**МIНIСТЕРСТВО  ОСВIТИ І НАУКИ  УКРАЇНИ**

**НАЦІОНАЛЬНИЙ   ТЕХНІЧНИЙ УНІВЕРСИТЕТ   УКРАЇНИ**

**“КИЇВСЬКИЙ  ПОЛІТЕХНІЧНИЙ  ІНСТИТУТ**

**ІМЕНІ ІГОРЯ СІКОРСЬКОГО”**

**Факультет прикладної математики**

**Кафедра програмного забезпечення комп’ютерних систем**

**Лабораторна робота № 9**

**з дисципліни “ Основи програмування ”**

**тема “Реляційна модель даних”**

|  |  |  |
| --- | --- | --- |
| **Виконав**  **студент I курсу**  **групи КП-82**  **Мельничук Олексій Геннадійович**  **варіант №15** |  | **Перевірив**  **“\_\_\_\_” “\_\_\_\_\_\_\_\_\_\_\_\_” 20\_\_\_ р.**  **викладач**  **Гадиняк Руслан Анатолійович**  **(прізвище, ім’я, по батькові)** |

**Київ 2019**

**Мета роботи**

Навчитись створювати мультивіконні графічні додатки.

Реалізувати ідентифікацію, аутентифікацію та авторизацію у програмі.

Опанувати різні види зв'язків між таблицями реляційної бази даних

**Постановка завдання**

Кроки виконання:

Скопіювати у проект даного завдання ваше GUI рішення лабораторної роботи №7.

Скопіювати у проект даного завдання модуль sqlite\_storage та файл бази даних data/sqlite/data.sqlite з лабораторної роботи №8.

Модель даних. Створити базу даних за схемою з ERD діаграми додатку А:

Створити у базі даних нову таблицю users з атрибутами:

id - цілочисельний автоінкрементний ключ

username - рядок, логін користувача

password\_hash - рядок, хеш пароля користувача

Обрати алгоритм хешування паролів (наприклад, MD5).

Додати через SQLite клієнт 2-3 записи у таблицю користувачів, password\_hash згенерувати з паролю за допомогою обраного алгоритму хешування.

Сформувати між таблицею users та таблицею основних сутностей відношення One-To-Many.

Для цього додати у таблицю основних сутностей foreign key поле user\_id.

Сформувати між таблицею основних сутностей та таблицею додаткових сутностей відношення Many-To-Many.

Для цього створити таблицю links з трьома полями:

PK Ідентифікатором зв'язку

FK Ідентифікатором запису з таблиці основних сутностей

FK Ідентифікатором запису з таблиці додаткових сутностей

Розширення функціональності сховища. Додати нові чисті віртуальні функції у абстрактний клас Storage і реалізувати їх у всіх класах-наслідниках (у CsvStorage та XmlStorage можна поставити методи-заглушки, які нічого не роблять):

Отримання опціонала користувача за його логіном та паролем

Отримання вектора тих основних сутностей, що належать залогіненому користувачу

Отримання вектора всіх додаткових сутностей, що належать даній основній сутності

Додавання вказаної додаткової сутності до вказаної основної сутності

Видалення вказаної додаткової сутності із вказаної основної сутності

Розширення графічного інтерфейсу. Вікна:

Реалізувати вікно аутентифікації:

Розмістити форму вводу логіна і пароля та кнопку підтвердження

UPD-1. Пароль має приховувати ввід користувача (echoMode=Password).

UPD-1. Після вводу даних вікно перевіряє чи існує в базі даних користувач з таким логіном та хешем паролю і показує головне вікно програми або сповіщає користувача про помилку аутентифікації (повідомлення у лейблі даного вікна або у стандартному діалоговому вікні).

UPD-1. Головне вікно програми після входу "знає" про аутентифікованого користувача. Для цього можна додати йому поле для зберігання структури чи ідентифікатора користувача, що увійшов в програму.

UPD-1. Вікно реєстрації користувачів опціональне. Може бути доступним лише перед входом користувача на вікні аутентифікації чи га головному вікні.

Меню головного вікна програми:

UPD-1. Видалити пункти меню Open i Save головного вікна та відкривати сховище по заданому у коді шляху. Або залишити пункт Open і дозволити користувачу обирати шлях до сховища і лише потім відкривати для нього вікно входу користувача.

UPD-1 (на майбутнє, КП-8х - можете не робити). Додати пункт меню Logout, що закриває сховище, "забуває" інформацію про аутентифікованого користувача і показує вікно аутентифікації для повторного входу, можливо іншого окристувача.

На головному вікні відображати тільки ті основні сутності, що належать поточному користувачу, що увійшов у систему.

Всі новостворені сутності будуть належати лише цьому користувачу.

На вікні редагування основної сутності додати графічний список додаткових сутностей, що зв'язані зв'язками з обраною основною сутністю:

Можна додати у цей список сутність із існуючих додаткових сутностей (створити новий зв'язок, а не додаткову сутність)

Можна видалити з цього списку обрану додаткову сутність (видаляється лише зв'язок, а не сама додаткова сутність)

UPD-1. Можливості створення і видалення додаткових сутностей опціональні.

**Тексти коду програм**

|  |
| --- |
| **main.cpp** |
| #include "mainwindow.h"  #include <QApplication>  int main(int argc, char \*argv[])  {  QApplication a(argc, argv);  MainWindow w;  w.show();  return a.exec();  } |

|  |
| --- |
| **storage.h** |
| #pragma once  #include <iostream>  #include <fstream>  #include <vector>  #include <sstream>  #include "optional.h"  #include "dino.h"  #include "period.h"  #include "user.h"  using namespace std;  class Storage  {  public:  virtual ~Storage() {}  virtual bool open() = 0;  virtual bool close() = 0;  // dinosaurs  virtual vector<Dino> getAllDinos(void) = 0;  virtual optional<Dino> getDinoById(int Dino\_id) = 0;  virtual bool updateDino(const Dino &dvno) = 0;  virtual bool removeDino(int Dino\_id) = 0;  virtual int insertDino(const Dino &dvno) = 0;  // periods  virtual vector<Per> getAllPers(void) = 0;  virtual optional<Per> getPerById(int Per\_id) = 0;  virtual bool updatePer(const Per &pr) = 0;  virtual bool removePer(int Per\_id) = 0;  virtual int insertPer(const Per &pr) = 0;  //new functs  virtual vector<Dino> getAllUserDino(int user\_id) = 0;  virtual optional<User> getUserAuth(string & username, string & password) = 0;  virtual vector<Per> getAllDinoPer(int dino\_id) = 0;  virtual bool insertDinoPer(int dino\_id, int per\_id) = 0;  virtual bool removeDinoPer(int dino\_id, int per\_id) = 0;  }; |

|  |
| --- |
| **csv\_storage.h** |
| #pragma once  #include <vector>  #include <string>  #include "dino.h"  #include "period.h"  #include "csv.h"  #include "optional.h"  #include "storage.h"  using std::string;  using std::vector;  class CsvStorage : public Storage  {  const string dir\_name\_;  vector<Dino> dinos\_;  vector<Per> pers\_;  static Dino rowToDino(const CsvRow &row);  static CsvRow DinoToRow(const Dino &dino);  static Per rowToPer(const CsvRow &row);  static CsvRow PerToRow(const Per &pr);  int getNewDinoId();  int getNewPerId();  public:  CsvStorage(const string & dir\_name) : dir\_name\_(dir\_name) { }  bool open();  bool close();  // Dinos  vector<Dino> getAllDinos();  optional<Dino> getDinoById(int Dino\_id);  bool updateDino(const Dino &dvno);  bool removeDino(int Dino\_id);  int insertDino(const Dino &dvno);  vector<Per> getAllPers();  optional<Per> getPerById(int Per\_id);  bool updatePer(const Per &pr);  bool removePer(int Per\_id);  int insertPer(const Per &pr);  //new functs  vector<Dino> getAllUserDino(int user\_id);  optional<User> getUserAuth(string & username, string & password);  vector<Per> getAllDinoPer(int dino\_id);  bool insertDinoPer(int dino\_id, int per\_id);  bool removeDinoPer(int dino\_id, int per\_id);  }; |

|  |
| --- |
| **csv\_storage.cpp** |
| #include "dino.cpp"  #include "period.cpp"  #include "csv\_storage.h"  #include <fstream>  #include <iostream>  using namespace std;  string fillStringFromFile(string fromw)  {  string csv;  string line;  ifstream myfile(fromw);  string::iterator it = line.end();  if (myfile.is\_open())  {  while (getline(myfile, line))  {  line+="\n";  csv += line;  }  myfile.close();  }  else  {  cerr << "File was not found: " << fromw << endl;  abort();  }  it = csv.end();  it--;  \*it = '\0';  return csv;  }  void writeToFile(string from, string tofile)  {  ofstream myfile(tofile);  if(myfile.is\_open())  {  myfile << from;  myfile.close();  }  else  {  cout << "Cannot open file\n"  << endl;  abort();  }  }  bool CsvStorage::open()  {  string dinos\_filename = this->dir\_name\_ + "/data.csv";  string dinos\_csv = fillStringFromFile(dinos\_filename);  CsvTable dinos\_table = Csv::createTableFromString(dinos\_csv);  for(CsvRow & row : dinos\_table)  {  Dino newone;  newone.id = stoi(row[0]);  newone.name = row[1];  newone.per = row[2];  newone.fam = row[3];  newone.disc = stoi(row[4]);  newone.mass = stod(row[5]);  this->dinos\_.push\_back(newone);  }  string pers\_filename = this->dir\_name\_ + "/data2.csv";  string pers\_csv = fillStringFromFile(pers\_filename);  CsvTable pers\_table = Csv::createTableFromString(pers\_csv);  for(CsvRow & row2 : pers\_table)  {  Per newone;  newone.id = stoi(row2[0]);  newone.name = row2[1];  newone.time = stod(row2[2]);  this->pers\_.push\_back(newone);  }  return true;  }  bool CsvStorage::close()  {  string dinos\_filename = this->dir\_name\_ + "/data.csv";  CsvTable newt;  for(Dino & a : this->dinos\_)  {  CsvRow row;  row.push\_back(to\_string(a.id));  row.push\_back(a.name);  row.push\_back(a.per);  row.push\_back(a.fam);  row.push\_back(to\_string(a.disc));  row.push\_back(to\_string(a.mass));  newt.push\_back(row);  }  string csv\_txt = Csv::createStringFromTable(newt);    writeToFile(csv\_txt, dinos\_filename);  string pers\_filename = this->dir\_name\_ + "/data2.csv";  CsvTable newt2;  for(Per & p : this->pers\_)  {  CsvRow row;  row.push\_back(to\_string(p.id));  row.push\_back(p.name);  row.push\_back(to\_string(p.time));  newt2.push\_back(row);  }  string csv\_txt2 = Csv::createStringFromTable(newt2);    writeToFile(csv\_txt2, pers\_filename);  return true;  }  vector<Dino> CsvStorage::getAllDinos()  {  return this->dinos\_;  }  optional<Dino> CsvStorage::getDinoById(int Dino\_id)  {  for(Dino & c : this->dinos\_)  {  if(c.id == Dino\_id)  {  return c;  }  }  return nullopt;  }  bool CsvStorage::removeDino(int Dino\_id)  {  int index = -1;  for(int i = 0; i < this->dinos\_.size(); i++)  {  if(this->dinos\_[i].id == Dino\_id)  {  index = i;  break;  }  }  if(index >= 0)  {  this->dinos\_.erase(this->dinos\_.begin() + index);  return true;  }  return false;  }  int CsvStorage::getNewDinoId()  {  int max\_id = 0;  for(Dino & d : this->dinos\_)  {  if(d.id > max\_id)  {  max\_id = d.id;  }  }  int new\_id = max\_id + 1;  return new\_id;  }  int CsvStorage::insertDino(const Dino &dvno)  {  int new\_id = this->getNewDinoId();  Dino copy = dvno;  copy.id = new\_id;  this->dinos\_.push\_back(copy);  return new\_id;  }  bool CsvStorage::updateDino(const Dino &dvno)  {  int Dino\_id = dvno.id;  for(Dino & c : this->dinos\_)  {  if(c.id == Dino\_id)  {  c.disc = dvno.disc;  c.fam = dvno.fam;  c.mass = dvno.mass;  c.name = dvno.name;  c.per = dvno.per;  return true;  }  }  return false;  }  vector<Per> CsvStorage::getAllPers()  {  return this->pers\_;  }  optional<Per> CsvStorage::getPerById(int id)  {  for(Per & p : this->pers\_)  {  if(p.id == id)  {  return p;  }  }  return nullopt;  }  bool CsvStorage::removePer(int id)  {  int index = -1;  for(int i = 0; i < this->pers\_.size(); i++)  {  if(this->pers\_[i].id == id)  {  index = i;  break;  }  }  if(index >= 0)  {  this->pers\_.erase(this->pers\_.begin() + index);  return true;  }  return false;  }  int CsvStorage::getNewPerId()  {  int max\_id = 0;  for(Per & p : this->pers\_)  {  if(p.id > max\_id)  {  max\_id = p.id;  }  }  int new\_id = max\_id + 1;  return new\_id;  }  int CsvStorage::insertPer(const Per &pr)  {  int new\_id = this->getNewPerId();  Per copy = pr;  copy.id = new\_id;  this->pers\_.push\_back(copy);  return new\_id;  }  bool CsvStorage::updatePer(const Per &pr)  {  int Per\_id = pr.id;  for(Per & p : this->pers\_)  {  if(p.id == Per\_id)  {  p.name = pr.name;  p.time = pr.time;  return true;  }  }  return false;  }  vector<Dino> CsvStorage::getAllUserDino(int user\_id)  {  vector<Dino> dinos;  Dino d;  d.id = 0;  d.name = "";  d.fam = "";  d.mass = 0;  d.per = "";  d.user\_id = 0;  d.disc = 0;  dinos.push\_back(d);  return dinos;  }  optional<User> CsvStorage::getUserAuth(string & username, string & password)  {  return nullopt;  }  vector<Per> CsvStorage::getAllDinoPer(int dino\_id)  {  vector<Per> pers;  Per p;  p.id = 0;  p.name = "";  p.time = 0;  pers.push\_back(p);  return pers;  }  bool CsvStorage::insertDinoPer(int dino\_id, int per\_id)  {  return false;  }  bool CsvStorage::removeDinoPer(int dino\_id, int per\_id)  {  return false;  } |

