File: tulostettavat/schema2.sql

```
1  DROP TABLE IF EXISTS BlockEditAccess;
2
3  DROP TABLE IF EXISTS BlockViewAccess;
4
5  DROP TABLE IF EXISTS UserGroupMember;
6
7  DROP TABLE IF EXISTS ReadRevision;
8
9  DROP TABLE IF EXISTS BlockRelation;
10
11  DROP TABLE IF EXISTS Block;
12
13  DROP TABLE IF EXISTS NewUser;
14
15  DROP TABLE IF EXISTS User;
16
17  DROP TABLE IF EXISTS UserGroup;
18
19  DROP TABLE IF EXISTS Answer;
20
21  DROP TABLE IF EXISTS UserAnswer;
22
23  DROP TABLE IF EXISTS AnswerTag;
24
25  DROP TABLE IF EXISTS ParMappings;
26
27  DROP TABLE IF EXISTS UserNotes;
28
29  DROP TABLE IF EXISTS ReadParagraphs;
30
31  DROP TABLE IF EXISTS Question;
32
33  DROP TABLE IF EXISTS Lecture;
34
35  DROP TABLE IF EXISTS LectureUsers;
36
37  DROP TABLE IF EXISTS LectureAnswer;
38
39  DROP TABLE IF EXISTS Message;
40
41  CREATE TABLE Answer (
42    id          INTEGER      NOT NULL,
43    task_id     VARCHAR(255) NOT NULL,
44    content     VARCHAR(255) NOT NULL,
45    points      VARCHAR(255), -- TODO: should this be of type REAL?
46    answered_on TIMESTAMP    NOT NULL,
47
48    CONSTRAINT Answer_PK
49    PRIMARY KEY (id)
50  );
51
52  CREATE TABLE UserAnswer (
53    id        INTEGER NOT NULL,
54    answer_id INTEGER NOT NULL,
55    user_id   INTEGER NOT NULL,
56
57    CONSTRAINT UserAnswer_PK
58    PRIMARY KEY (id),
```

```
  CONSTRAINT UserAnswer_id
  FOREIGN KEY (answer_id)
  REFERENCES Answer (id)
  ON DELETE NO ACTION
  ON UPDATE CASCADE
);

CREATE TABLE AnswerTag (
  id        INTEGER      NOT NULL,
  answer_id INTEGER      NOT NULL,
  tag       VARCHAR(255) NOT NULL,

  CONSTRAINT AnswerTag_PK
  PRIMARY KEY (id),
  CONSTRAINT AnswerTag_id
  FOREIGN KEY (answer_id)
  REFERENCES Answer (id)
  ON DELETE NO ACTION
  ON UPDATE CASCADE
);

CREATE TABLE UserGroup (
  id   INTEGER      NOT NULL,
  name VARCHAR(100) NOT NULL,

  CONSTRAINT UserGroup_PK
  PRIMARY KEY (id)
);


CREATE TABLE User (
  id        INTEGER      NOT NULL,
  name      VARCHAR(100) NOT NULL,
  real_name VARCHAR(100),
  email     VARCHAR(100),
  prefs     TEXT,
  pass      VARCHAR(128),

  CONSTRAINT User_PK
  PRIMARY KEY (id)
);

 CREATE TABLE NewUser (
   email   VARCHAR(100),
   pass    VARCHAR(128),
   created TIMESTAMP,

   CONSTRAINT NewUser_PK
   PRIMARY KEY (email)
 );

 CREATE TABLE Block (
   id                INTEGER NOT NULL,
   latest_revision_id INTEGER,
   type_id           INTEGER NOT NULL,
   description       VARCHAR(100), -- better name would be: tag
   created           TIMESTAMP,
   modified          TIMESTAMP,
   UserGroup_id      INTEGER NOT NULL,

   CONSTRAINT Block_PK
```

```
  PRIMARY KEY (id),
  CONSTRAINT Block_id
  FOREIGN KEY (UserGroup_id)
  REFERENCES UserGroup (id)
  ON DELETE NO ACTION
  ON UPDATE CASCADE
);


CREATE TABLE BlockRelation (
  parent_block_specifier   INTEGER NOT NULL,
  parent_block_revision_id INTEGER,
  parent_block_id          INTEGER NOT NULL,
  Block_id                 INTEGER NOT NULL,

  CONSTRAINT BlockRelation_PK
  PRIMARY KEY (Block_id),
  CONSTRAINT BlockRelation_id
  FOREIGN KEY (Block_id)
  REFERENCES Block (id)
  ON DELETE NO ACTION
  ON UPDATE CASCADE
);


