# Mongo DB & Spring

michael.schaffler@ciit.at

# 1. Introduction to Mongo DB

**What is Mongo DB?**

- MongoDB stores data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time
- The document model maps to the objects in your application code, making data easy to work with
- Ad hoc queries, indexing, and real time aggregation provide powerful ways to access and analyze your data
- MongoDB is a distributed database at its core, so high availability, horizontal scaling, and geographic distribution are built in and easy to use
- MongoDB is free and open-source. Versions released prior to October 16, 2018 are published under the AGPL. All versions released after October 16, 2018, including patch fixes for prior versions, are published under the Server Side Public License (SSPL) v1.

# 2. NoSQL Databases Explained

- **Data Model categories:**
  - Document Model
    - each record and its associated (i.e., related) data are typically stored together in a single, hierarchical document = a complex data structure, mostly in JSON
    - documents can contain many different key-value pairs, or key-array pairs, or even nested documents
    - each document can contain different fields
    - data can be queried based on any combination of fields in a document, with rich secondary indexes providing efficient access paths to support almost any query pattern
  - Graph Model
    - uses graph structures with nodes, edges and properties to represent data
    - useful in cases where traversing relationships are core to the application, like navigating social network connections, network topologies or supply chains
  - Key-Value and Wide Column Models
    - every item in the database is stored as an attribute name, or key, together with its value. The value, however, is entirely opaque to the system; data can only be queried by the key.
    - useful for representing polymorphic and unstructured data, as the database does not enforce a set schema across key-value pairs
    - Wide-column stores such as Cassandra and HBase are optimized for queries over large datasets, and store columns of data together, instead of rows

# 2. NoSQL Databases Explained

## The Benefits of NoSQL

When compared to relational databases, NoSQL databases are in some cases more scalable and provide superior performance, and their data model addresses several issues that the relational model is not designed to address:

- Large volumes of rapidly changing structured, semi-structured, and unstructured data
- Agile sprints, quick schema iteration, and frequent code pushes
- Object-oriented programming that is easy to use and flexible
- Geographically distributed scale-out architecture instead of expensive, monolithic architecture

# 3. Mongo DB - getting started

- Here we are connecting to a locally hosted MongoDB database called test with a collection named restaurants.

```
// 1. Connect to MongoDB instance running on localhost
MongoClient mongoClient = new MongoClient();

// Access database named 'test'
MongoDatabase database = mongoClient.getDatabase("test");

// Access collection named 'restaurants'
MongoCollection collection = database.getCollection("restaurants");
```

# 3. Mongo DB - getting started

- 5 example documents are being inserted into the restaurantscollection. Each document represents a restuarant with a name, star rating, and categories (stored as an array).

```
// 2. Insert
List documents = asList(
  new Document("name", "Sun Bakery Trattoria")
    .append("stars", 4)
    .append("categories",
      asList("Pizza", "Pasta", "Italian", "Coffee", "Sandwiches")),
  new Document("name", "Blue Bagels Grill")
    .append("stars", 3)
    .append("categories",
      asList("Bagels", "Cookies", "Sandwiches")),
  new Document("name", "Hot Bakery Cafe")
    .append("stars", 4)
    .append("categories",
      asList("Bakery", "Cafe", "Coffee", "Dessert")),
  new Document("name", "XYZ Coffee Bar")
    .append("stars", 5)
    .append("categories",
      asList("Coffee", "Cafe", "Bakery", "Chocolates")),
  new Document("name", "456 Cookies Shop")
    .append("stars", 4)
    .append("categories",
      asList("Bakery", "Cookies", "Cake", "Coffee")));

collection.insertMany(documents);
```

# 3. Mongo DB - getting started

- In this example, we run a simple query to get all of the documents in the restaurants collection and store them as an array.

```
// 3. Query
List results = collection.find().into(new ArrayList<>());
```

# 3. Mongo DB - getting started

- Indexes in MongoDB are similar to indexes in other database systems. MongoDB supports indexes on any field or sub-field of a document in a collection. Here, we are building an index on the name field with sort order ascending.

```
// 4. Create Index
collection.createIndex(Indexes.ascending("name"));
```

# 3. Mongo DB - getting started

- Using MongoDB’s aggregation pipeline, you can filter and analyze data based on a given set of criteria. In this example, we pull all the documents in the restaurants collection that have a category of Bakery using the $match operator and then group them by their star rating using the $group operator. Using the accumulator operator, $sum, we can see how many bakeries in our collection have each star rating.

```
// 5. Perform Aggregation
collection.aggregate(asList(match(eq("categories", "Bakery")),
  group("$stars", sum("count", 1))));

mongoClient.close();
```

# 4. Databases and Collections

- MongoDB stores BSON documents, i.e. data records, in collections; the collections in databases.
- Collections are analogous to tables in relational databases

# 4. Databases and Collections

- Create a Collection:
  - If a collection does not exist, MongoDB creates the collection when you first store data for that collection.
  - Both the insertOne() and the createIndex() operations create their respective collection if they do not already exist.

```
db.myNewCollection2.insertOne( { x: 1 } )
db.myNewCollection3.createIndex( { y: 1 } )
```

  - MongoDB provides the db.createCollection() method to explicitly create a collection with various options, such as setting the maximum size or the documentation validation rules.
  - To modify these collection options, use collMod
  - By default, a collection does not require its documents to have the same schema; starting in MongoDB 3.2, however, you can enforce document validation rules for a collection during update and insert operations.

# 4. Views

- Views are read-only; write operations on views will error.
- The following read operations can support views:
  - db.collection.find()
  - db.collection.findOne()
  - db.collection.aggregate()
  - db.collection.countDocuments()
  - db.collection.estimatedDocumentCount()
  - db.collection.count()
  - db.collection.distinct()

```
db.runCommand( { create: <view>, viewOn: <source>, pipeline: <pipeline> } )
```

- You can specify a default collation for a view at creation time. If no collation is specified, the view's default collation is the "simple" binary comparison collator. That is, the view does not inherit the collection's default collation.

