Министерство образования и науки Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

РАЗРАБОТКА ПРИЛОЖЕНИЙ НА C# С ИСПОЛЬЗОВАНИЕМ СУБД PostgreSQL

Утверждено Редакционно-издательским советом университета
в качестве учебного пособия

НОВОСИБИРСК
2015

УДК 004.65:004.43(075.8)
Р 177

Коллектив авторов:

И.А. Васюткина, *Г.В. Трошина*,
М.И. Бычков, *С.А. Менжулин*

Рецензенты:

д-р техн. наук, профессор *В.И. Гужов*
канд. техн. наук, доцент *С.П. Ильиных*

Работа подготовлена на кафедре вычислительной техники
для студентов, обучающихся по направлениям 09.03.01 и 09.03.04

Р 177 **Разработка приложений на C# с использованием СУБД PostgreSQL**: учебное пособие / И.А. Васюткина, Г.В. Трошина, М.И. Бычков, С.А. Менжулин. – Новосибирск: Изд-во НГТУ, 2015. – 143 с.

ISBN 978-5-7782-2699-9

Представлены основные приемы по использованию языка C# и баз данных PostgreSQL при создании информационных систем различного назначения. Учебное пособие предназначено для студентов, обучающихся по направлениям 09.03.01 – «Информатика и вычислительная техника», 09.03.04 – «Программная инженерия».

УДК 004.65:004.43(075.8)

ISBN 978-5-7782-2699-9

ПРЕДИСЛОВИЕ

Настоящее пособие содержит изложение основных вопросов по использованию языка C# и СУБД PostgreSQL при создании информационных систем (приложений) различного назначения.

Пособие рассчитано на студентов всех форм обучения направлений «Информатика и вычислительная техника» и «Программная инженерия», знакомых с языком запросов SQL и основами языка C#.

Цель написания данного учебного пособия – дать единую теоретическую базу, необходимую для выполнения курсовых, контрольных и лабораторных работ по курсам «Информационные системы», «Базы данных», «Безопасность баз данных», «Технология программирования».

Предлагаемое учебное пособие интегрирует и логически увязывает теоретический материал нескольких курсов и позволяет получить необходимый опыт практической разработки приложений.

Каждый из четырех разделов учебного пособия включает в себя примеры, упражнения, контрольные вопросы, необходимые для более глубокого понимания излагаемого материала и получения практических навыков его использования.

Основное внимание уделено вопросам построения баз данных, разработки графического интерфейса пользователя (GUI) в Microsoft Visual Studio.Net Windows Forms на языке C# и языковым средствам доступа к данным с использованием СУБД PostgerSQL, защите баз данных.

Учебное пособие может быть использовано при выполнении курсовых, контрольных и лабораторных работ по всем указанным дисциплинам.

1. ЭЛЕМЕНТЫ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

1.1. Особенности формирования пользовательского интерфейса

Пользовательский интерфейс – это средство взаимодействия пользователя с приложением. От того, как он разработан, будет зависеть время, которое требуется для его освоения, и эффективность работы с приложением. Правильно разработанный интерфейс должен учитывать ряд основных моментов:

1) учет потребностей пользователей приложения;

2) учет логики работы приложения и последовательность выполняемых действий;

3) оптимизация размещения элементов управления;

4) простота и привлекательность внешнего вида.

Требования, которые предъявляются к интерфейсам программных продуктов, разнообразны и зависят от конкретных задач. Однако в целом можно выявить много общих требований, которые необходимо определить еще до начала разработки интерфейса пользователя. Очень часто это игнорируют, и в результате интерфейсы строятся без учета закономерностей мышления и поведения человека.

Создание хороших интерфейсов требует от разработчика объемной и напряженной работы. При этом нужно учитывать и уметь применять как уже устоявшиеся подходы к разработке интерфейсов, так и новые, которые могут сделать интерфейс более удобным и практичным.

Интерфейс должен быть построен так, чтобы минимизировать как выполнение рутинных часто повторяющихся операций, так и время, необходимое для его освоения. Основное внимание следует уделять продуктивности интерфейса.

Интерфейсом желательно заниматься на начальных стадиях разработки приложения. На более поздних стадиях возможности улучшения качества взаимодействия между пользователем и приложением могут

быть потеряны. Начинать разработку интерфейса можно уже после того, как определены задачи, для решения которых предназначено создаваемое приложение.

На первом этапе следует определить, какие действия должен сделать пользователь, для того чтобы получить тот или иной результат. Необходимо также представить реакцию приложения на эти действия. В ходе разработки приложения возможны корректировки, связанные с изменением как ставящихся задач, так и интерфейса. По этой причине разработка интерфейса представляет собой повторяющийся процесс.

Интерфейс должен быть ориентирован на пользователя, отвечать его нуждам и учитывать его особенности. Интерфейсы не должны быть неоправданно сложны, запутаны, неэкономны и побуждающими к ошибкам. Следует также обращать внимание на необходимость быстрой реакции приложения на действия, осуществляемые пользователем.

Часто действия пользователя при работе с интерфейсом могут иметь несколько интерпретаций. Примером может служить нажатие на клавишу «Return», которая в одном случае означает вставку символа возврата каретки, а в другом – обеспечивает выполнение команды. Состояние интерфейса, при котором интерпретация действий пользователя остается неизменной, называют режимом.

Использование различных режимов (modes) может быть источником ошибок, путаницы и сложности в интерфейсе. Режимы создают проблемы, связанные с тем, когда привычные действия приводят к неожиданным результатам. Для устранения этих проблем целесообразно:

1) обеспечить четкое различие между режимами, например, использовать индикаторы текущего состояния системы;

2) не использовать режимы (не допускать неоднозначной интерпретации действий);

3) не использовать одинаковые команды в разных режимах.

Пользовательские настройки являются одним из примеров использования режимов. Обычно они ориентированы на пользователей с разными навыками и предпочтениями. Они, конечно же, расширяют возможности приложений, но вместе с тем создают и затруднения, связанные с необходимостью их изучения и использования.

Режимы, которые исчезают после однократного применения, называют временными режимами. Эти режимы создают меньше ошибок, чем постоянные режимы, так как они имеют меньше времени на то,

чтобы эти ошибки вызвать. Например, нажатие на любую кнопку приводит к выходу компьютера из режима ожидания, но после этого эти кнопки выполняют действия, для которых они предназначены.

Включение и удерживание элемента управления во время выполнения другого действия называют квазирежимом, или режимом, удерживаемым пользователем. Примером такого режима может быть выбор варианта в выпадающем меню. Использование таких режимов устраняет недостатки постоянных режимов. Однако квазирежимы зачастую требуют запоминания команд. Для сохранения эффективности работы с приложением число квазирежимов должно быть небольшим – примерно от 4 до 7.

Источником ошибок пользователя при работе с интерфейсом могут быть и неточные формулировки подписей элементов управления. Например, вместо формулировки Lock (Блокировка) лучше применить более точную формулировку Locked (Заблокировано).

Для нормальной работы интерфейса должны быть видимы только необходимые его элементы, те, что обеспечивают работу приложения в данный момент времени и определяют способ взаимодействия с приложением. Функция элемента управления и способ ее использования должны выявляться уже по одному его виду. Иногда эту роль могут выполнять пиктограммы. В разработке интерфейса следует оптимизировать качество восприятия, что важно с точки зрения эргономики.

Многие программные продукты предлагают сразу несколько методов достижения того или иного результата. Например, панель инструментов зачастую дублирует действия пунктов главного меню. Кроме того, некоторые команды можно выполнять как с помощью меню, так и с помощью сочетания клавиш.

Такая тактика предъявления действий на выбор в некоторых случаях может быть оправдана, но нужно учитывать, что это приводит к увеличению стоимости и сложности программного продукта и увеличивает время обучения работе с ним.

В тех случаях, когда этого не требуется в силу особых условий, следует придерживаться того, чтобы действие пользователя имело один и только один результат. Такой подход обусловливает только одно соответствие между причиной (командами) и следствием (действиями).

1.2. Формы. Основные сведения

Интерфейс проектов Windows Forms основан на визуальном представлении элементов управления и строится на основе форм Windows. Формы служат для создания окон программы. Windows Forms – это одна из технологий реализации графического интерфейса в платформе .Net. Она включает в себя кмножество типов, которые представлены двумя пространствами имен – System.Windows.Forms и System.Drawing. Первое из них служит для реализации элементов интерфейса, а второе – для рисования в клиентской области окна формы.

Формы являются основным компонентом интерфейса пользователя и представляют собой самые крупные визуальные объекты – контейнеры для размещения в них разнообразных элементов управления приложением.

Для работы с простыми приложениями нужна как минимум одна форма. Для работы с более сложными приложениями их может быть несколько. Одна из форм с именем Form1 создается автоматически при создании проекта Windows Forms. Другие формы могут быть включены в проект приложения с помощью меню или соответствующей кнопки на панели инструментов.

Как и все визуальные компоненты интерфейса, формы имеют свои свойства, методы и события. Управление ими может происходить как на этапе разработки интерфейса, так и в процессе работы приложения.

Для редактирования форм на этапе разработки интерфейса в Visual Studio существует специальный конструктор, в среде которого можно добавлять элементы управления, настраивать их свойства и свойства самих форм.

Добавить форму в проект на этапе разработки интерфейса можно путем выбора пункта меню «Проект –> Добавить форму Windows» (Project -> Add Windows Form…). В открывшемся окне Add New Item нужно выбрать категорию Windows Forms, и в разделе Шаблоны (Templates) выбрать шаблон с именем Windows Form, после чего можно заполнить поле Name, указав новое имя формы. Доступ к окну «Add New Item» можно получить и с помощью кнопки «Add Windows Form», расположенной на панели инструментов.

В результате добавления формы создается класс, производный от класса Form и имеющий по умолчанию имя Form2. Имя этого класса можно увидеть в поле Name панели свойств (Properties) формы. Имя формы в C# используется в пространстве имен для уникальной иден-

тификации класса Form2, и по этой причине не является средством доступа к экземпляру формы по умолчанию.

Получить доступ к свойствам текущей формы можно не только из панели свойств, но и из кода самой формы. Для этого необходимо использовать ключевое слово this. Например, для изменения заголовка формы можно использовать оператор this.Text=”Вторая форма”.

Доступ к свойствам первой (стартовой) формы можно получить из второй формы, создав переменную типа Form1 и присвоив ей значение new Form1().

Пример 1.1:

```
Form1 f1=new Form1();
f1.Text="Первая форма";
f1.Show();
```

Стартовая форма и метод Main

Метод Main() определяет точку входа в программу, и описание этого метода содержится в файле Program.cs. Файл виден в обозревателе решений. Если его содержимое не отображается в редакторе кода приложения, то нужно выделить этот файл в обозревателе решений, и в контекстном меню файла выбрать пункт View Code. После выполнения указанных действий в редакторе кода приложения появится описание метода Main():

```
static void Main()
    {
      Application.EnableVisualStyles();
      Application.SetCompatibleTextRenderingDefault(false);
      Application.Run(new Form1());
    }
```

С выполнения этого метода начинается работа с приложением. Если приложение содержит несколько форм, то одну из них нужно назначить стартовой. При запуске приложения она загружается первой.

Для того чтобы другую форму назначить стартовой, нужно в методе Main() поменять имя класса, которому принадлежит форма. Например, вместо Application.Run(new Form1()); записать Application.Run(new Form2());.

Класс Program, который создается в процессе создания проекта, уже реализует метод Main(), и по умолчанию будет запускать при старте форму по умолчанию (Forml). Стартовым объектом в C# считается любой класс, который реализует метод Main ().

Структура окна формы

Наиболее часто используются формы, окна которых содержат следующие составные части.

1. Строка заголовка окна, в которой находится системное меню и системные кнопки для управления окном (свернуть, раскрыть, закрыть). Системное меню располагается в левой части строки, а системные кнопки – в правой.

2. Строка меню приложения – для выполнения различных операций, например, с файлами, базами данных и т. д. Строка меню чаще всего располагается вверху окна формы, но может быть расположена и в любой другой части окна.

3. Инструментальная панель, содержащая элементы управления для ускоренного доступа к часто выполняемым операциям. Окно формы может содержать несколько инструментальных панелей, расположенных в различных частях формы. Чаще всего инструментальная панель располагается сразу за строкой меню.

4. Рабочая (клиентская) область, используемая пользователем, как правило, для отображения и редактирования информации.

5. Статусная панель (строка состояния), отображающая состояние системы, например, установленный режим. В эту панель можно выводить информационные сообщения о ходе работы приложения.

Кроме перечисленных элементов формы могут содержать боковые стационарные и выдвижные панели, окна контекстных меню, окна подсказок и т. д.

Виды окон

Все разнообразие окон можно представить тремя группами:

1) основные окна, представляющие все приложение и инициирующие создание других окон;

2) диалоговые окна, используемые для получения информации и запуска на выполнение вспомогательных задач приложения;

3) элементы управления (controls), представляющие собой дочерние окна, которые используются для выполнения команд пользователя или для работы с информацией.

Некоторые свойства и события форм

Базовым классом для всех визуальных элементов управления, включая формы, является класс Control. Свойства, методы и события этого класса наследуются всеми его потомками. Некоторые свойства и события форм приведены в табл. 1.1 и 1.2 соответственно.

Таблица 1.1

Свойства класса Form

Свойство	Описание
AcceptButton	Определяет кнопку, запуск действия которой будет продублирован нажатием на клавишу «Enter»
CancelButton	Определяет кнопку, запуск действия которой будет продублирован нажатием на клавишу «Esc»
FormBorderStyle	Свойство, определяющее тип границы окна формы (находится в категории Font). По умолчанию это свойство имеет значение Sizable. Для диалоговых окон применяется значение FixedDialog
Text	Строка, определяющая заголовок формы
MainMenuStrip	Ссылка на компонент, формирующий основное меню формы
DialogResult	Свойство, определяющее способ закрытия «модального» диалогового окна. Доступно только программным способом. Значения, которые может принимать свойство, ограничены определенным набором. Наиболее часто используемые значения: None, OK, Cancel
MinimizeBox	Логическое свойство, определяющее необходимость размещения кнопки минимизации у формы. Как правило, у диалоговых окон эта кнопка отсутствует
MaximizeBox	Логическое свойство, определяющее необходимость размещения кнопки максимизации у формы. Как правило, у диалоговых окон эта кнопка отсутствует
ShowInTaskbar	Логическое свойство, определяющее необходимость отображения формы на панели задач. Как правило, диалоговые окна на панели задач не отображаются

Методы отображения форм

Формы могут отображаться в двух режимах – модальном, и не модальном. В первом из них вызываемые формы до завершения работы с

ними, блокируют работу в других формах. Второй режим обеспечивает параллельную работу с другими окнами. Методы отображения форм представлены в табл. 1.3.

Таблица 1.2

События, возникающие при создании и уничтожении форм

Событие	Описание
Activated	Возникает при каждом получении формой фокуса ввода
Deactivated	Возникает, когда форма перестает быть активной (при переходе пользователя к работе с другим приложением)
Closing	Возникает в процессе закрытия формы и предоставляет возможность управлять процессом закрытия приложения
Closed	Возникает после закрытия формы, позволяя освободить все выделенные ресурсы (закрыть файлы, закрыть соединение с базой данных)
Load	Возникает после инициализации формы, но до её отображения на экране. Полезно для выполнения действий, подготавливающих форму к работе

Таблица 1.3

Методы отображения класса Form

Метод	Описание
Show	Запускает отображение обычного окна (простое отображение формы). При этом может осуществляться одновременно работа как в основной форме, так и в вызванной
ShowDialog	Запускает отображение модального окна (отображение формы в виде диалогового окна – в «модальном» режиме). При таком отображении происходит блокирование всех остальных форм приложения (в том числе и основной) до тех пор, пока работа с диалоговым окном не будет завершена. Результат завершения работы диалогового окна класса DialogResult возвращается как значение функции ShowDialog

Диалоговые окна отображаются в модальном режиме и, как правило, имеют две кнопки «OK» и «Cancel». Для первой кнопки свойство DialogResult должно быть установлено в значение «OK», а для второй это же свойство должно принимать значение «Cancel». Первая кнопка используется для подтверждения ввода данных в диалоговом окне, а вторая (Cancel) – для отказа от ввода данных в этом окне.

Модальные и немодальные формы
Отличия и пример создания

Модальные окна – это окна, которые в открытом состоянии блокируют работу с другими окнами. Переключиться на работу с другим окном можно только после закрытия модального окна. Форма со свойством перечисления FormBorderStyle, для которого установлено значение FixedDialog, относится к числу диалоговых окон.

Основное назначение модальных окон – поддержание диалога с пользователем. До завершения диалога модальную форму покинуть нельзя. Часто использование этого типа окон определяется необходимостью подтверждения каких-либо действий путем нажатия определенной кнопки. Широко используются кнопки с надписями «OK», «Cancel», «Yes», «No». Для открытия модальных окон используется метод ShowDialog.

Немодальные (обычные) окна равноправны со всеми другими окнами приложения. Они дают возможность параллельной работы в разных окнах. Для открытия немодальных окон используется метод Show(). Если переменная f ссылается на объект класса, производного от класса Form, то вызов метода f.Show() позволяет переключиться на этот объект. Метод же ShowDialog позволяет сделать это переключение только после закрытия модальной формы.

Другим отличием методов Show и ShowDialog является то, что метод Show при завершении не возвращает никакого значения, а метод ShowDialog возвращает одно из значений перечисления DialogResult, являющееся идентификатором, поясняющим причину закрытия диалога.

Следует заметить, что идентификаторы создаются автоматически дизайнером форм при добавлении в форму элементов управления и компонентов.

Перечисления могут содержать следующие причины закрытия диалога:

1) OK – работа с формой завершилась успешно (пользователь выполнил требуемую задачу);

2) Cancel – работа с формой завершилась не успешно (пользователь не выполнил требуемую задачу);

3) Abort – прервать;

4) Retry – повторить;

5) Ignore – пропустить;

6) Yes – пользователь ответил утвердительно на заданный вопрос;

7) No – пользователь ответил отрицательно на заданный вопрос.

При закрытии обычного окна объект окна уничтожается, и по этой причине повторно его вызвать с помощью метода Show() нельзя. При закрытии модального окна, созданного с использованием метода ShowDialog, экземпляр класса окна не уничтожается и поэтому не нужно перед каждым новым вызовом создавать экземпляр класса. Он может быть создан всего лишь раз, например, в методе Main() модуля Program.cs.

Еще одно отличие модального окна от обычного состоит в том, что свойства модального окна можно менять только в самом классе этого окна, в то время как свойства обычных окон можно менять из классов других окон.

Нужно иметь в виду, что любую обычную форму можно запустить как модальное окно, если вместо метода Show() использовать метод ShowDialog().

Пример создания и отображения обычного окна на основе класса Form2, производного от класса Form:

```
Form2 f2 = new Form2();
f2.Show();
```

Пример создания и отображения модального окна на основе класса Form2, производного от класса Form:

```
Form2 f2 = new Form2();
f2. ShowDialog();
```

Настройка модального диалогового окна

Пример создания и отображения модального окна на основе класса Form2, производного от класса Form

Используя кнопку Add New Item, добавим в проект приложения дополнительную форму класса Form2, производного от класса Form. В обозревателе решений увидим новый компонент Form2.cs. Выбирая в контекстном меню этого компонента пункт View Designer, отобразим его содержимое.

В конструкторе формы установим следующие значения свойств:

1) FormBorderStyle = FormBorderStyle.FixedDialog; //не изменять размер
2) ControlBox = false; //убрать из заголовка меню управления и кнопки
3) ShowInTaskbar = false; //не показывать в панели задач

Добавим на форму элемент управления Label и установим его значение Text в виде строки «Выберите одну из альтернатив: ».