CREATE TABLE ReadRevision (
  revision_id INTEGER NOT NULL PRIMARY KEY,
  Block_id    INTEGER NOT NULL,
  Hash        VARCHAR(128),

  CONSTRAINT ReadRevision_id
  FOREIGN KEY (Block_id)
  REFERENCES Block (id)
  ON DELETE CASCADE
  ON UPDATE CASCADE
);


CREATE TABLE UserGroupMember (
  UserGroup_id INTEGER NOT NULL,
  User_id      INTEGER NOT NULL,

  CONSTRAINT UserGroupMember_PK
  PRIMARY KEY (UserGroup_id, User_id),
  CONSTRAINT UserGroupMember_id
  FOREIGN KEY (UserGroup_id)
  REFERENCES UserGroup (id)
  ON DELETE NO ACTION
  ON UPDATE CASCADE,
  CONSTRAINT UserGroupMember_id
  FOREIGN KEY (User_id)
  REFERENCES User (id)
  ON DELETE NO ACTION
  ON UPDATE CASCADE
);


CREATE TABLE BlockViewAccess (
  visible_from TIMESTAMP NOT NULL,
  visible_to   TIMESTAMP,
  Block_id     INTEGER   NOT NULL,
```

```
  UserGroup_id INTEGER   NOT NULL,

  CONSTRAINT BlockViewAccess_PK
  PRIMARY KEY (Block_id, UserGroup_id),
  CONSTRAINT BlockViewAccess_id
  FOREIGN KEY (Block_id)
  REFERENCES Block (id)
  ON DELETE NO ACTION
  ON UPDATE CASCADE,
  CONSTRAINT BlockViewAccess_id
  FOREIGN KEY (UserGroup_id)
  REFERENCES UserGroup (id)
  ON DELETE NO ACTION
  ON UPDATE CASCADE
);


CREATE TABLE BlockEditAccess (
  editable_from TIMESTAMP NOT NULL,
  editable_to   TIMESTAMP,
  Block_id      INTEGER   NOT NULL,
  UserGroup_id  INTEGER   NOT NULL,

  CONSTRAINT BlockEditAccess_PK
  PRIMARY KEY (Block_id, UserGroup_id),
  CONSTRAINT BlockEditAccess_id
  FOREIGN KEY (Block_id)
  REFERENCES Block (id)
  ON DELETE NO ACTION
  ON UPDATE CASCADE,
  CONSTRAINT BlockEditAccess_id
  FOREIGN KEY (UserGroup_id)
  REFERENCES UserGroup (id)
  ON DELETE NO ACTION
  ON UPDATE CASCADE
);

CREATE TABLE ParMappings (
  doc_id    INTEGER NOT NULL,
  doc_ver   INTEGER NOT NULL,
  par_index INTEGER NOT NULL,
  new_ver   INTEGER NOT NULL,
  new_index INTEGER NOT NULL,
  modified  BOOLEAN NOT NULL,

  CONSTRAINT ParMappings_PK
  PRIMARY KEY (doc_id, doc_ver, par_index)
);


CREATE TABLE UserNotes (
  UserGroup_id INTEGER      NOT NULL,
  doc_id       INTEGER      NOT NULL,
  doc_ver      INTEGER      NOT NULL,
  par_index    INTEGER      NOT NULL,
  note_index   INTEGER      NOT NULL,
  content      VARCHAR(255) NOT NULL,
  created      TIMESTAMP    NOT NULL,
  modified     TIMESTAM,
  access       VARCHAR(20)  NOT NULL,
  tags         VARCHAR(20)  NOT NULL,
```

```
242    deprecated   BOOLEAN,
243
244    CONSTRAINT UserNotes_PK
245    PRIMARY KEY (UserGroup_id, doc_id, doc_ver, par_index,
note_index),
246
247    CONSTRAINT UserNotes_id
248    FOREIGN KEY (doc_id, doc_ver, par_index)
249    REFERENCES ParMappings (doc_id, doc_ver, par_index)
250    ON DELETE CASCADE
251    ON UPDATE RESTRICT
252  );
253
254
255  CREATE TABLE ReadParagraphs (
256    UserGroup_id INTEGER   NOT NULL,
257    doc_id       INTEGER   NOT NULL,
258    doc_ver      INTEGER   NOT NULL,
259    par_index    INTEGER   NOT NULL,
260    timestamp    TIMESTAMP NOT NULL,
261    deprecated   BOOLEAN,
262
263    CONSTRAINT ReadParagraphs_PK
264    PRIMARY KEY (UserGroup_id, doc_id, doc_ver, par_index),
265
266    CONSTRAINT ReadParagraphs_id
267    FOREIGN KEY (doc_id, doc_ver, par_index)