|  |
| --- |
| **xml\_storage.h** |
| #pragma once  #include <QtCore>  #include <QtXml>  #include <QString>  #include <QFile>  #include <QtXml>  #include <QDebug>  #include <QTextStream>  #include <string>  #include <vector>  #include "dino.h"  #include "period.h"  #include "optional.h"  #include "storage.h"  using std::string;  using std::vector;  class XmlStorage : public Storage  {  const string dir\_name\_;  vector<Dino> dinos\_;  vector<Per> pers\_;  int getNewDinoId();  int getNewPerId();  public:  XmlStorage(const string & dir\_name) : dir\_name\_(dir\_name) { }  bool open();  bool close();  vector<Dino> getAllDinos();  optional<Dino> getDinoById(int Dino\_id);  bool updateDino(const Dino &dvno);  bool removeDino(int Dino\_id);  int insertDino(const Dino &dvno);  vector<Per> getAllPers();  optional<Per> getPerById(int Per\_id);  bool updatePer(const Per &pr);  bool removePer(int Per\_id);  int insertPer(const Per &pr);  vector<Dino> getAllUserDino(int user\_id);  optional<User> getUserAuth(string & username, string & password);  vector<Per> getAllDinoPer(int dino\_id);  bool insertDinoPer(int dino\_id, int per\_id);  bool removeDinoPer(int dino\_id, int per\_id);  bool loadfromstr(string filename, int strtype);  }; |

|  |
| --- |
| **xml\_storage.cpp** |
| #include "xml\_storage.h"  using namespace std;  Dino domElementToDino(QDomElement & element)  {  Dino d;  d.id = element.attributeNode("id").value().toInt();  d.name = element.attributeNode("name").value().toStdString();  d.per = element.attributeNode("period").value().toStdString();  d.fam = element.attributeNode("family").value().toStdString();  d.disc = element.attributeNode("discovery").value().toInt();  d.mass = element.attributeNode("mass").value().toFloat();  return d;  }  Per domElementToPer(QDomElement & element)  {  Per p;  p.id = element.attributeNode("id").value().toInt();  p.name = element.attributeNode("name").value().toStdString();  p.time = element.attributeNode("time").value().toFloat();  return p;  }  bool errorcheck(QString & text, QDomDocument & doc)  {  QString errorMessage;  int errorLine;  int errorCol;  bool is\_parsed = doc.setContent(text, &errorMessage, &errorLine, &errorCol);  if(!is\_parsed)  {  qDebug() << "Error parsing xml: " << errorMessage;  qDebug() << "In line: " << errorLine;  qDebug() << "In colu: " << errorCol;  return false;  }  return true;  }  bool XmlStorage::loadfromstr(string filename, int strtype)  {  QString s\_filename = QString::fromStdString(filename);  QFile s\_file(s\_filename);  bool is\_opened = s\_file.open(QFile::ReadOnly);  if (!is\_opened)  {  cout << "file wasnt found" << endl;  return false;  }  QTextStream ts(&s\_file);  QString text = ts.readAll();  QDomDocument doc;  if(!errorcheck(text, doc))  {  return false;  }  QDomElement root = doc.documentElement();  for(int i = 0; i < root.childNodes().size(); i++)  {  QDomNode node = root.childNodes().at(i);  if(node.isElement())  {  QDomElement element = node.toElement();  if(strtype == 1)  {  Dino dino = domElementToDino(element);  this->dinos\_.push\_back(dino);  }  if(strtype == 2)  {  Per per = domElementToPer(element);  this->pers\_.push\_back(per);  }  }  }  s\_file.close();  return true;  }  bool XmlStorage::open()  {  string filename = this->dir\_name\_ + "/data.xml";  if(!loadfromstr(filename, 1))  {  return false;  }  string filename2 = this->dir\_name\_ + "/data2.xml";  if(!loadfromstr(filename2, 2))  {  return false;  }  return true;  }  QDomElement dinoToDomEl(QDomDocument & doc, Dino & dino)  {  QDomElement dino\_el = doc.createElement("dino");  dino\_el.setAttribute("id", dino.id);  dino\_el.setAttribute("name", dino.name.c\_str());  dino\_el.setAttribute("period", dino.per.c\_str());  dino\_el.setAttribute("family", dino.fam.c\_str());  dino\_el.setAttribute("discovery", dino.disc);  dino\_el.setAttribute("mass", dino.mass);  return dino\_el;  }  QDomElement perToDomEl(QDomDocument & doc, Per & per)  {  QDomElement per\_el = doc.createElement("period");  per\_el.setAttribute("id", per.id);  per\_el.setAttribute("name", per.name.c\_str());  per\_el.setAttribute("time", per.time);  return per\_el;  }  bool tofile(QDomDocument & doc, string & filename)  {  QString xml\_text = doc.toString(4);  QString s\_filename = QString::fromStdString(filename);  QFile file(s\_filename);  if(!file.open(QFile::WriteOnly))  {  qDebug() << "Couldn't open file to write : " << s\_filename;  return false;  }  QTextStream ts(&file);  ts << xml\_text;  file.close();  return true;  }  bool XmlStorage::close()  {  QDomDocument doc;  QDomElement root = doc.createElement("dinos");  for(Dino & dino : this->dinos\_)  {  QDomElement dino\_el = dinoToDomEl(doc, dino);  root.appendChild(dino\_el);  }  doc.appendChild(root);  string filename = this->dir\_name\_ + "/data.xml";  tofile(doc, filename);  QDomDocument doc2;  QDomElement root2 = doc2.createElement("periods");  for(Per & per : this->pers\_)  {  QDomElement per\_el = perToDomEl(doc2, per);  root2.appendChild(per\_el);  }  doc2.appendChild(root2);  string filename2 = this->dir\_name\_ + "/data2.xml";  tofile(doc2, filename2);  return true;  }  int XmlStorage::getNewDinoId()  {  int max\_id = 0;  for(Dino & d : this->dinos\_)  {  if(d.id > max\_id)  {  max\_id = d.id;  }  }  int new\_id = max\_id + 1;  return new\_id;  }  bool XmlStorage::removeDino(int Dino\_id)  {  int index = -1;  for(int i = 0; i < this->dinos\_.size(); i++)  {  if(this->dinos\_[i].id == Dino\_id)  {  index = i;  break;  }  }  if(index >= 0)  {  this->dinos\_.erase(this->dinos\_.begin() + index);  return true;  }  return false;  }  vector<Dino> XmlStorage::getAllDinos()  {  return this->dinos\_;  }  optional<Dino> XmlStorage::getDinoById(int Dino\_id)  {  for(Dino & c : this->dinos\_)  {  if(c.id == Dino\_id)  {  return c;  }  }  return nullopt;  }  int XmlStorage::insertDino(const Dino &dvno)  {  int new\_id = this->getNewDinoId();  Dino copy = dvno;  copy.id = new\_id;  this->dinos\_.push\_back(copy);  return new\_id;  }  bool XmlStorage::updateDino(const Dino &dvno)  {  int Dino\_id = dvno.id;  for(Dino & c : this->dinos\_)  {  if(c.id == Dino\_id)  {  c.name = dvno.name;  c.per = dvno.per;  c.fam = dvno.fam;  c.disc = dvno.disc;  c.mass = dvno.mass;  return true;  }  }  return false;  }  vector<Per> XmlStorage::getAllPers()  {  return this->pers\_;  }  optional<Per> XmlStorage::getPerById(int id)  {  for(Per & p : this->pers\_)  {  if(p.id == id)  {  return p;  }  }  return nullopt;  }  bool XmlStorage::removePer(int id)  {  int index = -1;  for(int i = 0; i < this->pers\_.size(); i++)  {  if(this->pers\_[i].id == id)  {  index = i;  break;  }  }  if(index >= 0)  {  this->pers\_.erase(this->pers\_.begin() + index);  return true;  }  return false;  }  int XmlStorage::getNewPerId()  {  int max\_id = 0;  for(Per & p : this->pers\_)  {  if(p.id > max\_id)  {  max\_id = p.id;  }  }  int new\_id = max\_id + 1;  return new\_id;  }  int XmlStorage::insertPer(const Per &pr)  {  int new\_id = this->getNewPerId();  Per copy = pr;  copy.id = new\_id;  this->pers\_.push\_back(copy);  return new\_id;  }  bool XmlStorage::updatePer(const Per &pr)  {  int Per\_id = pr.id;  for(Per & p : this->pers\_)  {  if(p.id == Per\_id)  {  p.name = pr.name;  p.time = pr.time;  return true;  }  }  return false;  }  vector<Dino> XmlStorage::getAllUserDino(int user\_id)  {  vector<Dino> dinos;  Dino d;  d.id = 0;  d.name = "";  d.fam = "";  d.mass = 0;  d.per = "";  d.user\_id = 0;  d.disc = 0;  dinos.push\_back(d);  return dinos;  }  optional<User> XmlStorage::getUserAuth(string & username, string & password)  {  return nullopt;  }  vector<Per> XmlStorage::getAllDinoPer(int dino\_id)  {  vector<Per> pers;  Per p;  p.id = 0;  p.name = "";  p.time = 0;  pers.push\_back(p);  return pers;  }  bool XmlStorage::insertDinoPer(int dino\_id, int per\_id)  {  return false;  }  bool XmlStorage::removeDinoPer(int dino\_id, int per\_id)  {  return false;  } |

|  |
| --- |
| **sqlite\_storage.h** |
| #ifndef SQLITE\_STORAGE\_H  #define SQLITE\_STORAGE\_H  #include "storage.h"  #include <QSqlDatabase>  class SqliteStorage : public Storage  {  const string dir\_name\_;  QSqlDatabase db\_;  public:  SqliteStorage(const string & dir\_name);  bool open();  bool close();  // dinosaurs  vector<Dino> getAllDinos(void);  optional<Dino> getDinoById(int Dino\_id);  bool updateDino(const Dino &dvno);  bool removeDino(int Dino\_id);  int insertDino(const Dino &dvno);  // periods  vector<Per> getAllPers(void);  optional<Per> getPerById(int Per\_id);  bool updatePer(const Per &pr);  bool removePer(int Per\_id);  int insertPer(const Per &pr);  };  #endif // SQLITE\_STORAGE\_H |