# 5. Capped Collections

- Capped collections are fixed-size collections that support high-throughput operations that insert and retrieve documents based on insertion order. Capped collections work in a way similar to circular buffers: once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection.

```
db.createCollection( "log", { capped: true, size: 100000 } )
```

- The size argument is always required, even when you specify max number of documents. MongoDB will remove older documents if a collection reaches the maximum size limit before it reaches the maximum document count.
- If the size field is less than or equal to 4096, then the collection will have a cap of 4096 bytes. Otherwise, MongoDB will raise the provided size to make it an integer multiple of 256.
- You can convert a non-capped collection to a capped collection

# 6. Documents

- MongoDB stores data records as BSON documents. BSON is a binary representation of JSON documents, though it contains more data types than JSON
- MongoDB documents are composed of field-and-value pairs
- The value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents. For example, the following document contains values of varying types:

```
var mydoc = {
               _id: ObjectId("5099803df3f4948bd2f98391"),
               name: { first: "Alan", last: "Turing" },
               birth: new Date('Jun 23, 1912'),
               death: new Date('Jun 07, 1954'),
               contribs: [ "Turing machine", "Turing test", "Turingery" ],
               views : NumberLong(1250000)
             }
```

# 6. Documents

- MongoDB stores data records as BSON documents. BSON is a binary representation of JSON documents, though it contains more data types than JSON
- MongoDB documents are composed of field-and-value pairs
  - Field names are strings, the field name _id is reserved for use as a primary key; its value must be unique in the collection, is immutable, and may be of any type other than an array
  - Top-level field names cannot start with the dollar sign ($) character
- The value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents. For example, the following document contains values of varying types:

```
var mydoc = {
               _id: ObjectId("5099803df3f4948bd2f98391"),
               name: { first: "Alan", last: "Turing" },
               birth: new Date('Jun 23, 1912'),
               death: new Date('Jun 07, 1954'),
               contribs: [ "Turing machine", "Turing test", "Turingery" ],
               views : NumberLong(1250000)
             }
```

# 6. Documents

- Embedded Documents
  - To specify or access an element of an array by the zero-based index position, concatenate the array name with the dot (.) and zero-based index position, and enclose in quotes:
  - For example, given the following field in a document:

```
{
   ...
   name: { first: "Alan", last: "Turing" },
   contact: { phone: { type: "cell", number: "111-222-3333" } },
   ...
}
```

  - To specify the number in the phone document in the contact field, use the dot notation "contact.phone.number".

# 6. Document Limitations

- Document Size Limit
  - The maximum BSON document size is 16 megabytes
- Document Field Order
  - MongoDB preserves the order of the document fields following write operations *except* for the following cases
- The _id Field
  - In MongoDB, each document stored in a collection requires a unique _id field that acts as a primary key. If an inserted document omits the _id field, the MongoDB driver automatically generates an ObjectId for the _id field.

# 7. The mongo Shell

- Connect to Mongo DB – example with authentication:
  - mongo --username alice --password --authenticationDatabase admin --host mongodb0.examples.com --port 28015
  - The maximum BSON document size is 16 megabytes
- Use db to display the database you are using
- Use <database> - to switch databases. You can switch to non-existing databases. When you first store data in the database, such as by creating a collection, MongoDB creates the database

```
use myNewDatabase
db.myCollection.insertOne( { x: 1 } );
```

# 7. The mongo Shell

- You can write scripts for the mongo shell in JavaScript that manipulate data in MongoDB or perform administrative operation.

**EXAMPLE:**

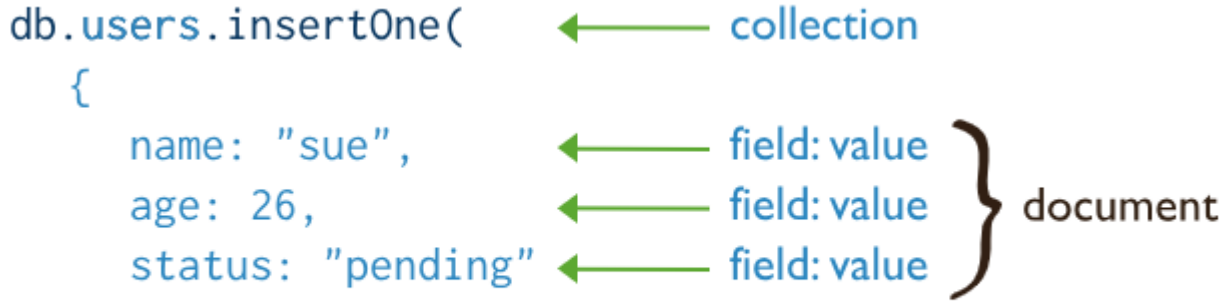
To print all items in a result cursor in mongo shell scripts, use the following idiom:

```
cursor = db.collection.find();
while ( cursor.hasNext() ) {
   printjson( cursor.next() );
}
```

```
mongo test --eval "printjson(db.getCollectionNames())"
```

```
load("myjstest.js")
```

# 8. CRUD Operations




```
db.inventory.insertMany([
   { item: "journal", qty: 25, tags: ["blank", "red"], size: { h: 14, w: 21, uom: "cm" } }
   { item: "mat", qty: 85, tags: ["gray"], size: { h: 27.9, w: 35.5, uom: "cm" } },
   { item: "mousepad", qty: 25, tags: ["gel", "blue"], size: { h: 19, w: 22.85, uom: "cm"
])
```


click to enlarge


# 9. SQL to MongoDB Mapping Chart

| | |
|---|---|
| database | database |
| table | collection |
| row | document or BSON document |
| column | field |
| index | index |
| table joins | `$lookup`, embedded documents |
| primary key<br><br>Specify any unique column or column combination as primary key. | primary key<br><br>In MongoDB, the primary key is automatically set to the _id field. |
| aggregation (e.g. group by) | aggregation pipeline<br><br>See the SQL to Aggregation Mapping Chart. |
| transactions | transactions |