Добавим на форму три управляющих элемента Button с надписями «Да», «Нет», «Отменить». В обработчиках событий Click этих элементов присвоим свойству DialogResult значение, соответствующее назначению кнопки, например, DialogResult = DialogResult.OK;.

На форму класса Form1 добавим элемент Label, и изменим его значение на пустую строку. Установим заголовок формы вида «Стартовое окно». Добавим на форму управляющий элемент – Button, в обработчике события Click которого пропишем код:

```
Form2 f2 = new Form2();
f2.Text = "Диалоговое окно"; // Заголовок окна
f2.ShowDialog(); // Открыть окно в режиме диалога
string str = "Вы нажали кнопку " + f2.DialogResult;
if (Convert.ToString(f2.DialogResult) == "OK") this.label1.Text = str;
if (Convert.ToString(f2.DialogResult) == "Cancel") this.label1.Text = str;
if (Convert.ToString(f2.DialogResult) == "Abort") this.label1.Text = str;
```

После запуска данного приложения, на экране появится стартовая форма вида, изображенного на рис. 1.1.

Нажатие на кнопку стартовой формы позволяет отобразить диалоговое окно вида, показанного на рис. 1.2.

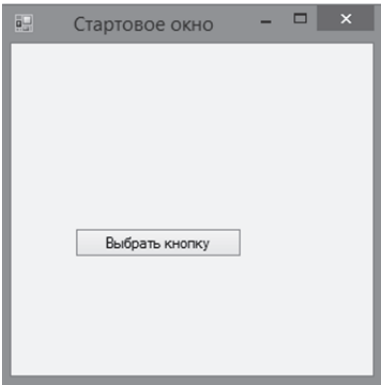


Рис. 1.1. Внешний вид окна стартовой формы

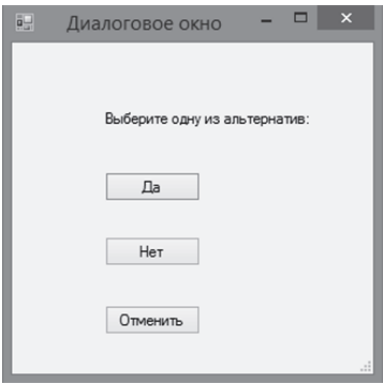


Рис. 1.2. Внешний вид диалогового окна

Если в панели свойств формы придадим свойству AcceptButton значение button1 (кнопка с надписью «Да»), а свойству CancelButton значение button3 (кнопка с надписью «Отменить»), то теперь нажатие на кнопку «Да» будет продублировано нажатием на кнопку «Enter», а нажатие на кнопку «Отменить» будет продублировано нажатием на кнопку «Esc».

Таким образом, в обработчиках событий командных кнопок свойству DialogResult формы диалогового окна присваивается одно из значений, содержащихся в перечислении DialogResult. При наступлении события нажатия на кнопку диалоговое окно исчезает с экрана, а метод ShowDialog, с помощью которого окно диалоговой формы отображается на экране, возвращает значение свойства DialogResult. Теперь диалоговую форму можно считать настроенной.

Особенностью диалоговых окон является то, что они для пользователя неизменяемы. В частности, размер их недоступен для изменения, в них нет элементов управления ControlBox, MinimizeBox и MaximizeBox. Если окна не имеют заголовков, то изменить их местоположение с помощью захвата мышью нельзя.

Диалоговые окна подразделяются на модальные и немодальные. Первые из них используются для выбора заранее определенных альтернатив, и до выбора альтернативы они блокируют работу остальных окон приложения. Их можно использовать для вставки, удаления и модификации таблиц, хранящихся в элементах управления DataGridView. Особенно это важно в тех случаях, когда требуется проверка правильности ввода данных.

Немодальные окна являются независимыми элементами интерфейса и могут работать одновременно с породившим их главным окном приложения. Эти окна часто используют для настройки параметров приложения и организации инструментальных панелей. Измененные в них параметры используются без закрытия этих окон.

Использование встроенных диалоговых панелей

Для организации диалога могут быть использованы и встроенные диалоговые панели. Например, в библиотеку Microsoft .NET Framework входит класс MessageBox, на базе которого можно создать соответствующее диалоговое окно. Для этого имеется набор различных вариантов метода MessageBox.Show, определяющих количество и тип кнопок управления выбором, а также задающих внешний вид диалоговых окон.

В структуру наиболее общего варианта метода MessageBox.Show входит текст сообщения, заголовок окна, кнопки выбора альтернатив, значок окна, кнопка по умолчанию и дополнительные параметры.

Так, например, из стартовой формы можно запустить на выполнение диалоговую панель MessageBox. Для этого настроим указанную панель в стартовой форме следующим образом:

```
MessageBoxButtons caps = MessageBoxButtons.YesNoCancel;
DialogResult result = MessageBox.Show("Выберите одну из альтернатив: ", "Диалоговое окно", caps);
string str = "Вы нажали кнопку " + result;
if (Convert.ToString(result) == "Yes") this.label1.Text = str;
if (Convert.ToString(result) == "No") this.label1.Text = str;
if (Convert.ToString(result) == "Cancel") this.label1.Text = str;
```

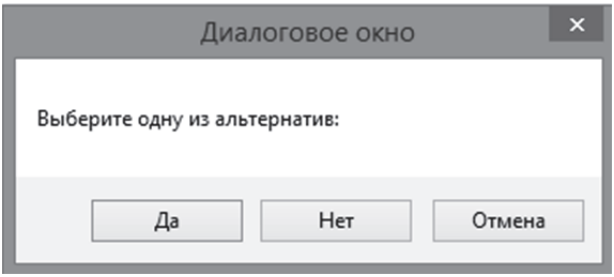


Рис. 1.3. Диалоговое окно на базе класса MessageBox

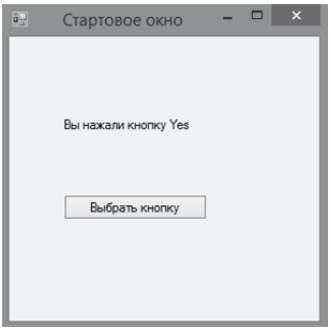


Рис. 1.4. Вид стартовой формы

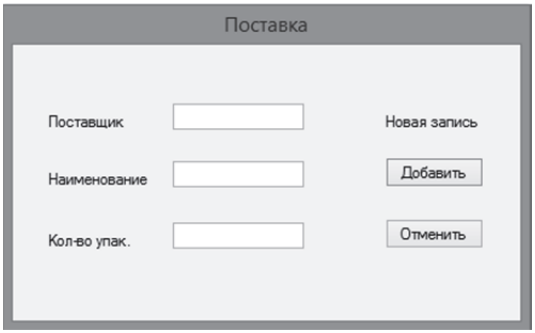


Рис. 1.5. Вид диалоговой формы

После нажатия кнопки «Выбрать кнопку» на экране получим диалоговое окно, как показано на рис. 1.3.

После нажатия на кнопку «Да» на экране получим стартовую форму, изображенную на рис. 1.4.

Упражнение 1.1.

Используя диалоговую форму, реализуйте ввод записей в таблицу данных, представленную с помощью элемента управления DataGridView. После настройки свойств диалоговая форма должна иметь вид, изображенный на рис. 1.5.

Добавление на форму календаря

Для ускорения ввода отдельных дат и их диапазонов в управляющие элементы формы на панели Toolbox имеется элемент управления – календарь. Календарь можно отобразить с помощью элемента управления MonthCalendar или DateTimePicker. Использование этих элементов позволяет не только упростить выбор требуемой даты, но и обеспечить гарантию того, что дата будет иметь правильный формат.

С помощью MonthCalendar можно отобразить календарь как для одного, так и для нескольких месяцев. Этот элемент позволяет выбирать либо отдельную дату, либо диапазон дат. Свойство MaxSelectionCount отвечает за установку максимального допустимого диапазона дат. Для выбора отдельной даты это свойство должно принять значение, равное единице.

Элемент управления DateTimePicker имеет два режима работы. В первом из них он позволяет выбрать только одну дату, во втором – отобразить время вместо даты. Визуальное представление этого элемента похоже на текстовое поле с раскрывающимся списком в виде стрелки. Нажатие на стрелку приводит к отображению календаря.

Извлечение даты из календаря зависит от того, какой из элементов управления установлен – MonthCalendar или DateTimePicker. Для элемента управления MonthCalendar для цели извлечения даты используется свойство Start. Для элемента управления DateTimePicker таким свойством является свойство Value.

Упражнение 1.2

1. В конструкторе формы добавьте на форму элементы управления DateTimePicker и TextBox, с именами, установленными по умолчанию.

2. Дважды щелкните элемент управления DateTimePicker, чтобы открыть обработчик событий по умолчанию в редакторе кода.

3. В открывшемся обработчике событий DateTimePicker_ValueChanged добавьте следующий код для добавления элементов в список:

```
this.textBox1.Text = Convert.ToString(this.dateTimePicker1.Value);
```

Теперь каждый выбор даты в календаре будет приводить к её отображению в элементе управления TextBox.

Перетащите этот элемент на поверхность разрабатываемой формы и снабдите его надписью «Дата поставки».

Свойство MaxSelectionCount отвечает за установку максимального допустимого диапазона дат. Для выбора отдельной даты это свойство должно принять значение, равное единице.

1.3. Настройка рабочего пространства интерфейса

Организовать рабочее пространство интерфейса пользователя можно как путем простого расположения элементов на форме и использования многооконной структуры, так и с применением принципа наследования, когда одни объекты допускают вложения в них других объектов. Такой принцип организации известен под названием объектная модель.

Контейнеры – это вместилища других объектов. Использование контейнерного дизайна пользовательского интерфейса приложения позволяет обособленно работать как с отельными элементами, так и выполнять групповые операции на всей совокупностью объектов, входящих в состав контейнера. К таким операциям относятся операции блокирования и разблокирования элементов, их отображения и скрытия, изменения порядка расположения и т. д.

Двухпанельный элемент SplitContainer

Визуально рабочее пространство контейнера SplitContainer представлено двумя панелями, отделенными друг от друга специальной полосой-разделителем. Ориентация разделителя может быть как вертикальной, так и горизонтальной. При вертикальной ориентации разделителя рабочее пространство контейнера делится на левую и правую панель, при горизонтальной ориентации – на верхнюю и нижнюю.

Каждая из панелей контейнера допускает произвольное количество вложений других контейнеров или одиночных элементов управления. Таким образом, все рабочее пространство может принимать сложную структуру, что часто бывает удобно при работе пользователя с прило-

жением. Так, например, при горизонтальной ориентации разделителя и вложении такого же контейнера в нижнюю панель можно получить структуру рабочего пространства, показанную на рис. 1.6.

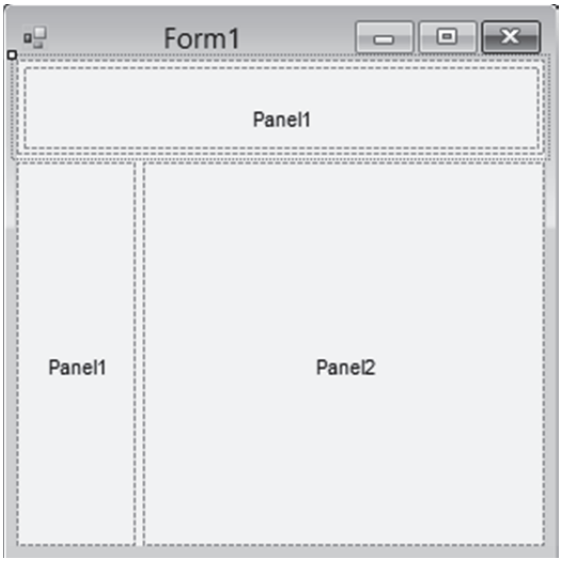


Рис. 1.6. Структура рабочего пространства формы, созданная вложением контейнера в нижнюю панель родительского контейнера

Часто в одной из панелей контейнера располагают некоторые объекты, а в другой отображают связанную с ними информацию. Например, при работе приложения с базой данных в одной из панелей может быть отображен список имен таблиц, а другая панель может служить для отображения содержимого этих таблиц.

Вид ориентации разделителя определяется значением свойства **Orientation**. Это свойство имеет два значения – **Horizontal** и **Vertical**, соответствующие горизонтальному и вертикальному расположению разделителя.

Открепление и удаление вложенных контейнеров

Открепить контейнер означает разорвать связь его с родительским объектом – соответствующей панелью или формой. Для этого прежде всего нужно выделить соответствующий контейнер, щелкнув правой

кнопкой мыши в его рабочей области и выбрав в контекстном меню соответствующий пункт, начинающийся словами **Select SplitContainer**.

Таблица 1.4

Широко применяемые свойства объекта SplitContainer

Имя свойства	Значения	Описание
FixedPanel	Panel1 Panel2 None	Определяет панель с фиксированными размерами при изменении размера SplitContainer
IsSplitterFixed	True False	Определяет возможность (False) или невозможность (True) перемещения разделителя при помощи клавиатуры или мыши
Orientation	Horizontal Vertical	Определяет ориентацию разделителя по вертикали или по горизонтали
SplitterDistance	Количество пикселей	Определяет расстояние (количество пикселей) от левой или верхней границы до разделителя
SplitterIncrement	Количество пикселей	Определяет минимальное расстояние (минимальное количество пикселей), на которое разделитель может быть перемещен пользователем
SplitterWidth	Количество пикселей	Определяет толщину разделителя, выраженную количеством точек
BorderStyle	None FixedSingle Fixed3D	Задает вид разделителя
SplitterWidth	Количество пикселей	Задает толщину разделителя в пикселях
Visible	True False	Управляет отображением (скрытием) контейнера в родительском объекте (чаще на форме)

После выделения в правом верхнем углу рабочей области контейнера появится значок, напоминающий закрашенную стрелку, направленную вправо. Нажатие левой кнопкой мыши на эту стрелку вызывает контекстное меню **SplitContainer Tasks** (Задачи). Первый пункт этого меню позволяет определить ориентацию разделителя в контейнере, а второй определяет связь выделенного контейнера с формой или панелью родительского контейнера.

Для удаления выделенного контейнера нужно выбрать пункт **Undock in Parent Container** (открепить от родительского контейнера), и в контекстном меню появившегося значка с левой верхней стороны контейнера выбрать пункт **Delete**.

1.4. Меню

Существует два способа создания меню. Первый из них осуществляется путем визуального переноса элемента меню с панели инструментов на форму и дальнейшего использования интерактивного редактора (конструктора) меню. Второй – предусматривает программное создание меню.

Первый способ создания меню не требует знания внутреннего устройства меню и реализуется очень просто – при разработке дизайна формы в результате диалога с редактором меню.

Второй способ является более гибким и позволяет динамически создавать меню в процессе выполнения приложения. В дальнейшем будет рассматриваться второй способ создания меню.

Создание меню с помощью конструктора

Добавление меню на форму осуществляется с помощью перетаскивания элемента MenuStrip из панели элементов Toolbox на форму. В верхней части формы появится панель меню с полем для ввода имени первого пункта меню.

При вводе имени добавятся два дополнительных элемента: один снизу – для ввода имени пункта меню второго уровня и второй справа – для ввода имени следующего пункта меню первого уровня.

По окончании ввода всех пунктов меню для них нужно создать обработчики событий. Заготовку функции обработчика событий можно получить путем двойного щелчка левой клавишей мыши по пункту меню. Далее в заготовке необходимо определить реакцию приложения на выбор данного пункта меню.

Программное создание меню. Обработка событий

1. В классе Form1 определим метод:

```
private void ChildClick(object sender, EventArgs e)
{ MessageBox.Show("Работает пункт подменю"); }
```

2. В методе загрузки формы выделим память под все меню:

```
MenuStrip m = new MenuStrip();
```

3. //Пункты меню первого уровня, создаваемые при загрузке формы

```
string[] rowmenu = new string[] { "File", "Edit", "View", "Project" };
```

```
4. // Определим цикл:
foreach (string dataRow in rowmenu)
{
 //Выделение памяти под пункты меню первого уровня
 ToolStripMenuItem m1 = new ToolStripMenuItem(dataRow);
 //Выделение памяти под пункты меню второго уровня
 foreach (string dataRow1 in rowmenu)
 {
 ToolStripMenuItem m2 = new ToolStripMenuItem
  (dataRow1.ToString(),
 null, new EventHandler(ChildClick));
 //Добавление пунктов меню второго уровня
 m1.DropDownItems.Add(m2);
 }
 //Добавим в меню m пункты меню первого уровня
 m.Items.Add(m1);
}
5. //Добавление меню в коллекцию Controls
this.Controls.Add(m);
```

После запуска приложения получим форму, изображенную на рис. 1.7.

Выбор пункта подменю будет приводить к появлению сообщения, изображенного на рис. 1.8.

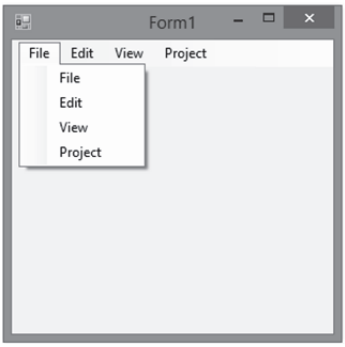


Рис. 1.7. Вид формы с меню

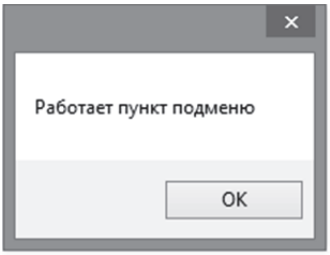


Рис. 1.8. Окно сообщения

Контекстное меню

Контекстное меню – это выпадающее меню, которое появляется при нажатии правой кнопкой мыши на объект, отображаемый на форме. Для создания этого меню используется класс ContextMenuStrip, который является контейнером для объектов ToolStripMenuItem. Объект ContextMenuStrip имеет графическое отображение в области компонентов формы.

В процессе создания контекстного меню объекты класса ToolStripMenuItem добавляются в коллекцию Items и для каждого из них создается обработчик события Click, обеспечивающий выполнение определенной задачи.

Связь контекстного меню с элементом управления, для которого оно предназначено, обеспечивается с помощью свойства ContextMenuStrip этого элемента. Для обеспечения этой связи нужно, чтобы свойство приняло значение имени контекстного меню.

После установления связи контекстное меню готово к работе. Выбор пункта этого меню будет приводить к выполнению обработчиками своих задач.

1.5. Панель инструментов ToolStrip

Общие сведения

Панели ToolStrip используются для быстрого выполнения команд. Иногда элементы этих панелей дублируют функции меню. При установке этого компонента на форму он по умолчанию располагается в верхней её части.

В режиме редактирования формы эта панель в левой своей части содержит кнопку со стрелкой вниз. Нажатие на стрелку приводит к появлению выпадающего меню.

Каждый пункт выпадающего меню обеспечивает выбор типа компонента, создаваемого на поверхности панели.

Можно выбрать следующие типы компонентов:

1) Button – кнопка;

2) Label – метка;

3) splitButton – кнопка, которая имеет дополнительную кнопку для вызова всплывающего меню. Основная и дополнительная кнопки работают независимо;

4) DropDownButton – кнопка, которая вызывает всплывающее меню. Всплывающее меню появится именно по нажатию кнопки без нажатия каких-либо дополнительных кнопок;

5) Separator – разделитель, который позволяет логически разделять группы кнопок;

6) comboBox – выпадающий список;

7) TextBox – текстовое поле ввода;

8) ProgressBar – индикатор процесса.

Нажатие на саму кнопку со стрелкой в режиме редактирования формы приводит к созданию на поверхности панели кнопки класса ToolstripButton, которая похожа на кнопку Button, но в отличие от нее имеет ряд дополнительных свойств: Checked, CheckOnClick и Checkstate.