268    REFERENCES ParMappings (doc_id, doc_ver, par_index)
269    ON DELETE CASCADE
270    ON UPDATE RESTRICT
271  );
272
273  CREATE TABLE Question (
274    question_id  INTEGER NOT NULL PRIMARY KEY,
275    doc_id       INTEGER NOT NULL,
276    par_index    INTEGER NOT NULL,
277    question_title    TEXT NOT NULL,
278    answer       TEXT,
279    questionJson TEXT
280  );
281
282  CREATE TABLE Lecture (
283    lecture_id   INTEGER,
284    lecture_code TEXT,
285    doc_id       INTEGER NOT NULL,
286    lecturer     INTEGER NOT NULL,
287    start_time   TEXT    NOT NULL,
288    end_time     TEXT,
289    password     TEXT,
290
291    PRIMARY KEY (lecture_id)
292  );
293
294  CREATE TABLE LectureUsers (
295    lecture_id INTEGER,
296    user_id    INTEGER,
297
298    FOREIGN KEY (lecture_id)
299    REFERENCES Lecture (lecture_id)
300    ON DELETE CASCADE,
301
```

```
    FOREIGN KEY (user_id)
    REFERENCES User (user_id)
    ON DELETE CASCADE
  );

  CREATE TABLE `Message` (
    msg_id     INTEGER PRIMARY KEY,
    lecture_id INTEGER NOT NULL,
    user_id    INTEGER NOT NULL,
    message    TEXT    NOT NULL,
    timestamp  TEXT    NOT NULL,

    FOREIGN KEY (lecture_id)
    REFERENCES Lecture (lecture_id)
    ON DELETE CASCADE,

    FOREIGN KEY (user_id)
    REFERENCES User (user_id)
    ON DELETE CASCADE
  );

  CREATE TABLE LectureAnswer (
    answer_id   INTEGER,
    user_id     INTEGER NOT NULL,
    question_id INTEGER NOT NULL,
    lecture_id  INTEGER NOT NULL,
    answer      TEXT    NOT NULL,
    answered_on TEXT    NOT NULL,
    points      REAL,
    PRIMARY KEY (answer_id)
  );
```

File: tulostettavat/timdb/lectureanswers.py

```
  __author__ = 'hajoviin'
  from contracts import contract
  from timdb.timdbbase import TimDbBase


  class LectureAnswers(TimDbBase):
      """
      LectureAnswer class to handle database for lecture answers
      """
       @contract
       def add_answer(self, user_id: "int", question_id: "int",
lecture_id:"int", answer: "string", answered_on: "string",
                      points: "float",
                      commit: 'bool'=True):
          """
          Adds answer to lecture question
          :param user_id: user id
          :param question_id: question id
          :param lecture_id: lecture id
          :param answer: answer
          :param answered_on: time of the answer
          :param points: points from the anwer
          :param commit: commit the database
          :return:
```

```
24          """
25          cursor = self.db.cursor()
26
27          cursor.execute("""
28              INSERT INTO LectureAnswer(user_id,
question_id,lecture_id, answer, answered_on,points)
29              VALUES (?,?,?,?,?,?)
30          """, [user_id, question_id, lecture_id, answer, answered_on,
points])
31
32          if commit:
33              self.db.commit()
34
35      def get_answers_to_question(self, question_id: "int", timestamp:
"string") -> 'list(dict)':
36          """
37          Gets answers from specific question
38          :param question_id: question id
39          :param timestamp: gets answer that came after this time.
