NoSQL project instructions for the course Advanced databases

Instructions are divided into two parts::

1. **Tutorial on basic MongoDB functionalities**
2. **Students' assignments**

Table Of Contents

1. Introduction to Basic MongoDB functionalities

1.1. Installation

Install MongoDB: https://www.mongodb.com/download-center/community

Note for Windows users:

- you can install it as a service and then run it in Services. Mongo inspects the mongod.cfg file from the bin directory for settings. It has a log file that can be useful later when writing MR queries because all output through the print function will be redirected to that file
- if you don't install Mongo as a service, you should run it manually from the terminal. Then all output will appear in the terminal. Here's an example of a "manual" boot from the installation directory:

```
D:\Program Files\MongoDB\Server\3.4> bin\mongod --vv --config "d:\Program
Files\MongoDB\Server\3.2\data\mongo.conf" --dbpath "d:\Program Files\MongoDB\Server\3.2\data\db"
```

1.2. Insert, update, delete

(Open a new terminal, as stated above) and run the mongo *shell*:

```
mongo
```

Try the help command in the *shell*:

```
help
```

Note that you can get more detailed descriptions for certain categories. You can quit the shell via CTRL+C.

It is recommended to use the shell. Still, if that is a problem, you can alternatively use other GUI tools that support shell commands, eg:
free: http://mongodb-tools.com/
You can also run the shell from an external client computer, see the part of the tutorial pertaining to the binary files.

See the available databases:

```
show dbs
```

The "local" database is an internal system database that mongo uses to store (meta)data and should not be tampered with. Find out your current database:

```
db
```

Mongo, by default, connects to the „test" database (why wasn't it visible with show dbs?).
Connect to the „advdb"database:

```
use advdb
db
```

Besides the mongo commands, mongo shell, being a javascript interpreter, can execute javascript code. Try it, eg.:

```
var d = new Date()
d
d.getY <press TAB nakon Y>   // you'll get getYear(), don't ask why it returns 115 ☺
var obj = { ime: 'Ana', voli: ['Milovana', 'Ivana']}
obj
```

Mongo shell can be started in a so called „blind mode", ie. You can assign an argument to the mongo shell (expression or e.g. js dile) that the shell will execute; handy for executing e.g. *scheduled scripts*.

1.2.1. Inserting records

Records are organized in collections, somewhat analogous to the relational DB tables. Unlike tables, collections do not have to be explicitly created as they will be created implicitly with the first insert (to the previously non-existing collection).
As for the records, the only schema rule is that every record has to have an "_id" attribute. If "_id" is not assigned on insert, it will be generated by Mongo.

In the advdb try the command listing all the collections:

```
show collections
```

As expected, there are none. Insert a record to the student collection:

```
db.student.save({ime:"Ana", prezime:"Kralj"})
```

Repeat:

```
show collections
```

Besides the student collection, there is also a new system.indexes collection used to store indexes.
Print the inserted record:

```
db.student.find()
```

and note the generated _id field. Its properties are detailed here: http://docs.mongodb.org/manual/reference/object-id/ , e.g. you can extract the date created from the _id field which can come in handy:

```
ObjectId().getTimestamp("<paste here _id>");
```

Now try to insert another two records with an explicitly assigned id:

```
db.student.save( {_id: 1, ime:"Eva", prezime:"Kralj"} )
db.student.find()
db.student.save( {_id: 1, ime:"Mirta", prezime:"Car"} )
db.student.find()
```

What happened?
Now try the insert command:

```
db.student.insert( {_id: 2, ime:"Maksim", prezime:"Beg"} )
db.student.find()
db.student.insert( {_id: 2, ime:"Maks", prezime:"Beg"} )
db.student.find()
```

What is the difference?

Does it make sense to use the insert without default _id field?

1.2.2. Update

Let us try to update an existing record. Assume that we have a collection used to count some events by weekdays. Insert a record for Monday:

```
db.counter.insert({_id: 'mon', cnt: 0})
db.counter.find()
```

Then try to increment the counter by one:

```
var mon = db.counter.findOne({_id: 'mon'});
mon
mon.cnt += 1;
db.counter.save(mon);
db.counter.find();
```

What is the problem with such an approach?