|  |
| --- |
| **sqlite\_storage.cpp** |
| #include "sqlite\_storage.h"  #include <QtSql>  #include <QCryptographicHash>  #include <QDebug>  SqliteStorage::SqliteStorage(const string & dir\_name) : dir\_name\_(dir\_name)  {  db\_ = QSqlDatabase::addDatabase("QSQLITE");  }  bool SqliteStorage::open()  {  QString path = QString::fromStdString(this->dir\_name\_) + "/data.sqlite";  db\_.setDatabaseName(path); // set sqlite database file path  bool connected = db\_.open(); // open db connection  if (!connected) {return false;}  return true;  }  bool SqliteStorage::close()  {  db\_.close();  return true;  }  Dino getDinoFromQuery(const QSqlQuery & query)  {  int id = query.value("id").toInt();  string name = query.value("name").toString().toStdString();  string period = query.value("period").toString().toStdString();  string family = query.value("family").toString().toStdString();  int discovered = query.value("discovered").toInt();  double mass = query.value("mass").toDouble();  Dino d;  d.id = id;  d.name = name;  d.per = period;  d.fam = family;  d.disc = discovered;  d.mass = mass;  return d;  }  void queryBindDino(const Dino &dvno, QSqlQuery & query)  {  query.bindValue(":name", QString::fromStdString(dvno.name));  query.bindValue(":period", QString::fromStdString(dvno.per));  query.bindValue(":family", QString::fromStdString(dvno.fam));  query.bindValue(":discovered", dvno.disc);  query.bindValue(":mass", dvno.mass);  query.bindValue(":user\_id", dvno.user\_id);  }  vector<Dino> SqliteStorage::getAllDinos(void)  {  vector<Dino> dinos;  QSqlQuery query("SELECT \* FROM dinosaurs");  while (query.next())  {  Dino d = getDinoFromQuery(query);  dinos.push\_back(d);  }  return dinos;  }  optional<Dino> SqliteStorage::getDinoById(int Dino\_id)  {  QSqlQuery query;  query.prepare("SELECT \* FROM dinosaurs WHERE id = :id");  query.bindValue(":id", Dino\_id);  if (!query.exec())  {  qDebug() << "get dinosaur error:" << query.lastError();  return nullopt;  }  if (query.next())  {  Dino d = getDinoFromQuery(query);  return d;  }  else {return nullopt;}  }  bool SqliteStorage::updateDino(const Dino &dvno)  {  QSqlQuery query;  query.prepare("UPDATE dinosaurs SET name = :name, period = :period, family = :family, discovered = :discovered, mass = :mass WHERE id = :id");  query.bindValue(":id", dvno.id);  queryBindDino(dvno, query);  if (!query.exec())  {  qDebug() << "updateDino error:" << query.lastError();  return false;  }  if(query.numRowsAffected() == 0)  {  return false;  }  return true;  }  bool SqliteStorage::removeDino(int Dino\_id)  {  QSqlQuery query;  query.prepare("DELETE FROM dinosaurs WHERE id = :id");  query.bindValue(":id", Dino\_id);  if (!query.exec())  {  qDebug() << "deleteDino error: " << query.lastError();  return false;  }  if(query.numRowsAffected() == 0) {return false;}  return true;  }  int SqliteStorage::insertDino(const Dino &dvno)  {  QSqlQuery query;  query.prepare("INSERT INTO dinosaurs (name, period, family, discovered, mass, user\_id) VALUES (:name, :period, :family, :discovered, :mass, :user\_id)");  queryBindDino(dvno, query);  if (!query.exec())  {  qDebug() << "adddino error:" << query.lastError();  return 0;  }  QVariant var = query.lastInsertId();  return var.toInt();  }  Per getPerFromQuery(const QSqlQuery & query)  {  int id = query.value("id").toInt();  string name = query.value("name").toString().toStdString();  double time = query.value("time").toDouble();  Per p;  p.id = id;  p.name = name;  p.time = time;  return p;  }  void queryBindDino(const Per &pr, QSqlQuery & query)  {  query.bindValue(":name", QString::fromStdString(pr.name));  query.bindValue(":time", pr.time);  }  vector<Per> SqliteStorage::getAllPers(void)  {  vector<Per> pers;  QSqlQuery query("SELECT \* FROM periods");  while (query.next())  {  Per p = getPerFromQuery(query);  pers.push\_back(p);  }  return pers;  }  optional<Per> SqliteStorage::getPerById(int Per\_id)  {  QSqlQuery query;  query.prepare("SELECT \* FROM periods WHERE id = :id");  query.bindValue(":id", Per\_id);  if (!query.exec())  {  qDebug() << "get period error:" << query.lastError();  return nullopt;  }  if (query.next())  {  Per p = getPerFromQuery(query);  return p;  }  else return nullopt;  }  bool SqliteStorage::updatePer(const Per &pr)  {  QSqlQuery query;  query.prepare("UPDATE periods SET name = :name, time = :time WHERE id = :id");  query.bindValue(":id", pr.id);  queryBindDino(pr, query);  if (!query.exec())  {  qDebug() << "updatePer error:" << query.lastError();  return false;  }  if(query.numRowsAffected() == 0)  {  return false;  }  return true;  }  bool SqliteStorage::removePer(int Per\_id)  {  QSqlQuery query;  query.prepare("DELETE FROM links WHERE id = :id");  query.bindValue(":id", Per\_id);  if (!query.exec())  {  qDebug() << "deletePer error: " << query.lastError();  return false;  }  if(query.numRowsAffected() == 0) {return false;}  return true;  }  int SqliteStorage::insertPer(const Per &pr)  {  QSqlQuery query;  query.prepare("INSERT INTO periods (name, time) VALUES (:name, :time)");  queryBindDino(pr, query);  if (!query.exec())  {  qDebug() << "addPeriod error:" << query.lastError();  return 0;  }  QVariant var = query.lastInsertId();  return var.toInt();  }  QString hashPassword(QString const & pass)  {  QByteArray pass\_ba = pass.toUtf8();  QByteArray hash\_ba = QCryptographicHash::hash(pass\_ba, QCryptographicHash::Md5);  QString pass\_hash = QString(hash\_ba.toHex());  return pass\_hash;  }  vector<Dino> SqliteStorage::getAllUserDino(int user\_id)  {  qDebug() << user\_id;  vector<Dino> dinos;  QSqlQuery query;  query.prepare("SELECT \* FROM dinosaurs WHERE user\_id = :user\_id");  query.bindValue(":user\_id", user\_id);  if (!query.exec())  {  qDebug() << query.lastError().text();  abort();  }  while (query.next())  {  Dino d = getDinoFromQuery(query);  qDebug() << d.name.c\_str();  dinos.push\_back(d);  }  return dinos;  }  optional<User> SqliteStorage::getUserAuth(string & username, string & password)  {  User user;  QString fhash = hashPassword(QString::fromUtf8(password.c\_str()));  string passhash = fhash.toUtf8().constData();  QSqlQuery query;  query.prepare("SELECT \* FROM users WHERE username = :un;");  query.bindValue(":un", username.c\_str());  if(!query.exec())  {  qDebug() << "Unknown error: " << query.lastError();  return nullopt;  }  if(query.next())  {  //if found user  string uhash = query.value("password\_hash").toString().toStdString();  if(passhash == uhash)  {  user.id = query.value("id").toInt();  qDebug() << user.id;  user.username = query.value("username").toString().toStdString();  qDebug() << user.username.c\_str();  user.password\_hash = query.value("password\_hash").toString().toStdString();  qDebug() << user.password\_hash.c\_str();  return user;  }  else  {  //uncorrect password  qDebug() << "Uncorrect password. Try again. Error: " << query.lastError();  User up;  up.id = -1;  return up;  }  }  qDebug() << "Unknown user";  User up;  up.id = -2;  return up;  }  vector<Per> SqliteStorage::getAllDinoPer(int dino\_id)  {  vector<Per> pers;  QSqlQuery query;  query.prepare("SELECT \* FROM links WHERE dino\_id = :dino\_id");  query.bindValue(":dino\_id", dino\_id);  if (!query.exec())  {  qDebug() << query.lastError().text();  abort();  }  while(query.next())  {  int pid = query.value("per\_id").toInt();  Per p = this->getPerById(pid).value();  pers.push\_back(p);  }  return pers;  }  bool SqliteStorage::insertDinoPer(int dino\_id, int per\_id)  {  QSqlQuery query3;  query3.prepare("INSERT INTO links VALUES (:dino\_id, :per\_id)");  query3.bindValue(":dino\_id", dino\_id);  query3.bindValue(":per\_id" , per\_id);  qDebug () << query3.lastQuery();  if (!query3.exec())  {  qDebug() << "addtoLinks error:" << query3.lastError();  return false;  }  return true;  }  bool SqliteStorage::removeDinoPer(int dino\_id, int per\_id)  {  QSqlQuery query3;  query3.prepare("DELETE FROM links WHERE dino\_id = :dino\_id AND per\_id = :per\_id");  query3.bindValue(":dino\_id", dino\_id);  query3.bindValue(":per\_id", per\_id);  if (!query3.exec())  {  qDebug() << "addtoLinks error:" << query3.lastError();  return false;  }  return true;  } |

|  |
| --- |
| **csv.cpp** |
| #include "csv.h"  CsvTable Csv::createTableFromString(const string & csv\_str)  {  int bracount = 0;  string buf;  const char \* p = csv\_str.c\_str();  CsvRow row;  CsvTable table;  while (1)  {  if (bracount == 0)  {  if (\*p == '\"')  {  bracount = 1;  }  else  {  if (\*p == '\n' || \*p == '\0')  {  row.push\_back(buf);  table.push\_back(row);  row.clear();  buf.clear();  }  else if(\*p == ',')  {  row.push\_back(buf);  buf.clear();  }  else  {  buf += \*p;  }  }  }  else  {  if (\*p == '\"')  {  if (\*(p + 1) == '\"')  {  buf += '\"';  p += 1;  }  else  {  row.push\_back(buf);  buf.clear();  if (\*(p + 1) == ',')  {  p += 1;  bracount = 0;  }  else if(\*(p + 1) == '\n' || \*(p + 1) == '\0')  {  table.push\_back(row);  row.clear();  p += 1;  bracount = 0;  }  else  {  std::cerr << "CSV format error: invalid char" << std::endl;  abort();  }  }  }  else  {  buf += \*p;  }  }  if (\*p == '\0')  {  break;  }  p += 1;  }  return table;  }  string Csv::createStringFromTable(const CsvTable &csv\_table)  {  bool escape = false;  string buf;  string::size\_type n = 0;  std::string bracount;  CsvRow row;  for (int i = 0; i < csv\_table.size(); i++)  {  row = csv\_table.at(i);  for (int j = 0; j < row.size(); j++)  {  n = 0;  bracount.clear();  std::string value = row.at(j);  if((value.find(',') != string::npos || value.find('\n') != string::npos) && value.find('\"') == string::npos)  {  buf += "\"";  buf += value;  buf += "\"";  }  else if(value.find('\"') != string::npos)  {  buf += "\"";  bracount = value;  while(n != string::npos)  {  if(n != 0)  {  bracount.insert(n, "\"");  n += 2;  }  n = value.find('\"', n);  }  buf += bracount;  buf += "\"";  }  else  {  buf += value;  }    if (j != csv\_table[i].size() - 1)  {  buf += ",";  }  }  if (i != csv\_table.size() - 1)  {  buf += "\n";  }  }  return buf;  } |

|  |
| --- |
| **dino.h** |
| #pragma once  #include <string>  #include <QMetaType>  using namespace std;  struct Dino  {  int id = 0;  string name = "";  string per = "";  string fam = "";  int disc = 0;  double mass = 0.0;  int user\_id = 0;  Dino();  Dino(int id, string name, string per, string fam, int disc, double mass, int user\_id);  };  Q\_DECLARE\_METATYPE(Dino) |

|  |
| --- |
| **period.h** |
| #pragma once  #include <string>  #include <QMetaType>  #include <QtCore>  using namespace std;  struct Per  {  int id = 0;  string name = "";  double time = 0.0;  Per();  Per(int id, string name, double time);  };  Q\_DECLARE\_METATYPE(Per) |

|  |
| --- |
| **user.h** |
| #pragma once  #include <string>  #include <QMetaType>  using namespace std;  struct User  {  int id = 0;  string username = "";  string password\_hash = "";  User();  User(int id, string username, string password\_hash);  };  Q\_DECLARE\_METATYPE(User) |

|  |
| --- |
| **dino.cpp** |
| #include "dino.h"  #include <QDebug>  Dino::Dino() { }  Dino::Dino(int id, string name, string per, string fam, int disc, double mass, int user\_id)  {  this->id = id;  this->name = name;  this->per = per;  this->fam = fam;  this->disc = disc;  this->mass = mass;  this->user\_id = user\_id;  } |

|  |
| --- |
| **period.cpp** |
| #include "period.h"  #include <QDebug>  Per::Per() { }  Per::Per(int id, string name, double time)  {  this->id = id;  this->name = name;  this->time = time;  } |

|  |
| --- |
| **user.cpp** |
| #include "user.h"  #include <QDebug>  User::User() { }  User::User(int id, string username, string password\_hash)  {  this->id = id;  this->username =username;  this->password\_hash = password\_hash;  } |