40          :return:
41          """
42          cursor = self.db.cursor()
43
44          cursor.execute("""
45              SELECT answer
46              FROM LectureAnswer
47              WHERE question_id = ? AND answered_on > ?
48          """, [question_id, timestamp])
49
50          return self.resultAsDictionary(cursor)
51
52
53      @contract
54      def get_answers_to_questions_from_lecture(self, lecture_id:
"int") -> "list(dict)":
55          """
56          Gets all the naswers to questions from specific lecture
57          :param lecture_id: lecture ID
58          :return:
59          """
60          cursor = self.db.cursor()
61
62          cursor.execute("""
63                          SELECT *
64                          FROM LectureAnswer
65                          WHERE lecture_id = ?
66          """, [lecture_id])
67
68          return self.resultAsDictionary(cursor)
69
70      @contract
71      def delete_answers_from_question(self, question_id: "int",
commit:"bool"=True):
72          """
73          Deletes answers from question
74          :param question_id: question
75          :param commit: commit to database
76          :return:
77          """
78          cursor = self.db.cursor()
79
```

```
80          cursor.execute("""
81                      DELETE FROM LectureAnswer
82                      WHERE question_id = ?
83          """, [question_id])
84
85          if commit:
86              self.db.commit()
```

File: tulostettavat/timdb/lectures.py

```
1   __author__ = 'localadmin'
2
3   from contracts import contract
4   from timdb.timdbbase import TimDbBase
5
6
7   class Lectures(TimDbBase):
8       @contract
9       def create_lecture(self, doc_id: "int", lecturer: 'int',
start_time: "string", end_time: "string",
10                         lecture_code: "string",
11                         password: "string", commit: "bool") -> "int":
12          cursor = self.db.cursor()
13
14          cursor.execute("""
15                             INSERT INTO Lecture(lecture_code,
doc_id,lecturer, start_time, end_time, password)
16                             VALUES (?,?,?,?,?,?)
17                             """, [lecture_code, doc_id, lecturer,
start_time, end_time, password])
18
19          if commit:
20              self.db.commit()
21          lecture_id = cursor.lastrowid
22
23          return lecture_id
24
25
26      @contract
27      def update_lecture(self, lecture_id: "int", doc_id: "int",
lecturer: 'int', start_time: "string", end_time: "string",
28                         lecture_code: "string",
29                         password: "string"):
30
31          cursor = self.db.cursor()
32
33          cursor.execute("""
34                          UPDATE Lecture
35                          SET lecture_code = ?, doc_id = ?, lecturer =
?, start_time = ?, end_time = ?, password = ?
36                          WHERE lecture_id = ?
37                          """, [lecture_code, doc_id, lecturer,
start_time, end_time, password, lecture_id])
38
39          self.db.commit()
40          return lecture_id
41
42      @contract
```

```
43      def delete_lecture(self, lecture_id: 'int', commit: 'bool'):
44          cursor = self.db.cursor()
45
46          cursor.execute(
47              """
48              DELETE
49              FROM Lecture
50              WHERE lecture_id = ?
51              """, [lecture_id])
52
53          if commit:
54              self.db.commit()
55
56      @contract
57      def delete_users_from_lecture(self, lecture_id: 'int', commit:
'bool'=True):
58          cursor = self.db.cursor()
59
60          cursor.execute(
61              """
62              DELETE FROM LectureUsers
63              WHERE lecture_id = ?
64              """, [lecture_id]
65          )
66
67          if commit:
68              self.db.commit()
69
70      @contract
71      def get_lecture(self, lecture_id: "int") -> 'list(dict)':
72          cursor = self.db.cursor()
73
74          cursor.execute(
75              """