Если установить свойство CheckOnClick = true, то при нажатии кнопка перейдет в состояние залипания и её состояние checked примет значение true, при повторном нажатии кнопка отпускается и её свойство состояние checked примет значение false. При CheckOnClick = false кнопка нажимается и тут же отпускается.

Компоненты splitButton и DropDownButton похожи по свойствам на классическую кнопку, но у них есть дополнительная возможность в виде жестко привязанного меню, и при выделении кнопки в редакторе появляется редактор для этого меню.

Компоненты ComboBox, TextBox и ProgressBar идентичны стандартным компонентам.

Напрямую в коллекцию Items объектов класса ToolStrip можно включать только элементы управления, производные от класса ToolStripItem. Другие элементы, например, такие как календарь DateTimePicker, можно включить в инструментальную полосу, используя класс ToolStripControlHost.

Пример программного создания инструментальной панели

В данном примере создается инструментальная панель и с помощью класса ToolStripControlHost в нее добавляется календарь DateTimePicker. Выбор даты в календаре приводит к ее отображению в метке label1, расположенной на форме ниже инструментальной панели (рис. 1.9).

```
    private void Form1_Load(object sender, EventArgs e)
{
      ToolStrip toolStrip = new ToolStrip();
      ToolStripLabel toolStripLabel = new ToolStripLabel();
      toolStrip.Items.Add(toolStripLabel);
      Controls.Add(toolStrip);
      ToolStripControlHost toolStripControlHost =
      new ToolStripControlHost(new DateTimePicker());
      ((DateTimePicker)toolStripControlHost.Control).ValueChanged +=
       delegate
    {
     this.label1.Text =
    ((DateTimePicker)toolStripControlHost.Control).Value.Date.ToString();
    };
     toolStrip.Items.Add(toolStripControlHost);
}
```

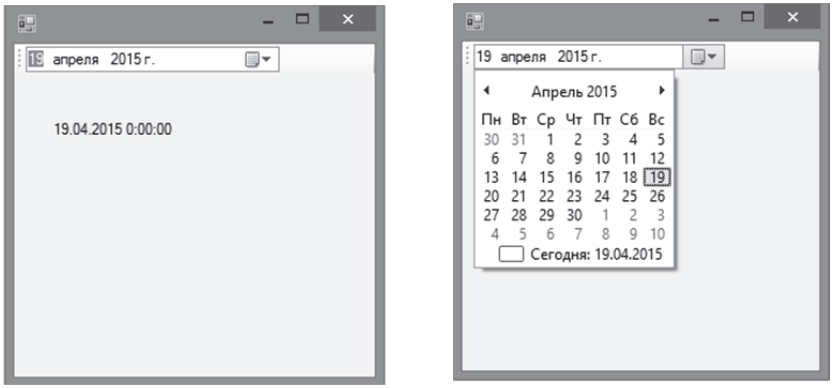


Рис. 1.9. Пример использования календаря из инструментальной панели

1.6. Элементы информирования пользователя

Элемент ToolTip (всплывающая подсказка)

Объекты ToolTip предоставляют пользователю приложения необходимую информацию (подсказку) о назначении требуемого элемента управления или, например, о формате вводимых данных. Использование таких объектов обеспечивает дружественность интерфейса и помогает в работе с приложением.

Информация предоставляется в всплывающем окне при задержке указателя на элементе управления. Для каждого элемента управления должна быть сформирована своя подсказка.

Объект ToolTip может быть получен с использованием панели инструментов ToolBox или программно с использованием оператора:

ToolTip toolTip1 = new ToolTip();.

При создании объекта ToolTip с помощью панели инструментов он отображается не на форме, а в области компонентов, расположенной под формой.

Пример создания подсказки:

```
private void Form1_Load(object sender, EventArgs e)
{
 toolTip1.SetToolTip(button1, "Элемент управления");
 toolTip1.ToolTipTitle = "Подсказка";
 toolTip1.ToolTipIcon = ToolTipIcon.Warning; //Добавление иконки
 toolTip1.IsBalloon = true; //IsBalloon – скруглить углы и сделать выноску
}
```

В данном случае наведение указателя на командную кнопку button1, расположенную на форме, будет приводить к появлению подсказки.

Строка состояния

Элемент StatusStrip (строка состояния) используется для того, чтобы отображать состояние работы приложения. На поверхности этого элемента можно создавать компоненты, которые и будут выполнять определенную информационную функцию. Например, компонент типа метки может отображать текстовую информацию.

При установке элемента StatusStrip на форму он по умолчанию располагается в нижней её части. В режиме редактирования формы строка состояния в левой своей части содержит кнопку со стрелкой вниз, нажатие на стрелку которой приводит к появлению выпадающего меню.

Каждый пункт выпадающего меню обеспечивает выбор типа компонента, создаваемого на поверхности панели.

Можно выбрать следующие типы компонентов:

1) statusLabel – метка;

2) ProgressBar – индикатор процесса;

3) SpiitButton – кнопка, которая имеет дополнительную кнопку для вызова всплывающего меню. Основная и дополнительная кнопки работают независимо;

4) DropDownButton – кнопка, которая вызывает всплывающее меню. Всплывающее меню появится именно при нажатии на кнопку без нажатия на какие-либо дополнительные кнопки.

В режиме редактирования формы нажатие на саму кнопку со стрелкой приводит к появлению на панели метки класса ToolstripstatusLabel, похожей на метку Label и также предназначенной для отображения информации.

1.7. Элементы отображения данных

Элемент управления DataGridView

Элемент управления DataGridView служит для организации представления данных в табличном виде. Его успешно можно использовать для просмотра и редактирования таблиц баз данных, содержащихся в наборах DataSet. С помощью этого элемента можно в разном виде отображать одни и те же данные, т. е. создавать их различные представления.

В качестве источника данных для элемента DataGridView может быть объект одного из следующих типов:

1) DataTable

2) DataView

3) DataSet

4) DataViewManager

Объекты класса DataTable имеет свойство DefaultView, которое и определяет их в качестве источника данных по умолчанию для отображения в DataGridView.

Пример 1.2.

dataSet.Tables["Students"].DefaultView;

DataGridView позволяет в своих ячейках кроме обычных данных отображать командные кнопки, комбинированные списки и изображения. Данные, отображаемые в DataGridView, хранятся локально и их изменение не приводит к изменению данных в источнике. С помощью DataGridView можно в табличном виде представить содержимое двухмерного массива данных.

Задать количество строк и столбцов в объекте DataGridView можно таким:

dataGridView1.RowCount=5;

dataGridView1.ColumnCount=5;

Добавить объект DataGridView в коллекцию Controls можно таким образом:

this.Controls.Add(DataGridView1);

Заполнить поля строки можно так:

this.DataGridView1.Rows.Add("one", "two", "three", "four");

Для доступа к ячейке DataGridView требуется указать номер строки и номер столбца, например:

dataGridView1.Rows[3].Cells[5].Value=”7”;

Для обеспечения связи DataGridView с набором DataSet в простейшем случае необходимо только установить значения свойств DataSource и DataMember. Первое из них определяет источник данных – это сам набор DataSet, а второе – элемент этого набора (таблицу).

Второе свойство необходимо в том случае, когда источник данных имеет в своем составе несколько таблиц. Если используется отдельный объект DataTable, не входящий в состав набора DataSet, то свойство DataCource необходимо установить на эту таблицу, а свойство DataMember можно не использовать.

В качестве источника данных для элемента DataGridView могут выступать данные объекта DataView, предназначенного для фильтрации и сортировки данных, содержащихся внутри таблиц, и представления результатов этих операций в виде наборов данных.

При использовании DataView в качестве источника в DataGridView отображаются не все данные таблиц, а только те, которые выбраны в

результате фильтрации. Фильтрация и сортировка с помощью DataView не приводят к изменениям внутри самих таблиц.

Создать объект класса DataView, настроенный на первую таблицу набора dataSet, можно оператором

```
DataView dv=new DataView(dataSet.Tables[0]);
```

Для отображения представления dv в объекте класса DataGridView можно поступить следующим образом:

```
DataGridView dgv = new DataGridView();
dgv.DataSource = dv;
```

В этом случае представление, сформированное в dv, будет содержать все записи таблицы dataTable. Изменить представление можно с помощью свойства RowFilter, присвоив ему значение в виде строки, содержащей критерии отбора данных, например:

```
dv.RowFilter="tirag>50";
```

Здесь отбор данных производится по значению поля tirag, и на экране монитора можно увидеть только те записи таблицы, которые подпадают под указанное ограничение.

Сортировку данных по возрастанию или убыванию можно выполнять в самом объекте класса DataGridView щелчком на заголовке того или иного столбца. Однако сортировать данные в этом элементе можно только по одному из столбцов.

Объект класса DataView обладает в этом случае большими возможностями и позволяет выполнять сортировку данных в формируемом им представлении по множеству столбцов.

Если сортировка производится по одному из столбцов, то этот столбец в объекте класса DataGridView помечается в заголовке изображением стрелки, направленной вверх (сортировка по возрастанию) или вниз (сортировка по убыванию).

При сортировке по нескольким столбцам объект класса DataGridView показывает изображение стрелки только в заголовке столбца, идущего первым в списке перечисленных для сортировки.

В программе установить порядок сортировки можно присвоением свойству Sort соответствующего строкового значения. Например:

```
dv.Sort=”tirag”;
dv.Sort=” tirag DESC, id ASC”;
```

Первый оператор выполняет сортировку по возрастанию поля tirag, а второй выполняет сортировку по возрастанию значения поля id и по

убыванию значения поля tirag, причем изображение стрелки появляется только в заголовке столбца tirag.

Данные, представленные в объектах класса DataGridView, отображаются в текстовом виде, но в источнике данных они могут быть типизированы. Сортировка на типизированных наборах, реализуемая как средствами класса DataGridView, так и средствами класса DataView производится на самих данных, а не на их строковом представлении.

Это обстоятельство позволяет, например, отсортировать строки по реальным датам, которые в строковом представлении могут иметь разный формат.

1.8. Привязка данных к элементам интерфейса

Связь DataGridView с источниками данных

Для связи DataGridView и других визуальных элементов формы с источниками данных используются два компонента – это компонент BindingSource и компонент TableAdapter. Первый из них инкапсулирует источник данных для визуальных элементов формы, а второй обеспечивает связь с источником данных. Помещая эти компоненты на форму, необходимо изменить модификатор доступа на public.

Оба компонента позволяют упростить процесс привязки элементов управления к базовым источникам данных и способствуют уменьшению объема программного кода. Вместе с тем они требуют достаточно сложной настройки и имеют ограниченные возможности управления данными.

Компонент BindingSource обеспечивает возможность универсальной привязки всех визуальных элементов управления формы к источникам данных. Если тип данных, содержащихся в источнике, реализует интерфейс NotifyPropertyChanged, то это позволяет компоненту BindingSource автоматически обнаруживать изменения в источнике данных.

Визуальные элементы управления, связанные с компонентом BindingSource, при изменении значений источника данных автоматически обновляются. При этом интерфейс NotifyPropertyChanged используется для уведомления участников привязки об изменении значения свойства.

Для того чтобы уведомление появилось, связанный тип должен либо реализовать интерфейс NotifyPropertyChanged, либо вызвать собы-

тие PropertyChanged для каждого свойства связанного типа. Связанный таким образом элемент управления DataGridView позволяет отслеживать изменение в источнике данных без повторной привязки.

Вызов метода NotifyPropertyChanged не требует указывать имя свойства в виде строкового аргумента, если используется атрибут CallerMemberName.

Пример связи таблицы в наборе dataSet с элементом dataGridView1:

```
private BindingSource bindingSource = new BindingSource();
bindingSource.DataSource = dataSet.Tables[0];
dataGridView1.DataSource = bindingSource;
```

Пример связи двух текстовых полей формы:

```
textBox2.DataBindings.Add(new Binding("Text", textBox1, "Text"));
```

Все изменения в textBox1 будут автоматически отображаться и в textBox2, но не наоборот. Здесь первое "Text" – это свойство textBox2, а второе – свойство textBox1.

Пример связи элемента таблицы в наборе dataSet с элементом textBox2:

```
dataSet.Clear();
adapter.Fill(dataSet, "knigi");
BindingSource bindingSource = new BindingSource();
textBox2.DataBindings.Add(new Binding("Text", dataSet,
"knigi.name"));
```

Здесь name – это поле (столбец) в таблице knigi в наборе данных dataSet, а Text – это свойство объекта textBox2. В результате в textBox2 появится первое значение поля name.

Компонент BindingSource является связующим звеном между источником данных и элементами управления формы. Он может обеспечить связь содержимого всей таблицы с одним элементом управления DataGridView (сложная привязка). С его помощью можно также связывать столбцы таблиц с более простыми элементами управления, такими как TextBox, Label и другими (простая привязка).

Если связь между данными и элементом управления формы осуществляется с помощью объекта BindingSource, то для навигации по

данным используется элемент управления BindingNavigator в виде панели инструментов для навигации.

Сохранение измененных данных (обновление источника данных) – это процесс их переноса из модели данных приложения (объект DataSet) обратно в источник данных, например, в базу данных PostgreSQL. Этот процесс обратный процессу загрузки данных в промежуточный набор DataSet, но более сложный, чем первый.

Процесс обновления источника данных происходит поэтапно. Сначала обновляется модель данных (набор данных DataSet) путем изменения, добавления или удаления записей, а затем производится сохранение измененного набора в базе данных.

Компонент DataView

Компонент класса DataView позволяет формировать различные (альтернативные) представления данных одной и той же таблицы. Далее эти представления можно отображать в элементе управления DataGridView. Различные представления таблиц формируются путем отбора данных по определенным правилам или критериям. Отбор данных не влияет на источник данных, а только позволяет сформировать требуемое представление данных для дальнейшего его отображения или использования.

Как правило, источниками данных для объектов класса DataView являются таблицы, содержащиеся в наборах данных DataSet, а для объектов DataGridView источником является сам объект класса DataView. Для задания источника данных для элемента DataGridView используется его свойство DataSource. Объект класса DataView обеспечивает фильтрацию строк и не позволяет фильтровать столбцы.

Основными свойствами класса DataView являются RowFilter и Sort. Первое из этих свойств используется для задания условия отбора записей из объекта DataTable, а второе задает правило сортировки записей. Для создания объекта класса DataView на базе существующей таблицы dataTable можно использовать следующий код:

```
DataView dataView=new DataView(dataTable);
```

После создания dataView можно установить настройки, которые будут обеспечивать допустимые действия над данными. Правило сортировки задает свойство Sort, которое предусматривает указания полей, по значениям которых будет производиться сортировка и порядок сортировки, задаваемый значениями ASC – сортировка по возрастанию

и DESC – по убыванию. По умолчанию установлен порядок сортировки ASC.

Пример 1.3. Использование объекта dataView для фильтрации и сортировки.

Исходная таблица dataTable:

	ID	Name	Age
▶	1	Ivanov	21
	2	Sidorov	19
*			

Задание порядка сортировки:

```
DataView dataView = new DataView(dataTable);
dataView.Sort = "ID DESC";
```

Представление таблицы в dataGridView:

	ID	Name	Age
▶	2	Sidorov	19
	1	Ivanov	21
*			

Задание критерия отбора данных:

```
DataView dataView = new DataView(dataTable);
dataView.RowFilter = "Name = 'Ivanov'";
```

Представление таблицы в dataGridView:

	ID	Name	Age
▶	1	Ivanov	21
*			

1.9. Работа с данными

Создание набора данных DataSet

Элементы набора DataSet имеют тип object. По этой причине выполнение действий над ними зачастую требует их преобразования в соответствующий тип. Типизированные объекты класса DataSet явля-

ются альтернативой не типизированных объектов, и в отличие от последних обеспечивают выявление ошибок несоответствия типов на этапе компиляции, а не в процессе выполнения программы. При использовании типизированных DataSet полные имена членов коллекций (например, Tables) допустимо заменять их более короткими именами. Кроме того, в период разработки имена типизированных членов данных отображаются в окнах среды разработки.

Пример создания пустого набора dataSet:

```
DataSet dataSet = new DataSet("Univercity ");
DataTable dataTable = new DataTable("Students");
dataSet.Tables.Add(dataTable);
```

Создание объектов DataTable

Пример создания и заполнения таблицы:

```
DataTable dataTable = new DataTable("Students");
dataTable.Columns.Add(new DataColumn("ID", typeof(int)));
DataRow dataRow1 = dataTable.NewRow();
dataRow1["ID"] = 1;
```

Упражнение 1.3

1. На форме установить объекты DataGridView, ToolStrip и ComboBox.
2. Установить на панель инструментов ToolStrip календарь DateTimePicker.
3. Создать набор данных DataSet, включив в него две таблицы, содержащие поля типа int, string, DateTime.
4. Заполнить поля таблиц данными.
5. В конструкторе формы обеспечить загрузку списка таблиц в выпадающий список ComboBox.
6. Создать объект DataViev и обеспечить его использование при отображении таблиц набора DataSet.
7. Обеспечить создание представлений отсортированных данных и их отображение в элементе управления DataGridView.
8. Используя метод ToShortDateString() с помощью календаря изменить даты в таблицах.

2. ОСНОВЫ РАБОТЫ В СУБД PostgreSQL

2.1. Установка PostgreSQL для Windows

Объектно-реляционная система управления базами данных (СУБД) PostgreSQL является одной из самых популярных СУБД, распространяемых с открытыми исходными текстами. Это означает, что любой пользователь может свободно работать с исходным текстом программы и подвергать его модификациям для своих целей без всяких ограничений. Последняя версия PostgreSQL доступна для скачивания на сайте http://www.postgresql.org. Установочный дистрибутив включает в себя установку PostgreSQL, утилиту pgAdmin, предназначенную для администрирования PostgreSQL, а также менеджер пакетов StackBuilder, который используется для загрузки дополнительных пакетов, расширяющих возможности СУБД PostgreSQL. Для того чтобы скачать дистрибутив, надо перейти на страницу загрузки (рис. 2.1), выбрать подходящую операционную систему (ОС) (Linux, Mac OS X или Windows) и загрузить дистрибутив.

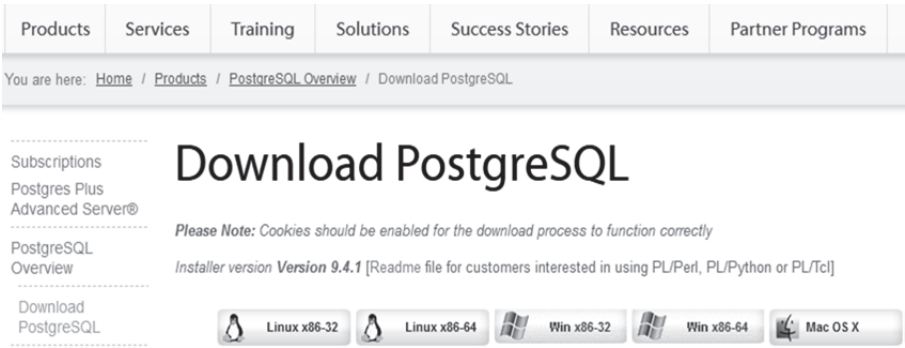


Рис. 2.1. Страница загрузки PostgreSQL

Установка для каждой операционной системы несколько отличается друг от друга. Ниже приведен порядок установки СУБД PostgreSQL

для операционной системы Windows как одной из самых распространенных операционных систем в настоящее время.

1. Для ОС Windows имя установочного файла имеет вид: postgresql-X.X.X-windows.exe, где X.X.X – это версия PostgeSQL. После запуска установочного файла вначале устанавливается Microsoft Visual C++ Redistributable (рис. 2.2), где находятся рабочие модули библиотек Visual C++.