Mongo has an update statement that has the **atomicity** property on **the document level**:

Db.collection.update(query, update, options)

Increment the counter using the update command. You can use Mongo's $inc function (a list of functions can be found here https://docs.mongodb.org/manual/reference/operator/update-field/).

```
db.counter.update( {_id: 'mon'}, { $inc: {cnt:1}})
```

Why is this better than the previous approach?
Add another field to our document, e.g. date modified timestamp – use the $set function:

```
db.counter.update( {_id: 'mon'}, { $set: {dateModified: null }});
db.counter.find();
```

We will update the dateModified on every operation (e.g. incrementing):

```
db.counter.update( {_id: 'mon'},
                { $inc: {cnt:1}, $set: {dateModified: new Date()}})
```

A field can be removed with the $unset operator, and renamed with the $rename operator. Try it.

Let's say we've changed our minds and want to record a timestamp of every operation (not just the last). We'll convert the dateModified field to an array and add an element to the array on every increment (we could've removed dateModified by using $unset, another command would've created a array):

```
db.counter.update( {_id: 'mon'}, { $set: {dateModified: [] }});
db.counter.update( {_id: 'mon'},
                { $inc: {cnt:1}, $push: {dateModified: new Date()}})
```

Repeat the second command multiple times, and inspect the contents (the $push command is analogous to the same Javascript command, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push).

Create another collection and add duplicate elements to the array. The $push command adds elements to the array without checking for duplicates. If you want unique elements (set), then you can use the $addToSet command instead. Try it.

Try the $pull operator that removes an element from the set. How does it behave when there are multiple duplicate elements?

Try the $pop operator that removes an element from the array; note that you can assign negative values as arguments (e.g. 1 and -1), for instance, if you want to undo the last increment:

```
db.counter.update( {_id: 'mon'} ,
                { $inc: {cnt: -1},  $pop: {dateModified : 1 } } )
```

Insert a few more records:

```
db.counter.insert({_id: 'tue', cnt: 1})
db.counter.insert({_id: 'wed', cnt: 2})
db.counter.insert({_id: 'thu', cnt: 3})
db.counter.insert({_id: 'fri', cnt: 4})
db.counter.insert({_id: 'sat', cnt: 5})
db.counter.insert({_id: 'sun', cnt: 6})
db.counter.find()
```

and try to reset them. An empty search criteria {} will select **all** the documents:

```
db.counter.update( { },
                { $set: {cnt: 0, dateModified: []}})
```

What happened? By default, if multiple records are selected, Mongo changes only the first one. That can be changed with multi flag:

```
db.counter.update( { },
                { $set: {cnt: 0, dateModified: []}},
               {multi: true})
```

If you want to change only one document, it is better to use the findAndModify command made just for that purpose. See the docs and try it: https://docs.mongodb.org/manual/reference/command/findAndModify/ .

1.2.3. Querying

You can retrieve records via find function:

```
db.collection.find(query, projection)
```

where both arguments are optional. The first argument gives the object used to select (filter) records, and the second argument defines the projection so that only a part of the document can be retrieved. The function returns a list of records (cursor) that can be iterated over. The Mongo shell iterates automatically and prints the first 20 records; you can continue to iterate using the "it" command.

> Before the retrieval example we are going to load three collections into Mongo **that will be required for solving homework tasks**. Download the NMBP_datasets.zip file from the course web page and follow the instructions for loading the data (they can be found together with the data).

Retrieve all documents:

```
db.cards.find();
it
```

Use the projection to retrieve only name and type fields:

```
db.cards.find({}, {name: 1, type: 1});
```

Obviously, mongo returns the _id field unless it is explicitly turned off:

```
db.cards.find({}, {_id:0, name: 1, type: 1});
```

_id is a special field and the only one that can be mixed (included and excluded) on projections, e.g. if we try to exclude the type field:

```
db.cards.find({}, {_id:0, name: 1, type: 0});
```

we'll get an error: *BadValue Projection cannot have a mix of inclusion and exclusion*.
In other words (with the exception of _id) it is possible to specify either attribute to include or to exclude from the result!