76              SELECT *
77              FROM Lecture
78              WHERE lecture_id = ?
79              """, [lecture_id]
80          )
81
82          return self.resultAsDictionary(cursor)
83
84      @contract
85      def get_lecture_by_name(self, lecture_code: "string", doc_id:
"int") -> 'list(dict)':
86          cursor = self.db.cursor()
87
88          cursor.execute(
89              """
90              SELECT *
91              FROM Lecture
92              WHERE lecture_code = ? AND doc_id = ?
93              """, [lecture_code, doc_id]
94          )
95
96          return self.resultAsDictionary(cursor)
97
98      @contract
99      def get_all_lectures_from_document(self, document_id:"int") ->
'list(dict)':
100          cursor = self.db.cursor()
```

```
101
102         cursor.execute(
103             """
104             SELECT *
105             FROM Lecture
106             WHERE doc_id = ?
107             """, [document_id]
108         )
109
110         return self.resultAsDictionary(cursor)
111
112     @contract
113     def join_lecture(self, lecture_id: "int", user_id: "int",
commit: "bool"=True):
114         cursor = self.db.cursor()
115
116         cursor.execute("""
117                         INSERT INTO LectureUsers(lecture_id,
user_id)
118                         VALUES (?,?)
119                         """, [lecture_id, user_id])
120         if commit:
121             self.db.commit()
122
123     @contract
124     def leave_lecture(self, lecture_id: "int", user_id: "int",
commit: "bool"=True):
125         cursor = self.db.cursor()
126
127         cursor.execute("""
128                         DELETE
129                         FROM LectureUsers
130                         WHERE lecture_id = ? AND user_id = ?
131                         """, [lecture_id, user_id])
132
133         if commit:
134             self.db.commit()
135
136     @contract
137     def get_document_lectures(self, doc_id: 'int', time: 'string'):
138         cursor = self.db.cursor()
139
140         cursor.execute("""
141                         SELECT lecture_code, start_time,end_time,
password
142                         FROM Lecture
143                         WHERE doc_id = ? AND end_time > ?
144                         ORDER BY lecture_code
145                         """, [doc_id, time])
146
147         return self.resultAsDictionary(cursor)
148
149     @contract
150     def get_lecture_by_code(self, lecture_code: 'string', doc_id:
'int') -> 'int':
151         cursor = self.db.cursor()
152
153         cursor.execute("""
154                         SELECT lecture_id, password
155                         FROM Lecture
156                         WHERE lecture_code = ? AND doc_id = ?
```

```
                        """, [lecture_code, doc_id])

        return cursor.fetchone()[0]

    @contract
    def check_if_correct_name(self, doc_id: 'int', lecture_code:
'string', lecture_id : 'int') -> 'int':
        cursor = self.db.cursor()

        cursor.execute("""
                        SELECT lecture_id
                        FROM Lecture
                        WHERE lecture_code = ? AND doc_id = ? AND
lecture_id != ?
                        """, [lecture_code, doc_id, lecture_id])

        answer = cursor.fetchall()

        if len(answer) >= 1:
            return False

        return True

    @contract
    def set_end_for_lecture(self, lecture_id: "int", end_time:
"string"):
        cursor = self.db.cursor()

        cursor.execute(
            """
            UPDATE Lecture
            SET end_time = ?
            WHERE lecture_id = ?
            """, [end_time, lecture_id]
        )

        self.db.commit()

    @contract
    def check_if_lecture_is_running(self, lecture_id: "int",
now="string") -> bool:
        cursor = self.db.cursor()

        cursor.execute(
            """
            SELECT lecture_id
            FROM Lecture
            WHERE lecture_id = ? AND end_time > ?
            """, [lecture_id, now]
        )

        lecture_id = cursor.fetchall()
        if len(lecture_id) <= 0:
            return False

        return True

    @contract
    def check_if_in_lecture(self, doc_id: "int", user_id: "int") ->
"tuple":
        """
```

```
213         Check if user is in lecture from specific document
214         :param doc_id: document id
215         :param user_id: user id
216         :return:
217         """
218
219         cursor = self.db.cursor()
220
221         cursor.execute("""
222                             SELECT lecture_id
223                             FROM Lecture
224                             WHERE doc_id = ?
225                             """, [doc_id])
226
227         lecture_ids = cursor.fetchall()
228         if len(lecture_ids) <= 0:
229             return False, 0
230
231         string_of_lectures = ""
232         comma = ""
233         for lecture in lecture_ids:
234             string_of_lectures += comma + str(lecture[0])
235             comma = ","
236
237         if len(string_of_lectures) <= 0:
238             return False, -1
239
240         cursor.execute("""
241                             SELECT lecture_id, user_id
242                             FROM LectureUsers
243                             WHERE lecture_id IN """ + "(" +
string_of_lectures + ")" + """ AND user_id = ?