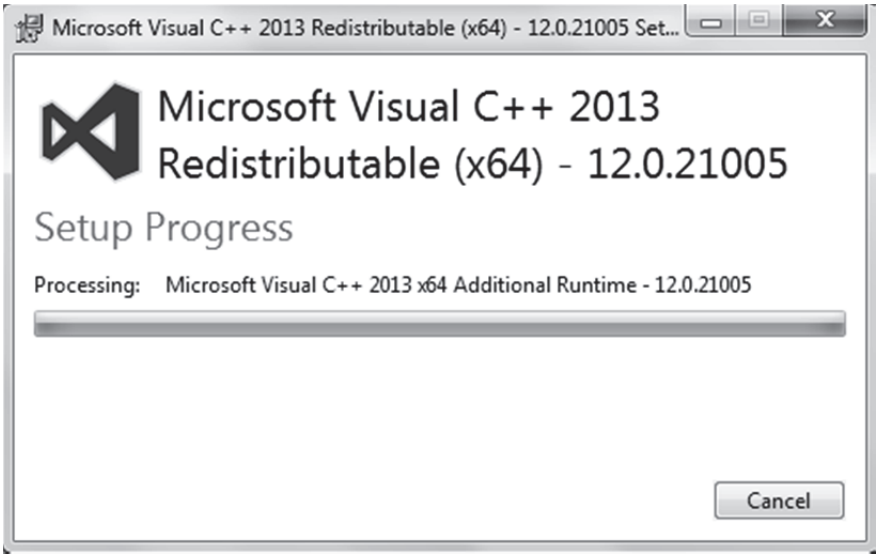


Рис. 2.2. Установка Microsoft Visual C++ Redistributable

2. После установки библиотек Visual C++ появляется окно мастера установки (рис. 2.3).

3. На следующем шаге необходимо выбрать директорию, где будет установлена программа (рис. 2.4).

4. Далее требуется выбрать каталог для хранения пользовательских данных (рис. 2.5).

5. В следующем окне требуется установить и подтвердить пароль, который будет использоваться для учетной записи суперпользователя – «postgres». При этом пользователь может изменить пароль суперпользователя в любое время после установки (рис. 2.6).

6. Далее необходимо указать порт подключения (рис. 2.7). Значение порта по умолчанию: 5432.

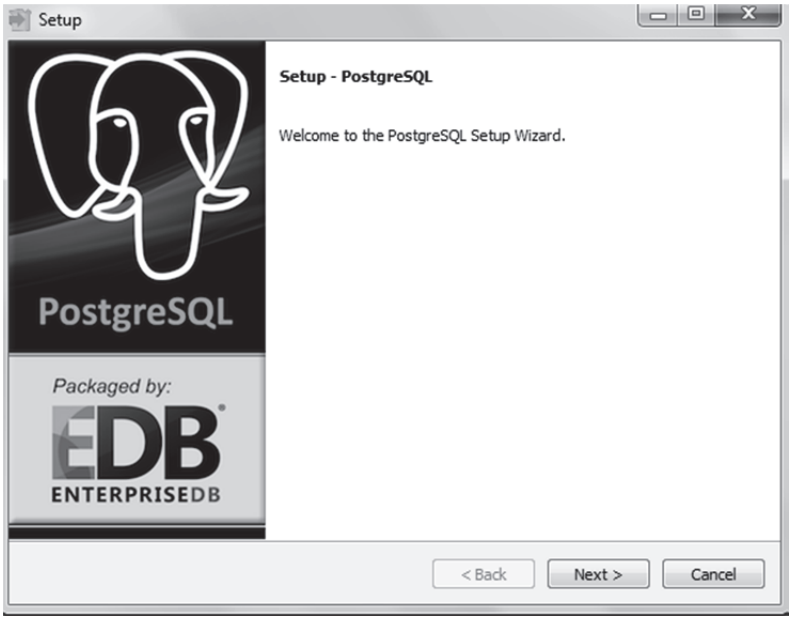


Рис. 2.3. Мастер установки PostgreSQL

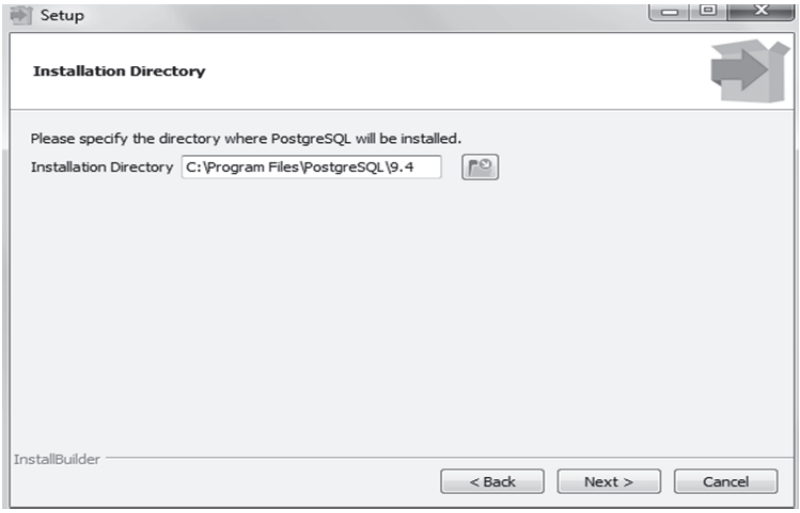


Рис. 2.4. Выбор директории для PostgreSQL

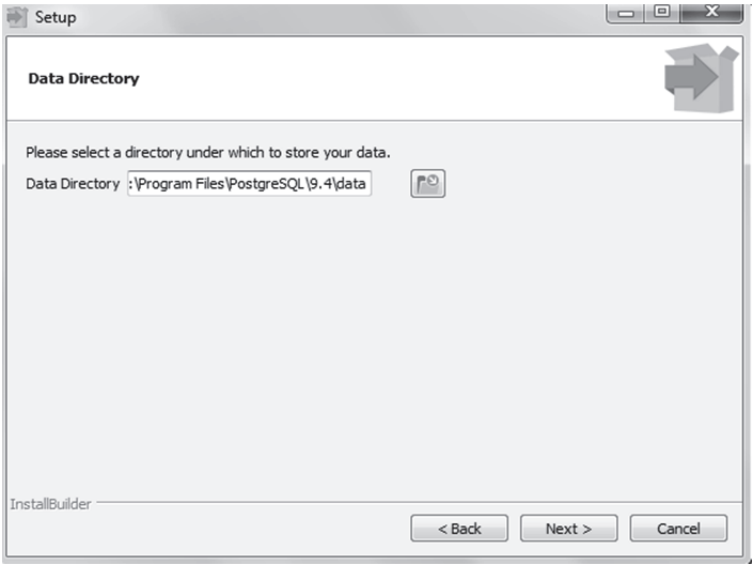


Рис. 2.5. Выбор каталога для пользовательских данных

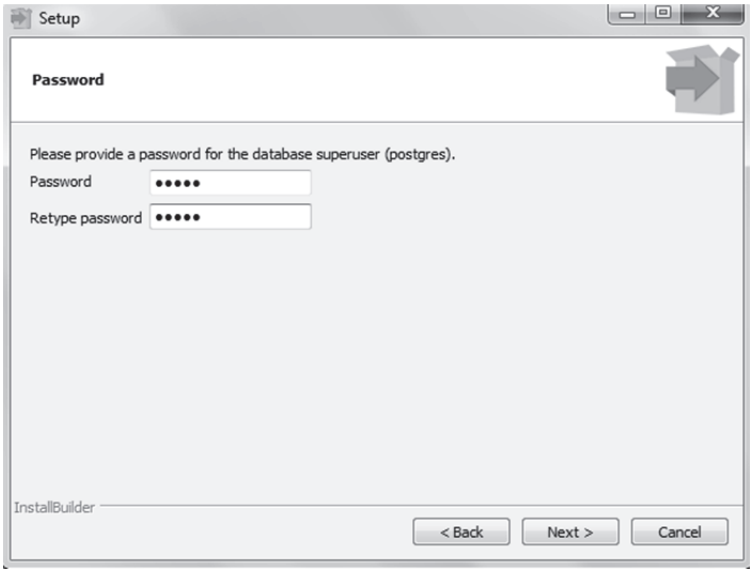


Рис. 2.6. Установка пароля

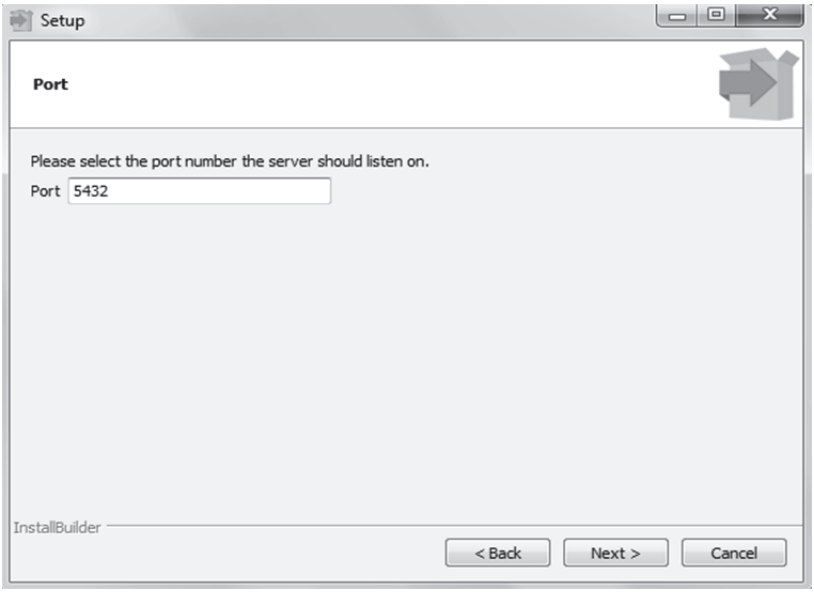


Рис. 2.7. Порт подключения

7. На следующем шаге установщик дает возможность указать локаль, которая будет указана при создании других баз данных по умолчанию. Этот параметр определяет кодировку данных. Параметр [Default locale] позволит установить подходящую локаль из окружающей среды (рис. 2.8).

8. Далее начнется распаковка дистрибутива на компьютер и отображение процесса установки, инициализация и запуск PostgreSQL (рис. 2.9).

9. После завершения установки PostgreSQL появляется окно для запуска Stack Builder (рис. 2.10). С помощью этой утилиты загружаются и устанавливаются дополнительные компоненты.

Ярлык для запуска мастера Stack Builder можно найти в меню Пуск в Windows. Там же присутствуют ярлыки для работы с pgAdmin и командной строкой SQL Shell (psql) (рис. 2.11).

Для удаления PostgreSQL необходимо запустить деинсталлятор, который автоматически создается при установке сервера PostgreSQL. Деинсталлятор присутствует в том же каталоге, где установлен и находится PostgreSQL (рис. 2.12).

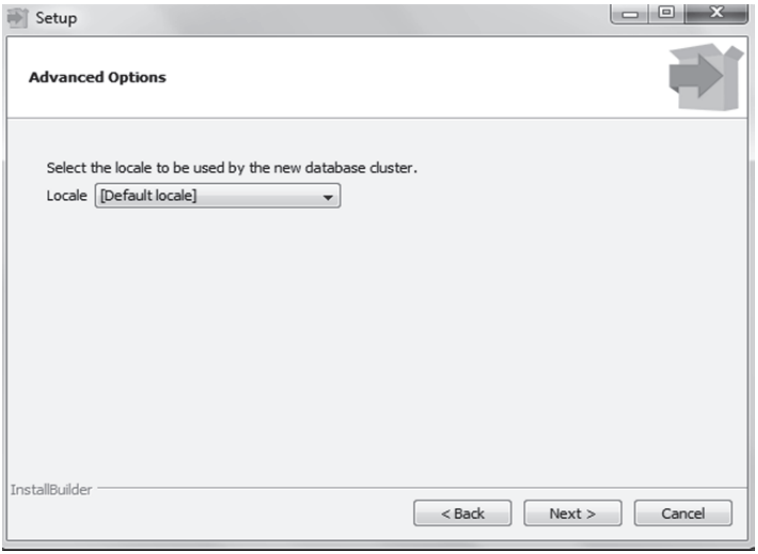


Рис. 2.8. Установка локаля

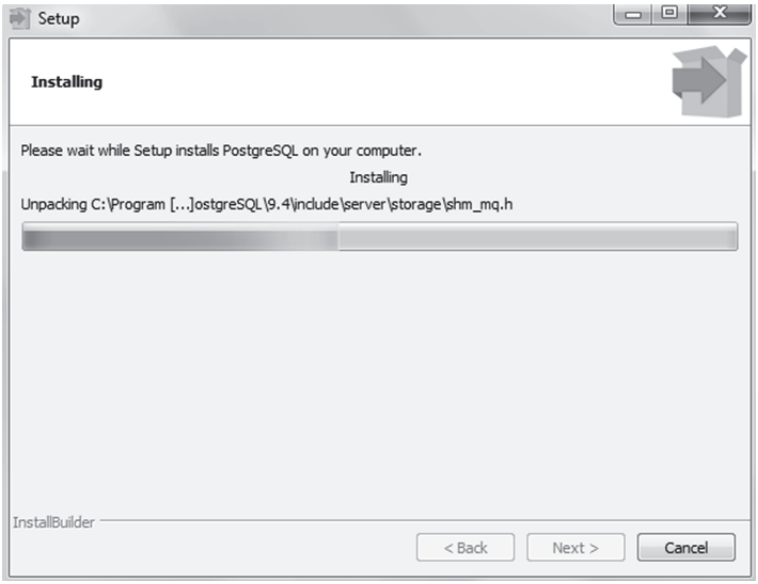


Рис. 2.9. Установка PostgreSQL

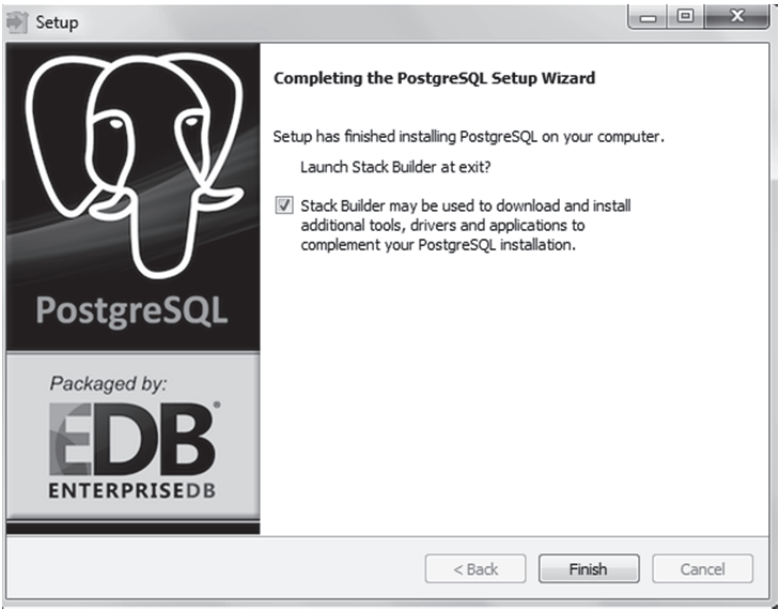


Рис. 2.10. Окно для установки Stack Builder

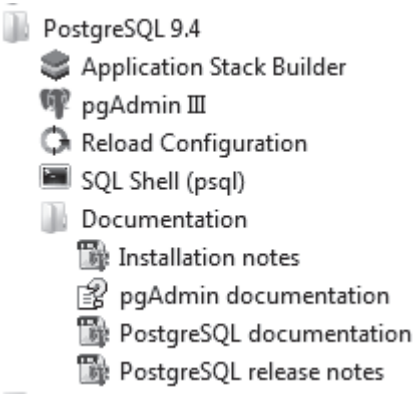


Рис. 2.11. Ярлыки для Stack Builder и pgAdmin

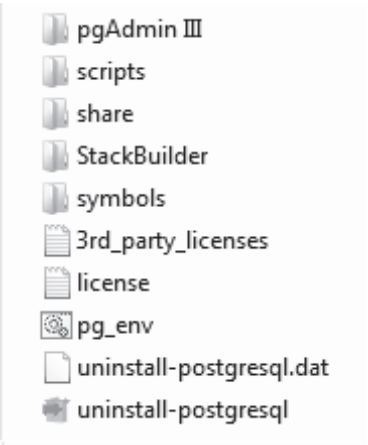


Рис. 2.12. Деинсталлятор PostgreSQL

При первом запуске SQL Shell (psql) пользователю последовательно предлагается указать сервер, базу данных, порт, имя пользователя и пароль. При этом по умолчанию предполагается, что сервер – это локальный компьютер, база данных – это база «postgres», порт – это значение 5432, имя пользователя – это имя «postgres». Если выбираются все параметры по умолчанию, то на каждом приглашении надо просто нажимать клавишу «Enter». Пароль пользователя «postgres» используется тот же самый, который был задан при установке. В итоге окно командной строки будет выглядеть примерно так, как показано на рис. 2.13 (для версии 9.4.1).

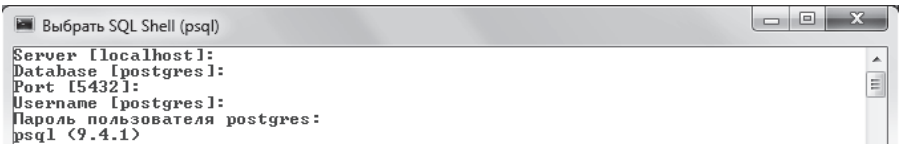


Рис. 2.13. Командная строка SQL Shell (psql)

Далее по тексту будет рассматриваться работа в командной строке SQL Shell (psql).

2.2. Создание пользователей

В PostgreSQL весьма важная роль отведена пользователям и группам пользователей. Информация о пользователях и группах пользователей хранится в системных каталогах. Учетные записи пользователей позволяют определить, какие именно действия разрешено выполнять в системе тому или иному пользователю. Пользователи и группы не связаны ни с какой конкретной базой данных и существуют как глобальные объекты баз данных.

Отметим, что в PostgreSQL по умолчанию создается один суперпользователь с именем «postgres». Все остальные пользователи создаются этим суперпользователем. Для создания новых пользователей в PostgreSQL используется команда:

CREATE USER имя_пользователя [[WITH] опции [...]]

В данной команде единственным обязательным параметром является имя нового пользователя, все остальные параметры не являются обязательными. Если выполнить создание пользователя командой:

CREATE USER имя_пользователя

то у этого пользователя не будет пароля и прав для создания баз данных и новых пользователей у него тоже не будет. Полный синтаксис команды CREATE USER выглядит примерно так:

CREATE USER пользователь

[WITH [SYSID uid]

[PASSWORD ‘пароль’]]

[CREATEDB | NOCREATEDB]

[CREATEUSER | NOCREATEUSER]

[IN GROUP группа [...]]

[VALID UNTIL ‘срок’]

В системе не допускается присутствие двух пользователей с одинаковыми именами. За ключевым словом WITH следуют опции SYSID и/или PASSWORD. Все остальные необязательные опции приводятся в произвольном порядке. Ниже перечислены ключевые слова и правила их использования.

SYSID uid. Пользователю назначается системный идентификатор uid. Если значение не указано, то идентификатор пользователя назначается автоматически (какое-либо уникальное число).

PASSWORD «пароль». Новому пользователю назначается пароль. Если значение не указано, то пароль не используется. Пароль можно установить или изменить позже, используя команду ALTER USER.

CREATEDB | NOCREATEDB. Ключевое слово CREATEDB предоставляет новому пользователю право создания баз данных, а также право уничтожения принадлежащих ему баз данных. Ключевое слово NOCREATEDB означает отсутствие прав создания баз данных. По умолчанию установлено NOCREATEDB.