# 9. SQL to MongoDB Mapping Chart (cont.)

| SQL Schema Statements | MongoDB Schema Statements |
| --- | --- |
| `CREATE TABLE people (`<br>`  id MEDIUMINT NOT NULL`<br>`  AUTO_INCREMENT,`<br>`  user_id Varchar(30),`<br>`  age Number,`<br>`  status char(1),`<br>`  PRIMARY KEY (id)`<br>`)` | Implicitly created on first `insertOne()` or `insertMany()` operation. The primary key `_id` is automatically added if `_id` field is not specified.<br><br>`db.people.insertOne( {`<br>`  user_id: "abc123",`<br>`  age: 55,`<br>`  status: "A"`<br>`} )`<br><br>However, you can also explicitly create a collection:<br><br>`db.createCollection("people")` |

# 9. SQL to MongoDB Mapping Chart (cont.)

```
ALTER TABLE people
ADD join_date DATETIME
```

Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level.

However, at the document level, `updateMany()` operations can add fields to existing documents using the `$set` operator.

```
db.people.updateMany(
    { },
    { $set: { join_date: new Date() } }
)
```

```
ALTER TABLE people
DROP COLUMN join_date
```

Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level.

However, at the document level, `updateMany()` operations can remove fields from documents using the `$unset` operator.

```
db.people.updateMany(
    { },
    { $unset: { "join_date": "" } }
)
```

# 9. SQL to MongoDB Mapping Chart (cont.)

| | |
|---|---|
| `CREATE INDEX idx_user_id_asc`<br>`ON people(user_id)` | `db.people.createIndex( { user_id: 1 } )` |
| `CREATE INDEX`<br>`idx_user_id_asc_age_desc`<br>`ON people(user_id, age DESC)` | `db.people.createIndex( { user_id: 1, age: -1 } )` |
| `DROP TABLE people` | `db.people.drop()` |

# 9. SQL to MongoDB Mapping Chart (cont.)

| SQL SELECT Statements | MongoDB find() Statements |
|---|---|
| `SELECT *`<br>`FROM people` | `db.people.find()` |
| `SELECT id,`<br>`  user_id,`<br>`  status`<br>`FROM people` | `db.people.find(`<br>`  { },`<br>`  { user_id: 1, status: 1 }`<br>`)` |
| `SELECT user_id, status`<br>`FROM people` | `db.people.find(`<br>`  { },`<br>`  { user_id: 1, status: 1, _id: 0 }`<br>`)` |

# 9. SQL to MongoDB Mapping Chart (cont.)

| | |
|---|---|
| `SELECT *`<br>`FROM people`<br>`WHERE age > 25`<br>`AND age <= 50` | `db.people.find(`<br>`{ age: { $gt: 25, $lte: 50 } }`<br>`)` |
| `SELECT *`<br>`FROM people`<br>`WHERE user_id like "%bc%"` | `db.people.find( { user_id: /bc/ } )`<br>-or-<br>`db.people.find( { user_id: { $regex: /bc/ } } )` |
| `SELECT *`<br>`FROM people`<br>`WHERE user_id like "bc%"` | `db.people.find( { user_id: /^bc/ } )`<br>-or-<br>`db.people.find( { user_id: { $regex: /^bc/ } } )` |
| `SELECT *`<br>`FROM people`<br>`WHERE status = "A"`<br>`ORDER BY user_id ASC` | `db.people.find( { status: "A" } ).sort( { user_id: 1 } )` |

# 9. SQL to MongoDB Mapping Chart (cont.)

| | |
|---|---|
| `SELECT COUNT(user_id)`<br>`FROM people` | `db.people.count( { user_id: { $exists: true } } )`<br>*or*<br>`db.people.find( { user_id: { $exists: true } } ).count()` |
| `SELECT COUNT(*)`<br>`FROM people`<br>`WHERE age > 30` | `db.people.count( { age: { $gt: 30 } } )`<br>*or*<br>`db.people.find( { age: { $gt: 30 } } ).count()` |
| `SELECT DISTINCT(status)`<br>`FROM people` | `db.people.aggregate( [ { $group : { _id : "$status" } } ] )`<br>or, for distinct value sets that do not exceed the BSON size limit<br>`db.people.distinct( "status" )` |

# 9. SQL to MongoDB Mapping Chart (cont.)

| SQL Update Statements | MongoDB updateMany() Statements |
| --- | --- |
| `UPDATE people`<br>`SET status = "C"`<br>`WHERE age > 25` | `db.people.updateMany(`<br>`{ age: { $gt: 25 } },`<br>`{ $set: { status: "C" } }`<br>`)` |
| `UPDATE people`<br>`SET age = age + 3`<br>`WHERE status = "A"` | `db.people.updateMany(`<br>`{ status: "A" } ,`<br>`{ $inc: { age: 3 } }`<br>`)` |