244                               """, [user_id])
245
246         result = cursor.fetchall()
247         if len(result) > 0:
248             return True, result[0][0]
249         else:
250             return False, -1
251
252     @contract
253     def get_users_from_leture(self, lecture_id: "int") ->
"list(dict)":
254         cursor = self.db.cursor()
255
256         cursor.execute("""
257                              SELECT user_id
258                              FROM LectureUsers
259                              WHERE lecture_id = ?
260                         """, [lecture_id])
261
262         return self.resultAsDictionary(cursor)
263
264     @contract
265     def update_lecture_starting_time(self, lecture_id: "int",
start_time: "string", commit: "bool"=True) -> "dict":
266         cursor = self.db.cursor()
267
268         cursor.execute("""
269                         UPDATE Lecture
270                         SET start_time = ?
```

```
                        WHERE lecture_id = ?
        """, [start_time, lecture_id])

        if commit:
            self.db.commit()

        cursor.execute("""
                    SELECT *
                    FROM Lecture
                    WHERE lecture_id = ?
        """, [lecture_id])

        return self.resultAsDictionary(cursor)[0]

    @contract
    def extend_lecture(self, lecture_id: "int", new_end_time:
"string", commit: "bool"=True):
        cursor = self.db.cursor()

        cursor.execute("""
                    UPDATE Lecture
                    SET end_time = ?
                    WHERE lecture_id = ?
        """,[new_end_time, lecture_id])

        if commit:
            self.db.commit()
```

File: tulostettavat/timdb/messages.py

```
__author__ = 'hajoviin'
from contracts import contract
from timdb.timdbbase import TimDbBase

# TODO: Also need to save the lecture_id to comfirm that we can only
show specific question.


class Messages(TimDbBase):
    @contract
    def delete_messages_from_lecture(self, lecture_id:"int", commit:
"bool"):
        cursor = self.db.cursor()
        cursor.execute(
            """
            DELETE FROM Message
            WHERE lecture_id = ?
            """, [lecture_id]
        )

        if commit:
            self.db.commit()

    @contract
    def get_message(self, msg_id: 'int'):
        cursor = self.db.cursor()
        cursor.execute("""
```

```
                        SELECT user_id,msg_id, message, timestamp
                        FROM Message
                        WHERE msg_id = ?
                        """, [msg_id])

        return self.resultAsDictionary(cursor)

    @contract
    def get_messages(self, lecture_id: 'int') -> 'list(dict)':
        """
        Gets the question
        :return: Questions as a list
        """
        cursor = self.db.cursor()
        cursor.execute("""
                        SELECT msg_id, user_id, message, timestamp
                        FROM Message
                        WHERE lecture_id = ?
                        """, [lecture_id]
        )

        return self.resultAsDictionary(cursor)

    @contract
    def get_new_messages(self, lecture_id: 'int',
client_last_message: 'int') -> 'list(dict)':
        cursor = self.db.cursor()
        cursor.execute("""
                        SELECT *
                        FROM Message
                        WHERE lecture_id = ? AND msg_id > ?
                        ORDER BY msg_id
                        DESC
                        """, [lecture_id,client_last_message ])

        return self.resultAsDictionary(cursor)

    def get_last_message(self, lecture_id: "int") -> 'list(dict)':
        cursor = self.db.cursor()
        cursor.execute("""
                        SELECT *
                        FROM Message
                        WHERE lecture_id = (?)