CREATEUSER | NOCREATEUSER. Ключевое слово CREATEUSER предоставляет пользователю право выполнять команды CREATE USER и DROP USER (создания и удаление пользователей). Ключевое слово NOCREATEUSER означает отсутствие прав создания и удаления пользователей и устанавливается по умолчанию.

IN GROUP группа [....]. Новый пользователь включается в группу с заданным именем. Допускается перечисление нескольких групп через запятую.

VALID UNTIL «срок». Срок действия пароля. Если параметр не задан, то срок действия пароля не ограничивается.

ENCRYPTED | UNENCRYPTED. Опция отвечает за то, как будет храниться пароль в системных каталогах (по умолчанию устанавливается параметр password_encryption – зашифрованный пароль).

При отсутствии ключевых слов CREATEDB и CREATEUSER создастся пользователь, не обладающий какими-либо привилегиями. Он не может ни создавать, ни уничтожать базы данных и других пользователей.

Пример 2.1. Создать пользователя ivanov с паролем на аутентификацию 123456, который остается действительным до 31 января 2016 года. Команда будет выглядеть следующим образом:

CREATE USER IVANOV WITH PASSWORD ‘123456’ VALID UNTIL 'Jan 31 2016';

При успешном завершении операции будет выдано сообщение: CREATE ROLE. Результат выполнения команды CREATE USER показан на рис. 2.14.

```
postgres=# CREATE USER IVANOV WITH PASSWORD ’123456’ VALID UNTIL ’Jan 31 2016’;

CREATE ROLE
postgres=#
```

Рис. 2.14. Команда создания нового пользователя

Для того чтобы просмотреть всех созданных пользователей, необходимо в командной строке набрать команду \du. После выполнения команды выведется таблица, содержащая имена пользователей и их атрибуты.

Модификация существующих пользователей выполняется с помощью команды ALTER USER. Синтаксис команды ALTER USER:

ALTER USER имя_пользователя

[WITH PASSWORD -пароль]

[CREATEDB | NOCREATEDB]

[CREATEUSER | NOCREATEUSER]

[VALID UNTIL 'срок']

Обязательный аргумент имя_пользователя определяет пользователя, для которого требуется изменить параметры. Команда ALTER USER также часто используется для изменения пароля и/или срока действия пароля. Если срок действия пароля не надо ограничивать, то можно использовать значение infinity (VALID UNTIL 'infinity'). При

успешном выполнении команды будет выдано сообщение: ALTER ROLE.

Пример 2.2. Изменить пароль пользователя IVANOV на «654321», установить неограниченный срок действия этого пароля, предоставить права суперпользователя. Команда записывается следующим образом:

ALTER USER IVANOV WITH PASSWORD ‘654321’ CREATEUSER VALID UNTIL 'infinity';

При успешном завершении операции будет выдано сообщение: ALTER ROLE. Результат выполнения команды ALTER USER показан на рис. 2.15.

```
postgres=# ALTER USER IVANOV WITH PASSWORD ’654321’ CREATEUSER VALID UNTIL ’infi
nity’;
ALTER ROLE
postgres=#
```

Рис. 2.15. Команда ALTER USER

Чтобы удалить пользователя, необходимо воспользоваться командой DROP USER. Синтаксис команды DROP USER:

DROP USER имя_пользователя;

Параметр имя_пользователя определяет имя пользователя, удаляемого из системы. При успешном завершении операции будет выдано сообщение: DROP ROLE. Управление правами пользователей более подробно рассмотрено в разделе 4 настоящего пособия.

2.3. Создание баз данных

В PostgreSQL для создания баз данных можно использовать команду CREATE DATABASE. Синтаксис команды CREATE DATABASE:

CREATE DATABASE имя_базы_данных
[[WITH] [OWNER [=] имя_пользователя]
[LOCATION [=] ‘каталог’ | DEFAULT]
[TEMPLATE [=] шаблон | DEFAULT]
[ENCODING [=] имя_кодировки|номер_кодировки | DEFAULT]
[CONNECTION LIMIT [=] количество | -1]]

Обязательный параметр имя_базы_данных определяет имя создаваемой базы данных, имена баз данных должны начинаться с алфавитного символа и длина не должна превышать 31 символа. За необяза-

тельным ключевым словом WITH можно указать дополнительные атрибуты.

OWNER [=] имя_пользователя. Параметр имя_пользователя определяет владельца базы данных. По умолчанию предполагается, что владельцем является пользователь, выполняющий команду.

LOCATION [=] ‘каталог’. Здесь определяется каталог, в котором будет храниться база данных. Если параметр LOCATION отсутствует или в нем указано ключевое слово DEFAULT, то PostgreSQL создает базу данных в каталоге данных по умолчанию, задаваемом переменной среды PGDATA. PostgreSQL позволяет создать в заданном каталоге любое количество баз.

TEMPLATE [=] шаблон. Имя шаблона, используемого в качестве прототипа для создания новой базы данных. Если шаблон не задан или используется конструкция TEMPLATE = DEFAULT, то PostgreSQL создает новую базу данных на основе шаблона template1.

ENCODING [=] имя_кодировки. Если ключевое слово ENCODING не задано или используется параметр DEFAULT, то PostgreSQL создает базу данных, используя кодировку по умолчанию. Обычно это кодировка SQL ASCII.

CONNECTION LIMIT [=] количество. Параметр определяет, сколько одновременных соединений возможно к базе данных. Если параметр не указан, то ограничение на количество соединений отсутствует.

Пример 2.3. Создать базу данных test1. Команда имеет следующий вид:

```
CREATE DATABASE test1;
```

При успешном завершении операции будет выдано сообщение: CREATE DATABASE. Результат выполнения команды CREATE DATABASE показан на рис. 2.16.

```
postgres=# create database test1;
CREATE DATABASE
postgres=#
```

Рис. 2.16. Команда CREATE DATABASE

Создатель базы данных автоматически является владельцем новой базы данных. Владельцу базы данных будут принадлежать все объекты, которые хранятся в созданной базе данных. Владелец базы данных имеет право предоставлять другим пользователям доступ к объектам базы данных.

Команда

postgres=# \connect имя_базы_данных;

используется для переключения из одной базы данных в другую.

Чтобы удалить базу данных, необходимо воспользоваться командой DROP DATABASE. При удалении базы данных уничтожаются все таблицы, данные, объекты, которые хранились в этой базе. Также происходит физическое удаление системных файлов, связанных с удаляемой базой данных. Необходимо помнить, что эту операцию нельзя отменить. Синтаксис команды DROP DATABASE:

DROP DATABASE имя_базы_данных;

Параметр имя_базы_данных определяет имя базы данных, которая подлежит удалению из системы. Эта команда может быть выполнена только владельцем базы данных. Отметим, что команда не может быть выполнена, если в момент удаления кто-нибудь другой подключен к удаляемой базе данных. При успешном завершении операции будет выдано сообщение: DROP DATABASE.

Для изменения атрибутов базы данных используется команда ALTER DATABASE. Только владелец базы данных или администратор могут изменять базу данных. Синтаксис команды ALTER DATABASE:

ALTER DATABASE имя_базы_данных

[RENAME TO новое_имя_базы_данных]

[OWNER TO имя_пользователя]

[WITH CONNECTION LIMIT количество]

Обязательный параметр имя_базы_данных определяет имя базы данных, у которой изменяются атрибуты. База данных, которая используется в настоящий момент, не может быть переименована (необходимо предварительно подключиться к другой базе данных). Команда

postgres=# \c postgres;

может использоваться для переключения в базу данных postgres.

RENAME TO новое_имя_базы_данных. Этот параметр осуществляет переименование базы данных. Только владелец базы данных или администратор может переименовать базу данных.

OWNER TO имя_пользователя. В параметре имя_пользователя определяется новый владелец базы данных.

WITH CONNECTION LIMIT количество. Параметр определяет, сколько одновременных соединений возможно к базе данных.

Пример 2.4. Переименовать базу данных test1 и подключиться к ней. Команды имеют следующий вид:

```
postgres=# ALTER DATABASE test1 RENAME TO test3;
postgres=# \c test3;
```

При успешном завершении операции будет выдано сообщение: ALTER DATABASE. Результат выполнения команды ALTER DATABASE показан на рис. 2.17.

```
Вы подключены к базе данных "test2" как пользователь "postgres".
test2=# ALTER DATABASE test1 RENAME TO test3;
ALTER DATABASE
test2=# \c test3;
```

Рис. 2.17. Команда ALTER DATABASE

2.4. Создание таблиц

Таблицы можно создать с помощью команды CREATE TABLE. Эта команда формирует пустую таблицу в текущей базе данных. Команда CREATE TABLE задает имена столбцов в определенном порядке, определяет типы данных и размеры столбцов. Каждая таблица должна иметь как минимум один столбец. Имя создаваемой таблицы должно быть отличным от имени любой другой таблицы, последовательности, индекса, представления или типа данных. Для разделения слов в имени таблицы принято использовать нижнее подчеркивание. Синтаксис команды CREATE TABLE:

```
CREATE [{ TEMPORARY | TEMP }] TABLE имя_таблицы
 ( { имя_поля тип_поля [ограничение_поля [,...] ] |
ограничение_таблицы } [,… ] )
   [ INHERITS (базовая_таблица [,… ] ) ]
   ограничение_поля ::=
                [ CONSTRAINT имя_ограничения_поля ]
                { NOT NULL | UNIQUE | PRIMARY KEY |
                DEFAULT значение |
                CHECK ( условие ) |
                REFERENCES внешняя_таблица [ ( внешнее_поле ) ]
                                [ MATCH FULL | MATCH PARTIAL ]
```

[ON DELETE операция]
[ON UPDATE операция]
[DEFERRABLE | NOT DEFERRABLE]
[INITIALLY DEFERRED | INITIALLY IMMEDIATE] }
ограничение_таблицы ::=
[CONSTRAINT имя_ограничения_таблицы]
{ UNIQUE (поле [,...]) |
PRIMARY KEY (поле [,...]) |
CHECK (условие) |
FOREIGN KEY (поле [,...])
REFERENCES внешняя_таблица [(внешнее_поле [,...])]
[MATCH FULL | MATCH PARTIAL]
[ON DELETE операция]
[ON UPDATE операция]
[DEFERRABLE | NOT DEFERRABLE]
[INITIALLY DEFERRED | INITIALLY IMMEDIATE] }
операция ::= { NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT }

Таблица будет принадлежать пользователю, который ее создал. Иногда после параметра тип_поля в круглых скобках необходимо указать размер, например, char(20). В основном указание размера необходимо использовать для типа данных CHAR, так как по умолчанию аргумент размера для этого типа поля выставляется равным единице. Далее описаны параметры, используемые в команде CREATE TABLE.

TEMPORARY | TEMP. Этот параметр является признаком временной таблицы. Таблица, созданная с использованием ключевых слов TEMPORARY или TEMP, автоматически уничтожается в конце текущего сеанса. Все индексы и ограничения будут также уничтожены вместе с таблицей в конце рабочего сеанса.

имя_таблицы. Имя создаваемой таблицы.

имя_поля. Имя поля в новой таблице. Перечисляемые имена полей указываются в круглых скобках и разделяются запятыми.

тип_поля. Указывается тип поля, при этом может быть указан стандартный тип или массив одного из стандартных типов.

ограничение_поля. Полное определение ограничения для данного поля. Ниже перечислены параметры ограничений полей.

– имя_ограничения_поля. Необязательное имя, присвоенное ограничению.

– NULL. Ключевое слово NULL разрешает, чтобы в поле хранилось псевдозначение NULL. Действует по умолчанию.

– NOT NULL. Поле не может содержать псевдозначеиие NULL. Ограничение NOT NULL эквивалентно ограничению CHECK (поле NOT NULL).

– UNIQUE. Ограничение гарантирует, что поле содержит только уникальные значения. Это ограничение автоматически устанавливается при создании уникального индекса для поля.

– PRIMARY KEY. Поле используется для однозначной идентификации записей. Ограничение первичного ключа эквивалентно установке ограничений UNIQUE и NOT NULL.

– DEFAULT. Значение по умолчанию, используемое в том случае, если значение поля не было указано в команде INSERT. Если значение по умолчанию не задано, поле заполняется псевдозначением NULL.

– CHECK. Значения поля проверяются на соответствие заданному условию.

– условие. Произвольное выражение, результатом которого является логическая величина. Указывается после секции CHECK.

– REFERENCES. Входные значения ограничиваемого поля сравниваются со значениями другого поля в заданной таблице.

– внешняя_таблица. Имя таблицы, используемой для проверки в ограничении внешнего ключа.

– внешнее_поле. Имя поля внешней таблицы, используемого для проверки в ограничении внешнего ключа. Поле должно принадлежать существующей таблице. Если имя поля не задано, то используется первичный ключ заданной таблицы.

– MATCH FULL | MATCH PARTIAL. Опция MATCH указывает, разрешается ли смешивание псевдозначений NULL и «обычных» значений (отличных от NULL) при вставке в таблицу, у которой внешний ключ ссылается на несколько полей.

– ON DELETE. При выполнении команды DELETE для заданной таблицы с ограничением на поле выполняется одна из следующих операций: NO ACTION – если удаление приводит к нарушению целостности ссылок, происходит ошибка (используется по умолчанию, когда операция не указана); RESTRICT – аналогично NO ACTION; CASCADE – удаление всех записей, содержащих ссылки на удаляе-

мую запись, поэтому при каскадном удалении необходимо действовать очень осторожно; SET NULL – поля, содержащие ссылки на удаляемую запись, заменяются псевдозначениями NULL; SET DEFAULT – полям, содержащим ссылки на удаляемую запись, присваивается значение по умолчанию.

– ON UPDATE. При выполнении команды UPDATE для заданной таблицы выполняется одна из операций, описанных выше. По умолчанию также используется значение NO ACTION. При выборе операции CASCADE все записи, содержащие ссылки на обновляемую запись, обновляются новым значением.

– DEFERRABLE | NOT DEFERRABLE. Значение DEFERRABLE позволяет отложить выполнение ограничения до конца транзакции (вместо немедленного выполнения после выполнения команды). Значение NOT DEFERRABLE означает, что ограничение всегда проверяется сразу же после выполнения очередной команды. В этом случае пользователь не может отложить проверку ограничения до конца транзакции. По умолчанию установлен именно этот вариант.

– INITIALLY DEFERRED | INITIALLY IMMEDIATE. Значение INITIALLY DEFERRED откладывает проверку ограничения до конца транзакции, а при установке значения INITIALLY IMMEDIATE проверка производится после каждой команды. По умолчанию устанавливается значение INITIALLY IMMEDIATE.

– ограничение_таблицы. Полное определение ограничения для создаваемой таблицы. Ограничения таблиц могут распространяться на несколько полей, ограничение поля всегда создает ограничение только для одного поля. Ниже перечислены параметры ограничений таблиц.

– имя_ограничеиия_таблицы. Необязательное имя, присвоенное ограничению.

– поле [, ...]. Имя поля (или разделенный запятыми список полей), для которых устанавливается ограничение таблицы.

– PRIMARY KEY | UNIQUE. Ключевые слова, при наличии которых для заданных полей автоматически строится индекс. UNIQUE означает, что комбинация значений полей, перечисленных за ключевым словом UNIQUE, принимает только уникальные значения. Ключевое слово PRIMARY KEY обеспечивает проверку уникальности и запрещает присутствие псевдозначений NULL среди значений заданного поля (или полей).

– CHECK (условие). Команда INSERT или UPDATE завершается успешно лишь при выполнении заданного условия. Если условие не выполняется, то записи не добавляются и не модифицируются.

– FOREIGN KEY. Ограничение внешнего ключа. Входные значения ограничиваемого поля сравниваются со значениями другого поля в заданной таблице. Синтаксис части, следующей за секцией FOREIGN KEY, идентичен синтаксису секции REFERENCES для ограничений полей.

– базовая _таблица. Имя таблицы, от которой новая таблица наследует поля.

Пример 2.5. Создать таблицу weather с полями город, температура, количество осадков, дата. Для создания такой таблицы необходимо воспользоваться командой:

CREATE TABLE weather (city char(30), temp int, osadki real, data date);

При успешном завершении операции будет выдано сообщение: CREATE TABLE. Результат выполнения команды CREATE TABLE показан на рис. 2.18.

```
Вы подключены к базе данных "test3" как пользователь "postgres".
test3=# CREATE TABLE weather (city char(30), temp int, osadki real, data date);
CREATE TABLE
```

Рис. 2.18. Команда CREATE TABLE

Для того чтобы вывести список всех таблиц, созданных в текущей базе данных, необходимо воспользоваться командой: postgres=# \dt+. Чтобы получить описание таблицы с указанием списка ее полей, необходимо задать команду: postgres=# \d имя_таблицы.

Если созданная таблица больше не нужна, то ее можно удалить, используя команду DROP TABLE. Только владелец и суперпользователь может удалить таблицу. Синтаксис команды DROP TABLE:

DROP TABLE имя_таблицы [, ...] [CASCADE | RESTRICT]

Имя существующей таблицы, удаляемой из базы данных, определяется параметром имя_таблицы. В одной команде можно удалить сразу несколько таблиц, имена которых перечисляются через запятую.

Если указан параметр CASCADE, то вместе с таблицей автоматически удаляются все объекты, которые зависят от таблицы (например, view – представления). Если указан параметр RESTRICT, то попытка удалить таблицу, если какие-либо объекты зависят от нее, окажется неудачной. Это значение установлено по умолчанию.

При успешном завершении операции будет выдано сообщение: DROP TABLE.

2.5. Вставка данных

С помощью команды INSERT можно вставлять новые строки в таблицу. Записи могут вставляться как по одной, так и группами. Каждый столбец, который не указан в команде INSERT, будет заполняться значениями по умолчанию, если такие были указаны, либо будет заполнен псевдозначениями NULL. Если в команде указан неправильный тип данных для какого-либо столбца, то PostgreSQL предпримет попытку преобразования типа. Все типы данных, используемые в команде, имеют соответствующие форматы. Константы, которые не являются простыми числовыми значениями, обычно должны быть заключены в одинарные кавычки ('). Чтобы вставлять данные в таблицу, пользователь должен иметь привилегию INSERT. Синтаксис команды INSERT:

INSERT INTO имя_таблицы [(поле [, ...])]

{ DEFAULT VALUES | VALUES (значение [, ...]) | запрос }

Форма записи команды, при которой необходимо соблюдать порядок полей таблицы:

INSERT INTO имя_таблицы VALUES (значение1, значение2, значение3, …);

Альтернативная форма записи позволяет перечислять поля явно:

INSERT INTO имя_таблицы (поле1, поле2, поле3, …)

VALUES (значение1, значение2, значение3, …);

При такой форме записи можно указать поля в другом порядке или даже опустить некоторые поля.

Параметры, используемые в команде INSERT, подробнее описаны ниже.

Параметр имя_таблицы определяет таблицу, в которую вставляются новые данные.

Имя поля, для которого задается значение, должно совпадать с именем одного из полей таблицы, хотя порядок перечисления полей в команде не обязан совпадать с естественным порядком их следования в таблице.

Параметр «значение» содержит константу или выражение, заносимое в таблицу. Если в команде задан список полей, то значение ассоциируется с соответствующим полем в списке, при этом перечисленные поля должны однозначно соответствовать выражениям в списке.

Если для какого-либо поля выражение относится к другому типу данных, то PostgreSQL пытается выполнить преобразование типа. Если преобразование не выполняется, то вся команда INSERT отменяется.

