**;

**boolean** background() **default** **false**;

**boolean** sparse() **default** **false**;

**int** expireAfterSeconds() **default** -1;

}

|  |  |
| --- | --- |
| Parameter | Usage |
| value() | The sort direction for the index |
| name() | The name of the index |
| unique() | Requires values in the index to be unique |
| dropDups() | Drop any duplicate values during the creation of a unique index. |
| background() | Create this index in the background. There are some [considerations](http://docs.mongodb.org/manual/tutorial/build-indexes-in-the-background/#considerations) to keep in mind. |
| sparse() | Create a [sparse](http://docs.mongodb.org/manual/core/index-sparse/) index |
| expireAfterSeconds() | Creates a [TTL Index](http://docs.mongodb.org/manual/core/index-ttl/) on a date field. |

### Id

Marks a field in an @Entity to be the “\_id” field in MongoDB.

### Property

An optional annotation instructing Morphia to persist the field in to the document given to MongoDB. By default, the field name is used as the property name. This can be overridden by passing a String with the new name to the annotation.

### Transient

Instructs Morphia to ignore this field when converting an entity to a document. The Java keyword transientcan also be used instead.

### Serialized

Instructs Morphia to serialize this field using JDK serialization. The field’s value gets converted to a byte[]and passed off to MongoDB.

**public** @interface Serialized {

String value() **default** Mapper.IGNORED\_FIELDNAME;

**boolean** disableCompression() **default** **false**;

}

|  |  |
| --- | --- |
| Parameter | Usage |
| value() | the field name to use in the document |
| disableCompression() | By default, Morphia compresses the byte[] after serialization. Setting this to true disables the compression. |

### NotSaved

Instructs Morphia to ignore this field when saving but will still be loaded.

Good for data migration.

### AlsoLoad

When a field gets remapped to a new name, you can either update the database and migrate all the fields at once or use this annotation to tell Morphia what older names to try if the current one fails. It is an error to have values under both the old and new key names when loading a document.

Good for data migration.

**public** @interface AlsoLoad {

String[] value();

}

|  |  |
| --- | --- |
| Parameter | Usage |
| value() | The array of names to try when loading the field |

### Version

Marks a field in an @Entity to control optimistic locking for that entity. If the versions change while modifying an entity (including deletes) a ConcurrentModificationException will be thrown. This field will be automatically managed for you – there is no need to set a value and you should not do so anyway. If another name beside the Java field name is desired, a name can be passed to this annotation to change the document’s field name.

@Entity

**class** MyClass {

...

@Version Long v;

}

### Reference

Marks fields as stored in another collection and which are linked (by a DBRef field). When the Entity is loaded, the referenced Entity can also be loaded. Any document referenced via an @Reference field must have already been saved in MongoDB or have the Java object’s @Id already assigned. Otherwise, no key can be copied in to the Key for storage in the database. If you’re always saving the referenced entity in the mapped collection (Datastore can be told to save in to a collection other than the mapped collection) a lot of space can be saved by using the idOnly() parameter to just save the key value.

**public** @interface Reference {

String value() **default** Mapper.IGNORED\_FIELDNAME;

Class<?> concreteClass() **default** Object.class;

**boolean** ignoreMissing() **default** **false**;

**boolean** lazy() **default** **false**;

**boolean** idOnly() **default** **false**;

}

|  |  |
| --- | --- |
| Parameter | Usage |
| value() | The field name to use in the document. Defaults to the Java field name. |
| ignoreMissing() | Ignore any missing documents |
| lazy() | Instructs Morphia to defer loading of the referenced document. |
| idOnly() | Instructs Morphia to only store the key of the referenced document rather than a fullDBRef |

### Embedded

In contrast to @Reference where a nested Java reference ends up as a separate document in a collection,@Embedded tells Morphia to embed the document created from the Java object in the document of the parent object. This annotation can be applied to the class of the embedded type or on the field holding the embedded instance.

**public** @interface Embedded {

String value() **default** Mapper.IGNORED\_FIELDNAME;

Class<?> concreteClass() **default** Object.class;

}

|  |  |
| --- | --- |
| Parameter | Usage |
| value() | The field name to use in the document. Defaults to the Java field name. |
| concreteClass() | The concrete class to use when instantiating the embedded entity |

### Lifecycle Annotations

There are various annotations which can be used to register callbacks on certain lifecycle events. These include Pre/Post-Persist (Save), and Pre/Post-Load.

