This page last changed on Jan 21, 2012 by patrik_nordwall.

Sculptor MongoDB Tutorial

MongoDB bridges the gap between key-value stores (which are fast and highly scalable) and traditional RDBMS systems (which provide rich queries and deep functionality).

I think this makes MongoDB very interesting for applications that need high-performance and/or scalability, but also prefer using a rich persistent domain model with complex associations. There are of course tradeoffs to be aware of, such as relaxed transactional guarantees.

The schema less structure is attractive from a developer productivity perspective, which is one of the two goals with Sculptor (quality is the other).

Table of Contents:

Part 1 - Setup Project for Business Tier

In this first part we will setup the project structure for maven and eclipse.

1. Use the following command (one line) to create a maven pom and file structure. You can change the groupId and artifactId if you like.

mvn archetype:generate -DarchetypeGroupId=org.fornax.cartridges -DarchetypeArtifactId=fornax-cartridges-sculptor-archetype-mongodb -DarchetypeVersion=2.1.0 -DarchetypeRepository=http://fornax-platform.org/nexus/content/repositories/public

Fill in groupId and artifactId:

```
Define value for groupId: : org.blogger
Define value for artifactId: : blog
Define value for version:  1.0-SNAPSHOT: :
Define value for package:  org.blogger: :
```

2. In the new directory, run `mvn eclipse:eclipse` to create an Eclipse project with the same dependencies as in the pom.

3. Open Eclipse and import the project.

Before you start serious development you should follow the instructions in the **Installation Guide**.

Part 2 - Business Tier Hands On

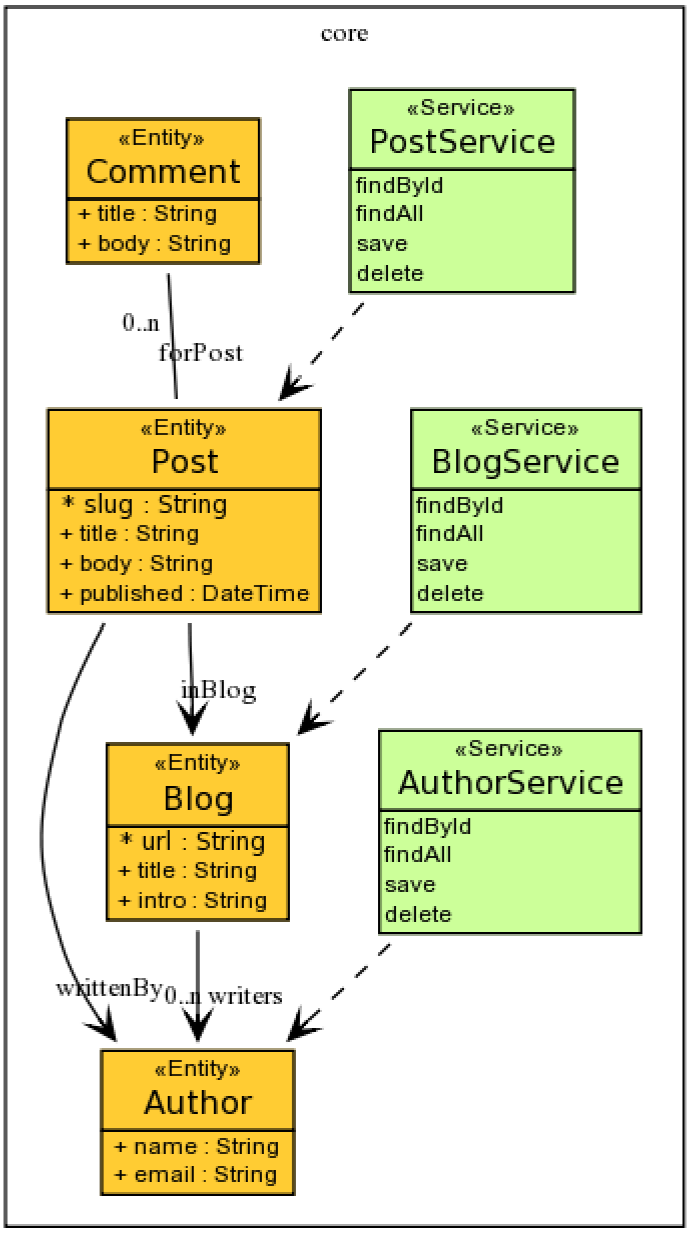
In this part we will write a Sculptor DSL file and generate code from it.

