Simple YAML Schemas

The core idea of `dbmisc` is to describe the database schema in a simple YAML file with one main entry for each table. Here is an example entry for a table `user` describing its columns and indices:

```
user:
  table:
    email: TEXT
    name: TEXT
    age: INTEGER
    female: BOOLEAN
    birthday: DATE
    last_edit: DATETIME
  unique_index:
    - email
  index:
    - age
```

The function dbCreateSQLiteFromSchema allows to quickly create an SQLite database from such a schema. More importantly, it also allows to update an existing database if the schema changed. This is quite helpful for me since smaller updates, like adding new columns, happen quite regularly in my projects.

`dbmisc` has several utility functions to get, insert or update data into a table that can use schemas.

Conveniently inserting data

For example, the following code can be used to insert an entry into our table `user`.

```
new_user = list(last_edit=Sys.time(), female=TRUE,
email="test@email.com",
  name="Jane Doe", birthday = "2000-01-01", gender="female")

dbInsert(db,table="user", new_user)
```

Note that the R object `new_user` has several *problems*:

- Its fields are in a different order than in the database table `user`.
- It contains the extra field `gender` that is not in the database table.
- It does not contain the field `age` that is specified in the database table.
- It has datetime and date variables whose conversion can in general be tricky.

Since I try to avoid writing boilerplate code, it is rather common in my Shiny apps that the R representation of my data has similar problems. Thanks to the information in the database schema, dbInsert can handle these issues automatically: the fields will be arranged in the right order, converted to the correct SQLite data types, additional fields in the R data set will just be ignored, and missing fields will be set to `NA`.

Of course, such auto-correction also has risks in so far that certain errors in the entered data may be less likely to be detected. However, in my personal experience from programming smaller to medium sized apps, the convenience not to have to write the boilerplate code to manually fit my R objects to the database definition outweighs the risks of overlooking bugs.

If you set the argument `run=FALSE`, `dbInsert` performs no database action but just returns the SQL statement that would be run:

```
dbInsert(db,table="user", new_user,run = FALSE)
```

```
"insert into user values (:email, :name, :age, :female, :birthday,
:last_edit)"
```

Note that `dbmisc` prepares in its default usage parametrized queries to avoid SQL-injections.

Opening a database connection with a schema

For `dbInsert` to perform these corrections, it needs to know the database schema. The simplest way is to assign it as attribute to the database connection `db`. This can be done, e.g. by opening the connection via the function dbConnectSQLiteWithSchema.

Side Remark: Use dbWithTransaction

Hint: When my Shiny apps insert, update or delete data they often have to modify several tables. Then I always combine the multiple dbInsert, dbUpdate or dbDelete commands inside a call to dbWithTransaction from the `DBI` package to reduce the risk that my database ends up in an undesirable state should errors occur in some subset of the commands.

Getting data with dbGet

The function dbGet facilitates data retrieval, in particular my most common task to get entries from a single table. Here is an example that selects a single entry from the `user` table:

```
dbGet(db, "user", list(email="albert.einstein@uni-ulm.de"))
```

(Yeah, Einstein was indeed born in Ulm, but that was long before our university was founded…) Of course, `dbGet` can handle also more complex queries, as described in its reference. If a schema is specified, data types should be all correctly converted from SQLite to R.