* @PreLoad - Called before mapping the datastore object to the entity (POJO); the DBObject is passed as an argument (you can add/remove/change values)
* @PostLoad - Called after mapping to the entity
* @PrePersist - Called before save, it can return a DBObject in place of an empty one.
* @PostSave - Called before the save call to the datastore
* @PostPersist - Called after the save call to the datastore

### Examples

[This](https://github.com/mongodb/morphia/blob/master/morphia/src/test/java/org/mongodb/morphia/TestQuery.java#L63) is one of the test classes.

All parameters and return values are optional in your implemented methods.

#### @PrePersist

Here is a simple example of an entity that always saves the Date it was last updated at.

**class** BankAccount {

@Id String id;

Date lastUpdated = **new** Date();

@PrePersist **void** prePersist() {lastUpdated = **new** Date();}

}

#### @EntityListeners

In addition, you can separate the lifecycle event implementation in an external class, or many.

@EntityListeners(BackAccountWatcher.class)

**public** **class** BankAccount {

@Id String id;

Date lastUpdated = **new** Date();

}

**class** BankAccountWatcher{

@PrePersist **void** prePersist(BankAccount act) {act.lastUpdated = **new** Date();}

}

## Querying

Morphia offers a fluent API with which to build up a query and map the results back to instances of your entity classes. It attempts to provide as much type safety and validation as possible. To this end, Morphia offers the Query<T> class which can be parameterized to the type of your entity.

### Creating a Query

The Datastore is the key class when using Morphia. Virtually all operations begin with the Datastore. To create the Query, we invoke the following code:

Query<Product> query = datastore.createQuery(Product.class);

createQuery() returns an instance of Query with which we can build a query.

### filter()

The first method of interest is filter(). This method takes two values: a condition string and a value. Thevalue parameter is, of course, the value to use when applying the condition clause. The conditionparameter is a bit more complicated. At its simplest, the condition is just a field name. In this case, the condition is assumed to be an [equality](http://docs.mongodb.org/manual/reference/operator/query/eq/) check. There is a slightly more complicated variant, however.

The condition value can also contain an operator. For example, to compare a numeric field against a value, you might write something like this:

query.filter(**"price >="**, 1000);

In this case, we’re instructing Morphia to add a filter using [$gte](http://docs.mongodb.org/manual/reference/operator/query/gte/). This would result in a query that looks like this:

{ price: { $gte: 1000 } }

The list of supported filter operations can be found in the [FilterOperator](https://github.com/mongodb/morphia/blob/master/morphia/src/main/java/org/mongodb/morphia/query/FilterOperator.java) class.

|  |  |
| --- | --- |
| Operator | Alias |
| $center |  |
| $centerSphere |  |
| $box |  |
| $eq | =, == |
| $ne | !=, <> |
| $gt | > |
| $gte | >= |
| $lt | < |
| $lte | <= |
| $exists | exists |
| $type | type |
| $not |  |
| $mod | mod |
| $size | size |
| $in | in |
| $nin | nin |
| $all | all |
| $elemMatch | elem, elemMatch |
| $where |  |
| $near | near |
| $nearSphere |  |
| $within (deprecated replaced by $geoWithin) | within |
| $geoNear | geoNear |
| $geoWithin | geoWithin |
| $geoIntersects | geoIntersects |

Each filter operator can either be referenced by its MongoDB “dollar operator” or by the aliases listed afterward. For example, with the equal operator, you can use the canonical $eq operator as you would when building a query in the shell or you could opt to use either the = or == aliases which might feel a little more natural to use than the dollar operators.

### field()

For those who would prefer more compile time validation of their queries, there is field(). This method takes only the field name and returns an instance of a [class](https://github.com/mongodb/morphia/blob/master/morphia/src/main/java/org/mongodb/morphia/query/FieldEnd.java) providing methods with which to define your filters. This approach is slightly more verbose but can be validated by the compiler to a much greater degree than filter() can be. To perform the same query as above, you’d write this:

query.field(**"price"**).greaterThanOrEq(1000);

This results in the exact same query as the filter() version but has the advantage that any typo in the operation name (method in this case) would easily be caught by an IDE or compiler. Which version you use is largely a question of preference.

