Morphia - Java

NSTALLATION GUIDE

The recommended way to get started using Morphia in your project is with a dependency management system.

```
<dependencies>
  <dependency>
    <groupId>dev.morphia</groupId>
    <artifactId>morphia</artifactId>
    <version>1.5.4-SNAPSHOT</version>
  </dependency>
</dependencies>


 dependencies {
    compile 'dev.morphia:morphia:1.5.4-SNAPSHOT'
 }
```

Setting up Morphia

he 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("dev.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 `MappingOptions` class. 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 the `Morphia` 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 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 foundhere.

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` 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 calling `datastore.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 to `Employee`. 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 practice `fetch()` 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 if 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 a 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 `UpdateOperations`instance:

```
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` 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.

ANNOTATIONS

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

Indexes

Indexes can be defined on each field directly for single field indexing or at the class level for compund indexes. To see the next few annotations in context, please refer to TestIndexCollections.java or TestIndexed.java in the Morphia source.

Index

The @Index documentation can be found here. 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 will be removed in the 2.x series. These options and more have been conglomerated in the @IndexOptions annotation.

Field

The @Field annotation defines indexing on a specific document field. Multiple instances of this annotation may be passed to the @Index annotation to define a compound index on multiple fields.

IndexOptions

The @IndexOptions annotation defines the options to apply to an index definition. This annotation replaces the fields found directly on the @Index annotation. This annotation was added to ensure that index options are consistent across the various index definition approaches.

Collation

The @Collation annotation defines the collation options to apply to the index definition. In addition to defining a collation as part of an index, a collation can be specified as part of a query as well. The Options classes provide facilities for specifying a specific collation to be used for any given operation. This collation does not have to match the one defined on the index but will, of course, be faster if it does.
See CountOptions, DeleteOptions, FindOptions, MapReduceOptions, and FindAndModifyOptions for more information.

Indexed

@Indexed, applied to a Java field, marks the field to be indexed by MongoDB. This is used for simple, single-field indexes. As stated above, the `options` value replaces the individual setting values on the `@Indexed` annotation itself.

Entity Mapping

Morphia provides a number of annotations providing for the customization of object mapping.

Entity

@Entity marks entities to be stored directly in a collection. This annotation is optional in most cases but is required if an entity is to be mapped to a specifically named collection. If no mapping is given, the collection is named after the class itself. There are two different mechanisms for mapping cross-object relationships in Morphia: references and embedding.

Reference

@Reference marks a field as a reference to a document stored in another collection and is linked (by a `DBRef`field). When the Entity is loaded, the referenced entity is also be loaded. Any object referenced via an `@Reference`field must have already have a non-null @Id value in the referenced entity. This can be done by either saving the referenced entities first or by manually assigning them ID values. By default, these referenced entities are automatically loaded by Morphia along with the referencing entity. This can result in a high number of database round trips just to load a single entity. To resolve this, `lazy = true` can be passed to the annotation. This will create a dynamic proxy which will lazily load the entity the first time it is referenced in code.

Fields annotated with `@Reference` will show up in MongoDB as `DBRef` fields by default. A `DBRef` stores not only the entity's ID value but also the collection name. In most cases, this is probably redundant information as the collection name is already encoded in the entity's

mapping information. To reduce the amount of storage necessary to track these references, use `idOnly = true` in the mapping. This will result in only the ID value being stored in the document.

Morphia 1.5.0 introduced a new experimental API help with some of the complications involved with the annotation based mapping. As this is an experimental API, details may change and the entire API might be removed. But users are encouraged to experiment with the API and provide any feedback via GitHub Issues. Further documentation can be found here.

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.

Validation

@Validation allows for the definition of a document validation schema to applied to all writes to MongoDB. Validation rules are specified on a per-collection basis using any query operators, with the exception of `$near`, `$nearSphere`, `$text`, and `$where`. This validation definition is done using the MongoDB query syntax as shown here:

```
@Validation("{ number : { $gt : 10 } }")
public class SomeEntity {
   ...
   private int number;
   ...
}
```

Various operations on Datastore and AdvancedDatastore can bypass this validation via their Options classes. For these operations, specify the `bypassDocumentValidation` option to disable document validation for a specific operation.
See InsertOptions, UpdateOptions, MapReduceOptions, and FindAndModifyOptions for more information.

Id

@Id marks a field in an entity to be the `_id` field in MongoDB. This annotation is required on all top level entities regardless of the presence of an @Entity annotation. If a class is marked with @Embedded this annotation is not required since embedded documents are not required to have `_id` fields.

Property

@Property is an optional annotation instructing Morphia to persist the field using the given name in the document saved in 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

@Transient instructs Morphia to ignore this field when converting an entity to a document. The Java keyword `transient` can also be used instead.

Serialized

@Serialized instructs Morphia to serialize this field using JDK serialization. The field's value gets converted to a `byte[]` and passed to MongoDB.

NotSaved

@NotSaved instructs Morphia to ignore this field when saving but will still be loaded from the database when the entity is read.

AlsoLoad

@AlsoLoad instructs Morphia to look for a field under different names than the mapped name. 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. These alternate names are not used in queries, however, so if there are queries against this field they should be updated to use the alternate names as well or the database should be updated such that every instance of the old name is renamed.

Version

@Version marks a field in an entity to control optimistic locking. If the versions change in the database 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. 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.

Lifecycle Annotations

There are various annotations which can be used to register callbacks on certain lifecycle events. These include Pre/Post-Persist, Pre-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.
- `@PreSave` - Called before the save call to the datastore
- `@PostPersist` - Called after the save call to the datastore

Examples

This 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();}

}
```

AGGREGATION

The aggregation framework 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 has extensive tutorials on such details. Rather, this guide will focus on the Morphia API. The examples shown here are taken from the tests in Morphia itself.

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

```
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, 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` 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 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 class. The options reference 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() 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 $out pipeline stage. $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 with aggregate() 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 of aggregate(), 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)` 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, text, and geospatial indexes. When defining text indexes there are certain restrictions which will be covered below. Full details for all these types are available in the manual.

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` annotation. This is a container annotation whose sole purpose is to hold a number of `@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(value = "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` annotation. An arbitrary number of `@Field`s 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` or `@Embedded`annotations. For most index types, this value is validated by default. An exception is made for text indexes 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

weight()

Optional

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

Index Options

Options for an index are defined on the `@IndexOptions`. More complete coverage can be found in the manual 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 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 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 to control how long MongoDB retains documents in this collection. The field listed must contain values that are dates.

language()

Optional

For text indexes, the language that determines the list of stop words and the rules for the stemmer and tokenizer. See Text Search Languages for the available languages and Specify a Language for Text Index for more information and examples. The default value is **english**.

languageOverride()

Optional

For text indexes, 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 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.

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 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.

partialFilter()

Optional

New in MongoDB 3.2, partial indexes only index the documents in a collection that meet a specified filter expression thereby reducing storage and maintenance costs. A partial filter is defined using a query as shown here:

```
@Indexes({@Index(options = @IndexOptions(partialFilter = "{ name : { $exists : true } }"),
      fields = {@Field(value = "name")})})
  public static class SomeClass { ... }
```

collation()

Optional

Collation allows users to specify language-specific rules for string comparison, such as rules for lettercase and accent marks. A collation can be defined using the `collation()` property on `@IndexOptions` and takes an `@Collation` instance.

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` 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` annotations are unnecessary. The options for the index are the same as defined above. 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 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 `$**` syntax. This special instruction tells MongoDB to create a text index on all fields with string content in a document. A 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.

LIFE CYCLE METHODS

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

- `@PrePersist` - Called before save, it can return a `DBObject` in place of an empty one.
- `@PreSave` - Called right before `DBCollection.save()` is called. Changes made to the entity will not be persisted; the `DBObject` can be passed as an argument (you can add/remove/change values)
- `@PostPersist` - Called after the save call to the database
- `@PreLoad` - Called before mapping the document from the database to the entity; the DBObject is passed as an argument (you can add/remove/change values)
- `@PostLoad` - Called after populating the entity with the values from the document

See the annotations guide for a full list of the annotations supported.

Examples

Here is a one of the test classes.

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

Example

Here is a simple example of an entity that always saves the Date it was last updated. Alternatively, the resulting serialized form can be passed back in just prior to sending the document to the database.

```
class BankAccount {
  @Id
  String id;
  Date lastUpdated = new Date();

  @PrePersist
  public void trackUpdate() {
    lastUpdated = new Date();
  }

  @PrePersist
  public void prePersist(final DBObject dbObj) {
    // perform operations on serialized form of the entity
  }
}
```

EntityListeners

If your application has more generalized life cycle events, these methods can be stored on classes external to your model. For example's sake, let's assume there's a need to digitally sign all documents before storing it in the database.

