МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Воронежский государственный технический университет»

Кафедра радиоэлектронных устройств и систем

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторной работы №6
для студентов специальности 11.05.01
«Радиоэлектронные системы и комплексы»
очной формы обучения

Воронеж 2024

УДК 681.3.06(07)
ББК 32.97я7

Составитель А. И. Сукачев

Информационные технологии: методические указания к выполнению лабораторной работы № 6 для студентов специальности 11.05.01 «Радиоэлектронные системы и комплексы» очной формы обучения / ФГБОУ ВО «Воронежский государственный технический университет»; сост.: А. И. Сукачев. – Воронеж: Изд-во ВГТУ, 2024. – 34 с.

В соответствии с рабочими учебными программами дисциплин приведены описания методов измерений и методик выполнения лабораторных работы, изложены теоретические сведения, лежащие в основе программирования на языке С++. По каждой лабораторной работе в описание включены: цель, основные теоретические сведения, порядок подготовки и проведения работы, перечень положений, которые необходимо отразить в выводах.

Предназначены для студентов специальности 11.05.01 «Радиоэлектронные системы и комплексы» очной формы обучения.

Методические указания подготовлены в электронном виде и содержатся в файле МУ_ИТ_ЛР6.pdf.

Ил. 32. Табл. 2. Библиогр.: 3 назв.

УДК 681.3.06(07)
ББК 32.97я7

Рецензент – А. В. Останков, д-р техн. наук, профессор
кафедры радиотехники ВГТУ

Издается по решению редакционно-издательского совета
Воронежского государственного технического университета

1. ЛАБОРАТОРНАЯ РАБОТА № 6
РАБОТА С БАЗАМИ ДАННЫХ

1.1. ОБЩИЕ УКАЗАНИЯ

1.1.1. ЦЕЛЬ РАБОТЫ

Целью работы является разработка типового интерфейса.

1.1.2. ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Основным содержанием работы является изучить особенности разработки ПО с учетом взаимодействия с системами управления базами данных (СУБД).В ходе работы идёт ознакомление с приведённым примером приложения, производится анализ используемых технологий. На основе полученных знаний выполняется индивидуальное задание.

1.2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.2.1. КЛАССЫ БАЗЫ ДАННЫХ

В табл. 1 приведены основные классы для работы с базами данных.

Таблица 1

Список классов для работы с базами данных

QSql	Содержит разные идентификаторы, используемые во всем модуле Qt SQL
QSqlDatabase	Обрабатывает подключение к базе данных
QSqlDriver	Абстрактный базовый класс для доступа к определенным базам данных SQL
QSqlDriverCreator	Класс шаблона, предоставляющий фабрику драйверов SQL для определенного типа драйверов
QSqlDriverCreatorBase	Базовый класс для фабрик драйверов SQL
QSqlError	Сведения об ошибке базы данных SQL
QSqlField	Управляет полями в таблицах и представлениях базы данных SQL
QSqlIndex	Функции для управления и описания индексов базы данных
QSqlQuery	Средства выполнения и управления инструкциями SQL
QSqlQueryModel	Модель данных только для чтения для результирующих наборов SQL
QSqlRecord	Инкапсулирует запись базы данных
QSqlRelationalTableModel	Редактируемая модель данных для одной таблицы базы данных, с поддержкой внешнего ключа
QSqlResult	Абстрактный интерфейс для доступа к данным из конкретных баз данных SQL
QSqlTableModel	Редактируемая модель данных для одной таблицы базы данных

1.2.2. ПОДКЛЮЧЕНИЕ К БАЗАМ ДАННЫХ

Чтобы получить доступ к базе данных с помощью QSqlQuery или QSqlQueryModel, создайте и откройте одно или несколько подключений к базе данных. Соединения с базой данных обычно определяются по имени соединения, *а не* по имени базы данных. Вы можете иметь несколько подключений к одной и той же базе данных. QSqlDatabase также поддерживает концепцию соединения по умолчанию, которое является безымянным соединением. При вызове QSqlQuery или QSqlQueryModel функции-члены, которые принимают аргумент имя соединения, если вы не передадите имя соединения, будет использоваться соединение по умолчанию. Создание соединения по умолчанию удобно, когда приложению требуется только одно подключение к базе данных.

Обратите внимание на разницу между созданием соединения и его открытием. Создание соединения включает в себя создание экземпляра класса QSqlDatabase. Соединение не может быть использовано до тех пор, пока оно не будет открыто. В следующем фрагменте кода показано, как создать соединение по умолчанию и затем открыть его:

```
  QSqlDatabase  db =  QSqlDatabase :: addDatabase(( "QMYSQL");
  db
дБ .setHostName(("bigblue");
  db
дБ .setDatabaseName(("flightdb");
  db
дБ .setUserName(("acarlson");
  db
дБ .setPassword(("1uTbSbAs");
  bool ok
bool ok = db.open();
```

Рис. 1. Листинг класса QSqlDatabase

Первая строка создает объект подключения, а последняя строка открывает его для использования. Кроме того, мы инициализируем некоторые сведения о соединении , включая имя базы данных , имя хоста , имя пользователя и пароль. В этом случае мы подключаемся к базе данных MySQL flightdb на хосте bigblue."QMYSQL". Аргумент к addDatabase() указывает тип драйвера базы данных, который будет использоваться для подключения. Набор драйверов баз данных, включенных в Qt, показан в таблице поддерживаемых драйверов баз данных.

Соединение в сниппете будет соединением по умолчанию, потому что мы не передаем второй аргумент в addDatabase() , который является именем соединения. Например, здесь мы устанавливаем два соединения базы данных MySQL с именем "first" и "second" (рис. 2).

```
 QSqlDatabase firstDB = QSqlDatabase::addDatabase(("QMYSQL", "first");

QSqlDatabase secondDB = QSqlDatabase::addDatabase(("QMYSQL", "second");
```

Рис. 2. Листинг

После того, как эти соединения были инициализированы, используйте open() для каждого из них, чтобы установить живые соединения. Если open() не удается, он возвращается false. В этом случае вызовите QSqlDatabase::lastError(), чтобы получить информацию об ошибке.

Как только соединение установлено, мы можем вызвать статическую функцию QSqlDatabase::database() из любого места с именем соединения, чтобы получить указатель на это соединение с базой данных. Если мы не передадим имя соединения, он вернет соединение по умолчанию. Например:

```
QSqlDatabase defaultDB = QSqlDatabase::database();
  ();
  QSqlDatabase firstDB = QSqlDatabase::database(("first");

  QSqlDatabase secondDB = QSqlDatabase::database(("second");
```

Рис. 3. Листинг кода для проверки соединения

Чтобы удалить соединение с базой данных, сначала закройте базу данных с помощью QSqlDatabase::close(), а затем удалите ее с помощью статического метода QSqlDatabase::removeDatabase().

1.2.3. ВЫПОЛНЕНИЕ ИНСТРУКЦИИЙ SQL

Класс QSqlQuery предоставляет интерфейс для выполнения инструкций SQL и навигации по результирующему набору запроса.

Классы QSqlQueryModel и QSqlTableModel, описанные в следующем разделе, предоставляют высокоуровневый интерфейс для доступа к базам данных. Если вы не знакомы с SQL, вы можете перейти непосредственно к следующему разделу (используя классы моделей SQL).

1.2.4. ВЫПОЛНЕНИЕ ЗАПРОСА

Чтобы выполнить инструкцию SQL, просто создайте объект QSqlQuery и вызовите QSqlQuery::exec() следующим образом:

```
QSqlQuery query;
  query;
  query.exec(("SELECT name, salary FROM employee WHERE salary > 50000");
```

Рис. 4. Листинг запроса

Конструктор QSqlQuery принимает необязательный объект QSqlDatabase, указывающий, какое соединение с базой данных следует использовать. В приведенном выше примере мы не указываем никакого соединения, поэтому используется соединение по умолчанию.

При возникновении ошибки exec() возвращает false значение. Эта ошибка затем доступна как QSqlQuery::lastError().