|  |
| --- |
| **add\_d.h** |
| #ifndef ADD\_D\_H  #define ADD\_D\_H  #include <QDialog>  #include <QIntValidator>  #include <QDoubleValidator>  #include <QRegExpValidator>  #include "dino.h"  namespace Ui {  class add\_d;  }  class add\_d : public QDialog  {  Q\_OBJECT  public:  explicit add\_d(QWidget \*parent = 0);  ~add\_d();  Dino getDino();  \  private slots:  void on\_buttonBox\_accepted();  void on\_buttonBox\_rejected();  private:  Ui::add\_d \*ui;  Dino dino;  QIntValidator int\_val;  QDoubleValidator double\_val;  QRegExpValidator regx\_val;  };  #endif // ADD\_D\_H |

|  |
| --- |
| **add\_d.cpp** |
| #include "add\_d.h"  #include "ui\_add\_d.h"  #include <QDebug>  #include <QMessageBox>  add\_d::add\_d(QWidget \*parent) :  QDialog(parent),  ui(new Ui::add\_d)  {  ui->setupUi(this);  double\_val.setNotation(QDoubleValidator::StandardNotation);  QLocale locale(QLocale::C);  double\_val.setLocale(locale);  ui->name->setText(QString::fromUtf8("Dinosaur"));  ui->fam->setText(QString::fromUtf8("Family"));  ui->disc->setValue(1000);  ui->mass->setValue(100.0);  ui->name->setValidator(&regx\_val);  ui->fam->setValidator(&regx\_val);  }  add\_d::~add\_d()  {  delete ui;  }  Dino add\_d::getDino()  {  return this->dino;  }  void add\_d::on\_buttonBox\_accepted()  {  if(ui->name->text().isEmpty() || ui->fam->text().isEmpty() || ui->disc->text().isEmpty() || ui->mass->text().isEmpty())  {  QMessageBox::warning(  this,  "WARNING",  "Some fields are empty",  QMessageBox::Ok);  return;  }  Dino d;  d.name = ui->name->text().toStdString();  d.fam = ui->fam->text().toStdString();  d.disc = ui->disc->value();  d.mass = ui->mass->value();  this->dino = d;  this->accept();  }  void add\_d::on\_buttonBox\_rejected()  {  this->reject();  } |

|  |
| --- |
| **add\_p.h** |
| #ifndef ADD\_P\_H  #define ADD\_P\_H  #include <QDialog>  #include <QIntValidator>  #include <QDoubleValidator>  #include <QRegExpValidator>  #include "period.h"  #include "dino.h"  #include "storage.h"  namespace Ui {  class add\_p;  }  class add\_p : public QDialog  {  Q\_OBJECT  public:  explicit add\_p(QWidget \*parent = 0, int did = 0, Storage \* storage = NULL);  ~add\_p();  Per getPer();  private slots:  void on\_cancel\_clicked();  void on\_add\_clicked();  void on\_listWidget2\_4\_itemSelectionChanged();  void fillWListPer();  private:  Ui::add\_p \*ui;  Per per;  QIntValidator int\_val;  QDoubleValidator double\_val;  QRegExpValidator regx\_val;  int did\_;  Storage \* storage\_;  };  #endif // ADD\_P\_H |

|  |
| --- |
| **add\_p.cpp** |
| #include "add\_p.h"  #include "ui\_add\_p.h"  #include "dino.h"  #include "mainwindow.h"  #include <QDebug>  #include <QMessageBox>  #include <QFileDialog>  #include <QMessageBox>  #include <QSqlDatabase>  #include <QSqlQuery>  add\_p::add\_p(QWidget \*parent, int did, Storage \* storage) :  QDialog(parent),  ui(new Ui::add\_p)  {  ui->setupUi(this);  did\_ = did;  storage\_ = storage;  this->fillWListPer();  // double\_val.setNotation(QDoubleValidator::StandardNotation);  // QLocale locale(QLocale::C);  // double\_val.setLocale(locale);  // ui->name->setText(QString::fromUtf8("Period"));  // ui->time->setValue(100000);  // ui->name->setValidator(&regx\_val);  }  add\_p::~add\_p()  {  delete ui;  }  Per add\_p::getPer()  {  return this->per;  }  void add\_p::on\_cancel\_clicked()  {  this->reject();  }  void add\_p::fillWListPer()  {  QVector<Per> pers = QVector<Per>::fromStdVector(storage\_->getAllPers());  QVariant var;  for(Per & per : pers)  {  string stext = per.name;  QString text = QString::fromStdString(stext);  QListWidgetItem \* new\_item = new QListWidgetItem(text);  var = QVariant::fromValue(per);  new\_item->setData(Qt::UserRole, var);  ui->listWidget2\_4->addItem(new\_item);  }  }  void add\_p::on\_add\_clicked()  {  QListWidgetItem \* item = ui->listWidget2\_4->currentItem();  Per per = item->data(Qt::UserRole).value<Per>();  int pid = per.id;  int did = did\_;  QSqlQuery query;  query.prepare("INSERT INTO links (dino\_id, per\_id) VALUES (:dino\_id, :per\_id)");  query.bindValue(":dino\_id", did);  query.bindValue(":per\_id", pid);  if (!query.exec())  {  qDebug() << "addlink error" ;  abort();  }  this->accept();  }  void add\_p::on\_listWidget2\_4\_itemSelectionChanged()  {  QListWidgetItem \* item = ui->listWidget2\_4->currentItem();  Per per = item->data(Qt::UserRole).value<Per>();  ui->lselected\_5->setText("Selected period:");  ui->lplname\_4->setText("Name:");  ui->lpltime\_4->setText("Length:");  ui->lpname\_4->setText(QString::fromUtf8(per.name.c\_str()));  ui->lptime\_4->setText(QString::number(per.time));  } |

|  |
| --- |
| **auth.h** |
| #ifndef AUTH\_H  #define AUTH\_H  #include <QDialog>  #include "storage.h"  namespace Ui {  class auth;  }  class auth : public QDialog  {  Q\_OBJECT  public:  explicit auth(QWidget \*parent = 0, Storage \* stor = NULL);  int getUID();  ~auth();  private slots:  QString hashPassword(QString const & pass);  void on\_loginb\_clicked();  void on\_back\_clicked();  void on\_create\_clicked();  private:  Ui::auth \*ui;  Storage \* astorage\_; // <-- STORAGE  int uid = -3;  };  #endif // AUTH\_H |

|  |
| --- |
| **auth.cpp** |
| #include "auth.h"  #include "ui\_auth.h"  #include <QDebug>  #include <QMessageBox>  #include "sqlite\_storage.h"  #include <QSqlQuery>  #include <QtSql>  #include <QSqlDatabase>  #include <QCryptographicHash>  #include <QDebug>  auth::auth(QWidget \*parent, Storage \* stor) :  QDialog(parent),  ui(new Ui::auth)  {  ui->setupUi(this);  astorage\_ = stor;  }  auth::~auth()  {  delete ui;  }  int auth::getUID()  {  int a = uid;  return a;  }  void auth::on\_loginb\_clicked()  {  if(ui->password->text().isEmpty() || ui->login->text().isEmpty())  {  QMessageBox::warning(  this,  "WARNING",  "Some fields are empty",  QMessageBox::Ok);  return;  }  string username = ui->login->text().toStdString().c\_str();  string password = ui->password->text().toStdString().c\_str();  if(astorage\_->getUserAuth(username, password) != nullopt)  {  if(astorage\_->getUserAuth(username, password).value().id == -1)  {  QMessageBox::warning(  this,  "WARNING",  "Uncorrect password. Try again",  QMessageBox::Ok);  return;  }  else if(astorage\_->getUserAuth(username, password).value().id == -2)  {  QMessageBox::warning(  this,  "WARNING",  "Unknown user. Try again",  QMessageBox::Ok);  return;  }  else  {  uid = astorage\_->getUserAuth(username, password).value().id;  qDebug() << uid;  qDebug() << "ok";  this->accept();  }  }  else if(astorage\_->getUserAuth(username, password) == nullopt)  {  QMessageBox::warning(  this,  "WARNING",  "Something went terribly wrong",  QMessageBox::Ok);  return;  }  }  void auth::on\_back\_clicked()  {  this->reject();  }  QString auth::hashPassword(QString const & pass)  {  QByteArray pass\_ba = pass.toUtf8();  QByteArray hash\_ba = QCryptographicHash::hash(pass\_ba, QCryptographicHash::Md5);  QString pass\_hash = QString(hash\_ba.toHex());  return pass\_hash;  }  void auth::on\_create\_clicked()  {  if(ui->password->text().isEmpty() || ui->login->text().isEmpty())  {  QMessageBox::warning(  this,  "WARNING",  "Some fields are empty",  QMessageBox::Ok);  return;  }  string username = ui->login->text().toStdString().c\_str();  string password = ui->password->text().toStdString().c\_str();  QString fhash = hashPassword(QString::fromUtf8(password.c\_str()));  string passhash = fhash.toUtf8().constData();  if(astorage\_->getUserAuth(username, password).value().id == -1)  {  QMessageBox::warning(  this,  "WARNING",  "This login is taken",  QMessageBox::Ok);  return;  }  QSqlQuery query;  query.prepare("INSERT INTO users (username, password\_hash) VALUES (:username, :password\_hash)");  query.bindValue(":username", username.c\_str());  query.bindValue(":password\_hash" , passhash.c\_str());  if (!query.exec())  {  qDebug() << "addtoUsers error:" << query.lastError();  abort();  }  QVariant var = query.lastInsertId();  uid = var.toInt();  this->accept();  } |

|  |
| --- |
| **edit\_d.h** |
| #ifndef EDIT\_D\_H  #define EDIT\_D\_H  #include <QDialog>  #include <QIntValidator>  #include <QDoubleValidator>  #include <QRegExpValidator>  #include "dino.h"  namespace Ui {  class edit\_d;  }  class edit\_d : public QDialog  {  Q\_OBJECT  public:  explicit edit\_d(Dino & dino, QWidget \*parent = 0);  ~edit\_d();  Dino getDino();  private slots:  void on\_buttonBox\_2\_accepted();  void on\_buttonBox\_2\_rejected();    private:  Ui::edit\_d \*ui;  QIntValidator int\_val;  QDoubleValidator double\_val;  QRegExpValidator regx\_val;  Dino dino;  };  #endif // EDIT\_D\_H |

|  |
| --- |
| **edit\_d.cpp** |
| #include "edit\_d.h"  #include "ui\_edit\_d.h"  #include <QDebug>  #include <QMessageBox>  edit\_d::edit\_d(Dino & dino, QWidget \*parent) :  QDialog(parent),  ui(new Ui::edit\_d)  {  ui->setupUi(this);  double\_val.setNotation(QDoubleValidator::StandardNotation);  QLocale locale(QLocale::C);  double\_val.setLocale(locale);  this->dino.id = dino.id;  ui->name->setText(QString::fromUtf8(dino.name.c\_str()));  ui->fam->setText(QString::fromUtf8(dino.fam.c\_str()));  ui->disc->setValue(dino.disc);  ui->mass->setValue(dino.mass);  ui->name->setValidator(&regx\_val);  ui->fam->setValidator(&regx\_val);  }  edit\_d::~edit\_d()  {  delete ui;  }  Dino edit\_d::getDino()  {  return this->dino;  }  void edit\_d::on\_buttonBox\_2\_accepted()  {  if(ui->name->text().isEmpty() || ui->fam->text().isEmpty() || ui->disc->text().isEmpty() || ui->mass->text().isEmpty())  {  QMessageBox::warning(  this,  "WARNING",  "Some fields are empty",  QMessageBox::Ok);  return;  }  this->dino.name = ui->name->text().toStdString();  this->dino.fam = ui->fam->text().toStdString();  this->dino.disc = ui->disc->value();  this->dino.mass = ui->mass->value();  this->accept();  }  void edit\_d::on\_buttonBox\_2\_rejected()  {  this->reject();  } |

|  |
| --- |
| **edit\_p.h** |
| #ifndef EDIT\_P\_H  #define EDIT\_P\_H  #include <QDialog>  #include <QIntValidator>  #include <QDoubleValidator>  #include <QRegExpValidator>  #include "period.h"  namespace Ui {  class edit\_p;  }  class edit\_p : public QDialog  {  Q\_OBJECT  public:  explicit edit\_p(Per & per, QWidget \*parent = 0);  ~edit\_p();  Per getPer();  private slots:  void on\_buttonBox\_accepted();  void on\_buttonBox\_rejected();  private:  Ui::edit\_p \*ui;  QIntValidator int\_val;  QDoubleValidator double\_val;  QRegExpValidator regx\_val;  Per per;  };  #endif // EDIT\_P\_H |