```
@EntityListeners(DigitalSigner.class)
public class BankAccount {
  @Id
  String id;
  Date lastUpdated = new Date();
}

class DigitalSigner {
  @PrePersist
  void prePersist(final Object entity, final DBObject dbObject) {
```

```
    dbObject.put("signature", sign(dbObject));
  }

}
```

No Delete Support

Because deletes are usually done with queries there is no way to support a Delete lifecycle event. If, or when, server-side triggers are enabled there may be some support for this, but even then it will be hard to imagine how this would logically fit.

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. The `value`parameter is, of course, the value to use when applying the `condition` clause. The `condition` parameter 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 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. 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 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

Operator	Alias
$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 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()` and `or()`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()` and `or()` take a `varargs` 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 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 does:

```
morphia.map(Greeting.class);
datastore.ensureIndexes();

datastore.save(new Greeting("good morning", "english"),
   new Greeting("good afternoon", "english"),
   new Greeting("good night", "english"),
   new Greeting("good riddance", "english"),
   new Greeting("guten Morgen", "german"),
   new Greeting("guten Tag", "german")),
   new Greeting("gute 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 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, this is an example of this feature in action:

```
ContainsRenamedFields user = new ContainsRenamedFields("Frank", "Zappa");
getDs().save(user);

ContainsRenamedFields found = getDs()
   .find(ContainsRenamedFields.class)
   .project("first_name", true)
   .get();
Assert.assertNotNull(found.firstName);
Assert.assertNull(found.lastName);

found = getDs()
   .find(ContainsRenamedFields.class)
   .project("firstName", true)
   .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 instructs Morphia to either include (`true`) or exclude (`false`) the field. 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. To save such instances back consider using `Datastore#merge(T)`

Limiting and Skipping

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

```
datastore.createQuery(Person.class)
   .asList(new FindOptions()
               .offset(1)
               .limit(10))
```

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 documentation for more detail.

Ordering

Ordering the results of a query is done via `Query.order(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 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. An example of this feature in action can be found in the unit tests 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
           .fetch(new FindOptions()
                       .cursorType(CursorType.Tailable));
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 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 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.

Raw Querying

You can use Morphia to map queries you might have already written using the raw Java API against your objects, or to access features which are not yet present in Morphia.

For example:

```
DBObject query = BasicDBObjectBuilder.start()
            .add("albums",
        new BasicDBObject("$elemMatch",
             new BasicDBObject("$and", new BasicDBObject[] {
                new BasicDBObject("albumId", albumDto.getAlbumId()),
                new BasicDBObject("album",
                   new BasicDBObject("$exists", false))})))
            .get();

Artist result = datastore.createQuery(Artist.cla
```

REFERENCES

Morphia supports two styles of defining references: the `@Reference` annotation and the experimental `MorphiaReference`. The annotation based approach is discussed here. This guide will cover the wrapper based approach.

important

This API is experimental. Its implementation and API subject to change based on user feedback. However, users are encouraged to experiment with the API and provide as much feedback as possible both positive and negative as this will likely be the approach used going forward.

An alternative to the traditional annotation-based approach is the `MorphiaReference` wrapper. This type can not be instantiated directly. Instead a `wrap()` method is provided that will construct the proper type and track the necessary

state. Currently, four different types of values are supported by `wrap()`: a reference to a single entity, a List of references to entities, a Set, and a Map. `wrap()` will determine how best to handle the type passed and create the appropriate structures internally. This is how this type might be used in practice:

```
private class Author {
   @Id
   private ObjectId id;
   private String name;
   private MorphiaReference<List<Book>> list;
   private MorphiaReference<Set<Book>> set;
   private MorphiaReference<Map<String, Book>> map;

   public Author() { }

   public List<Book> getList() {
      return list.get();
   }

   public void setList(final List<Book> list) {
      this.list = MorphiaReference.wrap(list);
   }

   public Set<Book> getSet() {
      return set.get();
   }

   public void setSet(final Set<Book> set) {
      this.set = MorphiaReference.wrap(set);
   }

   public Map<String, Book> getMap() {
      return map.get();
   }

   public void setMap(final Map<String, Book> map) {
      this.map = MorphiaReference.wrap(map);
   }
}