1.2.4. Comparison operators

To define the query criterions you can use the comparation operators described here:
http://docs.mongodb.org/manual/reference/operator/query-comparison/
For instance, retrieve all the cards with the power greater than 4:

```
db.cards.find({power: {$gt: 4}} );
```

The query returns no results. Why? Make sure that the power field exists:

```
db.cards.find({power: { $exists: true }} ).count();
```

What seems to be the problem?
Power field is not an integer, and so the comparison fails.
Convert *power* to integer:

```
db.cards.find({power: {$exists: true}}).forEach(function(obj) {
   obj.power = parseInt(obj.power);
   db.cards.save(obj);
});
```

Retrieve power 99 and pretty print it:

```
db.cards.find({power: {$eq: 99}} ).pretty()
```

Now, try to retrieve all cards with cmc greater than 10:

```
db.cards.find({cmc: {$gt: 10}}).count();
```

Or, the opposite (note the negatioin):

```
db.cards.find({cmc: {$not: {$gt: 10}}}).count();
```

Retrieva all cards having cmc one or ten, we'll use the $in operator (also try $nin):

```
db.cards.find({cmc: {$in: [1, 10]}}).count();
```

Equality and $in/$nin also work over arrays! Try (you can also use $eq):

```
db.cards.find({subtypes: "Human"} ).pretty()
```

Retrieve cards having both Human and Knight subtypes:

```
db.cards.find({subtypes:{$all: ["Human", "Knight"]}} ).count()
```

Let's add an additional nested object "abilities" to all the cards having "Human" subtypes:

```
db.cards.update(
        { subtypes: "Human" },
        { $set: {
            abilites: {
                canFly : "no",
                canWalk: "yes",
                canTalk : "yes"
            }
          }
         }
         , {multi: true}
       );
```

Nested attributes are referenced via dot-notation, e.g.:

```
db.cards.find( {"abilites.canFly": "no", subtypes : "Wizard"} ).count()
```

Also note the AND operator – comma!
One should be careful with nested documents, – what are the results of seemingly same query:

```
db.cards.find( {"abilites" : {canFly: "no"}, subtypes : "Wizard"} ).count()
```

How about the following two:

```
db.cards.find( {"abilites" : {canFly: "no", canWalk: "yes", canTalk : "yes"}, subtypes : "Wizard"} ).count();

db.cards.find( {"abilites" : {canTalk: "no", canWalk: "yes",  canFly: "yes"}, subtypes : "Wizard"} ).count();
```

Why (remember that mongo stores data in the BSON format)?

1.2.5. Cursor operations

Mongo implements a number of cursor commands http://docs.mongodb.org/manual/reference/method/js-cursor/ , but only sort, skip and limit will be commented here (often used for *paging*).
You can sort the results (also using the $orderby operator) in the following way (ascending by cmc, descending by _id):

```
db.cards.find({subtypes:"Wizard"}, {cmc: 1}).sort( {cmc: -1, _id: 1});
```

and then retrieve the second page of size 50:

```
db.cards.find({subtypes:"Wizard"}, {cmc: 1}
        ).sort( {cmc: -1, _id: 1}
        ).skip(50
        ).limit(50);
```

1.2.6. Map/Reduce

Mongo has powerful aggregation functionalities (http://docs.mongodb.org/master/core/aggregation-pipeline/) which, in general, work faster than M/R operations and are limited only by the available built-in operators/expressions. However, certain problems cannot be solved in that environment, whereas M/R is of "unlimited" functionalities because it is founded on user defined js functions that can

perform “anything”. In production systems, the aggregation pipeline should be the first choice. However, since we’re studying M/R as a part of the Course, with Mongo being just one of the implementations, we shall not discuss the aggregation pipeline here, but M/R instead.

Cards (but not all!) have a subtype field, e.g.:

```
"subtypes" : ["Human", "Wizard"], …
```

Let us find out how many different subtypes are there, that is, how many cards are there that have the subtype Human, how many have the subtype Wizard, etc.

The full map reduce syntax can be found here: http://docs.mongodb.org/master/reference/command/mapReduce/#dbcmd.mapReduce

Firstly, we define the map function that iterates the subtype field (if it exists):

```
var map = function() {
    if (this.subtypes !== undefined)
        this.subtypes.forEach( function(subtype) {
            emit( subtype, 1 );
        }
  );
};
```

For instance, this map function will emit two records for the subtype array shown above:

```
Human, 1
Wizard, 1
```

Reduce function receives all records grouped by key (subtype), and simply counts the records:

```
var reduce = function(key, values) {
    var rv = {
              subtype: key,
              count:0
             };
    values.forEach( function(value) {
        rv.count += value;
    });
    return rv;
};
```

> Mongo does not print the result of the MR command but the performance statistics. The result itself must be stored in a collection that can be viewed later. Note, for example, that it is not possible to sort the records with the MR command (Mongo always sorts them by key) as the result is saved in a collection, but they can be sorted by using the *find* command at a later time (analogous to saving to a table and then executing SELECT ORDERY BY).