|  |
| --- |
| **edit\_p.cpp** |
| #include "edit\_p.h"  #include "ui\_edit\_p.h"  #include <QDebug>  #include <QMessageBox>  edit\_p::edit\_p(Per & per, QWidget \*parent) :  QDialog(parent),  ui(new Ui::edit\_p)  {  ui->setupUi(this);  double\_val.setNotation(QDoubleValidator::StandardNotation);  QLocale locale(QLocale::C);  double\_val.setLocale(locale);  this->per.id = per.id;  ui->name->setText(QString::fromUtf8(per.name.c\_str()));  ui->time->setValue(per.time);  ui->name->setValidator(&regx\_val);  }  edit\_p::~edit\_p()  {  delete ui;  }  Per edit\_p::getPer()  {  return this->per;  }  void edit\_p::on\_buttonBox\_accepted()  {  if(ui->name->text().isEmpty() || ui->time->text().isEmpty())  {  QMessageBox::warning(  this,  "WARNING",  "Some fields are empty",  QMessageBox::Ok);  return;  }  this->per.name = ui->name->text().toStdString();  this->per.time = ui->time->value();  this->accept();  }  void edit\_p::on\_buttonBox\_rejected()  {  this->reject();  } |

|  |
| --- |
| **mainwindow.h** |
| #ifndef MAINWINDOW\_H  #define MAINWINDOW\_H  #include <QMainWindow>  #include "storage.h"  #include "csv\_storage.h"  #include "sqlite\_storage.h"  #include <QMainWindow>  #include <QtCore>  #include <QDebug>  #include <QFileDialog>  #include <QListWidgetItem>  #include <QMessageBox>  #include <QFileInfo>  #include "add\_d.h"  #include "add\_p.h"  #include "edit\_d.h"  #include "edit\_p.h"  #include "ui\_mainwindow.h"  namespace Ui {class MainWindow;  }  class MainWindow : public QMainWindow  {  Q\_OBJECT  public:  explicit MainWindow(QWidget \*parent = 0);  ~MainWindow();  private slots:  // void onNew();  void onOpen();  void on\_add\_clicked();  void on\_edit\_clicked();  void on\_remove\_clicked();  void on\_add2\_clicked();  void on\_edit2\_clicked();  void on\_remove2\_clicked();  void on\_listWidget2\_itemSelectionChanged();  void on\_listWidget\_itemSelectionChanged();  private:  Ui::MainWindow \*ui;  void loadfrom(QString & path);  void fillWListDino(int uid);  void fillWListPer(int did);  bool fileWrite(QString & path, QString & data);  bool defaultFile(QString & path);  Storage \* storage\_; // <-- STORAGE  int uid;  };  #endif // MAINWINDOW\_H |

|  |
| --- |
| **mainwindow.cpp** |
| #include "mainwindow.h"  #include "ui\_mainwindow.h"  #include "dino.h"  #include "period.h"  #include "auth.h"  #include <QDebug>  #include <QFileDialog>  #include <QMessageBox>  #include <QSqlDatabase>  #include <QSqlQuery>  MainWindow::MainWindow(QWidget \*parent) :  QMainWindow(parent),  ui(new Ui::MainWindow)  {  ui->setupUi(this);  // connect(ui->actionOption1, &QAction::triggered, this, &MainWindow::onNew);  connect(ui->actionOption2, &QAction::triggered, this, &MainWindow::onOpen);  connect(ui->actionExit, &QAction::triggered, this, &MainWindow::close);  }  //void MainWindow::onNew()  //{  // QFileDialog dialog(this);  // dialog.setFileMode(QFileDialog::Directory);  // QString current\_dir = QDir::currentPath();  // QString default\_name = "new\_SqlStorage";  // QString folder\_path = dialog.getSaveFileName(  // this,  // "Select New Storage Folder",  // current\_dir + "/" + default\_name,  // "Folders");  // if(!folder\_path.isEmpty())  // {  // QDir().mkdir(folder\_path);  // bool success = this->defaultFile(folder\_path);  // if(!success)  // {  // ui->listWidget->clear();  // ui->listWidget2->clear();  // ui->add->setEnabled(false);  // ui->add2->setEnabled(false);  // }  // }  //}  MainWindow::~MainWindow()  {  delete this->storage\_;  delete ui;  }  void MainWindow::onOpen()  {  qDebug() << "OPEN";  QString file\_path = QFileDialog::getExistingDirectory(  this, // parent  "Dialog Caption", // caption  "", // directory to start with  QFileDialog::ShowDirsOnly | QFileDialog::DontResolveSymlinks); // file name filter  qDebug() << file\_path;  if(!file\_path.isEmpty())  {  this->loadfrom(file\_path);  }  auth \* au = new auth(this, storage\_);  au->exec();  ui->add->setEnabled(false);  uid = au->getUID();  qDebug() << uid;  ui->listWidget->clear();  ui->listWidget2->clear();  this->fillWListDino(uid);  if(uid > 0)  {  ui->add->setEnabled(true);  }  }  void MainWindow::loadfrom(QString & path)  {  std::string filenamestd;  filenamestd = path.toUtf8().constData();  SqliteStorage \* sql\_storage = new SqliteStorage(filenamestd);  \  delete this->storage\_;  storage\_ = sql\_storage;  if(this->storage\_->open() == false)  {  QMessageBox::warning(  this,  "WARNING",  "Cannot open file!!!",  QMessageBox::Ok);  return;  }  }  void MainWindow::fillWListDino(int uid)  {  QVector<Dino> dinos = QVector<Dino>::fromStdVector(storage\_->getAllUserDino(uid));  QVariant var;  for(Dino & dino : dinos)  {  string stext = dino.name;  QString text = QString::fromStdString(stext);  QListWidgetItem \* new\_item = new QListWidgetItem(text);  var = QVariant::fromValue(dino);  new\_item->setData(Qt::UserRole, var);  ui->listWidget->addItem(new\_item);  }  }  void MainWindow::fillWListPer(int did)  {  QVector<Per> pers = QVector<Per>::fromStdVector(storage\_->getAllDinoPer(did));  QVariant var;  for(Per & per : pers)  {  string stext = per.name;  QString text = QString::fromStdString(stext);  QListWidgetItem \* new\_item = new QListWidgetItem(text);  var = QVariant::fromValue(per);  new\_item->setData(Qt::UserRole, var);  ui->listWidget2->addItem(new\_item);  }  }  bool MainWindow::fileWrite(QString &path, QString &data)  {  QFile file(path);  if(!file.open(QFile::WriteOnly | QFile::Text))  {  QMessageBox::warning(  this,  "WARNING",  "Cannot create storage!!!",  QMessageBox::Ok);  return false;  }  QTextStream out(&file);  out << data;  file.close();  return true;  }  bool MainWindow::defaultFile(QString &path)  {  QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");  QString filename = path + "/data.sqlite";  db.setDatabaseName(filename);  bool connected = db.open();  if (connected) {  QSqlQuery query;  query.prepare("CREATE TABLE dinosaurs (id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,"  " name TEXT NOT NULL,"  " period TEXT NOT NULL"  " family TEXT NOT NULL,"  " discovered INTEGER NOT NULL,"  " mass REAL NOT NULL,"  " user\_id INTEGER,"  " FOREIGN KEY(user\_id) REFERENCES users(id))");  query.exec();  query.prepare("CREATE TABLE periods ( id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,"  " name TEXT NOT NULL,"  " time REAL NOT NULL )");  query.exec();  query.prepare("CREATE TABLE links ( id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,"  " dino\_id INTEGER NOT NULL,"  " per\_id INTEGER NOT NULL,"  " FOREIGN KEY(per\_id) REFERENCES periods(id),"  " FOREIGN KEY(dino\_id) REFERENCES dinosaurs(id))");  query.exec();  query.prepare("CREATE TABLE users ( id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,"  " username TEXT NOT NULL,"  " password\_hash TEXT NOT NULL)");  query.exec();  query.prepare("INSERT INTO dinosaurs (name, period, family, discovered, mass, user\_id) "  "VALUES ('Dino1', 'Cretaceous', 'Dino1f', 123.4, 1), "  "('Dino2', 'Triassic', 'Dino2f', 234.0, 2)");  query.exec();  query.prepare("INSERT INTO periods (name, time) "  "VALUES ('Cretaceous', 145000000),"  "('Triassic, 251900000)");  query.exec();  query.prepare("INSERT INTO users (username, password\_hash) "  "VALUES ('qwerty', 'cf21666101224b0efdbc7226ee5ee36e'),"  "('John\_Yeehaw', 'd8bb63b6f25ce2e9b6f87cf08c4b8343')");  query.exec();  query.prepare("INSERT INTO links (dino\_id, per\_id) "  "VALUES (1, 1),"  "(2, 2)");  query.exec();  db.close();  // QString filename = path + "/data.sqlite";  // QString data = "1,Dino,Period,Family,999,999.0";  // bool dinow = this->fileWrite(filename, data);  // QString filename2 = path + "/data2.csv";  // QString data2 = "1,Period,999.0";  // bool perw = this->fileWrite(filename2, data2);  this->loadfrom(path);  return true;  // else return false;  }  else return false;  }  void MainWindow::on\_add\_clicked()  {  add\_d window(this);  if(window.exec())  {  QVariant var;  Dino dino = window.getDino();  dino.user\_id = uid;  dino.id = this->storage\_->insertDino(dino);  QListWidgetItem \* item = new QListWidgetItem;  var.setValue(dino);  const QString name = QString::fromUtf8(dino.name.c\_str());  item->setText(name);  item->setData(Qt::UserRole, var);  ui->listWidget->addItem(item);  }  }  void MainWindow::on\_edit\_clicked()  {  QList<QListWidgetItem \*> items = ui->listWidget->selectedItems();  Dino dino;  QVariant var;  foreach (QListWidgetItem \* item, items)  {  dino = item->data(Qt::UserRole).value<Dino>();  }  edit\_d window(dino, this);    if(window.exec())  {  dino = window.getDino();  this->storage\_->updateDino(dino);  foreach (QListWidgetItem \* item, items)  {  var.setValue(dino);  const QString name = QString::fromUtf8(dino.name.c\_str());  item->setText(name);  item->setData(Qt::UserRole, var);  }  }  // this->storage\_->close();  }  void MainWindow::on\_remove\_clicked()  {  QMessageBox::StandardButton remove;  remove = QMessageBox::question(  this,  "Remove",  "Are you sure?",  QMessageBox::Yes|QMessageBox::No);  if (remove == QMessageBox::Yes)  {  QList<QListWidgetItem \*> items = ui->listWidget->selectedItems();  Dino dino;  foreach(QListWidgetItem \* item, items)  {  dino = item->data(Qt::UserRole).value<Dino>();  this->storage\_->removeDino(dino.id);  delete ui->listWidget->takeItem(ui->listWidget->row(item));  }  }  // this->storage\_->close();  }  void MainWindow::on\_add2\_clicked()  {  QListWidgetItem \* qdino = ui->listWidget->currentItem();  Dino d = qdino->data(Qt::UserRole).value<Dino>();  int id = d.id;  add\_p window(this, id, storage\_);  if(window.exec())  {  // QVariant var;  // Per per = window.getPer();  // per.id = this->storage\_->insertPer(per);  // QListWidgetItem \* item = new QListWidgetItem;  // var.setValue(per);  // const QString name = QString::fromUtf8(per.name.c\_str());  // item->setText(name);  // item->setData(Qt::UserRole, var);  // ui->listWidget2->addItem(item);  }  // this->storage\_->close();  }  void MainWindow::on\_edit2\_clicked()  {  QList<QListWidgetItem \*> items = ui->listWidget2->selectedItems();  Per per;  QVariant variant;  foreach (QListWidgetItem \* item, items)  {  per = item->data(Qt::UserRole).value<Per>();  }  edit\_p window(per, this);    if(window.exec())  {  per = window.getPer();  this->storage\_->updatePer(per);    foreach (QListWidgetItem \* item, items)  {  variant.setValue(per);  const QString name = QString::fromUtf8(per.name.c\_str());  item->setText(name);  item->setData(Qt::UserRole, variant);  }  }  // this->storage\_->close();  }  void MainWindow::on\_remove2\_clicked()  {  QMessageBox::StandardButton remove;  remove = QMessageBox::question(  this,  "Remove",  "Are you sure?",  QMessageBox::Yes|QMessageBox::No);  if (remove == QMessageBox::Yes)  {  QListWidgetItem \* qdino = ui->listWidget->currentItem();  Dino d = qdino->data(Qt::UserRole).value<Dino>();  int did = d.id;  QListWidgetItem \* iper = ui->listWidget2->currentItem();  Per per = iper->data(Qt::UserRole).value<Per>();  int pid = per.id;  if(!this->storage\_->removeDinoPer(did, pid))  {  qDebug() << "Something went wrong";  }  delete ui->listWidget2->takeItem(ui->listWidget2->row(iper));  }  // this->storage\_->close();  }  void MainWindow::on\_listWidget2\_itemSelectionChanged()  {  QListWidgetItem \* item = ui->listWidget2->currentItem();  ui->edit2->setEnabled(true);  ui->remove2->setEnabled(true);  Per per = item->data(Qt::UserRole).value<Per>();  ui->lselected\_2->setText("Selected period:");  ui->lplname->setText("Name:");  ui->lpltime->setText("Length:");  ui->lpname->setText(QString::fromUtf8(per.name.c\_str()));  ui->lptime->setText(QString::number(per.time));  }  void MainWindow::on\_listWidget\_itemSelectionChanged()  {  QListWidgetItem \* item = ui->listWidget->currentItem();  ui->edit->setEnabled(true);  ui->remove->setEnabled(true);  Dino dino = item->data(Qt::UserRole).value<Dino>();  ui->lselected->setText("Selected dino:");  ui->ldlname->setText("Name:");  ui->ldlfam->setText("Family:");  ui->ldldisc->setText("Discovered:");  ui->ldlmass->setText("Mass");  ui->ldname->setText(QString::fromUtf8(dino.name.c\_str()));  ui->ldfam-> setText(QString::fromUtf8(dino.fam.c\_str()));  ui->lddisc->setText(QString::number(dino.disc));  ui->ldmass->setText(QString::number(dino.mass));  ui->listWidget2->clear();  this->fillWListPer(dino.id);  ui->add2->setEnabled(true);  // }  //now this to listselect1  } |

|  |
| --- |
| **add\_d.ui** |
|  |

|  |
| --- |
| **add\_p.ui** |
|  |

|  |
| --- |
| **auth.ui** |
|  |

|  |
| --- |
| **edit\_d.ui** |
|  |

|  |
| --- |
| **edit\_p.ui** |
|  |

|  |
| --- |
| **mainwindow.ui** |
|  |

**Робота коду**

Програма будується у QTCreator.