private class Book {
   @Id
   private ObjectId id;
   private String name;
   private MorphiaReference<Author> author;

   public Book() { }

   public Author getAuthor() {
```

```
        return author.get();
    }

    public void setAuthor(final Author author) {
        this.author = MorphiaReference.wrap(author);
    }
}
```

As you can see we have 3 different references from `Author` to `Book` and one in the opposite direction. It would also be good to note that the public API of those two classes don't expose the `MorphiaReference` externally. This is, of course, a stylistic choice but is the encouraged approach as it avoids leaking out implementation and mapping details outside of your model.

`setList()` accepts a `List<Book>` and stores them as references to `Book` instances stored in the collection as defined by the mapping metadata for `Book`. Because these references point to the mapped collection name for the type, we can get away with storing only the `_id` fields for each book. This gives us data in the database that looks like this:

```
> db.Author.find().pretty()
{
        "_id" : ObjectId("5c3e99276a44c77dfc1b5dbd"),
        "className" : "dev.morphia.mapping.experimental.MorphiaReferenceTest$Author",
        "name" : "Jane Austen",
        "list" : [
                ObjectId("5c3e99276a44c77dfc1b5dbe"),
                ObjectId("5c3e99276a44c77dfc1b5dbf"),
                ObjectId("5c3e99276a44c77dfc1b5dc0"),
                ObjectId("5c3e99276a44c77dfc1b5dc1"),
                ObjectId("5c3e99276a44c77dfc1b5dc2")
        ]
}
```

As you can see, we only need to store the ID values because the collection is already known elsewhere. However, sometimes we need to refer to documents stored in different collections. For example, if the generic type of the reference is a parent interface or class, we sometimes need to store extra information. For these cases, we store the references as full `DBRef` instances so we can track the appropriate collection for each reference. Using this version, we get data in the database that looks like this:

```
> db.jane.find().pretty()
{
        "_id" : ObjectId("5c3e99c06a44c77e5a9b5701"),
        "className" : "dev.morphia.mapping.experimental.MorphiaReferenceTest$Author",
        "name" : "Jane Austen",
        "list" : [
                DBRef("books", ObjectId("5c3e99c06a44c77e5a9b5702")),
```

```
                    DBRef("books", ObjectId("5c3e99c16a44c77e5a9b5703")),
                    DBRef("books", ObjectId("5c3e99c16a44c77e5a9b5704")),
                    DBRef("books", ObjectId("5c3e99c16a44c77e5a9b5705")),
                    DBRef("books", ObjectId("5c3e99c16a44c77e5a9b5706"))
            ]
}
```

In both cases, we have a document field called `list` but as you can see in the second case, we're not storing just the `_id` values but `DBRef` instances storing both the collection name, "books" in this case, and `ObjectId`values from the Books. This lets the wrapper properly reconstitute these references when you're ready to use them.

Note

Before we go too much further, it's important to point that, regardless of the type of the references, they are fetched lazily. So if you multiple fields with referenced entities, they will not be fetched until you call `get()` on the `MorphiaReference`. If the type is a `Collection` or a `Map`, all the referenced entities are fetched and loaded via a single query if possible. This saves on server round trips but does raise the risk of potential `OutOfMemoryError` problems if you load too many objects in to memory this way.

A `Set` of references will look no different in the database than the `List` does. However, `Map`s of references are slightly more complicated. A `Map` might look something like this:

```
> db.Author.find().pretty()
{
            "_id" : ObjectId("5c3e9cad6a44c77fa8f38f58"),
            "className" : "dev.morphia.mapping.experimental.MorphiaReferenceTest$Author",
            "name" : "Jane Austen",
            "map" : {
                        "Sense and Sensibility " : ObjectId("5c3e9cad6a44c77fa8f38f59"),
                        "Pride and Prejudice" : ObjectId("5c3e9cad6a44c77fa8f38f5a"),
                        "Mansfield Park" : ObjectId("5c3e9cad6a44c77fa8f38f5b"),
                        "Emma" : ObjectId("5c3e9cad6a44c77fa8f38f5c"),
                        "Northanger Abbey" : ObjectId("5c3e9cad6a44c77fa8f38f5d")
            }
}
```

References to single entities will follow the same pattern with regards to the `_id` values vs `DBRef` entries.

Note

Currently there is no support for configuring the `ignoreMissing` parameter as there is via the annotation. The wrapper will silently drop missing ID values or return null depending on the type of the reference. Depending on the response to this feature in generalcon

SCHEMA VALIDATION

Morphia provides annotations that allow developers to define document validations for a collection to be defined alongside the other mapping data on an entity's source. Schema validationprovides the capability to perform schema validation during updates and insertions. Validation rules are on a per-collection basis and can be defined via annotations just like indexes are.

Below we have a basic entity definition. Note the new annotation @Validation.