1. Modify the file named `model.btdesign` in the folder
`src/main/resources/`

2. Open the `model.btdesign` file with Sculptor DSL editor, double-click on it.
Add the following to the design file.

```
Application Blog {
    basePackage = org.blog

    Module core {

         Entity BlogPost {
            scaffold
            - Blog inBlog
            String slug key
            String title
            String body length="2000"
            Date published nullable
            - Author writtenBy
            - Set<Comment> comments
            def List<Comment> getSortedComments;

            Repository BlogPostRepository {
                List<@BlogPost> findPostsInBlog(@Blog blog);
                List<@BlogPost> findPostsWithComments => AccessObject;
                List<@BlogPost> findPostsWithGreatComments;
                protected findByCondition;
            }
        }

        ValueObject Comment {
            not aggregateRoot
            String title
            String body
            Timestamp timestamp
        }

        Entity Blog {
            scaffold
            String ^url key
            String title
            String intro length="300"
            - Set<Author> writers
        }

        Entity Author {
            scaffold
            String name required
            String emailAddress
        }
    }
}
```

Visualization of this model looks like this (generated by Sculptor):



3. Run `mvn clean install` to generate code and build. The JUnit test will fail.

4. Now it is your task to complete the failing tests. A start for implementing AuthorServiceTest may look like this:

```
private String authorId1;

@Before
public void initTestData() {
    Author author1 = new Author("Patrik");
    Author saved = authorService.save(SimpleJUnitServiceContextFactory.getServiceContext(), author1);
    authorId1 = saved.getId();
}

@Test
public void testFindById() throws Exception {
    Author found = authorService.findById(SimpleJUnitServiceContextFactory.getServiceContext(),
authorId1);
    assertEquals("Patrik", found.getName());
}
```

You need to download and install mongoDB. Start mongoDB with bin/mongod command.

When running the tests it will connect to a mongoDB instance running at localhost (default port 27017). The database will be created automatically and dropped after each test method (done by the @After dropDatabase method in the test case).

You can use `mvn -Dfornax.generator.force.execution=true -o -npu generate-sources` to speed up the builds,
-o == offline, -npu == no plugin upate.
Maven can be executed from inside Eclipse with External Tool Configuration (Fornax Maven Launcher as explained in **Installation Guide**) or with m2eclipse.
JUnit tests are typically run from inside Eclipse as usual.

Part 3 - Business Tier Explained

Most things that can be done with JPA target implementation can also be done with mongoDB. Therefore most of the descriptions in Advanced Tutorial are valid and I will not repeat everything, but I will elaborate on some topics.

Associations

An associated object can be stored as an embedded document, i.e. it belongs to parent object and cannot be shared between several objects. References to BasicType are always embedded. Aggregates are also embedded in the DBCollection of the parent object. Aggregates are defined with `belongsTo` or `not aggregateRoot` in the owned DomainObjects.

```
        Entity Cargo {
          - TrackingId trackingId key;
          - Location origin required;
          - Location destination required;
          - Itinerary itinerary nullable opposite cargo;
          - Set<HandlingEvent> events opposite cargo;

        }

        BasicType TrackingId {
          String identifier key
        }

        ValueObject Itinerary {
          belongsTo Cargo
          - Cargo cargo nullable opposite itinerary
          - List<Leg> legs
        }

        ValueObject Leg {
          belongsTo Cargo
          - CarrierMovement carrierMovement
          - Location from
          - Location to
        }
```

In above sample the TrackingId, Itinary and Leg are all stored toghether with the Cargo.

The other alternative is to store ids of the referred objects. Associations that are not owned are treated in this way. In the above sample the refereces to Location and HandlingEvent are unowned. In the domain objects there are generated getters that lazily fetch associated objects from the ids. This means that you don't have to work with the ids yourself, you can follow associations as usual, but be aware that an invocation of such a getter might need to query the database.

Referential integrity is not enforced. It must be handled by your program. Lazy getters of associations will not fail if referred to object is missing, they will return null for single value references and ignore missing objects for collection references. This means that you can cleanup dangling references by fetching objects, populate associations by invoking the getters and then save the object. There is also a populateAssociations repository operation to fetch all referred objects.

Avoid bidirectional associations

Within an aggregate it is not possible to use bidirectional associations or associations that creates cycles. Bidirectional associations cross aggregates are possible but complicated, because one end must be saved first, and the assignment of the id is done after the object is saved.