Користувачу дається можливість завантажити сховище та авторизуватися.

При відсутності логіну користувача у бд, неправильному паролі виводиться помилка і прохання ввести коректні дані.

Також можна зареєструвати нового користувача.

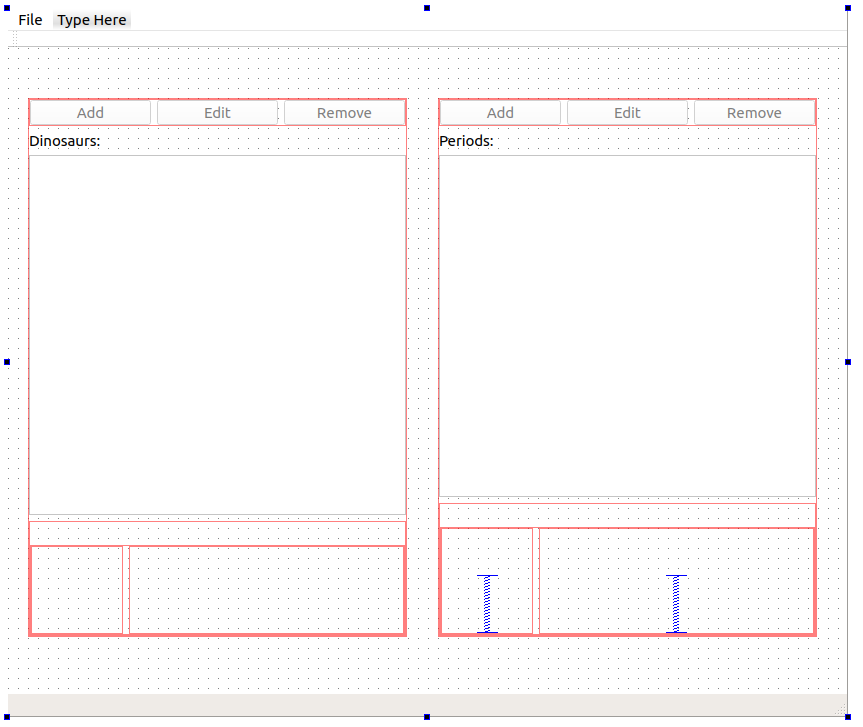
Списки основного вікна заповнюються елементами, що відповідають окремим сутностям, які належать користувачу.

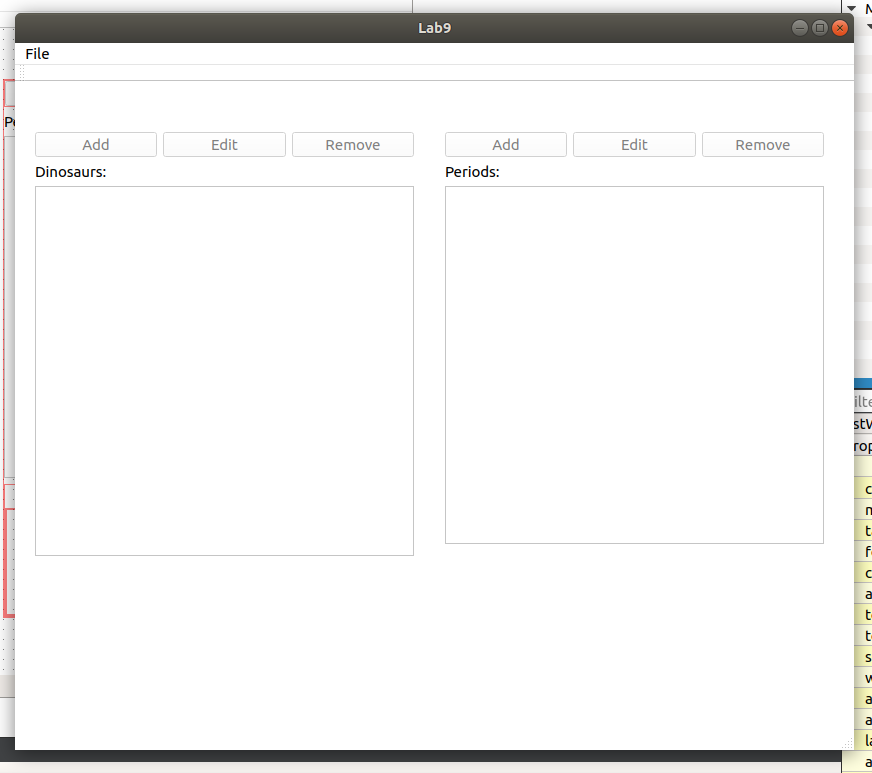
Користувач може дивитися інформацію про сутності, редагувати основні та додаткові сутності, прив’язувати додаткові сутності до основних, видаляти сутності.

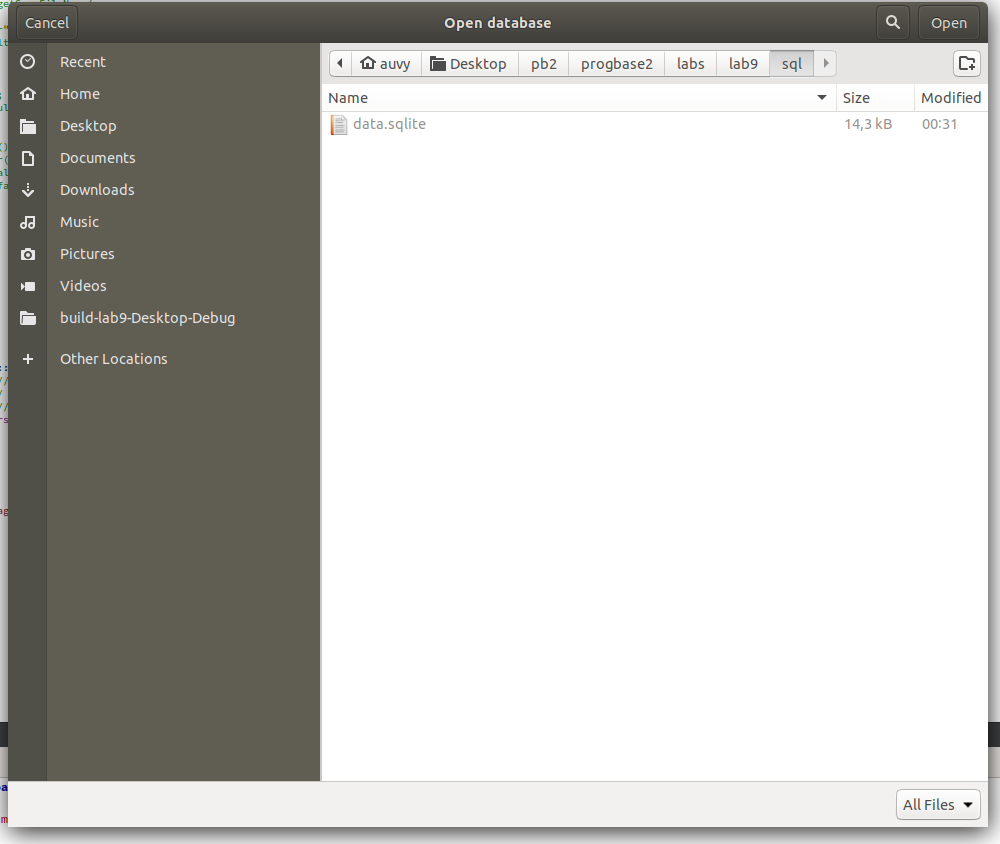
При створенні нової сутності поля заповняються стандартними записами.

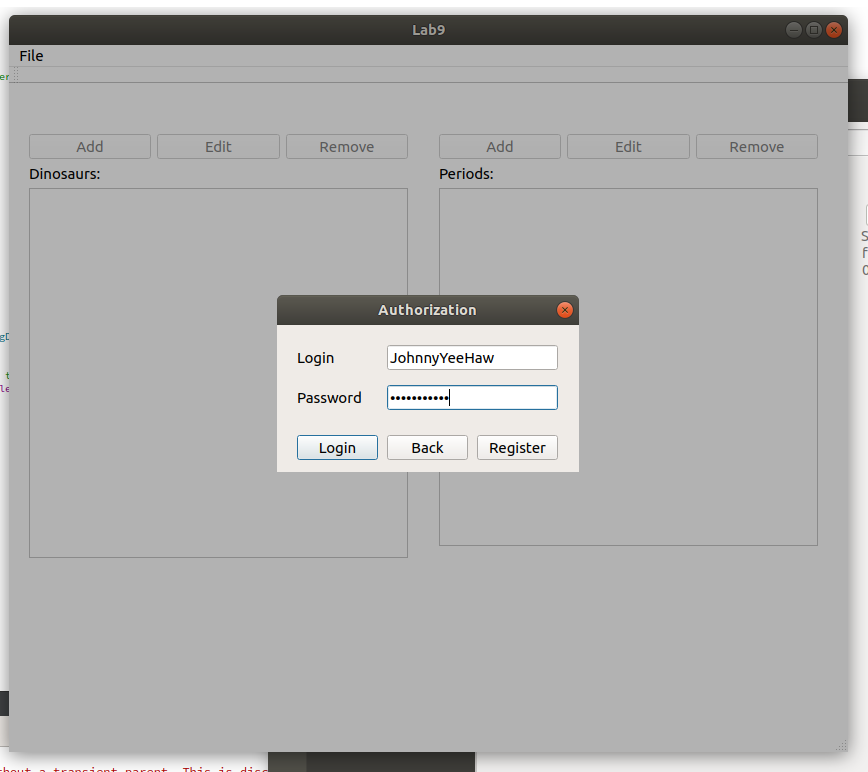
При пустих полях виводиться помилка.