##### NOTE

**Regardless of the approach used, the field name given can be either the Java field name or the document field name as defined by the**@Property**annotation on the field. Morphia will normalize the name and validate the name such that a query with a bad field name will result in an error.**

### Complex Queries

Of course, queries are usually more complex than single field comparisons. Morphia offers both and() andor() to build up more complex queries. An and query might look something like this:

q.and(

q.criteria(**"width"**).equal(10),

q.criteria(**"height"**).equal(1)

);

An or clause looks exactly the same except for using or() instead of and(), of course. For these clauses we use the criteria() method instead of field() but it is used in much the same fashion. and() andor() take a [varargs](https://docs.oracle.com/javase/8/docs/technotes/guides/language/varargs.html) parameter of type Criteria so you can include as many filters as necessary. If all you need is an and clause, you don’t need an explicit call to and():

datastore.createQuery(UserLocation.class)

.field(**"x"**).lessThan(5)

.field(**"y"**).greaterThan(4)

.field(**"z"**).greaterThan(10);

This generates an implicit and across the field comparisons.

### Text Searching

Morphia also supports MongoDB’s text search capabilities. In order to execute a text search against a collection, the collection must have a [text index](http://docs.mongodb.org/manual/core/index-text/) defined first. Using Morphia that definition would look like this:

@Indexes(@Index(fields = @Field(value = **"$\*\*"**, type = IndexType.TEXT)))

**public** **static** **class** Greeting {

@Id

**private** ObjectId id;

**private** String value;

**private** String language;

...

}

The $\*\* value tells MongoDB to create a text index on all the text fields in a document. A more targeted index can be created, if desired, by explicitly listing which fields to index. Once the index is defined, we can start querying against it like this [test](https://github.com/mongodb/morphia/blob/master/morphia/src/test/java/org/mongodb/morphia/query/TestTextSearching.java) does:

morphia.map(Greeting.class);

datastore.ensureIndexes();

datastore.save(**new** Greeting(**"good morning"**, **"english"**));

datastore.save(**new** Greeting(**"good afternoon"**, **"english"**));

datastore.save(**new** Greeting(**"good night"**, **"english"**));

datastore.save(**new** Greeting(**"good riddance"**, **"english"**));

datastore.save(**new** Greeting(**"guten Morgen"**, **"german"**));

datastore.save(**new** Greeting(**"guten Tag"**, **"german"**));

datastore.save(**new** Greeting(**"guten Nacht"**, **"german"**));

List<Greeting> good = datastore.createQuery(Greeting.class)

.search(**"good"**)

.order(**"\_id"**)

.asList();

Assert.assertEquals(4, good.size());

As you can see here, we create Greeting objects for multiple languages. In our test query, we’re looking for occurrences of the word “good” in any document. We created four such documents and our query returns exactly those four.

### Other Query Options

There is more to querying than simply filtering against different document values. Listed below are some of the options for modifying the query results in different ways.

#### Projections

[Projections](http://docs.mongodb.org/manual/tutorial/project-fields-from-query-results/) allow you to return only a subset of the fields in a document. This is useful when you need to only return a smaller view of a larger object. Borrowing from the [unit tests](https://github.com/mongodb/morphia/blob/master/morphia/src/test/java/org/mongodb/morphia/TestQuery.java), this is an example of this feature in action:

ContainsRenamedFields user = **new** ContainsRenamedFields(**"Frank"**, **"Zappa"**);

getDs().save(user);

ContainsRenamedFields found = getDs()

.find(ContainsRenamedFields.class)

.retrievedFields(**true**, **"first\_name"**).get();

Assert.assertNotNull(found.firstName);

Assert.assertNull(found.lastName);

found = getDs()

.find(ContainsRenamedFields.class)

.retrievedFields(**true**, **"firstName"**).get();

Assert.assertNotNull(found.firstName);

Assert.assertNull(found.lastName);

As you can see here, we’re saving this entity with a first and last name but our query only returns the first name (and the \_id value) in the returned instance of our type. It’s also worth noting that this project works with both the mapped document field name "first\_name" and the Java field name "firstName".