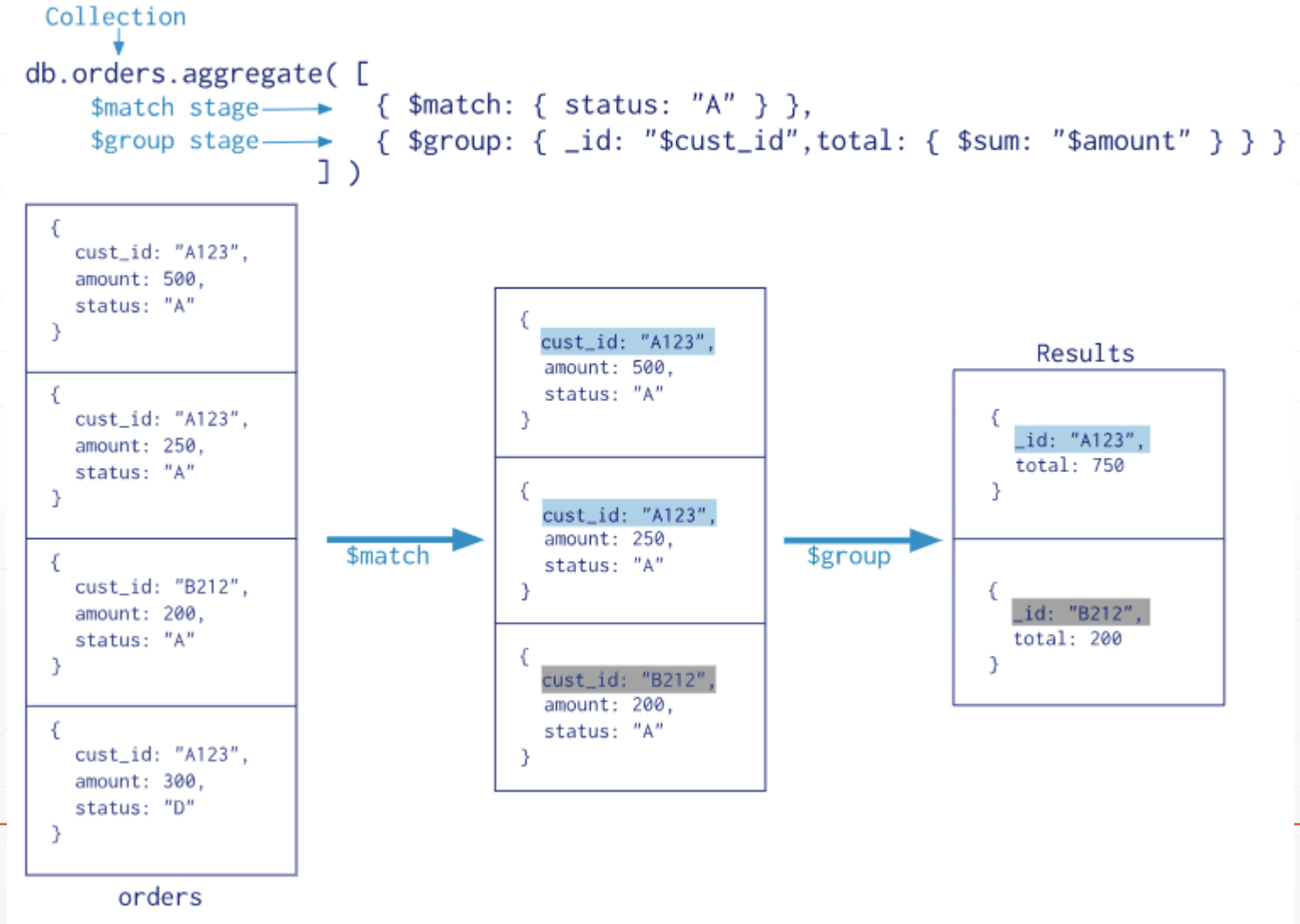
# 10. Retryable Writes

- Retryable writes allow MongoDB drivers to automatically retry certain write operations a single time if they encounter network errors, or if they cannot find a healthy primary in the replica sets or sharded cluster.
- The transaction commit and abort operations are retryable write operations. If the commit operation or the abort operation encounters an error, MongoDB drivers retry the operation a single time regardless of whether retryWrites is set to true.