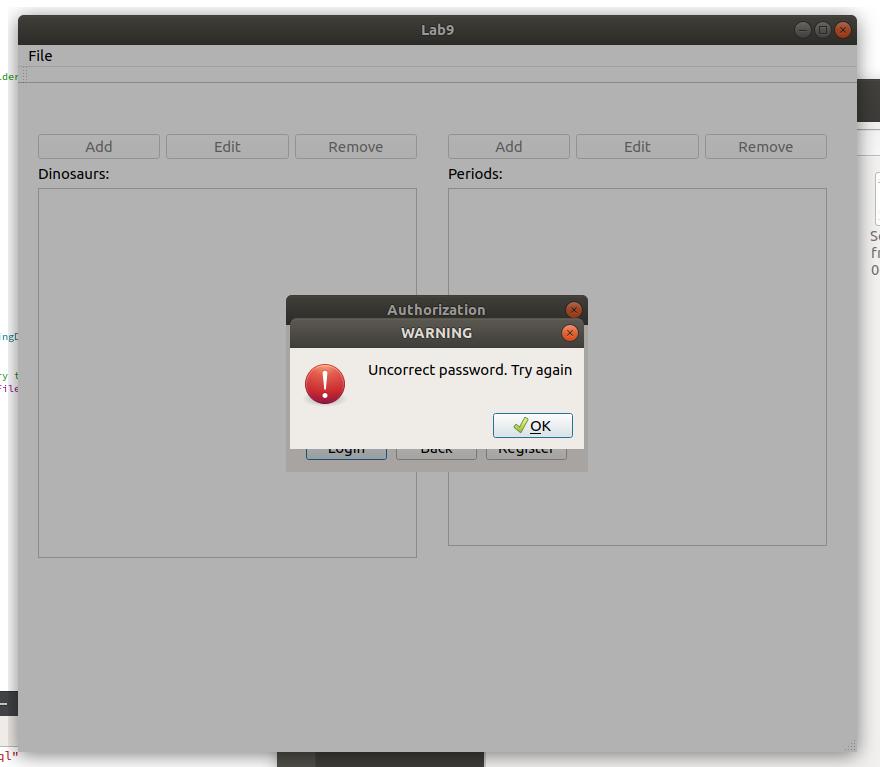
**Приклади результатів**

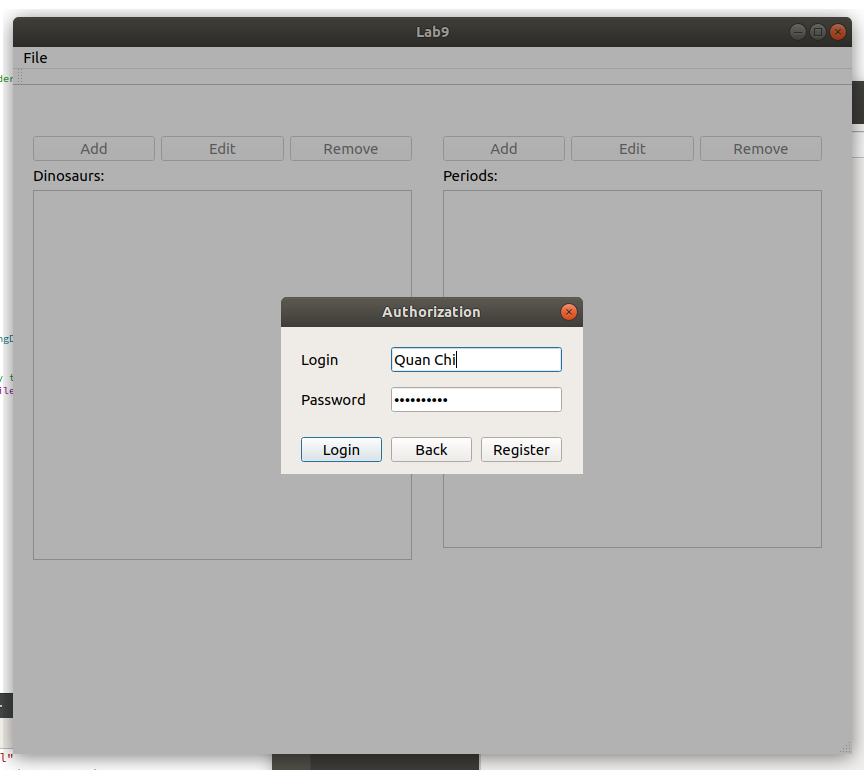
****

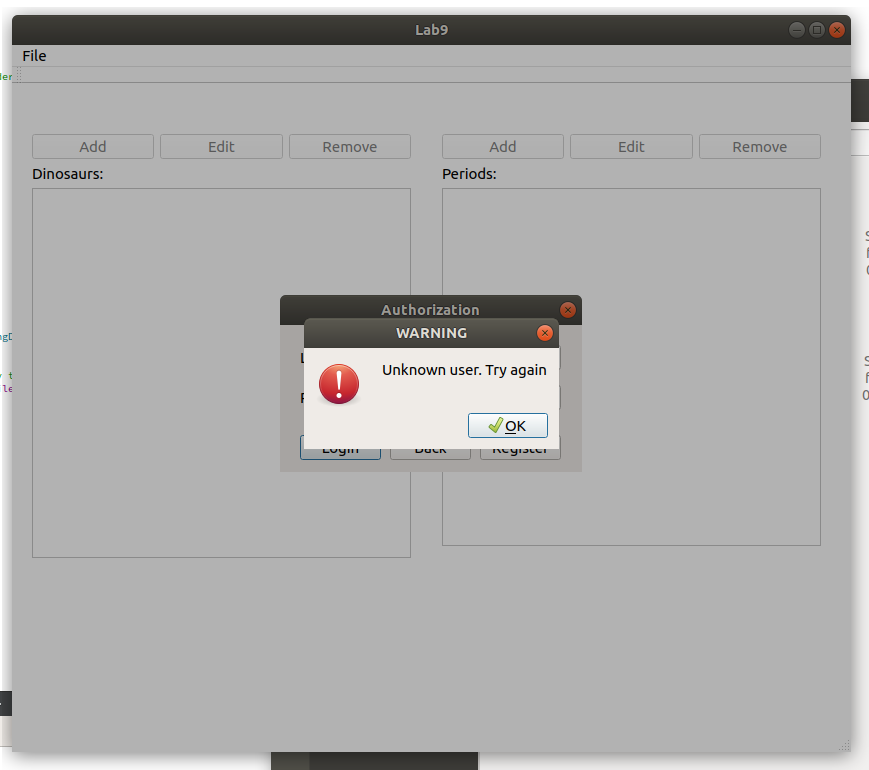
****

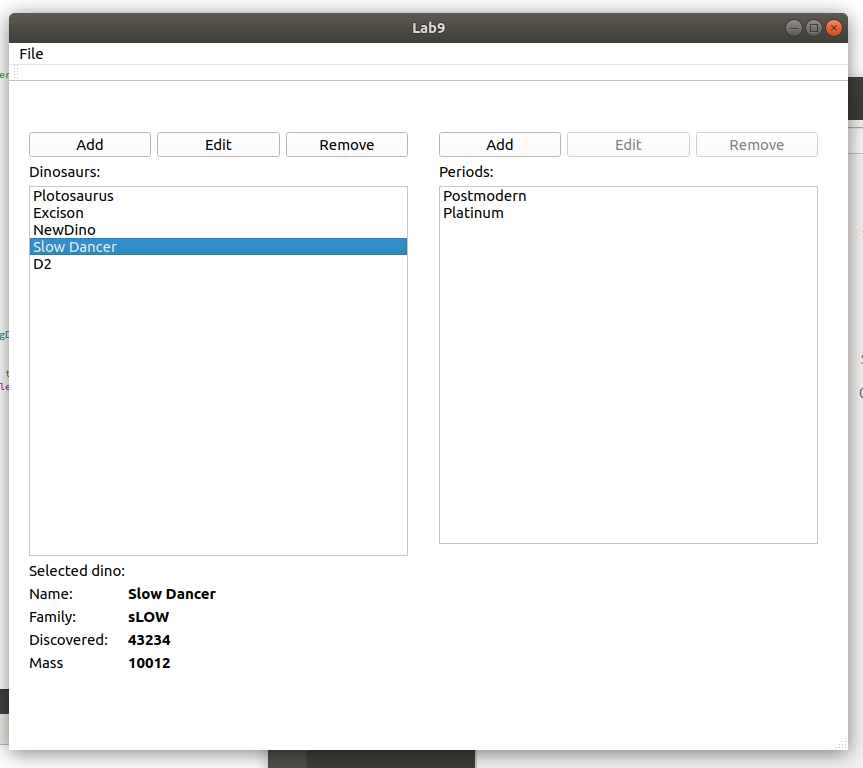
****

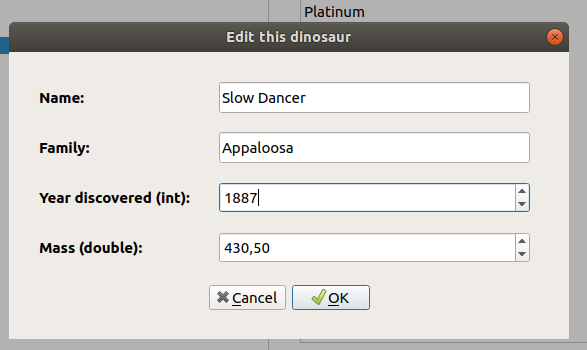
****

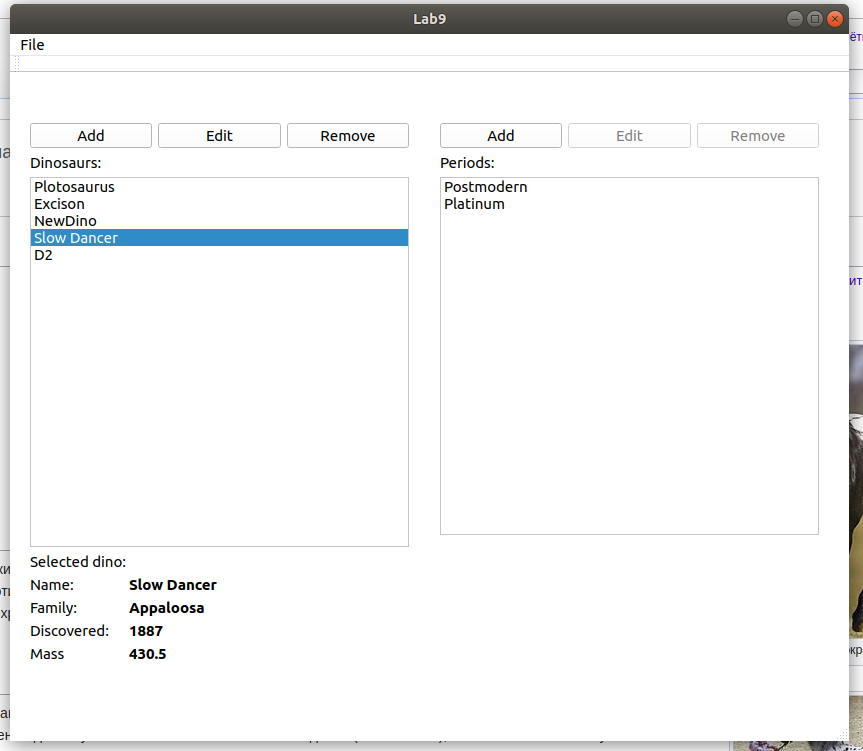
****

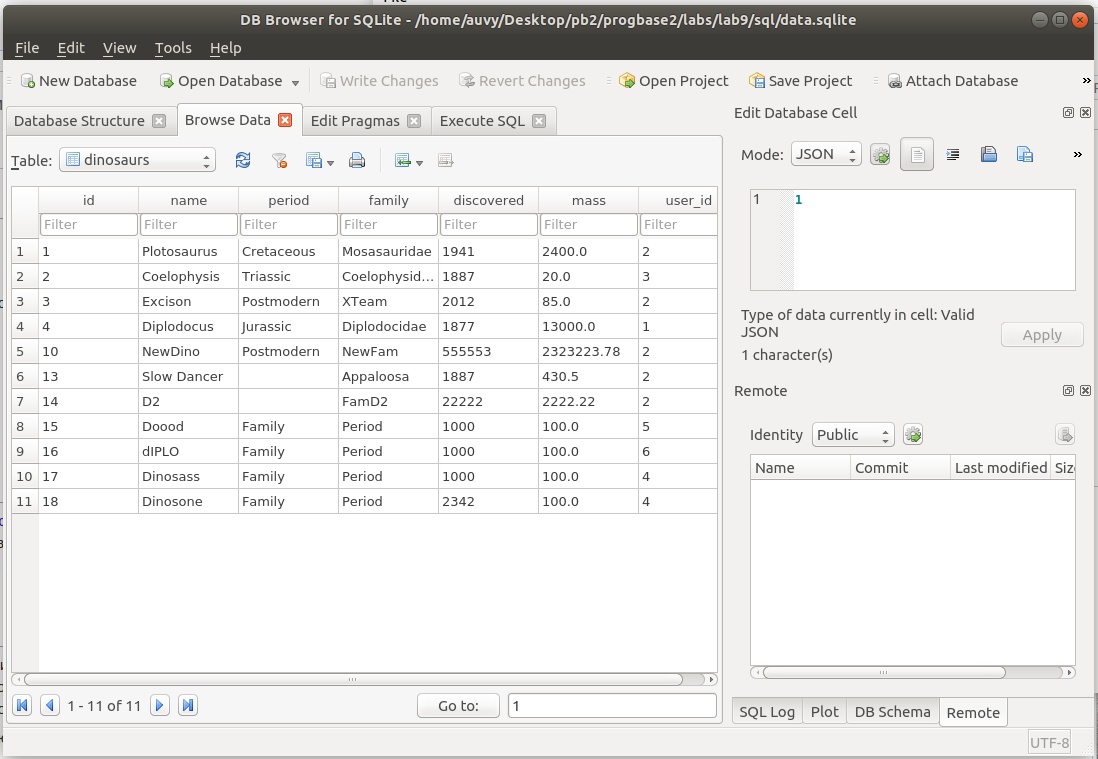


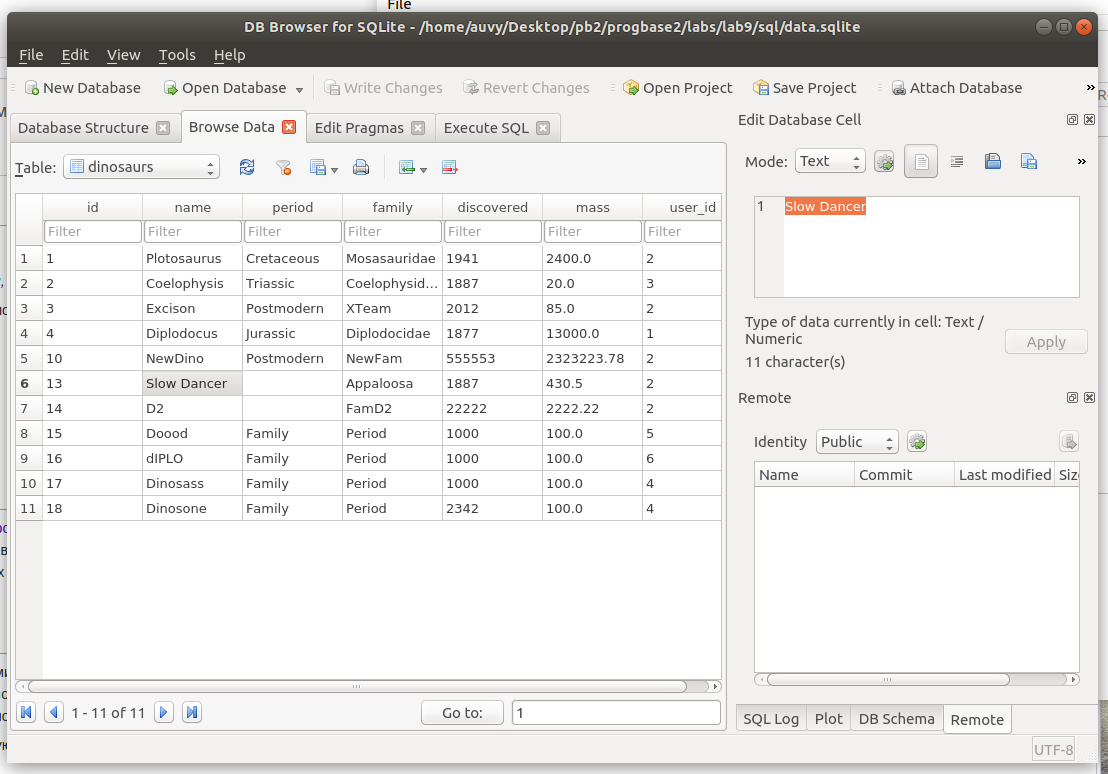


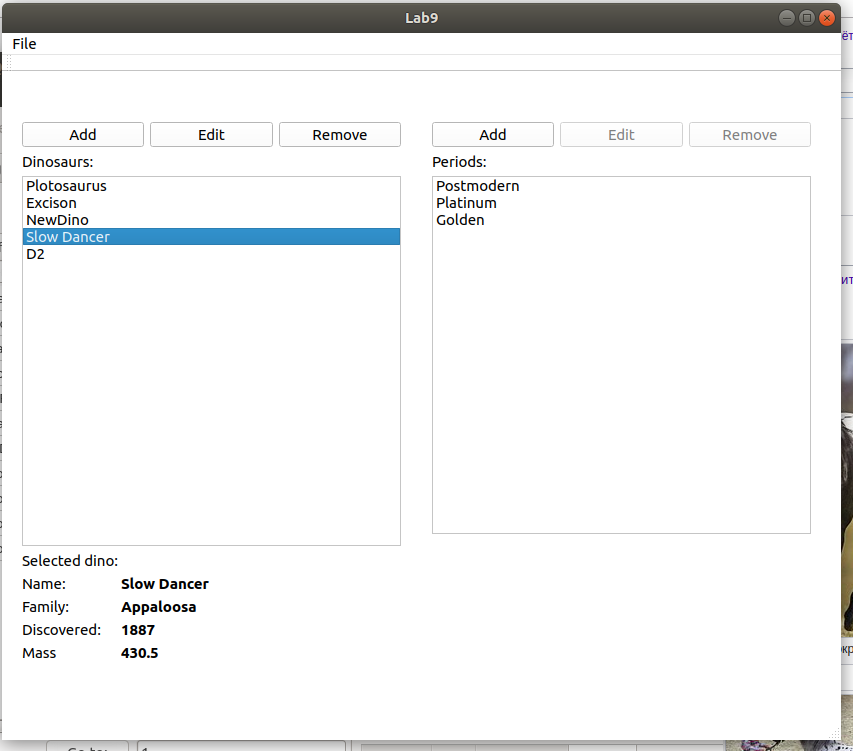