The boolean value passed in to the first parameter instructs Morphia to either include (true) or exclude (false) the fields listed. The second parameter is a [varargs](https://docs.oracle.com/javase/8/docs/technotes/guides/language/varargs.html) String parameter defining which fields are to be either included or excluded in the query results. It is not currently possible to list both inclusions and exclusions in one query.

##### IMPORTANT

**While projections can be a nice performance win in some cases, it’s important to note that this object can not be safely saved back to MongoDB. Any fields in the existing document in the database that are missing from the entity will be removed if this entity is saved. For example, in the example above if**found**is saved back to MongoDB, the**last\_name**field that currently exists in the database for this entity will be removed.**

#### Limiting and Skipping

Pagination of query results is often done as a combination of skips and limits. Morphia offersQuery.limit(int) and Query.offset(int) for these cases. An example of these methods in action would look like this:

datastore.createQuery(Person.class)

.offset(1)

.limit(10)

.asList()

This query will skip the first element and take up to the next 10 items found by the query. There’s a caveat to using skip/limit for pagination, however. See the [skip](http://docs.mongodb.org/manual/reference/method/cursor.skip) documentation for more detail.

#### Ordering

Ordering the results of a query is done via [Query.order(String)](http://mongodb.github.io/javadoc/org/mongodb/morphia/query/Query.html#order-java.lang.String-) . The javadoc has complete examples but this String consists of a list of comma delimited fields to order by. To reverse the sort order for a particular field simply prefix that field with a -. For example, to sort by age (youngest to oldest) and then income (highest to lowest), you would use this:

query.order(**"age,-income"**);

### Tailable Cursors

If you have a [capped collection](http://docs.mongodb.org/manual/core/capped-collections/) it’s possible to “tail” a query so that when new documents are added to the collection that match your query, they’ll be returned by the [tailable cursor](http://docs.mongodb.org/manual/reference/glossary/#term-tailable-cursor). An example of this feature in action can be found in the [unit tests](https://github.com/mongodb/morphia/blob/master/morphia/src/test/java/org/mongodb/morphia/TestQuery.java) in the testTailableCursors() test:

getMorphia().map(CappedPic.class);

getDs().ensureCaps(); *// #1*

**final** Query<CappedPic> query = getDs().createQuery(CappedPic.class);

**final** List<CappedPic> found = **new** ArrayList<CappedPic>();

**final** Iterator<CappedPic> tail = query.tail();

**while**(found.size() < 10) {

found.add(tail.next()); *// #2*

}

There are two things to note about this code sample:

1. This tells Morphia to make sure that any entity [configured](http://mongodb.github.io/guides/annotations/#entity) to use a capped collection has its collection created correctly. If the collection already exists and is not capped, you will have to manually [update](http://docs.mongodb.org/manual/core/capped-collections/#convert-a-collection-to-capped)your collection to be a capped collection.
2. Since this Iterator is backed by a tailable cursor, hasNext() and next() will block until a new item is found. In this version of the unit test, we tail the cursor waiting to pull out objects until we have 10 of them and then proceed with the rest of the application.

## Aggregation

The [aggregation framework](http://docs.mongodb.org/manual/aggregation) in MongoDB allows you to define a series (called a pipeline) of operations (called stages) against the data in a collection. These pipelines can be used for analytics or they can be used to convert your data from one form to another. This guide will not go in to the details of how aggregation works, however. The official MongoDB [documentation](http://docs.mongodb.org/manual/aggregation) has extensive tutorials on such details. Rather, this guide will focus on the Morphia API. The examples shown here are taken from the [tests](https://github.com/mongodb/morphia/blob/master/morphia/src/test/java/org/mongodb/morphia/aggregation/AggregationTest.java) in Morphia itself.

Writing an aggregation pipeline starts just like writing a standard query. As with querying, we start with theDatastore:

Iterator<Author> aggregate = datastore.createAggregation(Book.class)

.group(**"author"**, grouping(**"books"**, push(**"title"**)))

.out(Author.class, options);

createAggregation() takes a Class literal. This lets Morphia know which collection to perform this aggregation against. Because of the transformational operations available in the aggregation [pipeline](http://docs.mongodb.org/manual/core/aggregation-pipeline), Morphia can not validate as much as it can with querying so care will need to be taken to ensure document fields actually exist when referencing them in your pipeline.