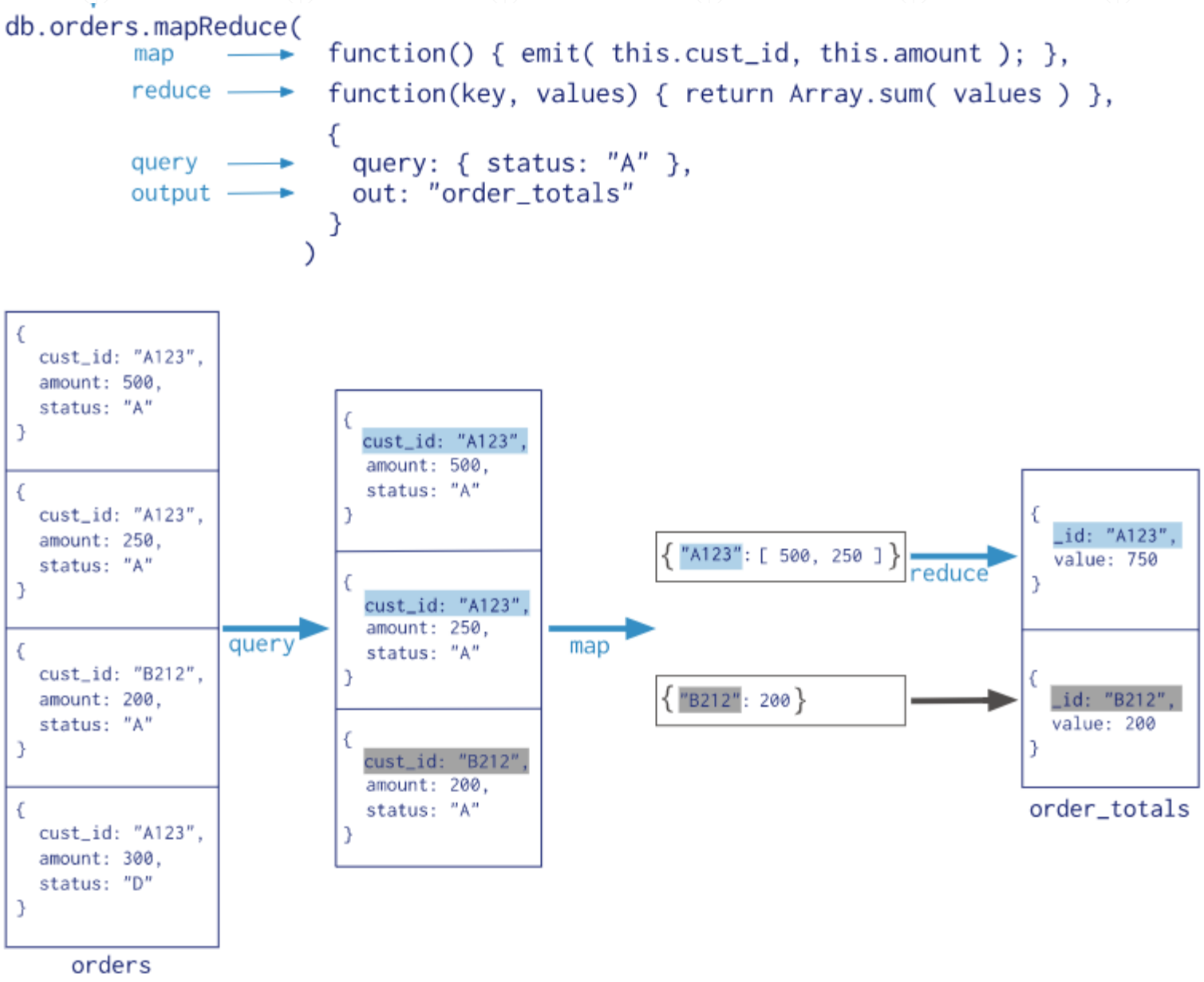
# 11. Aggregation

- Aggregation operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.
- MongoDB provides three ways to perform aggregation: the aggregation pipeline, the map-reduce function, and single purpose aggregation methods.

# 11. Aggregation Pipeline



# 11. Aggregation Map-Reduce



# 12. Data Modeling

- Flexible Schema
  - The documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection
  - To change the structure of the documents in a collection, such as add new fields, remove existing fields, or change the field values to a new type, update the documents to the new structure
- Document Structure
  - Embedded documents capture relationships between data by storing related data in a single document structure.

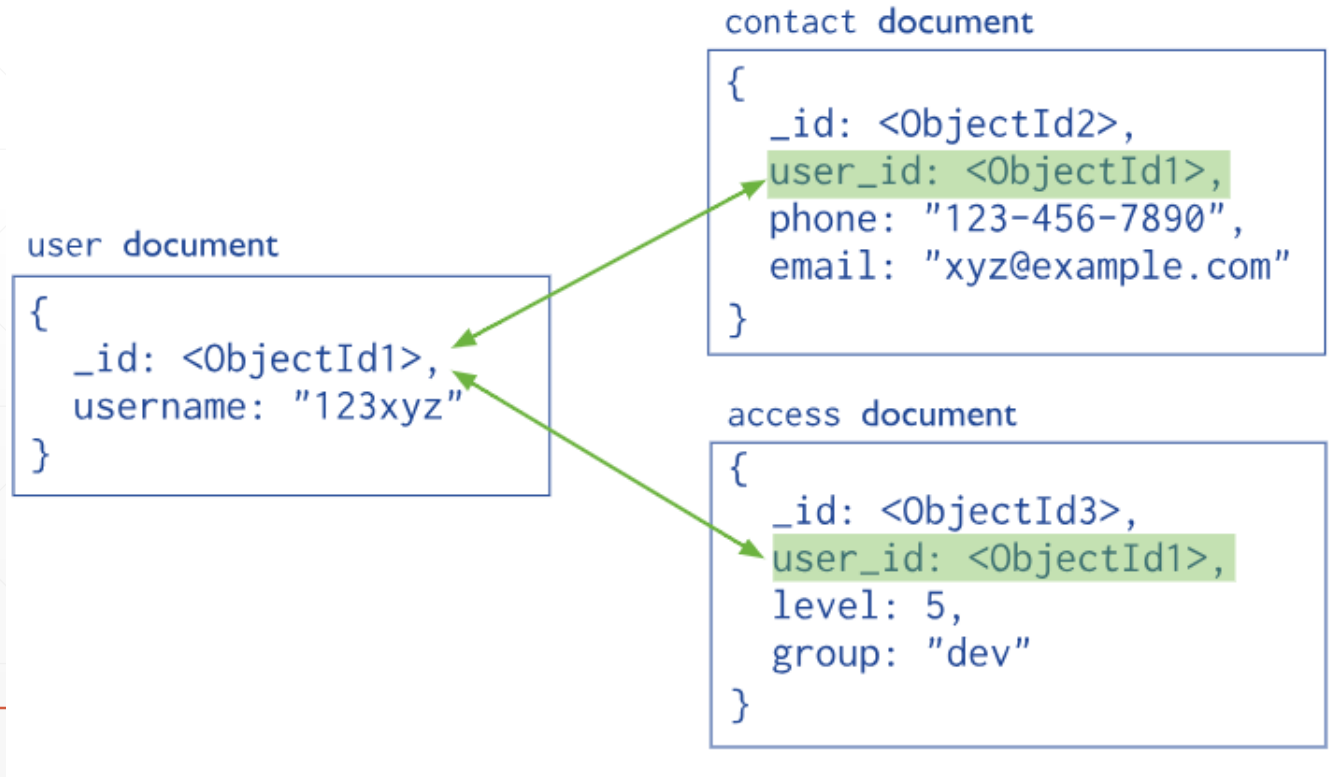
```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
             phone: "123-456-7890",
             email: "xyz@example.com"
           },
  access: {
            level: 5,
            group: "dev"
          }
}
```