The following statement executes the M/R function and stores the result into the mr_cards collection (it will be created by Mongo):

```
db.cards.mapReduce(
                    map,
                    reduce,
                    { out: "mr_cards" }
                   )
```

Inspect the result:

```
db.mr_cards.find()
```

Not the result we expected; what happened? It appears, that instead of addition, in certain cases concatenation occurred?

In order to debug this, we shall reduce the input set to only those that have subtype Atnelope:

```
db.cards.mapReduce(
                    map,
                    reduce,
                    { out: "mr_cards",
                      query: { subtypes: "Antelope" }}
                  );
db.mr_cards.find();
```

Works fine!? It appears that the error does not occur on the smaller set?
We'll revert to the entire set, but this time add the printout to reduce function, but only for one subtype ("Whale") to keep the printout comprehensible:

```
var reduce = function(key, values) {
   var rv = {
            subtype: key,
            count:0
           };
   values.forEach( function(value) {
      if (key === 'Whale') print (key + " counting = " + rv.count + " + " + tojson(value));
      rv.count += value;
   });
   if (key === 'Whale') print ("reduce for " + key + " returning " + rv.count);
   return rv;
};
```

Execute M/R again for all records and inspect the printout (visible in the terminal where mongod is run). It is apparent now that Whale works fine for a while, and then numbers become replaced with objects. Find out for yourselves what is going on, be sure to see the combinable reducer slides in the lectures and see the docs for the reduce function: http://docs.mongodb.org/master/reference/command/mapReduce/#mapreduce-reduce-cmd

Finally, the correct map and reduce functions:

```
var map = function() {
   if (this.subtypes !== undefined)
      this.subtypes.forEach( function(subtype) {
         emit( subtype, {count : 1} );
      }
  );
};

var reduce = function(key, values) {
   var rv = {
            subtype: key,
            count:0
           };
   values.forEach( function(value) {
      rv.count += value.count;
   });
   return rv;
};
```

Re-run the M/R. How much time is needed for the calculation?
Note that Mongo also provides the finalize function used to additionally process the results of the reduce phase. Let us use it, e.g. to transform the result format:

```
var fin = function (key, reducedVal) {
   return {count : reducedVal.count};
};

db.cards.mapReduce(
                    map,
                    reduce,
                    { out: "mr_cards",
                      finalize : fin
                    }
                   )
```

This outlines the basic M/R functionalities, as well as basic debugging process. More detailed instructions can be found on the official Mongo pages: http://docs.mongodb.org/master/tutorial/map-reduce-examples/
Note that Mongo also implements the incremental M/R (will not be covered in this course).

1.2.7. Indexes

Read the introduction to indexes at: http://docs.mongodb.org/master/core/indexes-introduction/
Answer the following questions:

- Why is, upon first collection creation, system.indexes collection also created? Inspect the contents of that collection.
- What index types are supported by Mongo?
- Why can't hashed index be used on range queries?
- What properties can an index have?

Inspect and compare query execution plans for two different attributes in our collection:

```
db.cards.find({_id: "Black Knight"}).explain();
db.cards.find({cmc: 3}).explain();
```

Create an index on the cmc attribute, and inspect again system.indexes and execution plans:

```
db.cards.createIndex({cmc : 1})
db.system.indexes.find()
db.cards.find({cmc: 3}).explain();
```

Also, for exercise, create a *multikey index* and *text index* on the appropriate attributes.

1.2.8. Binary files

Binary files (e.g. images) can also be stored to Mongo (using BSON BinData data type), but with limitation of maximum size of 16MB. If the files are larger in size, then GridFs should be used: http://docs.mongodb.org/manual/faq/developers/#faq-developers-when-to-use-gridfs

Of course, it can be used even for smaller files.