1.2.5. НАВИГАЦИЯ ПО РЕЗУЛЬТИРУЮЩЕМУ НАБОРУ

QSqlQuery предоставляет доступ к результирующему набору по одной записи за раз. После вызова exec() внутренний указатель QSqlQuery располагается на одну позицию *перед* первой записью. Мы должны вызвать QSqlQuery::next() один раз, чтобы перейти к первой записи, а затем next() снова несколько раз, чтобы получить доступ к другим записям, пока он не вернется false. Вот типичный цикл, который повторяется над всеми записями по порядку.

```
while (query(query.next()) {
      ()) {
      QString name = query.value((0).toString();
      ();
      int salary = query.value((1).toInt();
      ();
      qDebug()  << name << salary;
   };
   }
```

Рис. 5. Листинг цикла

Функция QSqlQuery::value() возвращает значение поля в текущей записи. Поля задаются как индексы, основанные на нуле. QSqlQuery::value() возвращает QVariant, тип , который может содержать различные типы данных C++ и core Qt, такие как int, QString и QByteArray. Различные типы баз данных автоматически сопоставляются с ближайшим эквивалентом Qt. Во фрагменте кода мы вызываем QVariant::toString() и QVariant::toInt() для преобразования вариантов в QString и int.

Вы можете перемещаться по набору данных с помощью QSqlQuery::next(), QSqlQuery::previous(), QSqlQuery::first(), QSqlQuery::last() и QSqlQuery:: seek(). Текущий индекс строки возвращается QSqlQuery::at(), и общее число строк в результирующем наборе доступно как QSqlQuery:: size() для баз данных, которые его поддерживают.

Чтобы определить, поддерживает ли драйвер базы данных заданную функцию, используйте QSqlDriver::hasFeature(). В следующем примере мы вызываем QSqlQuery::size(), чтобы определить размер результирующего набора базовой базы данных, поддерживающей эту функцию; в противном случае мы переходим к последней записи и используем позицию запроса, чтобы сообщить нам, сколько записей есть.

```
QSqlQuery query;
;
int numRows;
query;
query.exec(("SELECT name, salary FROM employee WHERE salary > 50000");

QSqlDatabase defaultDB = QSqlDatabase::database();
();
if (defaultDB(defaultDB.driver()()->hasFeature((QSqlDriver::QuerySize)) {
    numRows )) {
    numRows = query.size();
} ();
} else {
    {
    // this can be very slow
    query.last();
    numRows ();
    numRows = query.at() () + 1;
}
}
}
}
```

Рис. 6. Листинг кода проверки базы данных

Если вы перемещаетесь в пределах результирующего набора и используете next() и seek() только для просмотра вперед, вы можете вызвать QSqlQuery::setForwardOnly(true) перед вызовом exec(). Это простая оптимизация, которая значительно ускорит запрос при работе с большими результирующими наборами.

1.2.6. ВСТАВКА, ОБНОВЛЕНИЕ И УДАЛЕНИЕ ЗАПИСЕЙ

QSqlQuery может выполнять произвольные инструкции SQL, а не только SELECT. в следующем примере выполняется вставка записи в таблицу с помощью INSERT (рис. 7).

```
QSqlQuery query;
    query;
    query.exec(("INSERT INTO employee (id, name, salary) "
                "VALUES (1001, 'Thad Beaumont', 65000)");
```

Рис. 7. Листинг кода вставки значения в базу данных

Если вы хотите вставить много записей одновременно, часто более эффективно отделить запрос от фактических значений, которые вставляются. Это можно сделать с помощью заполнителей. Qt поддерживает два синтаксиса заполнителей: именованную привязку и позиционную привязку. Вот пример именованной привязки (рис. 8).

```
QSqlQuery query;
   query;
   query.prepare(("INSERT INTO employee (id, name, salary) "
                  "VALUES (:id, :name, :salary)");
   query
   query.bindValue((":id", 1001);
   query
   query.bindValue((":name", "Thad Beaumont");
   query
   query.bindValue((":salary", 65000);
   query
   query.exec();();
```

Рис. 8. Листинг кода, осуществляющего вставку большого числа записей

Вот пример позиционной привязки:

```
  QSqlQuery query;
query;
query.prepare(("INSERT INTO employee (id, name, salary) "
               "VALUES (?, ?, ?)");
query
query.addBindValue((1001);
query
query.addBindValue(("Thad Beaumont");
query
query.addBindValue((65000);
query
query.exec();();
```

Рис. 9. Листинг кода вставки записей с позиционной привязкой

Оба синтаксиса работают со всеми драйверами баз данных, предоставляемыми Qt. Если база данных поддерживает синтаксис изначально, Qt просто перенаправляет запрос в СУБД; в противном случае Qt имитирует синтаксис заполнителя путем предварительной обработки запроса. Фактический запрос, который в конечном итоге выполняется СУБД, доступен как QSqlQuery::executedQuery().

При вставке нескольких записей, вам нужно только вызвать QSqlQuery::prepare() один раз. Затем вы вызываете bindValue() или addBindValue(), а затем exec() столько раз, сколько необходимо.

Помимо производительности, одним из преимуществ заполнителей является то, что вы можете легко указать произвольные значения, не беспокоясь о том, чтобы избежать специальных символов.

Обновление записи аналогично вставке ее в таблицу (рис. 10).

```
QSqlQuery query;
   query;
   query.exec(("UPDATE employee SET salary = 70000 WHERE id = 1003");
```

Рис. 10. Листинг кода обновления записи

Можно также использовать именованную или позиционную привязку для связывания параметров с фактическими значениями.

Наконец, вот пример DELETE утверждения:

```
QSqlQuery query;
   query;
   query.exec(("DELETE FROM employee WHERE id = 1007");
```

Рис. 11. Листинг кода с именованной привязкой

1.2.7. ОПЕРАЦИИ

Если базовый компонент database engine поддерживает транзакции, QSqlDriver::hasFeature(QSqlDriver::Transactions) возвращает значение true. Вы можете использовать QSqlDatabase::transaction() для инициализации транзакции, за которой следуют команды SQL, которые вы хотите выполнить в контексте транзакции, а затем либо qsqldatabase::commit(), либо QSqlDatabase::rollback(). При использовании транзакций необходимо запустить транзакцию перед созданием запроса.

Пример приведен ниже:

```
QSqlDatabase::database()().transaction();
   ();
   QSqlQuery query;
   query;
   query.exec(("SELECT id FROM employee WHERE name = 'Torild Halvorsen'");

   if (query(query.next()) {
      ()) {
      int employeeId = query.value((0).toInt();
      query();
      query.exec(("INSERT INTO project (id, name, ownerid) "
               "VALUES (201, 'Manhattan Project', "
               + QString::number(employeeId) (employeeId) + ')');
   }

   }
   QSqlDatabase::database()().commit();();
```

Транзакции могут использоваться для обеспечения атомарности сложной операции (например, поиск внешнего ключа и создание записи) или для обеспечения средств отмены сложного изменения в середине.

1.3. ИСПОЛЬЗОВАНИЕ КЛАССОВ МОДЕЛЕЙ SQL

В дополнение к QSqlQuery, Qt предлагает три класса более высокого уровня для доступа к базам данных. Эти классы являются QSqlQueryModel, QSqlTableModel и QSqlRelationalTableModel.

Таблица 2

Список моделей

QSqlQueryModel	Модель только для чтения, основанная на произвольном запросе SQL.
QSqlTableModel	Модель чтения-записи, которая работает на одной таблице.
QSqlRelationalTableModel	Подкласс QSqlTableModel с поддержкой внешнего ключа.

Эти классы являются производными от QAbstractTableModel(который в свою очередь наследует от QAbstractItemModel) и позволяют легко представлять данные из базы данных в классе представления элементов, таких как QListView и QTableView. Это подробно объясняется в разделе Представление данных в табличном представлении.

Еще одним преимуществом использования этих классов является то, что они могут упростить адаптацию кода к другим источникам данных. Например, если вы используете QSqlTableModel и позже решите использовать XML-файлы для хранения данных вместо базы данных, это по существу просто вопрос замены одной модели данных на другую.

1.3.1. МОДЕЛЬ ЗАПРОСОВ SQL

QSqlQueryModel предлагает модель только для чтения, основанную на запросе SQL.