                        ORDER BY msg_id
                        DESC
                        LIMIT 1
                        """, [lecture_id])

        return self.resultAsDictionary(cursor)

    @contract
    def add_message(self, user_id: 'int', lecture_id: 'int',
message: 'str', timestamp: 'str',
                    commit: 'bool'=True) -> 'int':
        """ Creates a new message
        """

        cursor = self.db.cursor()
        cursor.execute("""
                         INSERT INTO
                         Message(user_id,lecture_id,message,timestamp)
```

```
85                          VALUES(?,?,?,?)
86                          """, [user_id, lecture_id, message,
timestamp])
87          if commit:
88              self.db.commit()
89          msg_id = cursor.lastrowid
90          return msg_id
```

File: tulostettavat/timdb/questions.py

```
1   __author__ = 'hajoviin'
2   from contracts import contract
3   from timdb.timdbbase import TimDbBase
4
5
6   class Questions(TimDbBase):
7       # TODO: Doesn't work until question table has been altered
8       @contract
9       def get_paragraphs_question(self, doc_id: 'int', par_index:
'int'):
10          """
11          Gets the questions of some paragraph
12          :param par_id: Paragraph to get question.
13          :return: The list of question from that paragraph
14          """
15
16          cursor = self.db.cursor()
17          cursor.execute("""
18                          SELECT question_id, question, answer
19                          FROM Question
20                          WHERE doc_id = ? AND par_index = ?
21                        """, [doc_id, par_index])
22          return self.resultAsDictionary(cursor)
23
24      @contract
25      def delete_question(self, question_id: 'int',
commit:'bool'=True):
26          """
27          Deletes the question from database
28          :param question_id: question to delete
29          """
30
31          cursor = self.db.cursor()
32
33          cursor.execute(
34              """
35                  DELETE FROM Question
36                  WHERE question_id = ?
37              """, [question_id])
38
39          if commit:
40              self.db.commit()
41
42      @contract
43      def get_question(self, question_id: 'int') -> 'list(dict)':
44          """
45          Gets question with specific id
46          :param question_id: question id
```

```
47          :return: the question
48          """
49          cursor = self.db.cursor()
50
51          cursor.execute(
52              """
53              SELECT *
54              FROM Question
55              WHERE question_id = ?
56              """, [question_id])
57
58          return self.resultAsDictionary(cursor)
59
60
61      @contract
62      def get_questions(self) -> 'list(dict)':
63          """
64          Gets the question
65          :return: Questions as a list
66          """
67          cursor = self.db.cursor()
68          cursor.execute("""SELECT id, question, answer FROM Question
"""")
69
70          return self.resultAsDictionary(cursor)
71
72      @contract
73      def add_questions(self, doc_id: 'int', par_index: 'int',
question_title: 'str', answer: 'str', questionJson: 'str',
74                        commit: 'bool'=True) -> 'int':
75          """
76          Creates a new questions
77          :param question_title: Question to be saved
78          :param answer: Answer to the question
79          :param commit: Commit or not to commit
80          :return: The id of the newly creater question
81          """
82
83          cursor = self.db.cursor()
84          cursor.execute("""
85                          INSERT INTO Question (doc_id, par_index,
question_title, answer, questionJson)
86                          VALUES(?,?,?,?,?)
87                          """, [doc_id, par_index, question_title,
answer, questionJson])
88          if commit:
89              self.db.commit()
90          question_id = cursor.lastrowid
91          return question_id
92
93
94      @contract
95      def update_question(self, question_id: 'int', doc_id: 'int',
par_index:'int', question_title: 'str', answer: 'str', questionJson:
'str') -> 'int':
96          """
97          Updates the question with particular id
98          """
99
100          cursor = self.db.cursor()
101          cursor.execute("""
```

```
102                         UPDATE Question
103                         SET doc_id = ?, par_index = ?,
question_title = ?, answer = ?, questionJson = ?
104                         WHERE question_id = ?
105                         """, [doc_id, par_index, question_title,
answer, questionJson, question_id])
106
107         self.db.commit()
108         return question_id
109
110     @contract
111     def get_doc_questions(self, doc_id: 'int') -> 'list(dict)':
112         """
113         Gets questions related to a specific document
114         """
115         cursor = self.db.cursor()
116         cursor.execute("""
117                     SELECT *
118                     FROM Question
119                     WHERE doc_id = ?
120                     """, [doc_id])
121
122         return self.resultAsDictionary(cursor)
123
124     def get_multiple_questions(self, question_ids: 'int[]') ->
'list(dict)':
125         """
126         Gets multiple questions
127         :param question_ids: quesitons ids as integet array
128         :return: list of dictionaries of the matching questions.