«Запрос» – это синтаксически правильная команда SELECT. Количество полей, возвращаемых запросом, должно соответствовать количеству полей в списке, а их типы данных должны быть совместимы. В сочетании с ключевым словом VALUES команда INSERT всегда вставляет ровно одну запись, а чтобы вставить несколько записей, можно воспользоваться данными, полученными в результате запроса. Данные из итогового набора запроса заносятся в таблицу, указанную в команде INSERT.

В случае успешного завершения команды INSERT будет выведена запись вида:

INSERT OID count,

где OID представляет собой идентификатор объекта вставленной записи, а count – общее количество вставленных в таблицу записей.

Пример 2.6. В таблицу weather из примера 2.5 необходимо добавить запись. В этом случае можно воспользоваться командой:

INSERT INTO weather VALUES ('Новосибирск', 25, 50.5, '2014-06-06');

Результат выполнения команды INSERT показан на рис. 2.19.

```
test3=# INSERT INTO weather VALUES ('Новосибирск', 25, 50.5, '2014-06-06');
INSERT 0 1
test3=#
```

Рис. 2.19. Команда INSERT

Добавить данные в таблицу с записью полей в произвольном порядке можно с помощью команды:

INSERT INTO weather (data, city, osadki, temp) VALUES ('2014-07-07', 'Томск', 60.6, 12);

Результат выполнения команды INSERT с записью полей в произвольном порядке показан на рис. 2.20.

```
test3=# INSERT INTO weather (data, city, osadki, temp) VALUES ('2014-07-07', 'To
мск', 60.6, 12);
INSERT 0 1
test3=#
```

Рис. 2.20. Запись полей в произвольном порядке при добавлении данных

2.6. Модификация таблиц

Для модификации таблиц и полей в PostgreSQL используется команда ALTER TABLE. Синтаксис команды ALTER TABLE:

ALTER TABLE имя_таблицы [*] [действие [, ...]]

[RENAME [COLUMN] имя_поля TO новое_имя_поля]

[RENAME TO новое_имя]

где «действие» может быть одним из выражений:

ADD [COLUMN] поле тип [ограничение_поля [...]]

DROP [COLUMN] поле [RESTRICT | CASCADE]

ALTER [COLUMN] поле [SET DATA] TYPE тип
[USING выражение]

ALTER [COLUMN] поле SET DEFAULT выражение

ALTER [COLUMN] поле DROP DEFAULT

ALTER [COLUMN] поле { SET | DROP } NOT NULL

ADD ограничение_таблицы

DROP CONSTRAINT имя_ограничения [RESTRICT | CASCADE]

DISABLE TRIGGER [имя_триггера | ALL | USER]

ENABLE TRIGGER [имя_триггера | ALL | USER]

INHERIT родительская_таблица

NO INHERIT родительская_таблица

OWNER TO новый_владелец

Атрибут имя_таблицы определяет имя таблицы, изменения в которую будут вноситься, атрибут поле определяет имя поля, в которое будут вноситься изменения. Параметры, используемые в команде ALTER TABLE, подробнее описаны ниже.

RENAME [COLUMN] имя_поля TO новое_имя_поля. Форма для изменения имени поля таблицы.

RENAME TO новое_имя. Форма для изменения имени таблицы (или индекса, последовательности или представления).

ADD [COLUMN]. Эта форма добавляет новый столбец в таблицу, используя тот же синтаксис, что и команда CREATE TABLE. Необходимо указать имя нового поля и его тип.

DROP [COLUMN]. Эта форма удаляет столбец из таблицы. Индексы и ограничения, связанные с этим полем, будут также автомати-

чески удаляться. То же самое происходит, если указано ключевое слово CASCADE. Если указан параметр RESTRICT, то не произойдет удаления поля, если какие-либо объекты зависят от него.

ALTER [COLUMN] поле [SET DATA] TYPE. Эта форма используется для изменения типа столбца таблицы. Индексы и простые ограничения таблицы, связанные с указанным полем, будут автоматически конвертированы для использования нового типа столбца. Дополнительный параметр USING определяет, как произвести преобразование типа.

ALTER [COLUMN] поле SET DEFAULT \ ALTER [COLUMN] поле DROP DEFAULT. Эти формы используются для установки и удаления значения для поля по умолчанию соответственно.

ALTER [COLUMN] поле { SET | DROP } NOT NULL. Эта форма используется для разрешения или отмены использования нулевых значений. Например, если в поле не могут быть использованы нулевые значения, следует использовать SET NOT NULL.

ADD ограничение_таблицы. Эта форма добавляет новые ограничения на таблицу, используя тот же синтаксис, что и команда CREATE TABLE.

DROP CONSTRAINT. Эта форма удаляет указанное ограничение на таблицу.

DISABLE TRIGGER \ ENABLE TRIGGER. Можно включить или отключить один триггер с указанным именем или все триггеры. Отключение или включение требует привилегий суперпользователя или владельца.

INHERIT родительская_таблица. Эта форма делает текущую таблицу наследником указанной родительской таблицы. Впоследствии запросы к родителю будут включать в себя записи, содержащиеся в данной таблице. Чтобы стать наследником, таблица должна содержать все те же столбцы, что и у родителя (но может помимо этих столбуов, содержать и дополнительные поля). Столбцы должны быть такого же типа данных, и если у родителя столбцы не нулевые, то и у наследника они также должны быть NOT NULL.

NO INHERIT родительская_таблица. Такая форма устраняет текущую таблицу из наследников указанной родительской таблицы. Запросы к родительской таблице больше не будут включать в себя сведения, взятые из текущей таблицы.

OWNER TO новый_владелец. Эта форма изменяет владельца таблицы.

Ключевое слово [COLUMN] во всех формах не является обязательным и включается в команду лишь для наглядности. Все действия, за исключением RENAME, могут быть объединены в список и применяться параллельно. Например, можно добавить несколько столбцов и\или изменить тип нескольких столбцов в одной команде. Это особенно полезно с большими таблицами, когда необходимо сделать только один проход по таблице.

Чтобы использовать команду ALTER TABLE, необходимо быть суперпользователем или владельцем таблицы. Чтобы сделать таблицу наследником, необходимо также быть владельцем родительской таблицы.

Если добавлен новый столбец с помощью ADD COLUMN, то все существующие строки таблицы инициализируются значением по умолчанию для нового столбца (NULL, если значение по умолчанию не указано).

Форма DROP COLUMN не физически удаляет колонку, а просто делает ее невидимой для SQL операций. После операций вставки и обновления в таблице будет храниться нулевое значение для столбца.

Если у таблицы есть наследники, то не допускается добавление, переименование или изменение типа столбца в родительской таблице, не сделав то же самое для потомков. То есть команда ALTER TABLE будет отклонена. Это гарантирует то, что наследники и родители будут всегда иметь одинаковый набор столбцов.

Чаще всего используются следующие команды для модификации таблицы:

– переименование таблицы

ALTER TABLE имя_таблицы RENAME TO новое_имя_таблицы;

– переименование поля таблицы

ALTER TABLE имя_таблицы RENAME имя_поля TO новое_имя_поля;

– добавление нового поля в таблицу

ALTER TABLE имя_таблицы ADD имя_поля тип_поля;

– изменения типа поля

ALTER TABLE имя_таблицы ALTER имя_поля TYPE тип_поля;

– удаление поля из таблицы

ALTER TABLE имя_таблицы DROP имя_поля;

где имя_таблицы – имя таблицы, в которую вносятся изменения;

имя_поля – имя создаваемого/изменяемого поля;

тип_поля – тип создаваемого/изменяемого поля.

Пример 2.7. В таблицу weather из примера 2.5 требуется добавить поле pressure (давление). В этом случае можно воспользоваться командой:

ALTER TABLE weather ADD pressure int;

Результат выполнения команды ALTER TABLE показан на рис. 2.21.

```
test3=# ALTER TABLE weather ADD pressure int;
ALTER TABLE
test3=#
```

Рис. 2.21. Команда для добавления поля в таблицу

Далее необходимо переименовать поле city в поле gorod, а затем изменить тип этого поля на char(25). Для этого надо выполнить следующие команды:

ALTER TABLE weather RENAME city TO gorod;

ALTER TABLE weather ALTER gorod TYPE char(25);

Результат выполнения команд показан на рис. 2.22.

```
test3=# ALTER TABLE weather RENAME city TO gorod;
ALTER TABLE
test3=# ALTER TABLE weather ALTER gorod TYPE char(25);
ALTER TABLE
test3=#
```

Рис. 2.22. Команды для переименования поля
и изменения типа поля таблицы

С помощью следующих команд владельцем таблицы weather становится пользователь Ivan:

CREATE USER IVAN

ALTER TABLE weather OWNER TO Ivan;

Результат выполнения команд показан на рис. 2.23.

```
test3=# create user ivan;
CREATE ROLE
test3=# ALTER TABLE weather OWNER TO Ivan;
ALTER TABLE
```

Рис. 2.23. Команды для изменения владельца таблицы

2.7. Модификация данных

Обновление данных можно выполнить с помощью команды UPDATE. Новые значения полей задаются в виде констант, идентификаторов других баз данных или выражений. Допускается обновление как поля в целом, так и подмножества его значений в соответствии с заданными условиями. Синтаксис команды UPDATE:

UPDATE [ONLY] имя_таблицы SET поле =

{ выражение | DEFAULT } [, ...]

[FROM список_источников]

[WHERE условие]

Ключевое слово ONLY означает, что обновление выполняется только в указанной таблице и не распространяется на производные таблицы (если они существуют). Применяется лишь в том случае, если таблица использовалась в качестве базовой при наследовании. По умолчанию обновление будет производиться в указанной таблице и во всех ее подтаблицах.

SET поле = выражение. Обязательная секция SET содержит перечисленные через запятую условия, определяющие новые значения обновляемых полей. Условия всегда имеют форму поле = выражение, где поле – имя обновляемого поля, а выражение описывает новое значение поля. Если установлено ключевое слово DEFAULT, то значение поля изменится на значение по умолчанию (которое будет NULL, если нет конкретных выражений по умолчанию).

FROM список_источников. Секция FROM принадлежит к числу нестандартных расширений PostgreSQL и позволяет обновлять поля значениями, взятыми из других наборов. Источником может быть таблица, представление или другой источник данных.

WHERE условие. В секции WHERE задается критерий отбора обновляемых записей. Если секция WHERE отсутствует, то поле обновляется во всех записях. По аналогии с командой SELECT может использоваться для уточнения выборки из источников, перечисленных в секции FROM. При отсутствии ключевого слова WHERE команда UPDATE обновляет заданное поле во всех записях таблицы. Обычно новое значение поля задается выражением, а не константой. Выражение, указанное после ключевого слова SET, вычисляется заново для каждой записи, а новое значение поля определяется динамически.

При обновлении таблиц необходимо наличие прав записи для обновляемых полей и прав чтения для полей, используемых после ключевого слова WHERE. В случае успешного завершения команды UPDATE, psql возвращает сообщение вида UPDATE count, где count – счетчик числа обновившихся строк.

Пример 2.8. Изменить в таблице weather значение поля data и temp для города Омска. В этом случае можно воспользоваться следующей командой:

UPDATE weather SET data = '2014-06-07', temp = 20 WHERE gorod = 'Омск';

Результат выполнения команды UPDATE показан на рис. 2.24.

```
test3=# UPDATE weather SET data = '2014-06-07', temp = 20 WHERE gorod = 'Омск';
UPDATE 1
test3=#
```

Рис. 2.24. Команда UPDATE

Далее требуется увеличить значение в поле «осадки» на пять единиц для всех городов, где температура колеблется от 15 до 17 градусов. Для этого выполним следующую команду:

UPDATE weather SET osadki = osadki + 5 WHERE temp >= 15 and temp <= 17;

Результат выполнения команды показан на рис. 2.25.

```
test3=# UPDATE weather SET osadki = osadki + 5 WHERE temp >= 15 and temp <= 17;
UPDATE 1
test3=#
```

Рис. 2.25. Команда для изменения значений в полях таблицы

Из рисунка видно, что после выполнения команды выдано сообщение UPDATE 1, так как в результате выполнения команды была изменена только одна запись в таблице.

Удаление записей из таблиц производится с помощью команды DELETE. Команда удаления одной или нескольких записей из базы имеет следующий синтаксис:

DELETE FROM [ONLY] имя_таблицы [WHERE условие]

После ключевого слова WHERE задается критерий, по которому в таблице выбираются удаляемые записи. При отсутствии ключевого слова WHERE из таблицы удаляются все записи. Ключевое слово WHERE практически всегда присутствует в команде DELETE. В ней определяются условия отбора удаляемых записей.

После выполнения команды будет выдано сообщение DELETE count, где count определяет количество удаленных записей. Если число равно 0, это означает, что ни одна запись не соответствовала критерию удаления или же таблица не содержала ни одной записи.

Ключевое слово ONLY предотвращает удаление записей из таблиц, производных от заданной таблицы. В этом случае операция удаления выполняется только в непосредственно указанной таблице. Если ключевое слово ONLY отсутствует, то операция удаления распространяется не только на заданную таблицу, но и на все производные таблицы. Команду DELETE можно использовать для очистки всей таблицы, если не указывать в ней условие.

Пример 2.9. Удалить из таблицы weather все записи, которые датируются до 2014-01-01. В этом случае можно воспользоваться следующей командой:

DELETE FROM weather WHERE data < '2014-01-01';

Результат выполнения команды DELETE показан на рис. 2.26.

```
test3=# DELETE FROM weather WHERE data < '2014-01-01';
DELETE 2
test3=#
```

Рис. 2.26. Команда DELETE

Из рис. 2.26 видно, что условию «до 2014-01-01» удовлетворяло две записи в таблице и они были удалены.

2.8. Объединение таблиц INHERITS

В PostgreSQL поддерживается механизм создания объектно-реляционных связей, называемый наследованием. Таблица может наследовать некоторые атрибуты своих полей от одной или нескольких других таблиц, что приводит к созданию отношений типа «предок-потомок». В результате производные таблицы («потомки») обладают теми же полями и ограничениями, что и их базовые таблицы («предки»), а также дополняются собственными полями.

Производная таблица создается командой CREATE TABLE, в которую включается ключевое слово INHERITS и имя базовой таблицы (или нескольких таблиц). Часть команды CREATE TABLE, относящаяся к наследованию, выглядит следующим образом:

CREATE TABLE имя_таблицы-потомка определение

INHERITS (имя_таблицы-предка [,...])

В этом определении производная таблица – имя создаваемой таблицы, определение – полное определение таблицы команды CREATE TABLE, а базовая таблица – таблица, структура которой наследуется новой таблицей. Дополнительные имена базовых таблиц перечисляются через запятую.

Пример 2.10. Существует таблица kinoteatr, необходимо создать новую таблицу rating_kinoteatr, которая наследует все поля таблицы kinoteatr, а также содержит поле rating. Ниже приведена команда, с помощью которой это можно сделать:

CREATE TABLE rating_kinoteatr (rating int) INHERITS (kinoteatr);

Результат выполнения команды показан на рис. 2.27.

```
test3=# CREATE TABLE rating_kinoteatr (rating int) INHERITS (kinoteatr);
CREATE TABLE
```

Рис. 2.27. Создание производной таблицы

Далее можно вывести структуру новой таблицы, используя команду:

test3=#\d rating_kinoteatr.

Отметим, что вставка данных в базовую таблицу никак не отражается на производной таблице. При модификации данных в производной таблице изменяется только содержимое производной таблицы, а все данные из базовой таблицы остаются без изменений.

2.9. Создание ограничений

В PostgreSQL предусмотрено несколько вариантов ограничения данных, участвующих в операциях вставки и обновления. Один из них заключается в установке ограничений для таблиц и полей.

Ограничение (constraint) представляет собой особый атрибут таблицы, который устанавливает критерии допустимости для содержимого ее полей. Соблюдение этих правил помогает предотвратить заполнение базы ошибочными или неподходящими данными.

Ограничения задаются после ключевого слова CONSTRAINT при создании таблицы командой CREATE TABLE. Они делятся на два типа – ограничения полей и ограничения таблиц. Ограничения полей всегда относятся только к одному полю, тогда как ограничения таблиц могут устанавливаться как для одного поля, так и для нескольких полей. В команде CREATE TABLE ограничения полей задаются сразу же после определения поля, тогда как ограничение таблицы устанавлива-

ется в специальном блоке, отделенном запятой от всех определений полей. Ниже описаны различные правила, устанавливаемые при помощи ограничений.

Синтаксис ограничения поля выглядит следующим образом:

[CONSTRAINT имя_ограничения]

{ NOT NULL | NULL | UNIQUE | PRIMARY KEY | DEFAULT значение | CHECK (условие) |

REFERENCES таблица [(поле)]

[MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]

[ON DELETE операция]

[ON UPDATE операция] }

[DEFERRABLE | NOT DEFERRABLE]

[INITIALLY DEFERRED | INITIALLY IMMEDIATE]

Определение следует в команде CREATE TABLE сразу же за типом ограничиваемого поля и предшествует запятой, отделяющей его от следующего поля. Ограничения могут устанавливаться для любого количества полей, а ключевое слово CONSTRAINT и идентификатор имя_ограничения не обязательны.

Существует несколько типов ограничений полей, задаваемых при помощи специальных ключевых слов. Некоторые из них косвенно устанавливаются при создании ограничений другого типа. Типы ограничений полей перечислены ниже.

NOT NULL. Поле не может содержать псевдозначение NULL. Ограничение NOT NULL эквивалентно ограничению CHECK (поле NOT NULL).

NULL. Поле может содержать нулевые значения, установлено по умолчанию.

UNIQUE. Поле не может содержать повторяющиеся значения. Следует учитывать, что ограничение UNIQUE допускает многократное вхождение псевдозначений NULL, поскольку формально NULL не совпадает ни с каким другим значением.

PRIMARY KEY. Автоматически устанавливает ограничения UNIQUE и NOT NULL, а для заданного поля создается индекс. В таблице может устанавливаться только одно ограничение первичного ключа.

DEFAULT значение. Пропущенные значения поля заменяются заданной величиной. Значение по умолчанию должно относиться к типу данных, соответствующему типу поля.

CHECK условие. Команда INSERT или UPDATE для записи завершается успешно лишь при выполнении заданного условия.

REFERENCES. Это ограничение состоит из нескольких, имеет несколько вариантов, которые перечислены ниже.

– REFERENCES таблица [(поле)]. Входные значения ограничиваемого поля сравниваются со значениями другого поля в заданной таблице. Если совпадения отсутствуют, то команды INSERT или UPDATE не выполняются. Если параметр поля не указан, то проверка выполняется по первичному ключу. Ограничение REFERENCES похоже на ограничение таблицы FOREIGN KEY, описанное в следующем пункте.

– MATCH FULL | MATCH PARTIAL. Ключевое слово MATCH указывает, разрешается ли смешивание значений NULL и обычных значений при вставке в таблицу, у которой внешний ключ ссылается на несколько полей. Таким образом, на практике MATCH приносит пользу лишь в ограничениях таблиц, хотя формально она может использоваться и при ограничении полей. Конструкция MATCH FULL запрещает вставку данных, у которых часть полей внешнего ключа содержит псевдозначение NULL (кроме случая, когда NULL содержится во всех полях). Если ключевое слово MATCH отсутствует, считается, что поля с псевдозначениями NULL удовлетворяют ограничению.

– ON DELETE операция. При выполнении команды DELETE для заданной таблицы с ограничиваемым полем выполняется одна из следующих операций: NO ACTION (если удаление приводит к нарушению целостности ссылок, то происходит ошибка; используется по умолчанию, если операция не указана), RESTRICT (аналогично NO ACTION), CASCADE (удаление всех записей, содержащих ссылки на удаляемую запись), SET NULL (поля, содержащие ссылки на удаляемую запись, заменяются псевдозначениями NULL), SET DEFAULT (полям, содержащим ссылки на удаляемую запись, присваивается значение по умолчанию).