Пример:

```
  QSqlQueryModel model;
model;
model.setQuery(("SELECT * FROM employee");



for ((int i = 0; i  i < model.rowCount(); (); ++i) {
   ) {
   int id = model.record(i)(i).value(("id").toInt();
   ();
   QString name = model.record(i)(i).value(("name").toString();
   ();
   qDebug()  << id << name;
};
```

```
}
```

После задания запроса с помощью QSqlQueryModel::setQuery(), вы можете использовать QSqlQueryModel:: record (int) для доступа к отдельным записям. Вы также можете использовать QSqlQueryModel::data() и любые другие функции, унаследованные от QAbstractItemModel.

Существует также перегрузка setQuery(), которая принимает объект QSqlQuery и работает с его результирующим набором. Это позволяет использовать любые функции QSqlQuery для настройки запроса (например, подготовленные запросы).

1.3.2. ТАБЛИЧНАЯ МОДЕЛЬ SQL

QSqlTableModel предлагает модель чтения-записи, которая работает на одной таблице SQL одновременно.

Пример:

```
QSqlTableModel model;
   model;
   model.setTable(("employee");
   model
   model.setFilter(("salary > 50000");
   model
   model.setSort((2, Qt::DescendingOrder);
   model);
   model.select();

   ();

   for ((int i = 0; i  i < model.rowCount(); (); ++i) {
      ) {
      QString name = model.record(i)(i).value(("name").toString();
      ();
      int salary = model.record(i)(i).value(("salary").toInt();
      ();
      qDebug()  << name << salary;
   };
   }
```

QSqlTableModel-это высокоуровневая альтернатива QSqlQuery для навигации и изменения отдельных таблиц SQL. Это обычно приводит к меньшему количеству кода и не требует знания синтаксиса SQL.

Используйте QSqlTableModel::record() для извлечения строки в таблице и QSqlTableModel::setRecord() для изменения строки. Например, следующий код увеличит зарплату каждого сотрудника на 10%.

```
for ((int i = 0; i  i < model.rowCount(); (); ++i) {
      ) {
      QSqlRecord record = model.record(i);
      (i);
      double salary = record.value(("salary").toInt();
      salary ();
      salary *= 1.1;
      record
      record.setValue(("salary", salary);
      model);
      model.setRecord(i(i, record);
   }
   model);
   }
   model.submitAll();();
```

Для доступа к данным также можно использовать qsqltablemodel::data() и QSqlTableModel::setData(), унаследованные от QAbstractItemModel. Например, вот как обновить запись с помощью setData() (рис. 12).

```
model.setData(model(model.index(row(row, column)), 75000);
   model
   model.submitAll();();
```

Рис. 12. Листинг кода доступа к данным

Вот как вставить строку и заполнить ее:

```
model.insertRows(row(row, 1);
   model
   model.setData(model(model.index(row(row, 0), 1013);
   model
   model.setData(model(model.index(row(row, 1), "Peter Gordon");
   model
   model.setData(model(model.index(row(row, 2), 68500);
   model
   model.submitAll();();
```

Вот как удалить пять последовательных строк:

```
model.removeRows(row(row, 5);
   model
   model.submitAll();();
```

Первый аргумент для QSqlTableModel::removeRows() - это индекс первой удаляемой строки.

Когда вы закончите изменять запись, вы всегда должны вызывать QSqlTableModel::submitAll(), чтобы гарантировать, что изменения записываются в базу данных.

Когда и действительно ли вам *нужно* вызвать submitAll(), зависит от стратегии редактирования таблицы. По умолчанию используется стратегия QSqlTableModel:: OnRowChange, которая указывает, что ожидающие изменения применяются к базе данных, когда пользователь выбирает другую строку. Другими стратегиями являются QSqlTableModel::OnManualSubmit (где все изменения кэшируются в модели до тех пор, пока вы не вызовете submitAll()) и QSqlTableModel:: OnFieldChange (где никакие изменения не кэшируются). Они в основном полезны, когда QSqlTableModel используется с представлением.

QSqlTableModel::OnFieldChange, похоже, дает обещание, что вам никогда не нужно будет явно вызывать submitAll(). Однако есть две подводные камни:

- Без какого-либо кэширования производительность может значительно снизиться.
- Если вы измените первичный ключ, запись может проскользнуть между вашими пальцами, когда вы пытаетесь заполнить ее.

2. РЕЛЯЦИОННАЯ ТАБЛИЧНАЯ МОДЕЛЬ SQL

QSqlRelationalTableModel расширяет QSqlTableModel для обеспечения поддержки внешних ключей. Внешний ключ - это сопоставление 1-к-1 между полем в одной таблице и полем первичного ключа другой таблицы. Например, если в book таблице есть поле с именем authorid, которое ссылается на поле таблицы authorid, мы говорим, что authorid это внешний ключ.

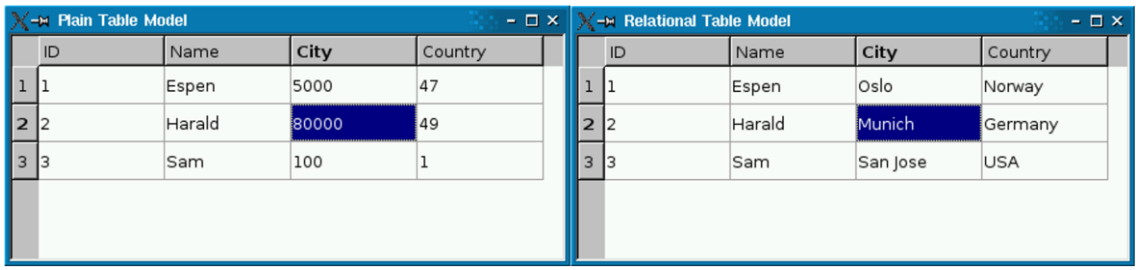

Рис. 13. Таблица базы данных

Снимок экрана слева показывает простую QSqlTableModel в QTableView. Внешние ключи (city и country) не разрешаются к удобочитаемым значениям. На снимке экрана справа показана модель QSqlRelationalTableModel с внешними ключами, разрешенными в удобочитаемые текстовые строки.

Следующий фрагмент кода показывает, как была настроена модель QSqlRelationalTableModel:

model->setTable(("employee");

model

model->setRelation((2, QSqlRelation("city", "id", "name"));
model
model->setRelation((3, QSqlRelation("country", "id", "name"));

Дополнительную информацию смотрите в документации QSqlRelationalTableModel.

2.1. ПРЕДСТАВЛЕНИЕ ДАННЫХ В ТАБЛИЧНОМ ВИДЕ

Классы QSqlQueryModel , QSqlTableModel и QSqlRelationalTableModel могут использоваться в качестве источника данных для классов представлений Qt, таких как QListView, QTableView и QTreeView. На практике QTableView является на сегодняшний день наиболее распространенным выбором, потому что результирующий набор SQL является по существу двумерной структурой данных.

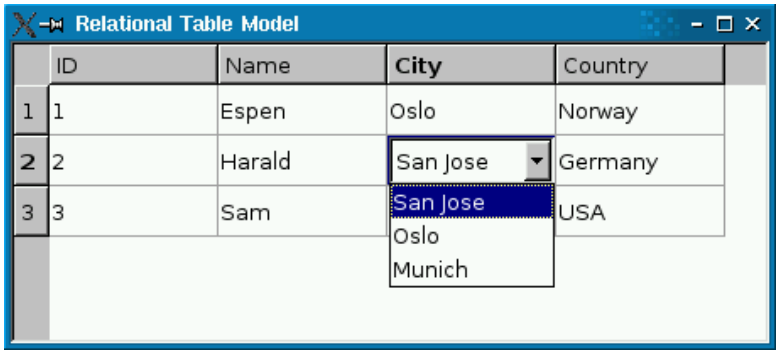

Рис. 14. Представление данных в таблице

В следующем примере создается представление на основе модели данных SQL:

QTableView *view = new QTableView;
view
view->setModel(model);
view(model);
view->show();();

Если модель является моделью чтения-записи (например, QSqlTableModel), представление позволяет пользователю редактировать поля. Вы можете отключить это, вызвав следующую функцию:

view- > > setEditTriggers((QAbstractItemView :: NoEditTriggers););

Можно использовать одну и ту же модель в качестве источника данных для нескольких представлений. Если пользователь редактирует модель с помощью одного из представлений, другие представления будут немедленно отражать изменения. Пример табличной модели показывает, как это работает.