Data Mapper

When working with mongoDB Java API the DBObject plays a central role. It is like a key value Map. Values can be of most types and also collections and other DBObjects for nested documents.

Sculptor generates data mapper classes that converts domain objects to DBObjects. It is useful to not have to write those mappers by hand. Since it is generated code it also runs at full speed, compared to alternative solutions using reflection or intermediate String JSON format.

You might need to add some hand written code to customize to the mappers. That is easy. Add hint="gapMapper" and you will have a subclass that you can implement yourself. This is very useful for doing data migration. In the model:

```
Entity Person {
    hint="gapMapper"
    Date birthDate
    - Gender sex !changeable
    - Ssn ssn key
    - PersonName name
}
```

In java:

```
@Override
    public Person toDomain(DBObject from) {
        if (from == null) {
            return null;
        }
        // converting from old name structure to new
        if (!from.containsField("name") && from.containsField("firstName") && from.containsField("lastName")) {
            BasicDBObject name = new BasicDBObject();
            name.put("first", from.get("firstName"));
            name.put("last", from.get("lastName"));
            from.put("name", name);
        }
        return super.toDomain(from);
    }

    @Override
    public DBObject toData(Person from) {
        return super.toData(from);
    }
```

It can also be good to know that fields marked with `transient` are are not stored, but they are loaded if they exist in the retrieved documents. This can also be used for data migration.

requestStart / requestDone

To ensure that you read your own writes MongoDB java driver recommends using requestStart/ requestDone. This is done by the DbManagerAdvice, which is automatically added in front of services. This is also necessary for optimistic locking.

Optimistic Locking

By default a version attribute is automatically added to each Entity and mutable persistent ValueObject. This is used for optimistic locking checks. You can skip this feature by specifying !optimisticLocking for the Domain Object.

Index

Indexes are defined and created automatically for the natural key fields, i.e. attributes and references marked with `key`. It is possible to define additional indices for attributes by marking them with `index`. The id (_id) field is always indexed.

In the following sample there will be one composite index for ssn.number, ssn.country and another index for birthDate.

```
        Entity Person {
          gap
          hint="gapMapper"
          Date birthDate index
          - Gender sex !changeable
          - Ssn ssn key
          - PersonName name

        }

        BasicType Ssn {
          String number key length="20"
          - @Country country key
        }
```

Indexes are defined in the generated method `indexes` in the mapper. It is possible for you to override this method and manually adjust the indices or add more indexes.

Repositories

Sculptor provides the following generic repository operations for use with mongoDB:

- findById
- findAll
- findByCondition
- findByKey
- findByKeys
- save
- delete
- countAll
- populateAssociations

findByCondition

mongoDB has good support for dynamic queries on any attribute. The easiest way to create a query is to use findByCondition , which is one of the built in repository operations.

In `model.btdesign`

```
Repository PostRepository {
    List<@Post> findPostsWithGreatComments;
    protected findByCondition;
}
```

The hand written java implementation:

```
public List<Post> findPostsWithGreatComments() {
    List<ConditionalCriteria> condition = criteriaFor(Post.class)
        .withProperty(comments().title()).ignoreCaseLike(".*great.*")
        .and().withProperty(published()).isNotNull()
        .orderBy(published()).build();
    return findByCondition(condition);
}
```

The `ConditionalCriteriaBuilder` builder supports conditions such as eq, like, between, lessThan, greaterThan, in. The order of the result can be specified with orderBy. Regular expression condition can be defined with like and ignoreCaseLike (ignore case).

Limitations of findByCondition when used with mongoDB:

- It is not possible to specifiy criteria on attibutes of unowned associations.
- OR-condition is not supported.

Pagination
Pagination is supported as described in Advanced Tutorial

Generation of Finder Operations

Generated finder methods, based on findByCondition, is supported as described in Advanced Tutorial. It has the same limitations as findByCondition when used with mongoDB, described above.