A file can be uploaded from the disk to GridFs using the mongofiles utility, e.g.:

```
mongofiles --host 192.168.56.12 --db advdb put -l "D:\path_to_image\img.jpg" img.jpg
```

http://docs.mongodb.org/manual/reference/program/mongofiles/

Note: Replication and fragmentation configuration shown in the following chapters is suitable only for testing and understanding of basic concepts and not for production servers/environment!

Note: These topics are discussed in the forth NoSQL lecture, so you can try them after that lecture. The commands are in Linux format, so Windows users will have to make some moderate changes to the syntax.

1.3. Replication

Read the basics: http://docs.mongodb.org/master/core/replication-introduction/ .
We'll create a replica set with three nodes, test the RW operations, and bring down the master node and cause elections.

Create three new folders where the three new instances will store their data:

```
mkdir -p /usr/mongo/rs0-0 /usr/mongo/rs0-1 /usr/mongo/rs0-2
```

Start three new terminals (you can even leave the standalone instance from the previous examples running side-by-side), log in, and execute the following statements in each of the terminals. Command format corresponds to the linux systems. Customize them for Windows if you use it. At the beginning of the document you have an example of how dbpath works in Windows.

```
mongod --port 27018 --dbpath /usr/mongo/rs0-0 --replSet rs0 --smallfiles --oplogSize 128
mongod --port 27019 --dbpath /usr/mongo/rs0-1 --replSet rs0 --smallfiles --oplogSize 128
mongod --port 27020 --dbpath /usr/mongo/rs0-2 --replSet rs0 --smallfiles --oplogSize 128
```

Now there are three separate instances running, each with their own directory for data storage, but they are not (yet) joined in a replica set.

Start the fourth ☺ terminal, run the mongo shell and connect to one of the instances:

```
mongo --port 27018
```

and configure the replica set:

```
rsconf = {
          _id: "rs0",
          members: [
                    {
                     _id: 0,
                     host: "127.0.0.1:27018"                }
                   ]
        }
rs.initiate( rsconf )
rs.conf()
```

Add the remaining two instances to the replica set (adjust the IP address if it is necessary). By adding the instances, a primary is elected, note the prompt changing in the shell:

```
rs.add("127.0.0.1:27019")
rs.add("127.0.0.1:27020")
```

Execute:

```
rs.conf();
rs.status();
```

Connect to PRIMARY (if not already connected) and save a record to default test database:

```
db.test.save({desc: "first"});
```

Quit the shell and connect to SECONDARY and execute:

```
mongo --port 27019
db.test.save({desc: "second"})
show collections
show dbs
```

Nothing works. Allow reading from the SECONDARY for the current connection:

```
rs.slaveOk()
```

and try again. How would you do that from e.g. web application?

Quit the shell and connect to the PRIMARY. Tell the PRIMARY to step down:

```
rs.stepDown()
```

and note what is happening in the terminals of the remaining two SECONDARY nodes – they are having an election!

Previous PRIMARY is still active in the RS, but now it is SECONDARY.

Connect to the new PRIMARY and shut it down:

```
use admin
db.shutdownServer()
```

Connect to the remaining nodes and inspect the status:

```
rs.status();
```

Shut down the remaining PRIMARY and see what happens.

1.4. Sharding

Read the introduction at: http://docs.mongodb.org/master/core/sharding-introduction/

What follows is optional, you will not be asked to reproduce this in the course exams. What follows is the set of statements that will setup the test sharding clusted with:

- one *config* server,
- one *routing* server, and
- two *shard* servers.

Shut down all servers from previous exercises!
Open a new, first terminal:

Create folder for the *config* server

```
mkdir -p /usr/mongo/configdb
```

and run it (on port 27019):

```
mongod --configsvr --dbpath /usr/mongo/configdb --port 27019
```

Open the second terminal and run the *router (mongos).* As an argument, give it the address of the *config* server:

```
mongos --configdb 127.0.0.1:27019
```

Mongos need no data folder and listens on the port 27017.