– ON UPDATE операция. При выполнении команды UPDATE для заданной таблицы выполняется одна из операций, описанных выше. По умолчанию используется значение NO ACTION. Если выбрана операция CASCADE, то все записи, содержащие ссылки на обновляемую запись, обновляются новым значением.

– DEFERRABLE | NOT DEFERRABLE. Значение DEFERRABLE позволяет отложить выполнение ограничения до конца транзакции (вместо немедленного выполнения после завершения команды). Зна-

чение NOT DEFERRABLE означает, что ограничение всегда проверяется сразу же после завершения очередной команды. В этом случае пользователь не может отложить проверку ограничения до конца транзакции. По умолчанию выбирается именно этот вариант.

– INITIALLY DEFERRED | INITIALLY IMMEDIATE. Ключевое слово INITIALLY задается только для ограничений, определенных с ключевым словом DEFERRED. Значение INITIALLY DEFERRED откладывает проверку ограничения до конца транзакции, а при установке значения INITIALLY IMMEDIATE проверка производится после каждой команды. При отсутствии ключевого слова INITIALLY по умолчанию используется значение INITIALLY IMMEDIATE.

Пример 2.11. Необходимо создать таблицу cinema со следующими ограничениями полей: поле id не может содержать значения меньше нуля, поле name не может содержать NULL значения, а также не может содержать пустые значения, поле zal должно содержать значение больше или равное трем. Ниже приведена команда, с помощью которой это можно сделать:

CREATE TABLE cinema (id int CHECK (id > 0), name char(12) NOT NULL CHECK (name <> ''), zal int CHECK (zal>= 3));

Результат выполнения команды показан на рис. 2.28.

```
test3=# CREATE TABLE cinema (  id int CHECK (id > 0), name char(12) NOT NULL CH
ECK (name <> ''),  zal int CHECK (zal>= 3) );
CREATE TABLE
test3=#
```

Рис. 2.28. Создание ограничений

Для просмотра всех ограничений, наложенных на поля таблицы, можно воспользоваться командой: test3=#\d cinema.

В ограничениях таблиц, в отличие от ограничений полей, могут участвовать сразу несколько полей таблицы. Синтаксис ограничения таблицы:

[CONSTRAINT имя_ограничения]

{ UNIQUE (поле [, ...]) | PRIMARY KEY (поле [, ...]) | CHECK (условие) | FOREIGN KEY (поле [, ...])

REFERENCES таблица [(поле [, …])]

[MATCH FULL | MATCH PARTIAL]

[ON DELETE операция]

[ON UPDATE операция] }

[DEFERRABLE | NOT DEFERRABLE]

[INITIALLY DEFERRED | INITIALLY IMMEDIATE]

Ключевое слово CONSTRAINT имя_ограничения определяет необязательное имя. Ограничениям рекомендуется присваивать содержательные имена. Имя также может пригодиться и для удаления ограничения.

PRIMARY KEY (поле [, ...]). Ограничение таблицы PRIMARY KEY имеет много общего с аналогичным ограничением поля. В ограничении таблицы PRIMARY KEY могут перечисляться несколько полей, разделенных запятыми. Для перечисленных полей автоматически строится индекс. Как и в случае с ограничением поля, комбинация значений всех полей должна быть уникальной и не может содержать NULL.

UNIQUE (поле [, ...]). Ограничение означает, что комбинация значений полей, перечисленных за ключевым словом UNIQUE, принимает только уникальные значения. Допускается многократное вхождение псевдозначения NULL.

CHECK (условие). Команда INSERT или UPDATE для записи завершается успешно лишь при выполнении заданного условия. Используется по аналогии с ограничениями полей, но CHECK может содержать ссылки на несколько полей.

FOREIGN KEY (поле [, ...]) REFERENCES таблица [(поле [, ...])]. В качестве прототипа для секции REFERENCES можно перечислить несколько полей. Синтаксис части, следующей за ключевым словом FOREIGN KEY, идентичен синтаксису ограничения REFERENCES для полей.

Команда

ALTER TABLE имя_таблицы DROP CONSTRAINT поле;

позволяет удалить ограничения у заданного поля таблицы.

Добавление ограничений в существующую таблицу. Команда ALTER TABLE позволяет включать ограничения в существующую таблицу. Установка ограничений в команде ALTER TABLE имеет следующий синтаксис:

ALTER TABLE имя_таблицы ADD [CONSTRAINT ограничение]

{ CHECK (условие) | FOREIGN KEY (поле [. ...])

REFERENCES таблица [(поле [....])]

[MATCH FULL | MATCH PARTIAL]

[ON DELETE операция]

[ON UPDATE операция] }

[DEFERRABLE | NOT DEFERRABLE]

[INITIALLY DEFERRED | INITIALLY IMMEDIATE]

Пример 2.12. Требуется установить новое ограничение для таблицы cinema, чтобы нельзя было добавлять фильмы, снятые в Праге. Ниже приведена команда, с помощью которой это можно сделать:

ALTER TABLE cinema ADD CONSTRAINT city CHECK (city <> 'Praga');

Результат выполнения команды показан на рис. 2.29.

```
test3=# ALTER TABLE cinema ADD CONSTRAINT city CHECK (city <> 'Praga');
ALTER TABLE
test3=#
```

Рис. 2.29. Добавление ограничений в существующую таблицу

Команда

ALTER TABLE cinema DROP CONSTRAINT city;

позволяет удалить ограничения у заданного поля city таблицы cinema (рис. 2.30).

```
test3=# ALTER TABLE cinema DROP CONSTRAINT city;
ALTER TABLE
test3=#
```

Рис. 2.30. Удаление ограничений

2.10. Формирование запросов

Команда SELECT, предназначенная для построения запросов и выборки данных из таблиц и представлений, является одной из самых важных команд SQL. Данные, возвращаемые в результате запроса, называются итоговым набором и состоят из записей и полей.

Данные итогового набора не хранятся на диске в какой-либо постоянной форме. Итоговый набор является лишь временным представлением данных, полученных в результате запроса. Структура полей итогового набора может соответствовать структуре исходной таблицы, но может и отличаться от нее. Итоговые наборы даже могут содержать поля, выбранные из других таблиц.

Ниже приведено общее определение синтаксиса SELECT.

SELECT [ALL | DISTINCT [ON (выражение [, ...])]]

* | выражение [[AS] имя] [, ...]

[FROM источник [, ...]]

[WHERE условие]

[GROUP BY критерий [, ...]]

[HAVING условие [, ...]]

[{ UNION | INTERSECT | EXCEPT } [ALL | DISTINCT] подзапрос]

[ORDER BY выражение [ASC | DESC | USING оператор] [, ...]]

[LIMIT { число | ALL }]

[[NATURAL] тип_объединения источник [ON условие | USING (список_полей)]] [, ...]

[OFFSET начало [ROW | ROWS]]

[FOR { UPDATE | SHARE } [OF таблица [, ...]] [NOWAIT] [...]]

где источник – это имя таблицы или подзапроса. При этом «источник» имеет следующий синтаксис:

FROM [ONLY] имя_таблицы [[AS] синоним [(синоним_поля [,...])]]

(запрос) [AS] синоним [(синоним_поля [, ...])]

[ONLY] table_name [*] [[AS] alias [(column_alias [, ...])]]

(select) [AS] alias [(column_alias [, ...])]

with_query_name [[AS] alias [(column_alias [, ...])]]

function_name ([argument [, ...]]) [AS] alias [(column_alias [, ...] | column_definition [, ...])]

function_name ([argument [, ...]]) AS (column_definition [, ...])

from_item [NATURAL] join_type from_item [ON join_condition | USING (join_column [, ...])]

ALL. Необязательное ключевое слово ALL указывает на то, что в выборку включаются все найденные записи.

DISTINCT [ON (выражение [, ...])]. Ключевое слово DISTINCT определяет поле или выражение, значения которого должны входить в итоговый набор не более одного раза.

Выражение [AS имя] [, ...]. В качестве выражения обычно указывается имя поля, хотя выражение также может быть константой, иденти-

фикатором, функцией. Перечисляемые выражения разделяются запятыми. Звездочка (*) является сокращенным обозначением всех полей.

FROM источник [, ...]. В секции FROM указывается источник, в котором PostgreSQL ищет заданные цели. В данном случае источник является именем таблицы или подзапроса. Допускается перечисление нескольких источников, разделенных запятыми.

[NATURAL] тип_объединения источник [ON условие | USING (список_полей)]. Источники FROM могут группироваться в секции JOIN с указанием типа объединения (INNER, FULL, OUTER, CROSS). В зависимости от типа объединения также может потребоваться уточняющее условие или список полей.

WHERE условие. Ключевое слово WHERE ограничивает итоговый набор заданными критериями. Условие должно возвращать простое логическое значение (true или false), но оно может состоять из нескольких внутренних условий, объединенных логическими операторами (например, AND или OR).

GROUP BY критерий [, ...]. Ключевое слово GROUP BY обеспечивает группировку записей по заданному критерию. Причем критерий может быть простым именем поля или произвольным выражением, примененным к значениям итогового набора.

HAVING условие [, ...]. Ключевое слово HAVING во многом похоже на ключевое слово WHERE, но условие проверяется на уровне целых групп, а не отдельных записей.

{ UNION | INTERSECT | EXCEPT } [ALL | DISTINCT] подзапрос. Выполнение одной из трех операций, в которых участвуют два запроса (исходный и дополнительный); итоговые данные возвращаются в виде набора с обобщенной структурой, из которого удаляются дубликаты записей. UNION – объединение (записи, присутствующие в любом из двух наборов). INTERSECT – пересечение (записи, присутствующие одновременно в двух наборах). EXCEPT – исключение (записи, присутствующие в основном наборе SELECT, но не входящие в подзапрос).

ORDER BY выражение. Сортировка результатов команды SELECT по заданному выражению. [ASC | DESC | USING оператор] – порядок сортировки, определяемый ключевым словом ORDER BY: по возрастанию (ASC) или по убыванию (DESC). С ключевым словом USING может задаваться оператор, определяющий порядок сортировки.

FOR { UPDATE | SHARE } [OF таблица [, ...]] [NOWAIT] [...]. Возможность монопольной блокировки возвращаемых записей.

В транзакционных блоках FOR UPDATE блокирует записи указанной таблицы до завершения транзакции. Заблокированные записи не могут обновляться другими транзакциями.

LIMIT {число | ALL}. Ограничение максимального количества возвращаемых записей или возвращение всей выборки (ALL).

OFFSET начало. Точка отсчета записей для ключевого слова LIMIT. Например, если в LIMIT установлено ограничение в 50 записей, а в OFFSET 10, то запрос вернет записи с номерами 10–50.

Необязательное ключевое слово DISTINCT исключает дубликаты из итогового набора.

Ниже описаны компоненты, относящиеся к ключевому слову FROM.

[ONLY] таблица. Имя таблицы, используемой в качестве источника для команды SELECT. Ключевое слово ONLY исключает из запроса записи всех таблиц-потомков.

[AS] синоним. Источникам FROM могут назначаться необязательные псевдонимы, упрощающие запрос. Ключевое слово AS является необязательным.

(запрос) [AS] синоним. В круглых скобках находится любая синтаксически правильная команда SELECT. Итоговый набор, созданный запросом, используется в качестве источника FROM так, словно выборка производится из статической таблицы. При выборке из подзапроса обязательно должен назначаться синоним.

(синоним_поля [, ...]). Синонимы могут назначаться не только всему источнику, но и его отдельным полям. Перечисляемые синонимы нолей разделяются запятыми и группируются в круглых скобках за синонимом источника FROM. Синонимы перечисляются в порядке следования полей в таблице, к которой они относятся.

Синонимы для названий полей. Наличие синонима для набора данных позволяет обращаться к нему при помощи точечной записи, что делает команды SQL более компактными и наглядными.

Синонимы можно назначать не только для источников данных, но и для отдельных полей внутри этих источников. Для этого за синонимом источника данных приводится список синонимов полей, разделенный запятыми и заключенный в круглые скобки. Список синонимов полей представляет собой последовательность идентификаторов, перечисленных в порядке следования полей в структуре таблицы (слева направо). Список синонимов полей не обязан содержать данные обо

всех полях. К тем полям, для которых не были заданы синонимы, можно обращаться по обычным именам.

Команда SELECT с ключевым словом INTO TABLE создает новую таблицу, структура и содержимое которой определяются итоговым набором запроса. Синтаксис команды выглядит следующим образом:

SELECT выборка INTO [TABLE] новая_таблица FROM таблицы_источники WHERE условие.

По сути, это тот же самый запрос, только создается новая таблица, куда и записываются результаты выборки.

2.11. Представления

При работе с SQL нередко возникают ситуации, когда один и тот же запрос приходится использовать повторно. Кроме того, крайне неэффективно пересылать большие наборы данных по сети на сервер PostgreSQL при выполнении стандартных процедур.

В таких случаях как правило используются представления (views). Представление можно рассматривать как хранимый запрос, на основе которого создается объект базы данных. Этот объект очень похож на таблицу, но в его содержимом динамически отражается состояние только тех записей, которые были заданы при создании. Представления являются гибкими и универсальными. Они могут строиться на основе как простых и стандартных запросов к одной таблице, так и чрезвычайно сложных запросов, в которых задействовано несколько таблиц. Представления можно создать с помощью команды CREATE VIEW, синтаксис которой выглядит следующим образом:

CREATE VIEW имя_представления AS запрос;

где имя_представления – это имя (идентификатор) создаваемого представления, запрос – команда SELECT, определяющая содержимое представления.

Представления значительно упрощают получение нужных данных. Вместо того чтобы вводить длинный запрос, достаточно ввести простую команду SELECT. Команда удаления представления имеет следующий синтаксис:

DROP VIEW имя_представления

Удаление представления не отражается на данных, которые использовались представлением. Представление всего лишь обеспечивает доступ к данным других таблиц и потому может удаляться без потери данных (хотя запрос, на котором оно основано, конечно, теряется).

Контрольные вопросы

1. Опишите порядок установки СУБД PostgreSQL для операционной системы Windows.
2. Что необходимо указать пользователю при первом запуске SQL Shell (psql)?
3. Какая команда используется для создания новых пользователей в PostgreSQL?
4. С помощью какой команды выполняется модификация существующих пользователей?
5. Какую команду в PostgreSQL можно использовать для создания баз данных?
6. С помощью какой команды можно удалить базу данных?
7. Какая команда используется для изменения атрибутов базы данных?
8. Какая команда используется для создания таблиц?
9. С помощью какой команды можно получить описание таблицы с указанием списка ее полей?
10. Каким образом можно вставлять новые строки в таблицу?
11. Какая команда используется для модификации таблиц и полей?
12. Каким образом можно выполнить обновление данных в таблице?
13. Какая команда осуществляет удаление записей из таблиц?
14. Каким образом выполняется объединение таблиц с использованием ключевого слова inherits?
15. Что такое ограничение? Как можно задать ограничение?
16. Какие типы ограничений полей существуют?
17. Каким образом можно добавить ограничения в существующую таблицу?
18. С помощью какой команды можно увидеть все ограничения, наложенные на поля таблицы?
19. Как можно удалить ограничение?
20. С какой целью в запросах используется ключевое слово DISTINCT?
21. В каких случаях используется представление?
22. С помощью какой команды можно создать представление?
23. Перечислите преимущества использования синонимов.

3. ТЕХНОЛОГИЯ РАБОТЫ С СУБД POSTGRESQL В ADO.NET

Функциональность большинства приложений направлена на извлечение, отображение и модификацию данных.

В настоящее время разработчики перешли от простых клиентских приложений с локальными базами данных к распределенным системам, основанным на централизованных базах данных и выделенных серверах. Сегодня свыше 95 % всех информационных систем так или иначе используют базы данных. Вот почему вопросы организации взаимодействия приложения с базой данных являются актуальными.

ADO.NET представляет собой набор библиотек, входящих в Microsoft .NET Framework и предназначенных для взаимодействия с различными хранилищами данных из .NET-приложений. Библиотеки ADO.NET включают в себя все необходимые классы для подключения к источникам данных практически произвольного формата, выполнения запросов к этим источникам и получения результата.

3.1. Архитектура ADO.NET

Так как ADO.NET представляет собой набор классов для организации взаимодействия клиентского приложения с базой данных, рассмотрим объектную модель ADO.NET. На рис. 3.1 показаны классы, составляющие объектную модель ADO.NET.

Объекты, расположенные в левой части, называются подсоединенными и необходимы для управления соединением, транзакциями, выборкой данных и передачей измененных данных в БД. Они непосредственно взаимодействуют с базой данных.

Объекты, расположенные в правой части, называются отсоединенными. Они позволяют работать с данными автономно. Отсоединенные объекты не взаимодействуют непосредственно с БД для получения и модификации хранимых данных. Можно сказать, что они не взаимодействуют со всеми объектами левой части иерархии объектов.

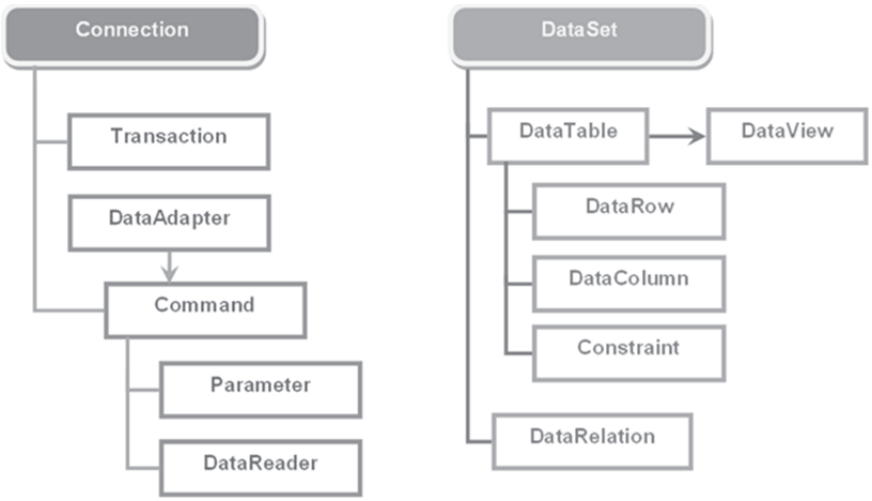


Рис. 3.1. Иерархия объектов ADO.NET

При организации доступа к данным с помощью ADO.NET исключительно важную роль играют объекты, изображенные на рис. 3.1. Рассмотрим их более подробно.

Объект **Connection** представляет соединение с источником данных. С его помощью можно задать тип источника данных, его местонахождение, параметры доступа и ряд других атрибутов. Перед тем как начать взаимодействие с источником данных, необходимо установить подключение к нему с помощью объекта Connection.

Объект **Command** представляет запрос к источнику данных, вызов хранимой процедуры или прямой запрос на возврат содержимого конкретной таблицы. Как известно, существует несколько типов запросов. Часть из них возвращают данные, извлекаемые из источника данных, другие изменяют записи, третьи – управляют структурой БД. С помощью объекта Command возможно выполнить любой из перечисленных типов запросов. При выполнении запроса, не возвращающего записи, необходимо использовать метод ExecuteNonQuery объекта Command, а при извлечении данных из БД – метод ExecuteReader, который, возвращает объект DataReader, позволяющий просматривать полученные в результате запроса записи.

Объект **DataReader** предназначен для максимально быстрой выборки и просмотра возвращаемых запросом записей. Однако он позво-

ляет просматривать весь результирующий набор записей путем перемещения от одной записи к другой и, таким образом, просматривать только одну запись за один раз. DataReader не имеет возможности для обновления значений записей и поэтому может работать в режиме «только для чтения», за счет чего обладает высокой производительностью.