```
@Entity("validation")
@Validation("{ number : { $gt : 10 } }")
public class DocumentValidation {
   @Id
   private ObjectId id;
   private String string;
   private int number;
   private Date date;

   ...
}
```

In this case, only one value is supplied to the annotation. This string value is the query that will be used to match against any new documents or updated documents. Should this query fail to match the new document, validation will fail on the document and it will be rejected. In addition to the required query, there are two other values that can be configured based on your needs: `level` with a default of `STRICT` and `action` with a default of `ERROR`.
A `MODERATE` validation level does not apply rules to updates on existing invalid documents.
An `action`setting of `WARN` will merely log any validation violations.

UPDATING

There are two basic ways to update your data: insert/save a whole Entity or issue an update operation.

Updating (on the server)

The update method on Datastore is used to issue a command to the server to change existing documents. The effects of the update command are defined via UpdateOperations methods.

The Field Expression

The field expression, used by all update operations, can be either a single field name or any dot-notation form (for embedded elements). The positional operator ($) can also be used in the in the field expression for array updates. To illustrate, consider the entity here:

```
@Entity("hotels")
public class Hotel
{
  @Id
  private ObjectId id;

  private String name;
  private int stars;

  @Embedded
  private Address address;

  List<Integer> roomNumbers = new ArrayList<Integer>();

  // ... optional getters and setters
}

@Embedded
public class Address
{
  private String street;
  private String city;
  private String postalCode;
  private String country;

  // ... optional getters and setters
}
```

set()/unset()

To change the name of the hotel, one would use something like this:

```
UpdateOperations ops = datastore
   .createUpdateOperations(Hotel.class)
   .set("name", "Fairmont Chateau Laurier");
datastore.update(updateQuery, ops);
```

This also works for embedded documents. To change the name of the city in the address, one would use something like this:

```
UpdateOperations ops = datastore
   .createUpdateOperations(Hotel.class)
   .set("address.city", "Ottawa");
datastore.update(updateQuery, ops);
```

Values can also be removed from documents as shown below:

```
UpdateOperations ops = datastore
   .createUpdateOperations(Hotel.class)
   .unset("name");
datastore.update(updateQuery, ops);
```

After this update, the name of the hotel would be `null` when the entity is loaded.

inc()/dec()

To simply increment or decrement values in the database, updates like these would be used:

```
// increment 'stars' by 4
UpdateOperations ops = datastore
   .createUpdateOperations(Hotel.class)
   .inc("stars");
datastore.update(updateQuery, ops);

// increment 'stars' by 4
ops = datastore
   .createUpdateOperations(Hotel.class)
   .inc("stars", 4);
datastore.update(updateQuery, ops);

// decrement 'stars' by 1
ops = datastore
   .createUpdateOperations(Hotel.class)
   .dec("stars");  // same as .inc("stars", -1)
datastore.update(updateQuery, ops);

// decrement 'stars' by 4
ops = datastore
   .createUpdateOperations(Hotel.class)
   .inc("stars", -4);
datastore.update(updateQuery, ops);
```

push()/addToSet()

`push()` is used to add a value to an array field:

```
ops = datastore
   .createUpdateOperations(Hotel.class)
   .push("roomNumbers", 11);
datastore.update(updateQuery, ops);
```

This will issue a `$push` operation adding `11` to the list. This might result in duplicated values in this field. If the values should be unique, use `addToSet()` instead:

```
ops = datastore
   .createUpdateOperations(Hotel.class)
   .addToSet("roomNumbers", 11);
datastore.update(updateQuery, ops);
```

`push()` and `addToSet()` can take either single values or a `List` of values. The `push()` methods can also optionally take a PushOptions instance allowing for tweaking how the values are added to the list. See the manual for more information about the various modifiers available.

removeFirst()/removeLast()/removeAll()

To remove values from a list, use `removeFirst()`, `removeLast()`, or `removeAll()`:

```
//given roomNumbers = [ 1, 2, 3 ]
ops = datastore
   .createUpdateOperations(Hotel.class)
   .removeFirst("roomNumbers");
datastore.update(updateQuery, ops);  // [ 2, 3 ]

//given roomNumbers = [ 1, 2, 3 ]
ops = datastore
   .createUpdateOperations(Hotel.class)
   .removeLast("roomNumbers");
datastore.update(updateQuery, ops);  // [ 1, 2 ]
ops = datastore
   .createUpdateOperations(Hotel.class)
   .removeLast("roomNumbers");
datastore.update(updateQuery, ops);  // [ 1 ]
ops = datastore
   .createUpdateOperations(Hotel.class)
   .removeLast("roomNumbers");
datastore.update(updateQuery, ops);  // []   empty array

//given roomNumbers = [ 1, 2, 3, 3 ]
ops = datastore
   .createUpdateOperations(Hotel.class)
   .removeAll("roomNumbers", 3);
datastore.update(updateQuery, ops);  // [ 1, 2 ]

//given roomNumbers = [ 1, 2, 3, 3 ]
ops = datastore
   .createUpdateOperations(Hotel.class)
   .removeAll("roomNumbers", Arrays.asList(2, 3));
datastore.update(updateQuery, ops);  // [ 1 ]
```

updateFirst()

In the default driver and shell this is the default behavior. In Morphia we feel like updating all the results of the query is a better default (see below).

```
  {
    name: "Fairmont",
    stars: 5
  },
  {
    name: "Last Chance",
    stars: 3
  }
ops = datastore.createUpdateOperations(Hotel.class).inc("stars", 50);

// morphia exposes a specific updateFirst to update only the first hotel matching the query
datastore
  .updateFirst(datastore
    .find(Hotel.class)
    .order("stars"),
    ops);  // update only Last Chance
datastore
  .updateFirst(datastore
    .find(Hotel.class)
    .order("-stars"),
    ops); // update only Fairmont
```

Multiple Operations

You can also perform multiple update operations within a single update.

```
//set city to Ottawa and increment stars by 1
ops = datastore
  .createUpdateOperations(Hotel.class)
  .set("city", "Ottawa")
  .inc("stars");
datastore.update(updateQuery, ops);

//if you perform multiple operations in one command on the same property, results will vary
ops = datastore
  .createUpdateOperations(Hotel.class)
  .inc("stars", 50)
  .inc("stars");  //increments by 1
ops = datastore
  .createUpdateOperations(Hotel.class)
  .inc("stars")
  .inc("stars", 50);  //increments by 50
```

```
//you can't apply conflicting operations to the same property
ops = datastore
   .createUpdateOperations(Hotel.class)
   .set("stars", 1)
   .inc("stars", 50); //causes error
```

createIfMissing (overload parameter)

All of the update methods on `Datastore` are overloaded and accept a `createIfMissing` parameter

```
ops = datastore
   .createUpdateOperations(Hotel.class)
   .inc("stars", 50);

//update, if not found create it
datastore
   .updateFirst(datastore
      .createQuery(Hotel.class)
      .field("stars").greaterThan(100),
   ops, true);

// creates { "_id" : ObjectId("4c60629d2f1200000000161d"), "stars" : 50 }
```

VALIDATION EXTENSION

This is a simple extension to Morphia to process JSR 303 Validation Annotations.

Using

Add this at the start of your application (or wherever you create your morphia instances).

```
new ValidationExtension(morphia);
```

Example

Here is a simple example using (as an example) Hibernate validation:

```
...
import org.hibernate.validator.constraints.Email;
...

@Entity
public class Userlike {
        @Id ObjectId id;
```

```
        @Email String email;
}
```

Implementation

This is a lightweight wrapper around the JSR 303 API. It installs a simple global entity interceptor which listens to all life cycle methods needed for validation. You can use any implementation of JSR 303 by just adding it to the classpath.

You can look at the code here.

Dependencies

Manual

- Hibernate Validator

Maven

If you use Maven to manage your project, you can reference Morphia as a dependency:

```
<dependency>
  <groupId>dev.morphia.morphia</groupId>
  <artifactId>validation</artifactId>
  <version>1.5.4-SNAPSHOT</version>
</dependency>
```