Open two new terminals. Create data folders for two new shard servers:

```
mkdir -p /usr/mongo/sh1 /usr/mongo/sh2
```

and run them:

```
mongod --port 27020 --dbpath /usr/mongo/sh1 --smallfiles --oplogSize 128
mongod --port 27021 --dbpath /usr/mongo/sh2 --smallfiles --oplogSize 128
```

Open the fifth terminal☺, and connect via *shell to the routing server*:

```
mongo --host 127.0.0.1 --port 27017
```

Finally, add the *sharding servers (repeat for)*:

```
sh.addShard( "127.0.0.1:27020" )
sh.addShard( "127.0.0.1:27021" )
```

and check the status:

```
sh.status();
```

Currently, nothing is sharded. Enable sharding on the shtest database:

```
use shtest;
sh.enableSharding("shtest");
```

and enable and define sharding on cards collection (currently non-existing) on the _id attribute:

```
sh.shardCollection("shtest.cards", { "_id": "hashed" } )
```

Repeat the loading process for the AllCards.json described previously, and see how the data is sharded:

```
db.cards.getShardDistribution()
sh.status();
```

2. Students' assignments

Students need to complete two assignments described in the following text.

2.1. NoSQL1: Minimal portal based on MongoDB (10 points)

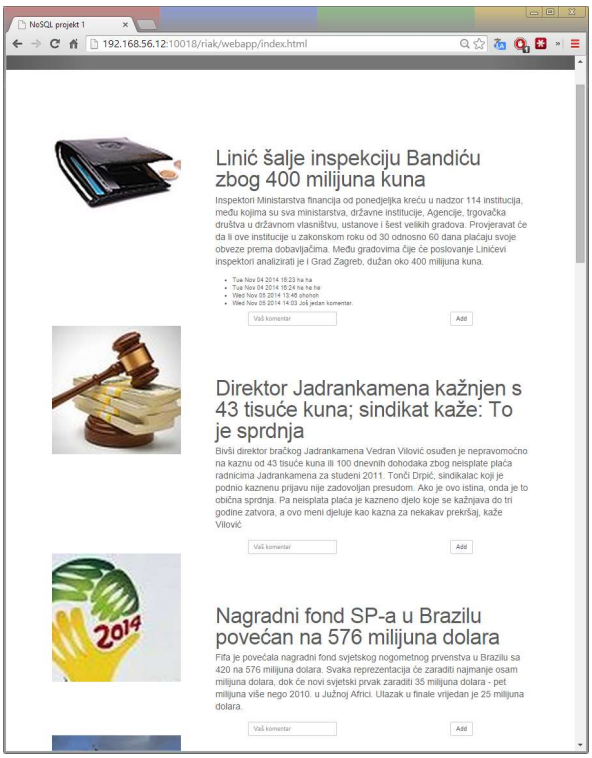
Construct a web portal which displays N most recent (npr. N=10) articles.

> **?** Figure out how to do that, i.e. use the appropriate data structure.
> E.g.: assuming the database contains 100.000 articles it **is not a good strategy** to fetch all 100.000 articles to the client, sort them, and take the top ten.

Each article should (roughly) take the following form:

Title	Image
Text	
Author	

Implement article comments: add a text field for submitting comments beneath every article.
You do not need to construct an interface for entering new articles (but are welcome to do so if you want). You can enter new articles by hand.
Database should contain at least 10.000 articles. You can generate or download a set of data from the Internet, but you need to (programmatically) load at least 10.000 news.
For example:



2.2. NoSQL2: MapReduce (3 + 7 = 10 points)

Write a:
(a) (3 points) MapReduce query returning a list of articles sorted descending by the number of comments.

(b) (7 points) MapReduce query that for each author returns 10 most used words. Use the concept of "word" in the simplest possible way (a series of letters separated with space, coma of full stop).

You don't need to perform any lexical transformations (like stemming). You do not need to return the 11th word if it has the same number of uses as the 10th one.

Incomplete solutions will be awarded partial points (i.e. all word used by an author instead of 10 most used ones).

For both tasks:

You can use arbitrary technology, there is a large number of drivers for MongoDB:
http://docs.mongodb.org/ecosystem/drivers/

Tasks will be personally delivered, i.e. a solution must be demonstrated.

Incomplete solutions (e.g. only the first task, without commenting on the news) will be considered.
The project solutions should contain a readme.txt document in the root directory explaining:
• How the task is solved (what technology, how test data is entered, etc.)
• How to run a solution

Students can be asked to:

• Explain something from the solution (main.txt)
• Explain some concept in the Instructions (e.g. *replica* set)
• launch i.e. demonstrate a solution (and e.g. add news to the portal)
• Make some minor modifications (e.g. modify M / R query slightly)
• etc.