Просмотр классов отображение заголовка в верхней части для надписывания столбцов. Чтобы изменить текст заголовка, вызовите setHeaderData() на модели. Метки заголовка по умолчанию соответствуют именам полей таблицы. Например:

```
model->setHeaderData((0, Qt::Horizontal, QObject::tr(("ID"));
  model
  model->setHeaderData((1, Qt::Horizontal, QObject::tr(("Name"));
  model
  model->setHeaderData((2, Qt::Horizontal, QObject::tr(("City"));
  model
  model->setHeaderData((3, Qt::Horizontal, QObject::tr(("Country"));
```

QTableView также имеет вертикальный заголовок слева с номерами, идентифицирующими строки. Если вы вставляете строки программно с помощью QSqlTableModel::insertRows(), новые строки будут помечены звездочкой (*) до тех пор, пока они не будут отправлены с помощью submitAll() или автоматически при переходе пользователя к другой записи (предполагая, что стратегия редактирования qsqltablemodel:: OnRowChange).

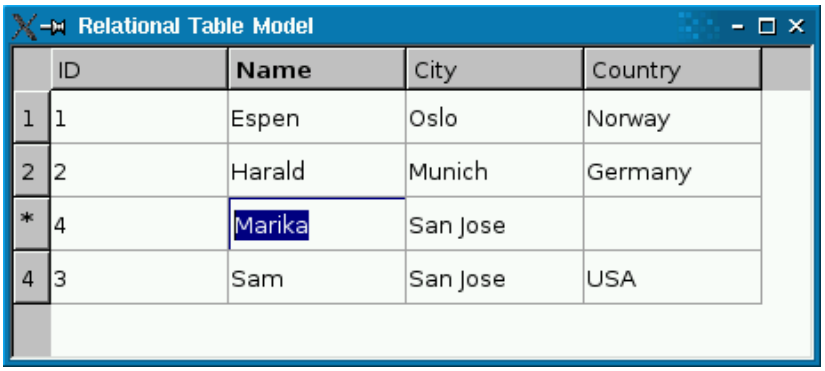

Рис. 15. Пример пометки новых строк

Аналогично, если вы удалите строки с помощью removeRows (), строки будут отмечены восклицательным знаком (!) до тех пор, пока не будет представлено изменение.

Элементы в представлении отображаются с помощью делегата. Делегат по умолчанию, QItemDelegate, обрабатывает наиболее распространенные типы данных (int, QString, QImage и т. д.). Делегат также отвечает за предоставление виджетов редактора (например, combobox), когда пользователь начинает редактировать элемент в представлении. Вы можете создать свои собственные делегаты, создав подкласс QAbstractItemDelegate или QItemDelegate. Дополнитель-

ную информацию смотрите в разделе Программирование моделей и представлений.

QSqlTableModel оптимизирован для работы с одной таблицей одновременно. Если вам нужна модель чтения-записи, которая работает с произвольным результирующим набором, вы можете подкласс QSqlQueryModel и переопределить flags() и setData(), чтобы сделать его чтение-запись. Следующие две функции делают поля 1 и 2 модели запроса редактируемыми:

```
Qt::ItemFlags EditableSqlModel EditableSqlModel::flags(
      (
      const QModelIndex &index) ) const
{
   {
   Qt::ItemFlags flags  flags = QSqlQueryModel::flags(index);
   (index);
   if (index(index.column() () == 1 || index.column() () == 2)
      flags
      flags |= Qt::ItemIsEditable;
   ;
   return flags;
}

bool EditableSqlModel;
}

bool EditableSqlModel::setData((const QModelIndex &index, const QVariant &value, int /* role */)
{

{
   if (index(index.column() () < 1 || index.column() () > 2)

      return false;



   QModelIndex primaryKeyIndex = QSqlQueryModel::index(index(index.row()(), 0);

   int id = data(primaryKeyIndex)(primaryKeyIndex).toInt();

   clear();
```

```
    bool ok;
    ();

    clear();

    bool ok;
    if (index(index.column() () == 1) {
       ok  {
       ok = setFirstName(id(id, value.toString());
    } ());
    } else {
       ok {
       ok = setLastName(id(id, value.toString());
    }
    refresh();
    ());
    }
    refresh();
    return ok;
};
}
```

Вспомогательная функция setFirstName() определяется следующим образом:

```
bool EditableSqlModel EditableSqlModel::setFirstName((int personId, const QString &firstName)
{
    )
{
    QSqlQuery query;
    query;
    query.prepare(("update person set firstname = ? where id = ?");
    query
    query.addBindValue(firstName);
    query(firstName);
    query.addBindValue(personId);
    (personId);
    return query.exec();
}();
}
```

Функция setLastName() аналогична. Смотрите пример модели запроса для полного исходного кода.

Создание подклассов модели позволяет настраивать ее различными способами: можно предоставлять подсказки для элементов, изменять цвет фона, предоставлять вычисляемые значения, предоставлять различные значения для просмотра и редактирования, обрабатывать значения null специально и многое другое. Дополнительную информацию смотрите в разделе Программирование моделей и представлений, а также в справочной документации QAbstractItemView.

Если все, что вам нужно, это решить внешний ключ к более удобной для человека строке, вы можете использовать QSqlRelationalTableModel. Для достижения наилучших результатов следует также использовать делегат QSqlRelationalDelegate, предоставляющий Редакторы combobox для редактирования внешних ключей.

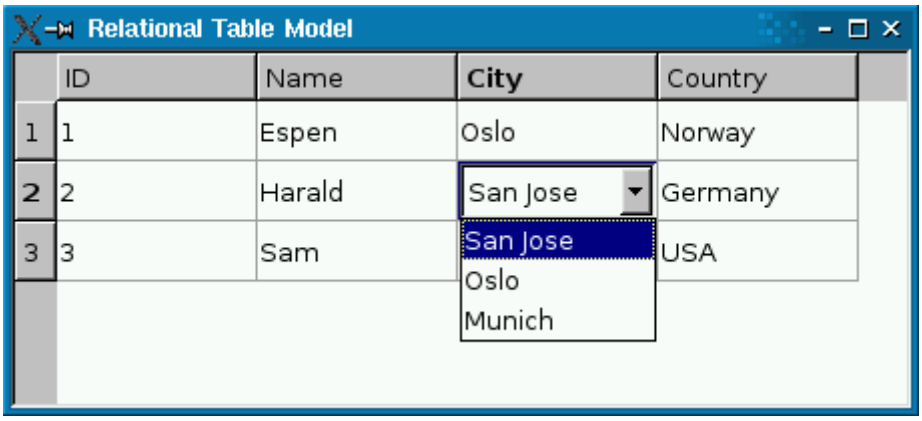

Рис. 16. Использование внешнего ключа

Пример реляционной табличной модели иллюстрирует, как использовать QSqlRelationalTableModel в сочетании с QSqlRelationalDelegate для предоставления таблицам поддержки внешнего ключа.

2.2. СОЗДАНИЕ ФОРМ С ПОДДЕРЖКОЙ ДАННЫХ

С помощью описанных выше моделей SQL содержимое базы данных может быть представлено другим компонентам модели/представления. Для некоторых приложений достаточно представить эти данные с помощью стандартного представления элемента, такого как QTableView. Однако пользователям приложений на основе записей часто требуется пользовательский интерфейс на основе форм, в котором данные из определенной строки или столбца таблицы базы данных используются для заполнения виджетов редактора на форме.

Такие формы с поддержкой данных могут быть созданы с помощью класса QDataWidgetMapper, общего компонента модели/представления, который используется для сопоставления данных из модели в определенные виджеты в пользовательском интерфейсе.

QDataWidgetMapper работает с определенной таблицей базы данных, сопоставляя элементы в таблице на основе строки за строкой или столбца за столбцом. В результате использование QDataWidgetMapper с моделью SQL так же просто, как и использование его с любой другой табличной моделью.