Example of generated finders in the BlogPostRepository:

```
Repository BlogPostRepository {
    findByTitle(String title) condition="title i= :title";
    findPostsInBlog(@Blog inBlog) orderBy="title";
}
```

Note that this feature is not activated by default. To allow sculptor to generate finder operations add the following line into your sculptor-generator.properties file.

```
generate.repository.finders=true
```

After activation all repository operations starting with 'find', except built in operations, are potential candidates for finder generation. To prevent a generation and to use handwritten code for a special operation, add the keyword 'gap'.

```
PersonRepository {
    List<@Person> findPersonByName(String name) gap;
}
```

Own AccessObject
You can also easily create your own Access Object with full access to the underlaying DBCollection. For example, a query that finds a blog Posts with comments. In model.btdesign:

```
    Repository PostRepository {
        List<@Post> findPostsWithComments => AccessObject;
    }
```

You write the implementation yourself and it may look something like this:

```
public class FindPostsWithCommentsAccessImpl extends FindPostsWithCommentsAccessImplBase {
    @Override
    public void performExecute() {
        DBObject query = new BasicDBObject();
        query.put("comments", new BasicDBObject("$not", new BasicDBObject("$size", 0)));
        DBCursor cur = getDBCollection().find(query);

        List<Post> mappedResult = new ArrayList<Post>();
        for (DBObject each : cur.hasNext()) {
            Post eachResult = (Post) getDataMapper().toDomain(each);
            mappedResult.add(eachResult);
        }

        setResult(mappedResult);
    }
}
```

MongoDB specific code directly in Repository

The default implementation of a Repository consists of an implementation class and Access Objects. The intention is a separation of concerns between the domain and the data layer. Repository is close to the business domain and Access Objects are close to the data layer. The MongoDB specific code is located in the Access Object, and not in the Repository.

For some systems this separation might be overkill and you might prefer to implement the data access directly in the Repository. Sculptor supports this design out-of-the-box. You only have to specify a property in sculptor-generator.properties to have MongoDB support directly in the Repository implementation. The starting point is the method `getDbManager`.

```
generate.repository.dbManagerSupport=true
```

MongoDB documentation

Documentation of queries: http://www.mongodb.org/display/DOCS/Querying

Configuration

It is possible to define configuration options for the MongoDB connection. It is done in `spring.properties`. You find available configuration parameters (default values) in `generated-spring.properties`. Documentation of options is here. Example:

```
mongodb.dbname=Blog
mongodb.url1=localhost:27017
mongodb.url2=otherhost:27017
mongodbOptions.connectionsPerHost=10
```

When running JUnit test the dbname is suffixed with `-test`.

Custom Datastore Names

By default the names in the data store are the same as the names in the Java DomainObjects. In case you need to use other names it is possible to define that in model with `databaseTable` and `databaseColumn`. Discriminator name and value is also possible to define when inheritance is used.

```
abstract Entity Media {
  databaseTable="Products"
  discriminatorColumn="type"
  String title !changeable
}

Entity Book extends @Media {
  discriminatorValue="B"
  String isbn key length="20" databaseColumn="KEY"
}

Entity Movie extends @Media {
  discriminatorValue="M"
  String urlIMDB key databaseColumn="KEY"
  Integer playLength
}
```

Part 4 - Show me the GUI

The generated Web CRUD GUI of Sculptor works fine with mongoDB.

All you need to do to is to run two more maven archetypes:

- fornax-cartridges-sculptor-archetype-parent
- fornax-cartridges-sculptor-archetype-jsf