### The Pipeline

Aggregation operations are comprised of a series stages. Our example here has only one stage: group(). This method is the Morphia equivalent of the [$group](http://docs.mongodb.org/manual/reference/operator/aggregation/group/) operator. This stage, as the name suggests, groups together documents based on the given field’s values. In this example, we are collecting together all the books by author. The first parameter to group() defines the \_id of the resulting documents. Within this grouping, this pipeline takes the books fields for each author and extracts the title. With this grouping of data, we’re then push()ing the titles in to an array in the final document. This example is the Morphia equivalent of an [example](http://docs.mongodb.org/manual/reference/operator/aggregation/group/#group-title-by-author) found in the aggregation tutorials. This results in a series of documents that look like this:

{ "**\_id**" : **"Homer"**, "**books**" : **[ "The Odyssey", "Iliad" ]** }

{ "**\_id**" : **"Dante"**, "**books**" : **[ "The Banquet", "Divine Comedy", "Eclogues" ]** }

### Executing the Pipeline

There are two basic ways to execute an aggregation pipeline: aggregate() and out(). These methods are Morphia’s cues to send the pipeline to MongoDB for execution. In that regard, both are similar. In practice, how the results are processed is even very similar. The differences, however, can have huge implications on the performance of your application. aggregate() by default will use the ‘inline’ method for returning the aggregation results. This approach has the same 16MB limitation that all documents in MongoDB share. We can changes this behavior using the [AggregationOptions](http://api.mongodb.org/java/3.0/com/mongodb/AggregationOptions.html) class. The optionsreference we passed to out() also applies to aggregate().

### Aggregation Options

There are a handful options here but there’s one that deserves some extra attention. As mentioned, the aggregation pipeline, by default, returns everything “inline” but as of MongoDB 2.6 you can tell the aggregation framework to return a cursor instead. This is what the value of[AggregationOptions#getOutputMode()](http://api.mongodb.org/java/3.0/com/mongodb/AggregationOptions.html#getOutputMode--) determines. By setting the output mode to CURSOR, MongoDB can return a result size much larger than 16MB. The options can also be configured to update the batch size or to set the time out threshold after which an aggregation will fail. It is also possible to tell the aggregation framework to use disk space which allows, among other things, sorting of larger data sets than what can fit in memory on the server.

### $out

But this example doesn’t use aggregate(), of course, it uses out() which gives us access to the $outpipeline stage. [$out](http://docs.mongodb.org/manual/reference/operator/aggregation/out/) is a new operator in MongoDB 2.6 that allows the results of a pipeline to be stored in to a named collection. This collection can not be sharded or a capped collection, however. This collection, if it does not exist, will be created upon execution of the pipeline.

##### IMPORTANT

Any existing data in the collection will be replaced by the output of the aggregation.

Using out() is implicitly asking for the results to be returned via a cursor. What is happening under the covers is the aggregation framework is writing out to the collection and is done. Morphia goes one extra step further and executes an implicit find on the output collection and returns a cursor for all the documents in the collection. In practice, this behaves no differently than setting the output mode to CURSOR withaggregate() and your application need not know the difference. It does, of course, have an impact on your database and any existing data. The use of $out and out() can be greatly beneficial in scenarios such as precomputed aggregated results for later retrieval.

### Typed Results

out() has several variants. In this example, we’re passing in Author.class which tells Morphia that we want to map each document returned to an instance of Author. Because we’re using out() instead ofaggregate(), Morphia will use the mapped collection for Author as the output collection for the pipeline. If you’d like to use an alternate collection but still return a cursor of Author instances, you can use[out(String,Class,AggregationOptions)](http://mongodb.github.io/javadoc/org/mongodb/morphia/aggregation/AggregationPipeline.html#out-java.lang.String-java.lang.Class-com.mongodb.AggregationOptions-) instead.