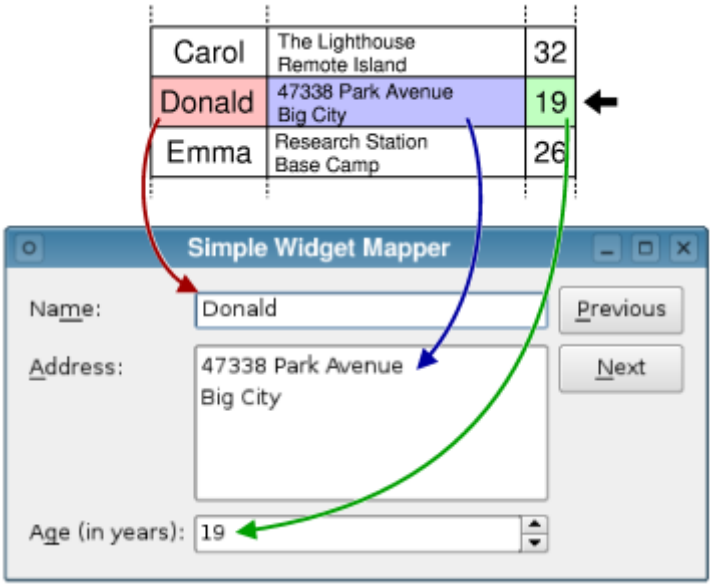

Рис. 17. Получение информации с помощью QDataWidgetMapper

В примере показано, как информация может быть представлена для быстрого доступа с помощью QDataWidgetMapper и набора простых виджетов ввода.

3. ЛАБОРАТОРНЫЕ ЗАДАНИЯ И МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ИХ ВЫПОЛНЕНИЮ

3.1. СОЗДАНИЕ ТАБЛИЦ В POSTGRESQL

В качестве СУБД использовалась свободная PostgreSQL. Она обладает всем необходимым нам функционалом.

Для начала работы с Postgres нужно запустить pgAdmin4.exe. Это инструмент для упрощения администрирования на сервере PostgreSQL, включенный в ее базовый комплект. По сути он представляет собой графический клиент для работы с сервером, через который в удобном виде можно создавать, удалять, изменять базы данных и управлять ими. Главным достоинством pgAdmin является то, что он позволяет управлять СУБД без непосредственного ввода SQL-команд.

После запуска pgAdmin4 на компьютере создается сервер для работы с базами данных.

Интерфейс pgAdmin состоит из меню (рис.18) и обозревателя (Рис.19).

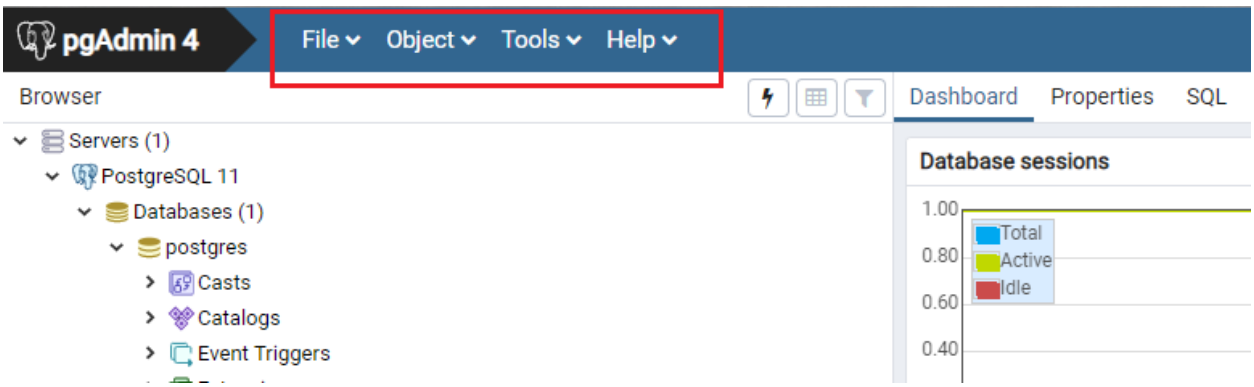

Рис. 18. Меню pgAdmin4

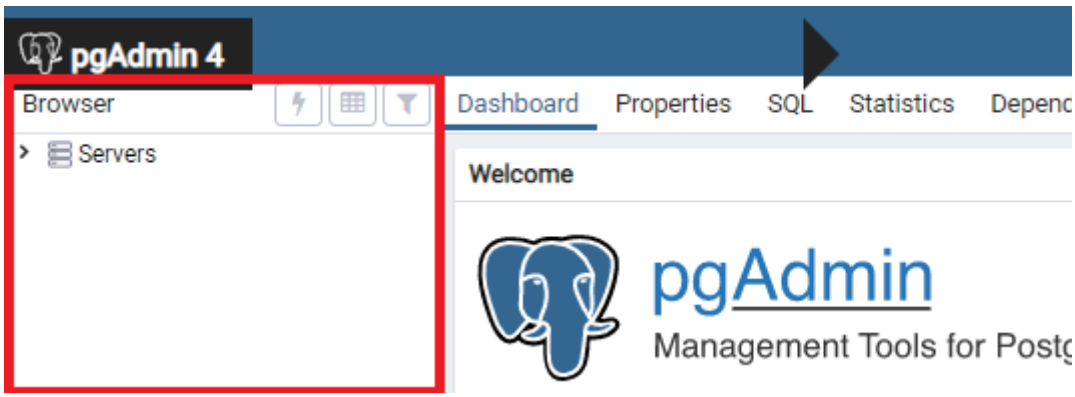

Рис. 19. Обозреватель pgAdmin4

Сначала в обозревателе отображена только вкладка Servers. При щелчке по ней появится дочерняя ветвь с названием PostgreSQL 11. После нее идут ветви Databases, Login/Group Roles и Tablespaces.

Login/Group Roles содержит т.н. роли. Это пользователи с разными правами доступа. По умолчанию их 9, среди них postgres. В рамках обучения можно воспользоваться и им, хотя на практике это нерационально с точки зрения безопасности, поскольку пользователь postgres обладает очень широкими правами доступа, среди которых запись, изменение и удаление данных. Обозреватель с раскрытыми упомянутыми ветвями приведен на рис. 20.

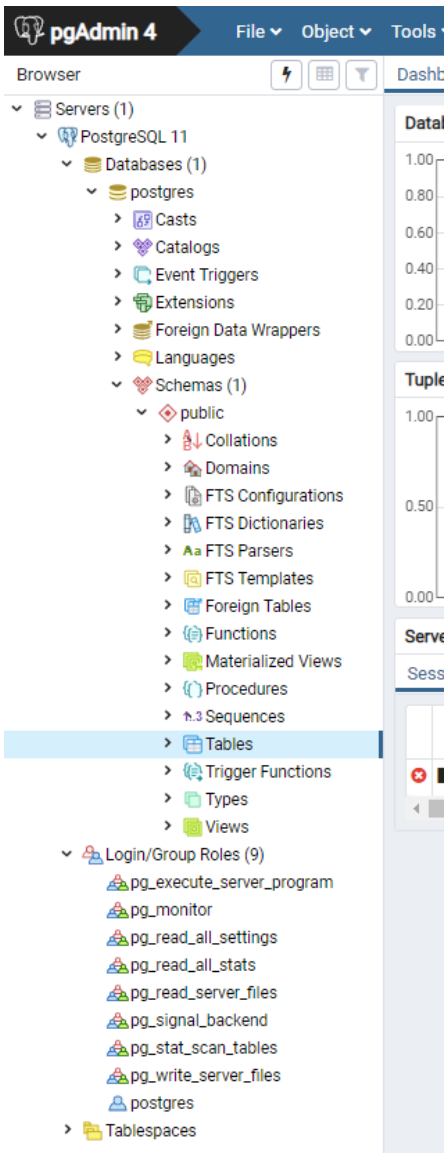

Рис. 20. Обозреватель pgAdmin4 с раскрытыми Databases и Login/Group Roles

При открытии Databases появится список имеющихся баз данных. По умолчанию такая база одна, и это postgres. Это стандартная БД, создаваемая системой при инициализации.