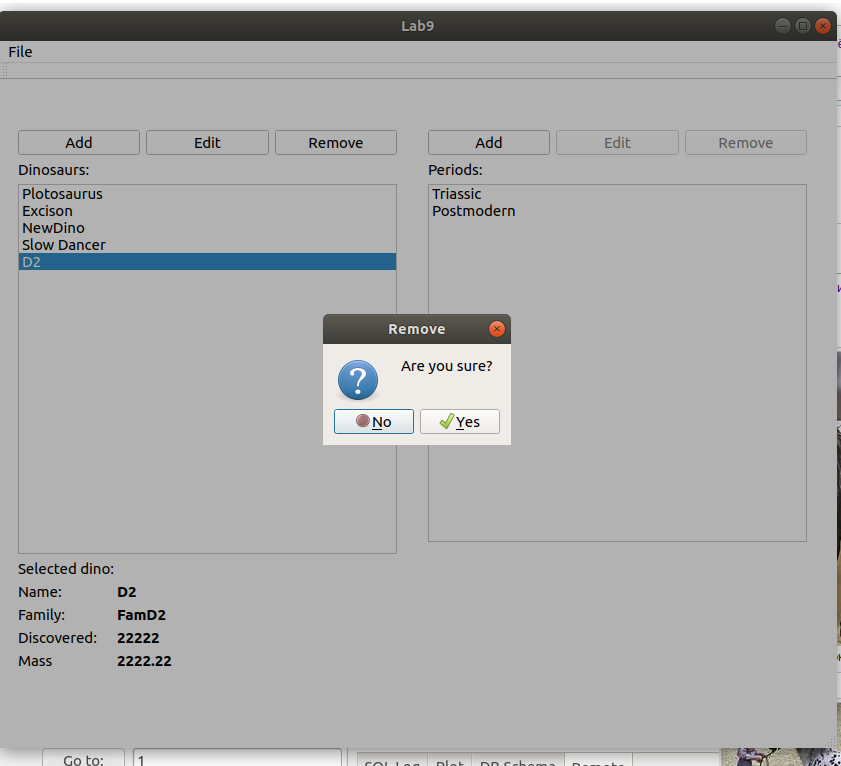


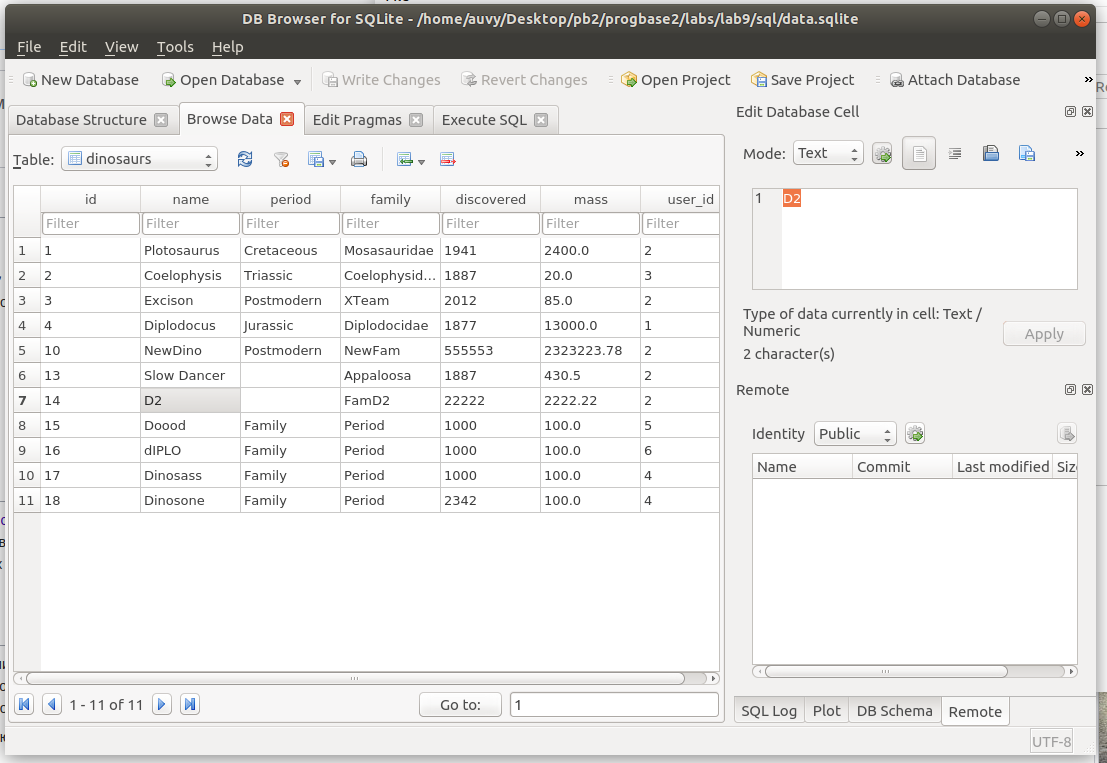


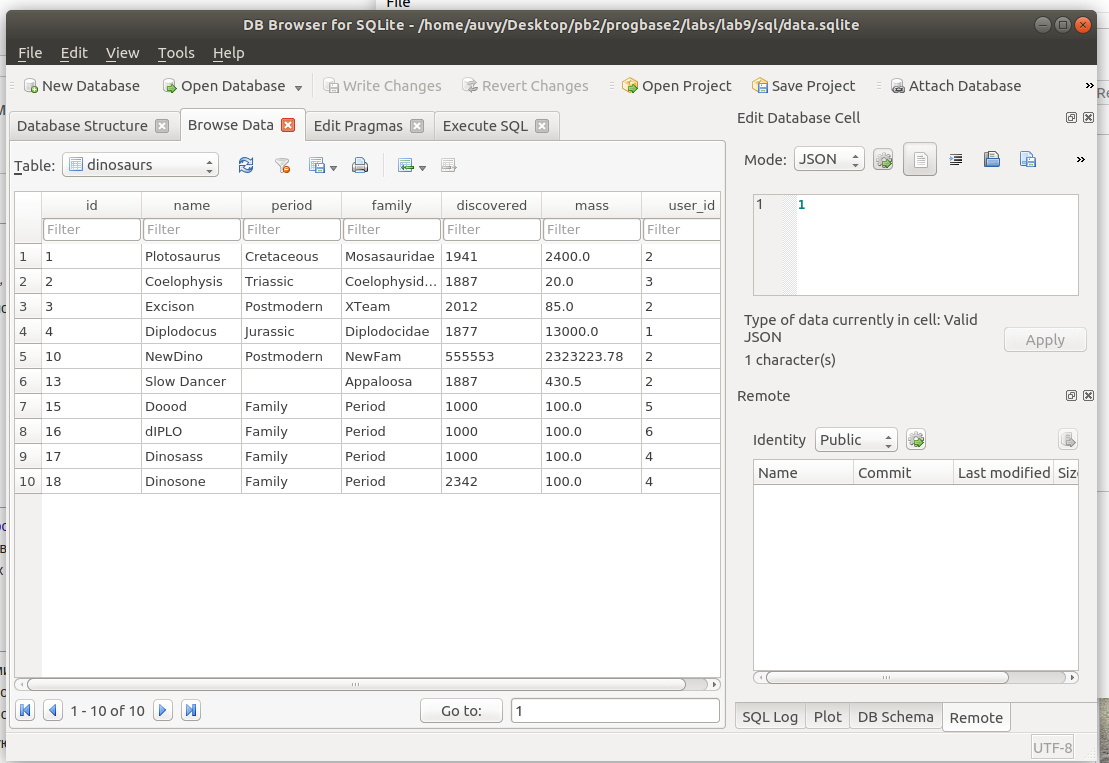












**Висновки**

Лабораторна допомогла у практиці зі створення мультвіконних графічних додатків. Була реалізовані базові можливості керування аккаунтом користувача.

Були опановані зв’язки між таблицями, а саме one-to-many, а також був ознайомлений з поняттям foreign key.