## Indexing

Morphia provides annotations that allow developers to define indexes for a collection to be defined alongside the other mapping data on an entity’s source. In addition to the familiar ascending/descending index types, Morphia and MongoDB support [TTL](http://docs.mongodb.org/manual/core/index-ttl/), [text](http://docs.mongodb.org/manual/core/index-text/), and [geospatial](http://docs.mongodb.org/manual/applications/geospatial-indexes/) indexes. When defining [text](http://mongodb.github.io/morphia/1.1/guides/indexing/#text-indexing) indexes there are certain restrictions which will be covered below. Full details for all these types are available in the[manual](http://docs.mongodb.org/manual/indexes).

There are two ways to define indexes: at the class level and at the field level.

### Class Level Indexes

Class level indexing begins with the [@Indexes](http://mongodb.github.io/morphia/1.1/guides/annotations/#indexes) annotation. This is a container annotation whose sole purpose is to hold a number of [@Index](http://mongodb.github.io/morphia/1.1/guides/annotations/#index) annotations. This annotation has two primary components to cover here: fields and options. An index definition would take the following form:

@Entity

@Indexes({

@Index(fields = @Field(**"field2"**, type = DESC)),

@Index(fields = @Field(**"field3"**, options = @IndexOptions(name = **"indexing\_test"**)))

})

**public** **class** IndexExample {

@Id

**private** ObjectId id;

**private** String field;

@Property

**private** String field2;

@Property(**"f3"**)

**private** String field3;

}

### Fields

The fields to use in an index definition are defined with the [@Field](http://mongodb.github.io/morphia/1.1/guides/annotations/#field) annotation. An arbitrary number of@Fields can be given but at least one must be present.

#### value()

Indicates which field to use for indexing. The name used for the field can be either the Java field name or the mapped document field name as defined in the class’s mapping via, e.g., the [@Property](http://mongodb.github.io/morphia/1.1/guides/annotations/#property) or [@Embedded](http://mongodb.github.io/morphia/1.1/guides/annotations/#embedded)annotations. For most index types, this value is validated by default. An exception is made for [text indexes](http://mongodb.github.io/morphia/1.1/guides/indexing/#text-indexing)as discussed below.

#### type()

Default: IndexType.ASC

Indicates the “type” of the index (ascending, descending, geo2D, geo2d sphere, or text) to create on the field.

See[*IndexType*](http://mongodb.github.io/javadoc/org/mongodb/morphia/utils/IndexType.html)

#### weight()

Optional

Specifies the weight to use when creating a text index. This value only makes sense when direction isIndexType.TEXT.

### Index Options

Options for an index are defined on the [@IndexOptions](http://mongodb.github.io/morphia/1.1/guides/annotations/#indexoptions). More complete coverage can be found in the[manual](http://docs.mongodb.org/manual/reference/method/db.collection.createIndex/#options) but we’ll provide some basic coverage here as well.

#### background()

Default: false

This value determines if the index build is a blocking call or not. By default, creating an index blocks all other operations on a database. When building an index on a collection, the database that holds the collection is unavailable for read or write operations until the index build completes. For potentially long running index building operations, consider the **background** operation so that the MongoDB database remains available during the index building operation. The MongoDB [manual](http://docs.mongodb.org/manual/core/index-creation/#background-construction) has more detail.

#### disableValidation()

Default: false

When ensuring indexes in the database, Morphia will attempt to ensure that the field names match either the Java field names or the mapped document names. Setting this to true disables this validation.

#### dropDups()

Default: false

When defining a [unique](http://mongodb.github.io/morphia/1.1/guides/indexing/#unique) index, if there are duplicate values found, the index creation will. Setting this value to true will instruct MongoDB to drop the documents with duplicate values.

##### IMPORTANT

As of MongoDB version 3.0, the dropDups option is no longer available.

#### expireAfterSeconds()

Optional

Specifies a value, in seconds, as a [TTL](http://docs.mongodb.org/manual/core/index-ttl/) to control how long MongoDB retains documents in this collection. The field listed must contain values that are dates.

#### language()

Optional

For [text indexes](http://mongodb.github.io/morphia/1.1/guides/indexing/#text-indexing), the language that determines the list of stop words and the rules for the stemmer and tokenizer. See [Text Search Languages](http://docs.mongodb.org/manual/reference/text-search-languages/) for the available languages and [Specify a Language for Text Index](http://docs.mongodb.org/manual/tutorial/specify-language-for-text-index)for more information and examples. The default value is **english**.