Создадим новую базу данных, чтобы иметь возможность вручную задать ее параметры и разобраться в механике процесса. Для этого требуется нажать правой кнопкой мыши по Databases, затем выбрать Create—Database (Рис.4). При этом стоит уделить внимание полю Owner (Владелец), в котором задается пользователь, имеющий полные права доступа к создаваемой таблице. По умолчанию это postgres, в данном случае менять ничего не нужно. Зададим имя БД — db. Для завершения создания базы данных служит кнопка Save.

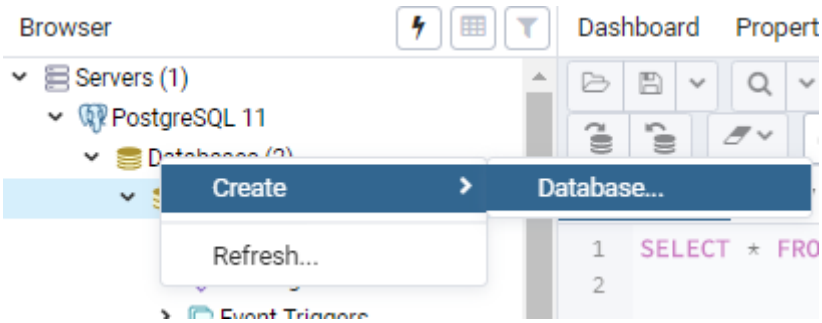

Рис. 21. Создание БД

По той же причине теперь нужно создать нового пользователя. Для этого правой кнопкой мыши нажимаем на Login/Group Roles и затем Create — Login/Group Role. В появившемся окне зададим имя нового пользователя – user.

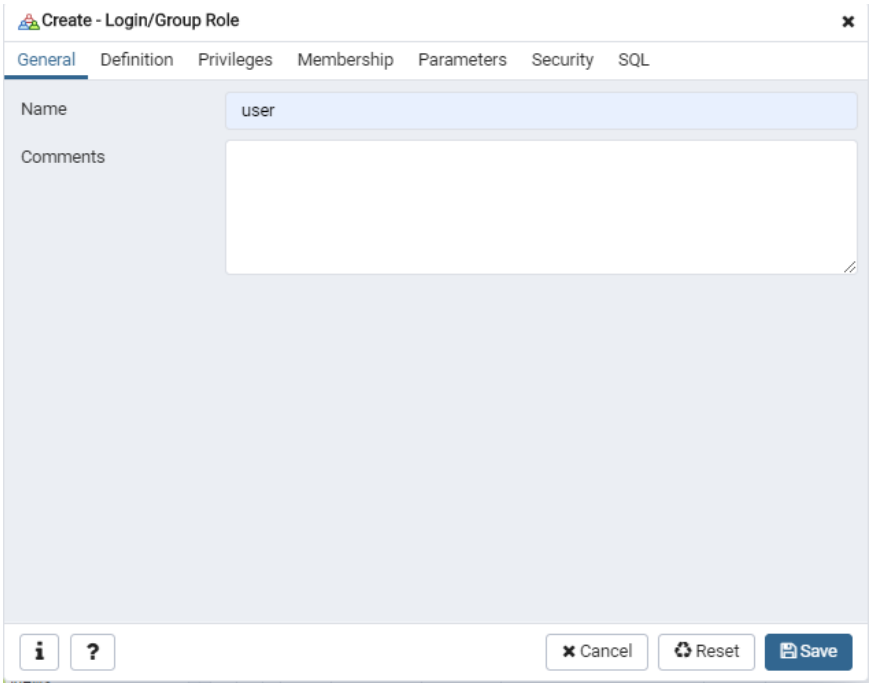

Рис. 22. Общие настройки создаваемого пользователя

Далее на вкладке Definition есть возможность:

1. Задать пароль пользователя в поле Password.
2. Дату истечения его прав доступа в Account Expires.
3. Количество подключений к БД, которые он может выполнить, — Connection Limit.

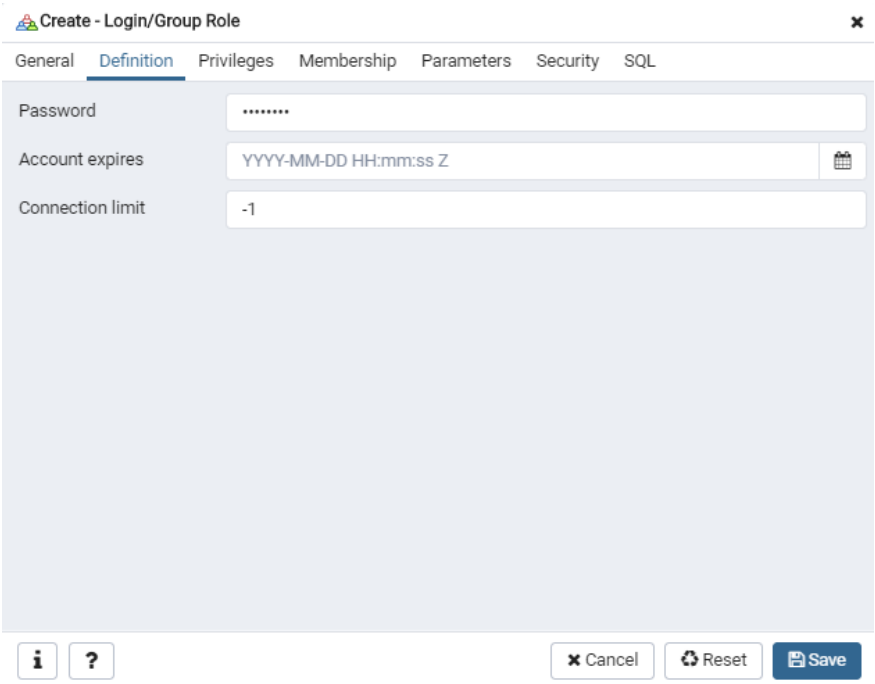

Рис. 23. Настройки доступа создаваемого пользователя

В вкладке Privileges настраиваются права пользователя:

1. Переключатель Can login (Возможность подключения) определяет, сможет ли данный пользователь подключаться в БД. Он должен быть выставлен в положение Yes.
2. Переключатель Superuser (Администратор) задает, сможет ли этот пользователь менять права других пользователей и т.д. Оставляем в состоянии No.
3. Переключатель Create roles определяет, может ли пользователь создавать других пользователей. Оставляем No.
4. Create Databases задает право пользователя создавать базы данных. Переводим в Yes.
5. Can initiate streaming replication and backups — может ли пользователь запускать потоковую репликацию и создание резервных копий.

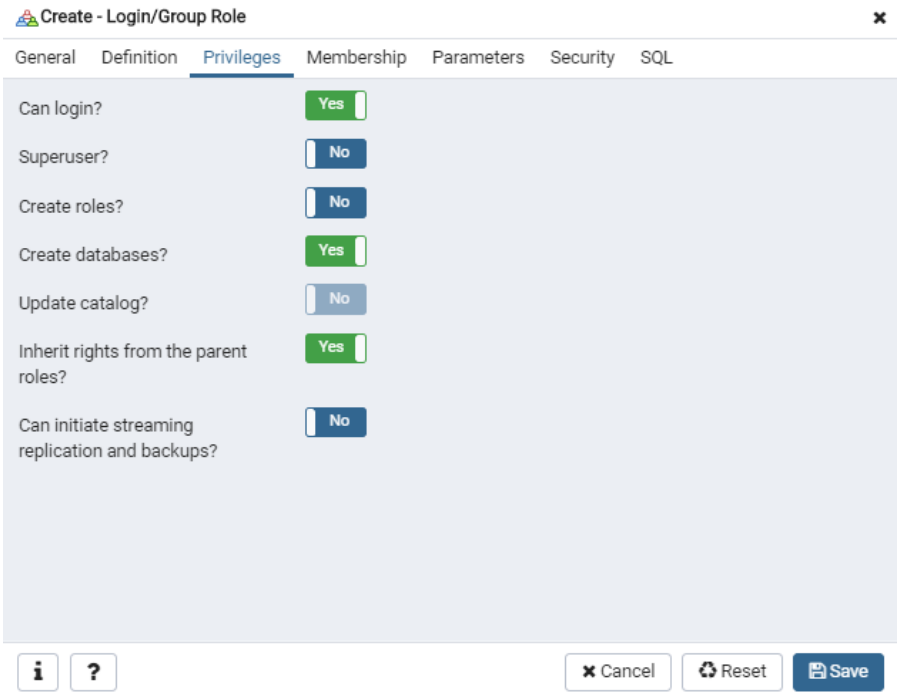

Рис. 24. Настройка привилегий (прав) создаваемого пользователя

После создания и настройки нового пользователя необходимо сделать его владельцем созданной ранее базы данных db. Это делается следующим образом: делается щелчок правой кнопкой мыши по названию БД, выбирается Properties, и во вкладке Owner назначается нужный пользователь.