Embedded sub-document

Embedded sub-document

# 12. Data Modeling (cont.)

- References store the relationships between data by including links or *references* from one document to another.
- Applications can resolve these references to access the related data. Broadly, these are *normalized* data models



# 12. Data Modeling (cont.)

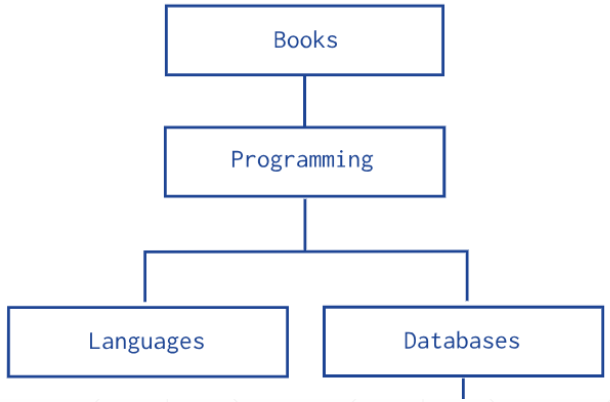
- Schema Validation - Validation occurs during updates and inserts. When you add validation to a collection, existing documents do not undergo validation checks

```
db.createCollection("students", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         required: [ "name", "year", "major", "gpa", "address.city", "address.street" ],
         properties: {
            name: {
               bsonType: "string",
               description: "must be a string and is required"
            },
            gender: {
               bsonType: "string",
               description: "must be a string and is not required"
            },
```

```
db.createCollection( "contacts",
   { validator: { $or:
      [
         { phone: { $type: "string" } },
         { email: { $regex: /@mongodb\.com$/ } },
         { status: { $in: [ "Unknown", "Incomplete" ] } }
      ]
   }
} )
```

# 12. Data Modeling (cont.)

- Model Tree Structures with Parent References




```
db.categories.insert( { _id: "Databases", parent: "Programming" } )
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming", parent: "Books" } )
db.categories.insert( { _id: "Books", parent: null } )
```

```
db.categories.find( { parent:"Programming"} )
```

# 13. Transactions

- In MongoDB, an operation on a single document is atomic.
- Because you can use embedded documents and arrays to capture relationships between data in a single document structure instead of normalizing across multiple documents and collections, this single-document atomicity obviates the need for multi-document transactions for many practical use cases.

### Transactions and the mongo Shell

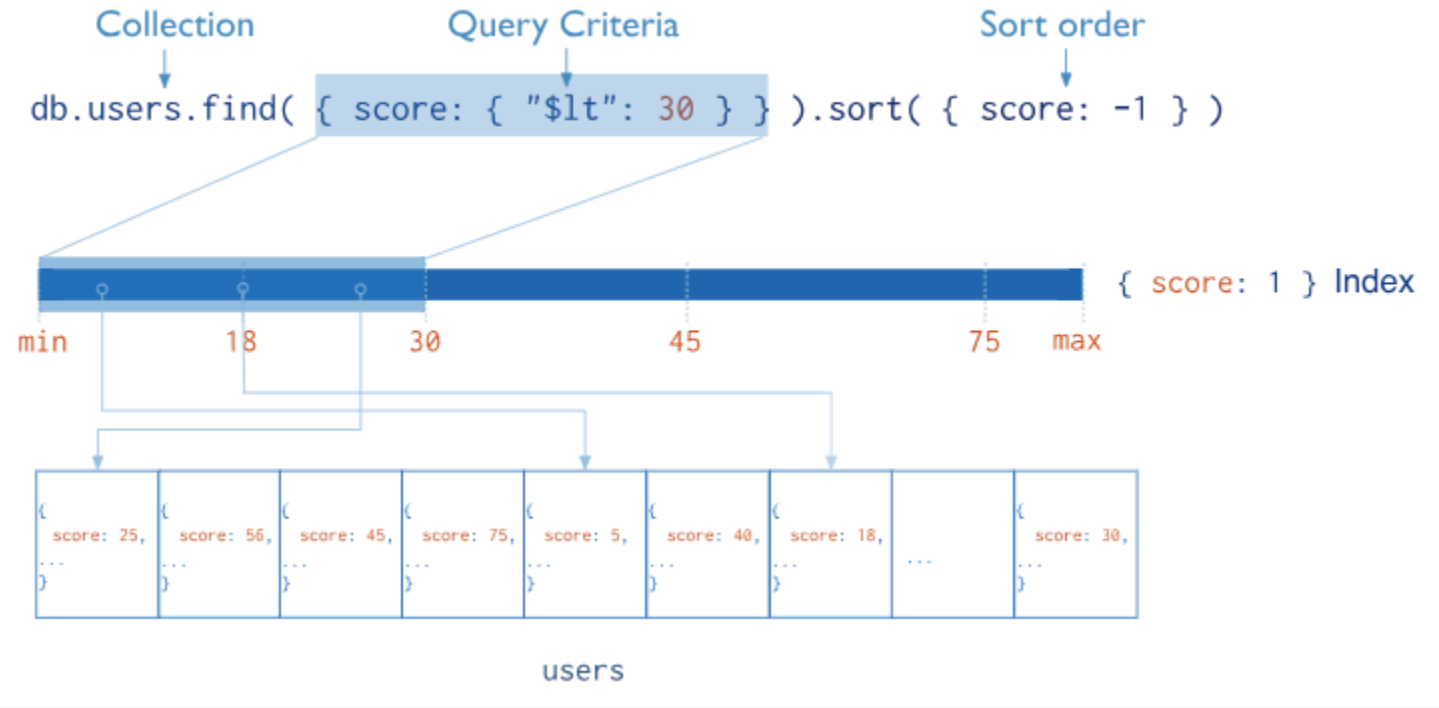
The following mongo shell methods are available for transactions:

- `Session.startTransaction()`
- `Session.commitTransaction()`
- `Session.abortTransaction()`

- By default, transactions waits up to 5 milliseconds to acquire locks required by the operations in the transaction. If the transaction cannot acquire its required locks within the 5 milliseconds, the transaction aborts.