#### languageOverride()

Optional

For [text indexes](http://mongodb.github.io/morphia/1.1/guides/indexing/#text-indexing), the name of the field in the collection’s documents that contains the override language for the document. The default value is **language**. See [Use any Field to Specify the Language for a Document](http://docs.mongodb.org/manual/tutorial/specify-language-for-text-index/#specify-language-field-text-index-example)for an example.

#### name()

Optional

The name of the index. If unspecified, MongoDB generates an index name by concatenating the names of the indexed fields and the sort order.

Whether user specified or MongoDB generated, index names including their full namespace (i.e. database.collection) cannot be longer than the [Index Name Limit](http://docs.mongodb.org/manual/reference/limits/#Index-Name-Length).

#### sparse()

Default: false

If true, the index only references documents with the specified field. These indexes use less space but behave differently in some situations (particularly sorts). See [Sparse Indexes](http://docs.mongodb.org/manual/core/index-sparse/) for more information.

#### unique()

Default: false

Creates a unique index so that the collection will not accept insertion of documents where the index key or keys match an existing value in the index. Specify true to create a unique index.

### Field Level Indexes

Field level indexing is a simpler approach to defining a basic, single key index. These indexes are defined by applying the [@Indexed](http://mongodb.github.io/morphia/1.1/guides/annotations/#indexed) annotation to a particular field on a class. Because the index definition is applied at the field level, the index is created using only that field and so the [@Field](http://mongodb.github.io/morphia/1.1/guides/annotations/#field) annotations are unnecessary. The options for the index are the same as defined [above](http://mongodb.github.io/morphia/1.1/guides/indexing/#options). A field level index definition would look like this:

@Entity

**private** **class** FieldIndex {

@Id

**private** ObjectId id;

@Indexed(options = @IndexOptions(unique = **true**))

**private** String name;

**private** String color;

}

### Text Indexing

Morphia’s indexing supports MongoDB’s text indexing and search functionality as we’ve briefly seen above. Full details can be found in the [manual](http://docs.mongodb.org/manual/core/index-text/) but there are a few Morphia specific details to cover. Indexed field names are validated by default but validation is disabled when an index is defined using MongoDB’s [$\*\*](http://docs.mongodb.org/manual/core/index-text/#text-index-wildcard)syntax. This special instruction tells MongoDB to create a text index on all fields with string content in a document. A [compound index](http://docs.mongodb.org/manual/core/index-text/#compound-index) can be created incorporating a text index but it’s important to note there can only be one text index on a collection.

A wild card text index declaration would look like this:

@Indexes(@Index(fields = @Field(value = **"$\*\*"**, type = TEXT)))

##### IMPORTANT

A collection can have at most one text index.

# API DOCUMENT

[see](http://mongodb.github.io/morphia/1.1/issues-help/)

# Issues & Help

We are lucky to have a vibrant MongoDB Java community with lots of varying experience of using Morphia. We often find the quickest way to get support for general questions is through the [Morphia google group](https://groups.google.com/forum/#!forum/morphia),[mongodb-user google group](http://groups.google.com/group/mongodb-user), or through [stackoverflow](https://stackoverflow.com/questions/tagged/morphia). Please also refer to our own [support channels](http://www.mongodb.org/about/support)documentation. If you have a question or think you’ve encountered a bug, the mailing list is the place to start.

## Bugs / Feature Requests

If you think you’ve found a bug or want to see a new feature in the Morphia, please open an issue on [github](https://github.com/mongodb/morphia/issues). Please provide as much information as possible (including version numbers) about the issue type and how to reproduce it.

If you’ve identified a security vulnerability in a driver or any other MongoDB project, please report it according to the [instructions here](http://docs.mongodb.org/manual/tutorial/create-a-vulnerability-report).

## Pull Requests

We are happy to accept contributions to help improve Morphia. We will guide user contributions to ensure they meet the standards of the codebase. Please ensure that any pull requests include documentation, tests and also pass a the gradle checks.

To get started check out the source and work on a branch:

$ git clone https://github.com/mongodb/morphia.git

$ cd morphia

$ git checkout -b myNewFeature

Finally, ensure that the code passes gradle checks.

$ ./gradlew check

# Source Code

[See](https://github.com/mongodb/morphia)