Для создания таблицы в списке элементов БД db требуется открыть ветвь Schemas. По умолчанию в ней находится схема public. Нам понадобится еще одна схема, создадим ее. Для этого: ПКМ на Schemas, затем Create — Schema. Задаем имя новой схемы — private, и нажимаем Save.

После этого в обозревателе открываем private. В нем выбираем Tables, содержащую таблицы. Сначала список таблиц будет пуст. Создадим таблицу. Для этого необходимо щелкнуть по Tables правой кнопкой мыши и выбрать Create—Table.

Откроется окно конфигурации будущей таблицы.

- Во вкладке General в поле Name указывается имя таблицы (в нашем случае city) и в поле Owner ее владелец.

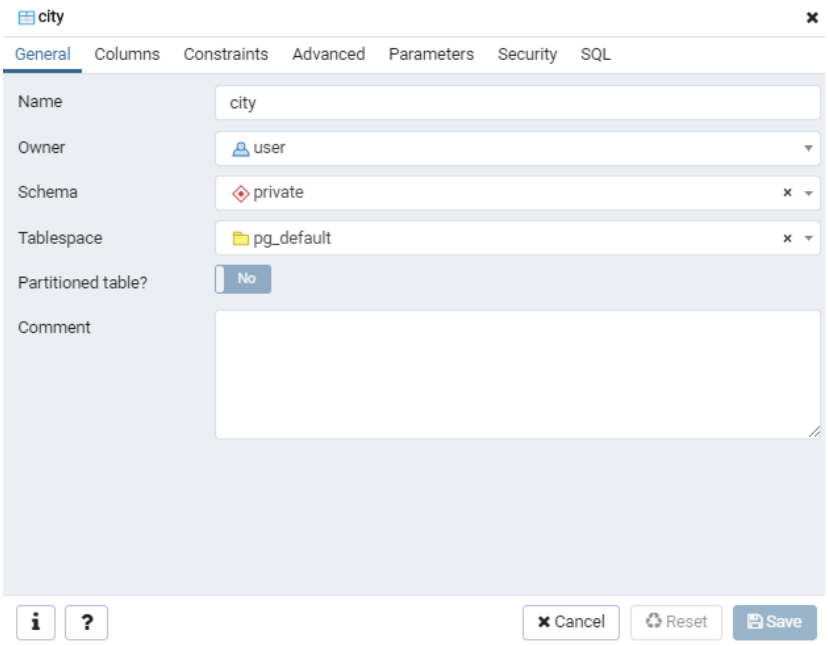

Рис. 25. Общие настройки таблицы

- Во вкладке Columns требуется создать новую колонку в создаваемой таблице, кликнув по «+».
- Колонке присваиваем имя idcity, тип данных (Data type) integer. Переводим переключатели Primary key и Not NULL в Yes, тем самым обозначив данную колонку как первичный ключ таблицы. Это уникальное в рамках данной таблицы значение, по которому будут идентифицироваться все ее записи. Это поле требуется, чтобы в дальнейшем осуществлять связи записей, что уменьшает риск возникновения двойственности данных. Также это поле служит для обращения к записи.
- Создаем вторую колонку с именем city и типом данных text.

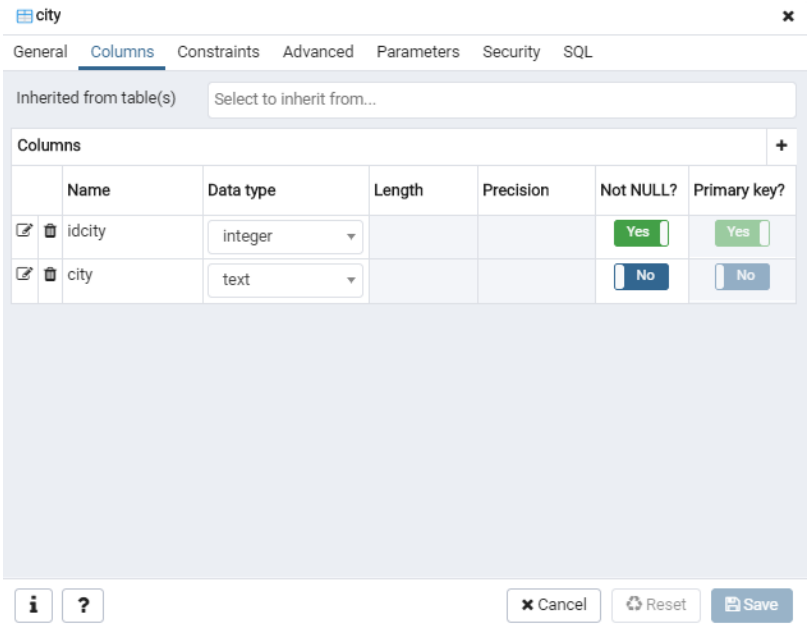

Рис. 26. Окно настройки колонок

Таким же образом нужно создать таблицу country с колонками idcountry и name.

Добавим записи в таблицу city. Для этого требуется кликнуть правой кнопкой мыши по ней и выбрать View/Edit Data—All Rows и внести данные: запись с именем Voronezh и idcity 1, запись с именем Moscow и idcity 2 и запись с именем New York и idcity 3.

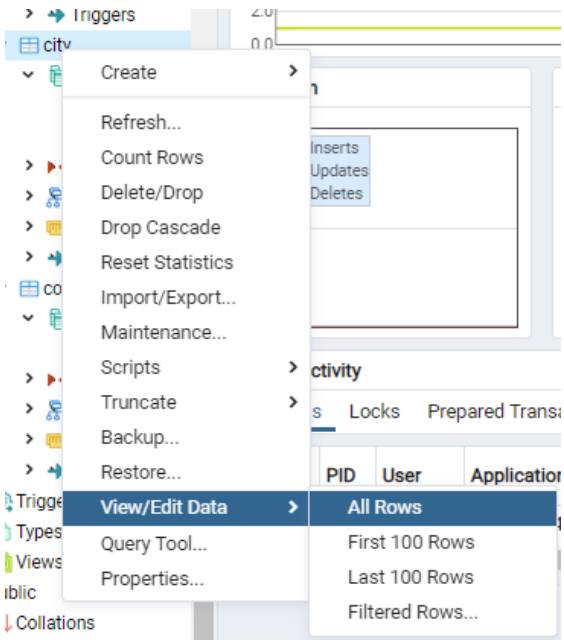

Рис. 27. Активация режима просмотра таблицы

Вот что должно получиться:

	idcity [PK] integer	city text
1	3	New York
2	1	Voronezh
3	2	Moscow

Рис. 28. Таблица городов

Аналогично вносим записи в таблицу country: Russia с idcontry 1 и USA с idcontry 2.

Создадим таблицу People в схеме private. В ней будет храниться информация о людях. Указанным выше способом вносим в нее записи: Pavel с idpeople 1, Ivan с idpeople 2 и Joe с idpeople 3.

Далее в схеме public создаем таблицу ConnectionPeopleToCity, с помощью которой будет задаваться связь людей с городами. В ней добавляем колонки idpeople, idcity, idcountry. Все они должны быть Not NULL, а idpeople — еще и Primary key. В упомянутые колонки необходимо внести соответствующие id из предыдущих таблиц. Это свяжет внесенные ранее города, страны и личности в рамках общего для них реестра. В итоге получится так:

	idpeople [PK] integer	idcity integer	idcountry integer
1	1	2	1
2	2	1	1
3	3	3	2

Рис. 29. Таблица взаимосвязей людей, городов и стран

На этом можно обзор основных возможностей PostgreSQL и создание БД с его помощью можно считать завершенным.