129         """
130
131         cursor = self.db.cursor()
132         for_db = str(question_ids)
133         for_db = for_db.replace("[", "")
134         for_db = for_db.replace("]", "")
135         cursor.execute("""
136                     SELECT *
137                     FROM Question
138                     WHERE question_id IN (""" + for_db + """)
139                     """)
140
141         return self.resultAsDictionary(cursor)
```

File: tulostettavat/timdb/timdb2.py

```
1  """
2  Another version of TimDb that stores documents as whole.
3  """
4
5  import sqlite3
6  from contracts import contract
7  from timdb.notes import Notes
8  from timdb.users import Users
9  from timdb.images import Images
10  from timdb.documents import Documents
11  from timdb.answers import Answers
12  from timdb.readings import Readings
```

```
13  from timdb.questions import Questions
14  from timdb.messages import Messages
15  from timdb.lectures import Lectures
16  from timdb.folders import Folders
17  from timdb.lectureanswers import LectureAnswers
18  import os
19
20
21  TABLE_NAMES = ['BlockEditAccess',
22                 'BlockViewAccess',
23                 'UserGroupMember',
24                 'ReadRevision',
25                 'BlockRelation',
26                 'Block',
27                 'User',
28                 'UserGroup',
29                 'Question',
30                 'Messages',
31                 'Lectures',
32                 'LectureAnswers']
33
34
35  class TimDb(object):
36      """Handles saving and retrieving information from TIM
database."""
37
38      @contract
39      def __init__(self, db_path: 'str', files_root_path: 'str',
current_user_name='Anonymous'):
40          """Initializes TimDB with the specified database and root
path.
41
42          :param db_path: The path of the database file.
43          :param files_root_path: The root path where all the files
will be stored.
44          """
45          self.files_root_path = os.path.abspath(files_root_path)
46
47          # TODO: Make sure that files_root_path is valid!
48
49          self.blocks_path = os.path.join(self.files_root_path,
'blocks')
50          for path in [self.blocks_path]:
51              if not os.path.exists(path):
52                  os.makedirs(path)
53
54          self.db = sqlite3.connect(db_path)
55          self.db.row_factory = sqlite3.Row
56          self.notes = Notes(self.db, files_root_path, 'notes',
current_user_name)
57          self.readings = Readings(self.db, files_root_path, 'notes',
current_user_name)
58          self.users = Users(self.db, files_root_path, 'users',
current_user_name)
59          self.images = Images(self.db, files_root_path, 'images',
current_user_name)
60          self.documents = Documents(self.db, files_root_path,
'documents', current_user_name)
61          self.answers = Answers(self.db, files_root_path, 'answers',
current_user_name)
62          self.questions = Questions(self.db, files_root_path,
```

```
'questions', current_user_name)
63          self.messages = Messages(self.db, files_root_path,
'messages', current_user_name)
64          self.lectures = Lectures(self.db, files_root_path,
'lectures', current_user_name)
65          self.folders = Folders(self.db, files_root_path, 'folders',
current_user_name)
66          self.lecture_answers = LectureAnswers(self.db,
files_root_path, 'lecture_answers', current_user_name)
67
68      def clear(self):
69          """Clears the contents of all database tables."""
70          for table in TABLE_NAMES:
71              self.db.execute('delete from ' + table)  # TABLE_NAMES
is constant so no SQL injection possible
72
73      def commit(self):
74          """Commits any changes to the database"""
75          self.db.commit()
76
77      def close(self):
78          """Closes the database connection."""
79          self.db.commit()
80          self.db.close()
81
82      def initializeTables(self, schema_file='schema2.sql'):
83          """Initializes the database from the schema2.sql file.
84          NOTE: The database is emptied if it exists."""
85          with open(schema_file, 'r') as schema_file:
86              self.db.cursor().executescript(schema_file.read())
87          self.db.commit()
88
```