# 14. Indexes

- Indexes support the efficient execution of queries in MongoDB.
- Without indexes, MongoDB must perform a *collection scan*, i.e. scan every document in a collection, to select those documents that match the query statement.



# 15. Security

| Authentication | Authorization | TLS/SSL | Enterprise Only |
| --- | --- | --- | --- |
| Authentication | Role-Based Access Control | TLS/SSL (Transport Encryption) | Kerberos Authentication |
| SCRAM | Enable Auth | Configure mongod and mongos for TLS/SSL | LDAP Proxy Authentication |
| x.509 | Manage Users and Roles | TLS/SSL Configuration for Clients | Encryption at Rest |
| | | | Auditing |

# 16. Spring Data MongoDB Repositories

- Spring Data includes repository support for MongoDB. As with the JPA repositories discussed earlier, the basic principle is that queries are constructed automatically, based on method names.
- In fact, both Spring Data JPA and Spring Data MongoDB share the same common infrastructure. You could take the JPA example from earlier and, assuming that City is now a Mongo data class rather than a JPA @Entity, it works in the same way, as shown in the following example:
- You can customize document scanning locations by using the @EntityScan annotation

```
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;

public interface CityRepository extends Repository<City, Long> {

        Page<City> findAll(Pageable pageable);

        City findByNameAndStateAllIgnoringCase(String name, String state);

}
```

# 17. MongoDB und Spring Beispiel

- Das folgende Beispiel demonstriert:
- wie Sie eine MongoDB über Spring Data mit Spring ansprechen können,
- wie Sie durch Nutzung des Spring-Data-Repository-Konzepts bequem MongoDB-Abfragemethoden automatisch generieren lassen und verwenden,
- wie automatisch/implizit Java-Objekte in MongoDB-Documents und umgekehrt konvertiert werden (ODM, Object Document Mapping), inklusive Enums und Listen von Objekten, und
- wie Sie mit Spring Boot sehr einfach eine ausführbare Jar-Datei erstellen, welche alle benötigten Abhängigkeiten beinhaltet.

# 17. MongoDB und Spring Beispiel (cont.)

- Im Maven-Projektkonfigurationsdatei

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
    <version>1.2.8.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-mongodb</artifactId>
    <version>1.8.2.RELEASE</version>
  </dependency>
</dependencies>
```

# 17. MongoDB und Spring Beispiel (cont.)

- Person.java

```
@Document(collection = "personen4")
public class Person
{
   @Id
   private String        id;
   private String        vorname;
   private String        nachname;
   private int           groesse;
   private LocalDate     geburtstag;
   private List<Adresse> adressen;

   public Person()
   {
   }

   public Person( String vorname, String nachname, int groesse, LocalDate geburtstag, Adresse... adressen )
   {
      this.vorname    = vorname;
      this.nachname   = nachname;
      this.groesse    = groesse;
      this.geburtstag = geburtstag;
      this.adressen   = Arrays.asList( adressen );
   }

   public String        getVorname()    { return vorname; }
   public String        getNachname()   { return nachname; }
   public int           getGroesse()    { return groesse; }
   public LocalDate     getGeburtstag() { return geburtstag; }
   public List<Adresse> getAdressen()   { return adressen; }
   public void setVorname(    String vorname )  { this.vorname  = vorname; }
   public void setNachname(   String nachname ) { this.nachname = nachname; }
   public void setGroesse(    int groesse )     { this.groesse  = groesse; }
   public void setGeburtstag( LocalDate geburtstag )   { this.geburtstag = geburtstag; }
   public void setAdressen(   List<Adresse> adressen ) { this.adressen   = adressen; }

   @Override
   public String toString()
   {
      return ( geburtstag == null && adressen == null )
             ? "Person={vorname='" + vorname + "', nachname='" + nachname + "', groesse=" + groesse + "}"
             : "Person={vorname='" + vorname + "', nachname='" + nachname + "', groesse=" + groesse +
                                     "', geburtstag=" + geburtstag + "', adressen=" + adressen + "}";
   }
```

# 17. MongoDB und Spring Beispiel (cont.)

- Personrepository.java - Beachten Sie, dass die find-Abfragemethoden nicht ausprogrammiert werden müssen.

```
import java.time.LocalDate;
import java.util.List;
import org.springframework.data.mongodb.repository.MongoRepository;
import mongodbmitspring.Person.Adresse;

public interface PersonRepository extends MongoRepository<Person,String>
{
   public Person findByVornameAndNachname( String vorname, String nachname );

   public List<Person> findByVorname( String vorname );

   public List<Person> findByNachname( String nachname );

   public List<Person> findByGroesse( int groesse );

   public List<Person> findByGroesseGreaterThan( int groesse );

   public List<Person> findByGroesseBetween( int from, int to );

   public List<Person> findByGeburtstag( LocalDate geburtstag );

   public List<Person> findByGeburtstagGreaterThan( LocalDate geburtstag );

   public List<Person> findByGeburtstagBetween( LocalDate from, LocalDate to );
```

# 17. MongoDB und Spring Beispiel (cont.)

- MongoSpringBeispiel.java

```
package mongodbmitspring;

import java.time.LocalDate;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import mongodbmitspring.Person.Adresse;
import mongodbmitspring.Person.LandEnum;