3.2. НАСТРОЙКА ВЗАИМОДЕЙСТВИЯ QT CREATOR И POSTGRESQL

Для того, чтобы Qt мог взаимодействовать с СУБД, нужно указать ему путь к ее драйверу либо скопировать этот драйвер в директорию компилятора. Он находится по пути: папка_с_PostgreSQL\pgAdmin\bin. Папка назначения: Qt\папка с версией установленного Qt в качестве названия\папка с компилятором\bin. Теперь возможна корректное взаимодействие Qt и PostgreSQL.

Итак, среде Qt теперь известно местоположение драйвера СУБД. Теперь нужно активизировать его работу в самом Qt. Для этого в файле main.cpp вызывается статический метод QSqlDatabase::addDatabase(“”) .В него необходимо передать строку, обозначающую идентификатор драйвера СУБД. В нашем случае это QPSQL.

Для успешного подключения к БД требуется четыре параметра:

- Имя БД — передается в метод QSqlDatabase::setDatabaseName();

- Имя присоединяющегося пользователя — передается в метод QSqlDatabase::setUserName();
- Имя компьютера, на котором размещена БД — передается в метод QSqlDatabase::setHostName();
- Пароль — передается в метод QSqlDatabase::setPassword() .

```
#include "widget.h"
#include <QApplication>
#include <QtSql>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QSqlDatabase db = QSqlDatabase::addDatabase("QPSQL");
    db.setHostName("localhost");
    db.setUserName("user");
    db.setDatabaseName("db");
    db.setPassword("00000000");
    if(!db.open())
    {
        qDebug()<<"error db:"<<db.lastError();
    }
    Widget w;
    w.resize(950, 550);
    w.show();
    return a.exec();
}
```

Рис. 30. Подключение к базе данных

Методы должны вызываться из объекта, созданного с помощью статического метода

Само соединение осуществляется методом QsglDatabase::open(). Значение, возвращаемое им, рекомендуется проверять. В случае возникновения ошибки, информацию о ней можно получить с помощью метода QSglDatabase::lastError(), который возвращает объект класса QSqlError. Его содержимое можно вывести на экран с помощью метода qDebug() . Если вы хотите получить строку с ошибкой, то можно вызвать метод text() объекта класса QSqlError.

При помощи объекта класса QslDatabase можно также получить и мета-информацию о базе данных — например, о таблицах. Это можно сделать конечно же только после подключения к самой базе данных. Следующий пример

при помощи метода QSqlDatabase:: tables () получает информацию об именах всех таблиц, которые находятся в базе данных и отображает их:

3.3. ИСПОЛНЕНИЕ КОМАНД SQL (ВТОРОЙ УРОВЕНЬ)

Для исполнения команд SQL после установки соединения можно использовать класс QSqlQuery. Запросы (команды) оформляются в виде обычной строки, которая передается в конструктор или в метод QSqlQuery::exec(). При передаче запроса в конструктор запуск команды будет выполняться автоматически при создании объекта.

Класс QSqlQuery предоставляет возможность навигации. Например, после выполнения запроса SELECT можно перемещаться по отобранным данным при помощи методов next(), previous(), first(), last() и seek(). С помощью метода next() мы перемещаемся на следующую строку данных, а вызов метода previous() перемещает нас на предыдущую строку данных. Методы first o и last() помогут нам установить первую и последнюю строку данных соответственно. Метод seek() устанавливает текущей строку данных, целочисленный индекс которой указан в параметре. Количество строк данных можно получить вызовом метода size().

```
void NewModel::refresh()
{
    clear();
    insertColumns(0,3);
    QSqlQuery query;
    query.exec("select* from db");
    qDebug()<<query.lastError();
    int id;
    QString code;
    QString university;
    QSqlRecord rec = query.record();
    while (query.next())
    {
        id = query.value(rec.indexOf("id")).toInt();
        code = query.value(rec.indexOf("code")).toString();
        university = query.value(rec.indexOf("university")).toString();
        insertRow(rowCount());
        setData(index(rowCount()-1,0),id);
        setData(index(rowCount()-1,1),code);
        setData(index(rowCount()-1,2),university);
    }
}
```

Рис. 31. Выполнение запроса SELECT

Дополнительные сложности возникают с запросом INSERT. Дело в том, что в запрос нужно внедрять данные. Для этого можно воспользоваться двумя методами: prepare() и bindValue(). В методе prepare () мы задаем шаблон, данные в который подставляются методами bindValue() . Например:

query.prepare("INSERT INTO название таблицы (названия колонок через запятую)"

"VALUES (:имя колонки, :имя колонки);");

query.bindValue(":имя колонки", значение);

Можно также прибегнуть и к известному из интерфейса ODBC методу использования безымянных параметров:

query.prepare ("INSERT INTO название (имена колонок через запятую)" "VALUES(?, ?, ?, ?);");

query.bindValue(значение первой колонки);

query.bindValue(значение второй колонки);

и т.д.

Есть и третий вариант - воспользоваться классом OString, в частности, методом Qstring::arg(), с помощью которого имеется возможность выполнить подстановку значений данных.

Разумеется, в рассматриваемом случае можно было бы сразу вставить данные в текст запроса, поскольку они известны заранее, но в реальных приложениях данные чаще всего представляют собой значения выражений, и такой подход не срабатывает.

Воспользуемся вторым вариантом.

```
void SQLTreeModel::updateTree(QString y)
{
    insertRow(rowCount());
    setData(index(rowCount()-1,0),rowCount());
    setData(index(rowCount()-1,1), y);
    setData(index(rowCount()-1,2), NULL);
    QSqlQuery query;
    query.prepare("insert into names values (?, ?, ?)");
    QVariant new_id = rowCount();
    query.addBindValue(new_id);
    QVariant new_name = y;
    query.addBindValue(new_name);
    QVariant new_id_parent = NULL;
    query.addBindValue(new_id_parent);
    query.exec();
}

void SQLTreeModel::updateTreeChild(QString y, QModelIndex sqlindex)
{
    QString str = data(index(sqlindex.row(),0,sqlindex.parent())).toString()+"-"+QString::number(rowCount(sqlindex));
    itemFromIndex(sqlindex)->setChild(rowCount(sqlindex),0,new QStandardItem(str));
    itemFromIndex(sqlindex)->setChild(rowCount(sqlindex)-1,1,new QStandardItem(y));
    itemFromIndex(sqlindex)->setChild(rowCount(sqlindex)-1,2,new QStandardItem(data(index(sqlindex.row(),0,sqlindex.parent())).toString()));
    QSqlQuery query;
    query.prepare("insert into names values (?, ?, ?)");
    QVariant new_id = str;
    query.addBindValue(new_id);
    QVariant new_name = y;
    query.addBindValue(new_name);
    QVariant new_id_parent = data(index(sqlindex.row(),0,sqlindex.parent())).toString();
    query.addBindValue(new_id_parent);
    query.exec();
}
```

Рис. 32. Обращение к БД и внесение новых значений в таблицу

Вывод: разработка с использованием СУБД обладает множеством преимуществ, среди которых важнейшее — возможность сохранять, читать и изменять данные программы как локально, так и удаленно.

4. УКАЗАНИЯ ПО ОФОРМЛЕНИЮ ОТЧЕТА

Отчет должен содержать название и цель работы, краткие теоретические сведения, а также код с комментариями.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Саммерфилд М. Qt. Профессиональное программирование. Разработка кроссплатформенных приложений на C++ / М. Саммерфилд – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 560 с.
2. Шлее М. Qt 5.10. Профессиональное программирование на C++ / М. Шлее – СПб: БХВ-Петербург, 2018. – 1072 с.
3. Бланшет Ж. QT 4: программирование GUI на C++ / Ж. Бланшет – М.: КУДИЦ-ПРЕСС, 2007. – 546 с.

ОГЛАВЛЕНИЕ

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторной работы №6
для студентов специальности 11.05.01
«Радиоэлектронные системы и комплексы»
очной формы обучения

Составитель
Сукачев Александр Игоревич

Издается в авторской редакции

Подписано к изданию 18.03.2024.
Уч.-изд. л. 1,7.

ФГБОУ ВО «Воронежский государственный технический университет»
394006 Воронеж, ул. 20-летия Октября, 84