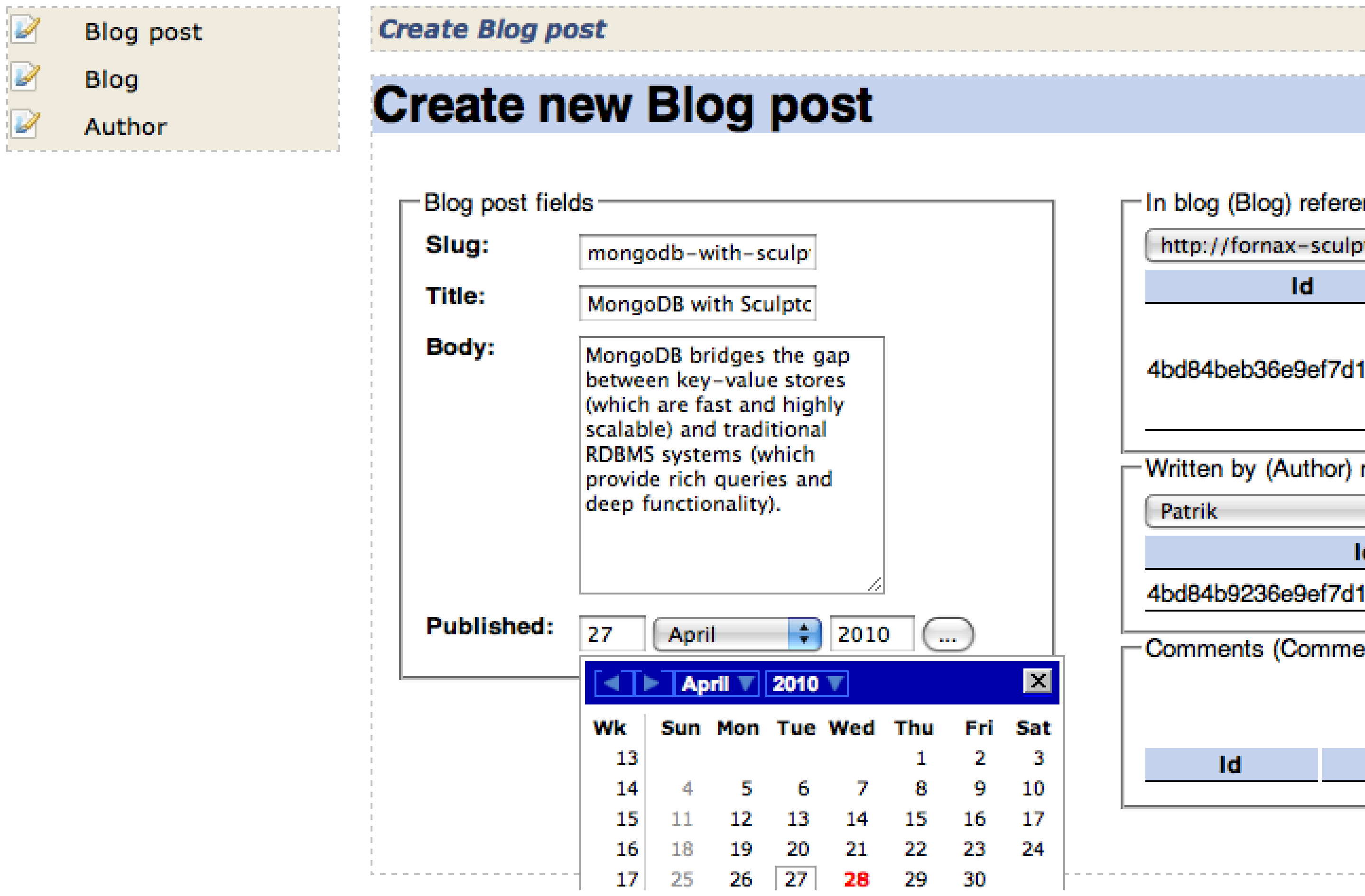
It is explained in Archetype Tutorial how to run them. Note that fornax-cartridges-sculptor-archetype for the business tier corresponds to fornax-cartridges-sculptor-archetype-mongodb, which you already have run in Part 1 of this tutorial.

You have now created a multi module maven project and therefore you should change pom.xml in business tier project. Add parent element. Remove properties and repositories elements, since they are located in parent pom.xml.

```
<parent>
        <artifactId>blog-parent</artifactId>
        <groupId>org.blogger</groupId>
        <version>1.0-SNAPSHOT</version>
        </parent>
```

Thereafter you start Jetty with mvn jetty:run from the blog-web project. Note that no installation is needed. Jetty is launched from maven.

Open http://localhost:8080/blog-web in your browser.



Blog post
Blog
Author

Create Blog post

Create new Blog post

Blog post fields

Slug: mongodb-with-sculp
Title: MongoDB with Sculptc
Body: MongoDB bridges the gap between key-value stores (which are fast and highly scalable) and traditional RDBMS systems (which provide rich queries and deep functionality).
Published: 27 April 2010 ...

April 2010

Wk	Sun	Mon	Tue	Wed	Thu	Fri	Sat
13					1	2	3
14	4	5	6	7	8	9	10
15	11	12	13	14	15	16	17
16	18	19	20	21	22	23	24
17	25	26	27	28	29	30	

In blog (Blog) refere
http://fornax-sculp
Id
4bd84beb36e9ef7d1

Written by (Author) r
Patrik
Id
4bd84b9236e9ef7d1

Comments (Comme
Id

Note that the CRUD GUI is intended for administrative sections of an application. Without customization (manual coding) it is not good enough for a public blog.

Source

The complete source code for this tutorial is available in Subversion.

Web Access (read only):
http://fisheye3.cenqua.com/browse/fornax/trunk/cartridges/sculptor/mongodb-sample

Anonymous Access (read only):
https://fornax.svn.sourceforge.net/svnroot/fornax/trunk/cartridges/sculptor/mongodb-sample