@SpringBootApplication
public class MongoSpringBeispiel implements CommandLineRunner
{
   private static final Adresse AACHEN = new Adresse( "Markt", 52072, "Aachen", LandEnum.DE );
   private static final Adresse KOELN  = new Adresse( "Markt", 50667, "Köln",   LandEnum.DE );
   private static final Adresse WIEN   = new Adresse( "Markt",  1010, "Wien",   LandEnum.AT );

   @Autowired
   private PersonRepository repo;

   public static void main( String[] args )
   {
      SpringApplication.run( MongoSpringBeispiel.class, args );
   }

   @Override
   public void run( String... args ) throws Exception
   {
      insertPersonen();
      findPersonen();
      updatePerson();
      repo.deleteAll();
   }

   void insertPersonen()
   {
      repo.save( new Person( "Anton", "Alfa", 184, LocalDate.of( 1960, 1, 2 ), WIEN, AACHEN ) );
      repo.save( new Person( "Anton", "Beta", 173, LocalDate.of( 1970, 3, 4 ), AACHEN ) );
      repo.save( new Person( "Berta", "Beta", 190, LocalDate.of( 1980, 5, 6 ), KOELN ) );
      repo.save( new Person( "Cäsar", "Zulu", 175, LocalDate.of( 1990, 7, 8 ), KOELN ) );
   }

   void findPersonen()
```

# 17. MongoDB und Spring Beispiel (cont.)

- MongoSpringBeispiel.java

```
void findPersonen()
{
   LocalDate dt70 = LocalDate.of( 1970, 1, 1 );
   LocalDate dt90 = LocalDate.of( 1990, 1, 1 );
   printPersonen( "findAll():",                              repo.findAll() );
   printPersonen( "findByVorname('Anton')",                  repo.findByVorname( "Anton" ) );
   printPersonen( "findByNachname('Beta')",                  repo.findByNachname( "Beta" ) );
   printPersonen( "findByGroesse(184)",                      repo.findByGroesse( 184 ) );
   printPersonen( "findByGroesseGreaterThan(175)",           repo.findByGroesseGreaterThan( 175 ) );
   printPersonen( "findByGroesseBetween(172,176)",           repo.findByGroesseBetween( 172, 176 ) );
   printPersonen( "findByGeburtstagBetween(1970,1990)",      repo.findByGeburtstagBetween( dt70, dt90 ) );
   printPersonen( "findByAdressen(Aachen)",                  repo.findByAdressen( AACHEN ) );
   printPersonen( "findByAdressenOrt('Wien')",               repo.findByAdressenOrt( "Wien" ) );
   printPersonen( "findByAdressenPlz(50667)",                repo.findByAdressenPlz( 50667 ) );
   printPersonen( "findByAdressenPlzOrderByGroesse(50667)", repo.findByAdressenPlzOrderByGroesse( 50667 ) );
}

void updatePerson()
{
   Person antonAnnet = repo.findByVornameAndNachname( "Anton", "Alfa" );
   antonAnnet.setVorname( "Annet" );
   repo.save( antonAnnet );
   printPersonen( "update('Anton Alfa' -> 'Annet Alfa')", repo.findAll() );
   System.out.println();
}

static void printPersonen( String titel, List<Person> personen )
{
   System.out.println( "\n" + titel + ":" );
   personen.forEach( System.out::println );
}
```

# 17. MongoDB und Spring Beispiel (cont.)

- java -jar target/MongoDbMitSpring.jar

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.2.8.RELEASE)

... --- [main] mongodbmitspring.MongoSpringBeispiel     : Starting MongoSpringBeispiel ... (D:\Mei
... --- [main] s.c.a.AnnotationConfigApplicationContext : Refreshing org.springframework.context.a
... --- [main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on sta

findAll()::
Person={vorname='Anton', nachname='Alfa', groesse=184', geburtstag=1960-01-02', adressen=[Adresse=
Person={vorname='Anton', nachname='Beta', groesse=173', geburtstag=1970-03-04', adressen=[Adresse=
Person={vorname='Berta', nachname='Beta', groesse=190', geburtstag=1980-05-06', adressen=[Adresse=
Person={vorname='Cäsar', nachname='Zulu', groesse=175', geburtstag=1990-07-08', adressen=[Adresse=

findByVorname('Anton'):
Person={vorname='Anton', nachname='Alfa', groesse=184', geburtstag=1960-01-02', adressen=[Adresse=
Person={vorname='Anton', nachname='Beta', groesse=173', geburtstag=1970-03-04', adressen=[Adresse=

findByNachname('Beta'):
Person={vorname='Anton', nachname='Beta', groesse=173', geburtstag=1970-03-04', adressen=[Adresse=
Person={vorname='Berta', nachname='Beta', groesse=190', geburtstag=1980-05-06', adressen=[Adresse=

findByGroesse(184):
Person={vorname='Anton', nachname='Alfa', groesse=184', geburtstag=1960-01-02', adressen=[Adresse=

findByGroesseGreaterThan(175):
Person={vorname='Anton', nachname='Alfa', groesse=184', geburtstag=1960-01-02', adressen=[Adresse=
Person={vorname='Berta', nachname='Beta', groesse=190', geburtstag=1980-05-06', adressen=[Adresse=

findByGroesseBetween(172,176):
Person={vorname='Anton', nachname='Beta', groesse=173', geburtstag=1970-03-04', adressen=[Adresse=
Person={vorname='Cäsar', nachname='Zulu', groesse=175', geburtstag=1990-07-08', adressen=[Adresse=

findByGeburtstagBetween(1970,1990):
Person={vorname='Anton', nachname='Beta', groesse=173', geburtstag=1970-03-04', adressen=[Adresse=
Person={vorname='Berta', nachname='Beta', groesse=190', geburtstag=1980-05-06', adressen=[Adresse=
```

# Mongo DB & Spring

michael.schaffler@ciit.at