Объект **Transaction** позволяет осуществлять группировку записей в логическую единицу работы, называемой транзакцией. Транзакция логически объединяет несколько различных действий, связанных с манипулированием данными, в единое целое. В процессе выполнения действий, осуществляемых в рамках транзакции, СУБД обычно кэширует изменения, вносимые этими действиями в данные до момента завершения транзакции. Это позволяет производить отмену любых изменений, выполненных в рамках транзакции, в случае, если хотя бы одно из действий транзакции завершилось неудачно.

Объект **Parameter** позволяет вводить в SQL-запросы параметры, значение которых может быть задано непосредственно перед исполнением запроса. За счет этого отпадает необходимость каждый раз изменять текст самого запроса.

Объект **DataAdapter** представляет собой связующее звено между отсоединенными объектами ADO.NET и базой данных. С его помощью осуществляется заполнение таких объектов, как DataSet или DataTable, значениями, полученными в результате выполнения запроса к базе данных, для последующей автономной работы с ними. Помимо этого DataAdapter реализует эффективный механизм выполнения обновления данных, хранимых в базе данных, изменениями, внесенными в данные объектов DataSet и DataTable.

Объект **DataTable** позволяет просматривать данные в виде наборов записей и столбцов. Фактически он представляет собой аналог таблицы БД, размещенный в памяти. Достоинством такой организации является возможность автономной работы с данными. Это означает, что после установления соединения с базой данных, чтения данных и заполнения ими объекта DataTable можно отключиться от источника данных и продолжать работать с ним в автономном режиме. При этом, однако, возникают и побочные эффекты, один из которых связан с тем, что человек, работающий с автономным набором данных, не может увидеть изменений, вносимых в данные в этот момент другими пользователями.

Объект **DataColumn** представляет собой столбец объекта DataTable. Набор же всех столбцов объекта DataTable представляет собой

коллекцию Columns. Посредством этого объекта можно получить доступ к значению ячейки соответствующего столбца.

Объект **DataRow** представляет собой строку объекта DataTable. Набор всех строк этого объекта представляет собой коллекцию Rows. DataRow очень часто используется для доступа к значению конкретного поля определенной записи. При этом применяется свойство Item.

Объект **DataSet** представляет собой отсоединенный набор данных, который может рассматриваться как контейнер для объектов DataTable. DataSet позволяет организовывать внутри себя структуру, полностью соответствующую реальной структуре таблиц и связей между ними в БД. Это удобно в том случае, когда при работе с базой данных необходимы данные из разных таблиц. Все изменения, которые вносятся в данные, хранящиеся в DataSet, кэшируются в объектах DataRow. Когда возникает необходимость передачи изменений из DataSet в базу данных, возможно осуществить передачу только измененных данных, вместо того чтобы передавать все данные объекта. Это значительно снижает объем данных, передаваемых между клиентским компьютером и сервером.

Объект **DataRelation** представляет собой описание связей между таблицами реляционной базы данных. Он предоставляется объектом DataSet и позволяет организовывать взаимосвязи между таблицами отсоединенного набора данных объекта DataSet. DataRelation можно настроить таким образом, чтобы изменения значения первичного ключа родительской таблицы автоматически передавались (каскадировались) дочерним записям, а при удалении записи в родительской таблице автоматически удалялись записи в дочерних таблицах, связанных с ней.

Объект **DataView** предназначен для организации возможности просмотра содержимого DataTable различными способами. Это относится к таким операциям, как сортировка и фильтрация записей. Так как обычно требуется просматривать содержимое с разными установками фильтрации и сортировки, то необходимо создавать несколько объектов DataView, связанных с одним объектом DataTable.

Поставщики данных. Одной из основных идей, лежащих в основе ADO.NET, является наличие поставщиков данных. Поставщик данных – это набор классов, предназначенных для взаимодействия с хранилищем данных определенного типа. За счет этого модель ADO.NET является чрезвычайно гибкой и расширяемой.

Каждый поставщик данных может обеспечивать доступ только к базе данных определенного формата. Для доступа к базе данных

PostgreSQL используется поставщик Npgsql. Также можно применять поставщика данных OleDb для .NET или ODBC для .NET, обеспечивающих доступ к любым данным, для которых существует драйвер OleDb либо ODBC соответственно.

Каждый поставщик .NET реализует одинаковые базовые классы – Connection, Transaction, DataAdapter, Command, Parameter, DataReader, имена которых зависят от поставщика. Например, у поставщика Npgsql существует объект NpgsqlDataAdapter, у поставщика ODBC – OdbcDataAdapter и т. д. У каждого поставщика данных существует собственное пространство имен. Хотя все поставщики относятся к пространству имен System.Data, каждый из них содержит свой подраздел этого пространства. Например, объект NpgsqlDataAdapter находится в пространстве имен Npgsql.

Итак, в приложениях использовать библиотеки ADO.NET можно двумя различными способами: в подключенном режиме или в автономном режиме, а также с помощью очень востребованной в настоящее время технологии EDM (Entity Data Model), которая реализуется в .NET библиотекой Entity Framework.

В качестве примера организации работы с базой данных в ADO.NET рассмотрим создание подключения и выполнение запросов к базе данных AutoLot, которая представлена таблицей inventory, содержащей записи об автомобилях с параметрами: идентификационный номер, марка, цвет, имя любимца.

```
CREATE TABLE inventory
(
 carid serial NOT NULL,
 mark character(20),
 color character(20),
 petname character(20),
 CONSTRAINT "CarID" PRIMARY KEY (carid)
)
```

3.2. ADO.NET – подключенный уровень

Помимо работы с разными поставщиками данных подключение к базе данных можно выполнить несколькими способами:

1) через создание источника данных ODBC;

2) без создания источника данных, задав все параметры подключения в самом приложении;

3) указав параметры подключения в конфигурационном файле, который потом считывается приложением. Этот подход является более гибким, так как не требует при изменениях параметров перекомпиляции приложения.

Для того чтобы приложение .NET могло осуществлять взаимодействие с базой данных, необходимо установить соединение. Наиболее типичным сценарием работы приложения с БД является следующий.

1. Создать, настроить и открыть объект подключения.

2. Создать и настроить объект команды, указав объект подключения в аргументе конструктора или через свойство Connection.

3. Вызвать метод ExecuteReader() настроенного объекта команды.

4. Обработать каждую запись с помощью метода Read() объекта чтения данных.

Подключение к БД. Для подключения к базе данных во время выполнения программы необходимо создать объект Connection, а также задать его свойства, определяющие текущие параметры подключения. Основным параметром, устанавливающим необходимые опции для подключения к БД, является строка соединения, которая представляет собой набор пар «имя-значение», разделенных точкой с запятой. Существует несколько фрагментов информации, указываемых в строке подключения, которые необходимы практически всегда. Перечислим и прокомментируем их назначение.

1. Сервер, на котором находится база данных. Если СУБД, к которой осуществляется подключение, расположена на клиентском компьютере, то вместо имени сервера необходимо указывать имя localhost либо IP-адрес 127.0.0.1.

2. Имя базы данных, к которой производится подключение.

3. Способ аутентификации пользователя. Существующие клиент-серверные СУБД (к которым относится SQL Server, Oracle, PostgreSQL и ряд других) позволяют указывать в строке подключения имя пользователя и пароль, которые будут нужны для проверки возможности доступа к базе данных, либо использовать параметры текущего пользователя.

Рассмотрим технологию работы с базой данных двух поставщиков данных СУБД PostgreSQL: ODBC и Npgsql.

3.3. Использование поставщика данных ODBC

Создание источника данных. Технология ODBC. Технология ODBC в общем случае не самая быстрая для работы с большинством СУБД. Однако она поддерживает очень большой процент существующих на данный момент СУБД и, что очень важно, проверена временем.

Операционная система Windows осуществляет настройку параметров ODBC средствами системных утилит. Для этого нужно выполнить следующие действия.

1. Прежде всего необходимо, чтобы в системе был установлен драйвер psqlODBC для СУБД PostgreSQL.

2. Далее необходимо зарегистрировать источник данных в системе. Для этого нужно открыть системное меню Пуск → Панель управления → Администрирование → Источники данных (ODBC) (путь приведен для операционной системы Windows). В появившемся окне нужно открыть вкладку Драйверы и убедиться в наличии установленного драйвера ODBC для PostgreSQL (рис. 3.2).

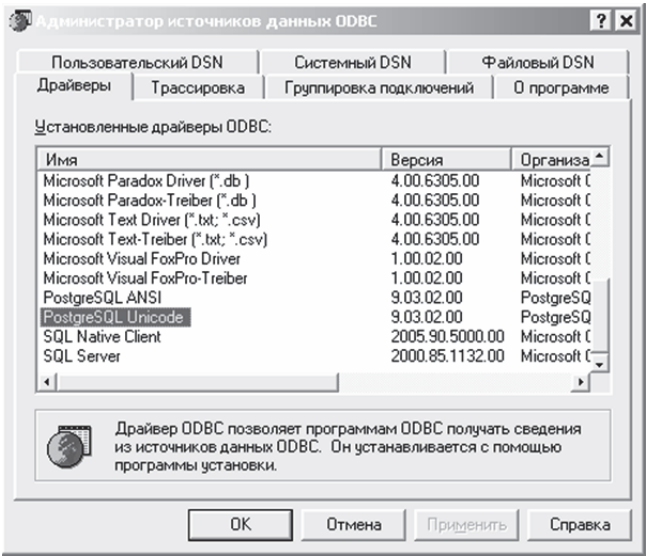


Рис. 3.2. Список драйверов ODBC

3. Выбрать вкладку Пользовательский DSN, на которой будет представлен список доступных источников данных.

4. Для того чтобы добавить новый источник данных, следует нажать кнопку Добавить и выбрать драйвер PostgreSQL. По кнопке OK перейти в следующее открывшееся окно и задать параметры соединения (рис. 3.3).

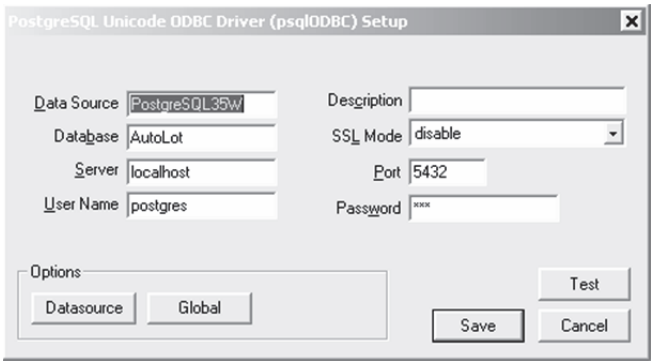


Рис. 3.3. Задание параметров базы данных

5. Далее сохранить параметры соединения по кнопке Save. Источник данных создан и готов к использованию локальными приложениями (рис. 3.4).

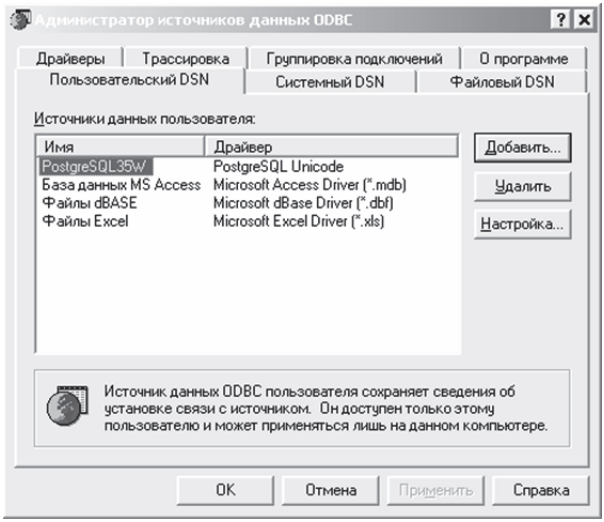


Рис. 3.4. Список доступных источников данных

Стоит заметить, что приведенные выше действия необходимы лишь для удобства при подключении к базе данных через ODBC из приложения.

Подключение к БД через источник данных ODBC. Для создания программного соединения с базой данных с использованием технологии ODBC можно использовать класс System.Data.Odbc.OdbcConnection. Параметры соединения задаются с помощью публичного атрибута класса ConnectionString. Пример строки соединения с использованием источника данных:

```
OdbcConnection oCon = new OdbcConnection();
// задание созданного источника данных
   oCon.ConnectionString = "Dsn=PostgreSQL35W";
   oCon.Open();
```

Подключение без установления источника данных. Подключение производится прямо в приложении путем создания объекта подключения OdbcConnection и задания ему строки подключения с параметрами:

```
OdbcConnection oCon = new OdbcConnection();
// настройка и задание строки подключения
oCon.ConnectionString = "DRIVER={PostgreSQL
Unicode};Server=localhost;
User=postgres;Port=5432;Password=123;Database=AutoLot;";
// подключение
 oCon.Open();
```

Следует учитывать, что при подключении к базе данных может произойти сбой, в результате которого установить соединение с ней окажется невозможно. Это может быть особенно актуальным при размещении базы данных на другом сервере, который в момент подключения может оказаться недоступным. Для того чтобы неудавшаяся попытка соединения с базой данных не приводила к фатальным последствиям при работе приложения, необходимо использовать конструкции try catch, позволяющие адекватно реагировать на возникшую ошибку:

```
try
{
        oCon.Open();
```

```
        //вывод сообщений в элемент Label в окно приложения
         label2.Text = "Сервер: "+ oCon.ServerVersion;
         label2.Text += "\n Соединение:" + oCon.ToString();
        // выполнение запросов к БД
} catch(Exception ex)
{
         label2.Text = "При соединении с БД произошла ошибка ";
         label2.Text += ex.Message;
}finally
{
         oCon.Close();
         label2.Text += "\n Закрытие соединения: ";
         label2.Text += oCon.State.ToString();
}
```

Для обслуживания соединения с базой данных расходуются ресурсы компьютера, на котором выполняется приложение. В связи с этим рекомендуется открывать соединение как можно позже, а закрывать как можно раньше, сразу после того, как все необходимые действия с объектами базы данных были выполнены. Кроме того, желательно строить программный код так, чтобы соединение с базой данных закрывалось при любом исходе соединения с ней. В предыдущем примере код, расположенный в блоке finally, выполнится при любом исходе попытки подключения, что гарантирует освобождение ресурсов сервера, не дожидаясь момента, когда память будет освобождена сборщиком мусора.

Задание строки подключения в конфигурационном файле. В конфигурационный файл app.config добавить строку подключения к базе данных:

```
<configuration>
…
<appSettings>
 <add key="DBConnectionString" value=
"Server=localhost;Port=5432;User=postgres;Password=123;
Database=AutoLot"/>
```

```
</appSettings>
</configuration>
```

2. Затем в приложении извлечь параметры подключения из конфигурационного файла. Для этого обязательно в модуле нужно подключить пространства имен System.Configuration.

```
oCon.ConnectionString = ConfigurationSettings.
AppSettings ["DBConnectionString"];
   // и далее подключение
oCon.Open();
   . . . .
```

3.4. Использование поставщика данных Npgsql

Npgsql является поставщиком данных PostgreSQL в ADO.NET. Для его использования нужно в проект добавить ссылки на Npgsql.dll и Mono.Security.dll или скопировать эти файлы в проект.

Создание объекта подключения можно реализовать также в операторе using. Ключевое слово using упрощает работу с объектами, которые реализуют интерфейс IDisposable. Интерфейс IDisposable содержит один метод Dispose(), который используется для освобождения ресурсов, захваченных объектом. Метод будет вызван сразу же по завершении оператора using.

```
using (NpgsqlConnection oCon = new NpgsqlConnection())
{
// настройка, задание строки подключения
oCon.ConnectionString =
"Server=localhost;Port=5432;User=postgres;Password=123;Database=
AutoLot;";
// подключение
oCon.Open();
. . .
}
```

Этого поставщика можно использовать и при подключении к базе данных через конфигурационный файл аналогично рассмотренному ранее примеру.

3.5. Выполнение команд над наборами данных

Одним из основных элементов из набора классов ADO.NET, способным выполнять любой SQL-оператор, является класс Command. Для того чтобы использовать класс Command, необходимо установить тип команды, установить текст запроса на языке SQL и привязать ее к соединению с БД.

Существует три типа команд класса Command (табл. 3.1).

Таблица 3.1

Задание типа SQL-команды

CommandType. Text	Выполнение оператора SQL, текст которого устанавливается в свойстве CommandText. Используется по умолчанию
CommandType. StoredProcedure	Выполнение хранимой процедуры, имя которой установлено в свойстве CommandText
CommandType. TableDirect	Выполнение опроса всей таблицы базы данных, имя которой задается в свойстве CommandText. Этот тип команды используется для обратной совместимости с некоторыми драйверами OLE DB

Пример создания объекта Command для выполнения SQL-запроса:

```
string strSQL = "SELECT * FROM inventory";
OdbcCommand myCommand = new OdbcCommand(strSQL, oCon);
```

Для выполнения созданной команды необходимо использовать один из следующих методов (табл. 3.2).

Таблица 3.2

Методы выполнения SQL-запроса

ExecuteReader()	Выполнение запроса SQL и возврат объекта DataReader, представляющего однонаправленный курсор, с доступом только для чтения для команды Select
ExecuteNonQuery()	Выполнение SQL-команд, предназначенных для вставки, изменения, удаления записей БД. Результатом работы команды является количество строк, обработанных командой
ExecuteScalar()	Выполнение SQL-команды и возврат первой строки результата запроса. Обычно используется для выполнения команды, содержащей агрегирующие функции типа COUNT(), MAX() и т. д.

При использовании метода ExecuteReader() создается объект DataReader, с помощью которого можно организовать перебор всех строк возвращенного набора данных. Для этого необходимо реализовать цикл по строкам результирующего набора данных. Объект DataReader представляет собой один из самых простых и быстрых способов получения доступа к данным БД.

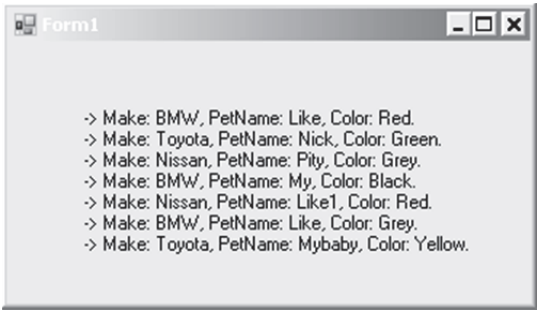


Рис. 3.5. Результат выполнения запроса

Пример реализации запроса к базе данных для получения набора данных через команды поставщика ODBC (рис. 3.5):

```
// Создание объекта команды на языке SQL.
string strSQL = "SELECT * FROM inventory";
OdbcCommand myCommand = new OdbcCommand(strSQL, oCon);
// Получение объекта чтения данных
using (OdbcDataReader myDataReader =
myCommand.ExecuteReader())
{
   // Просмотр всех результатов.
   string s = "";
   while (myDataReader.Read())
{
   s += string.Format("-> Make: {0}, PetName: {1}, Color: {2}.",
myDataReader["mark"].ToString().Trim(),
myDataReader["petname"].ToString().Trim(),
myDataReader["color"].ToString().Trim());
```

```
            s += "\n";
        }
        label1.Text = s;
    }
    oCon.Close();
```

Пример реализации запроса к базе данных для получения набора данных через команды поставщика Npgsql:

```
// Создание объекта команды на языке SQL.
string strSQL = "SELECT * FROM inventory";
NpgsqlCommand myCommand = new NpgsqlCommand(strSQL, oCon);
// Получение объекта чтения данных
using (NpgsqlDataReader myDataReader =
myCommand.ExecuteReader())
{
// Просмотр всех результатов.
string s = "";
while (myDataReader.Read())
{
s += string.Format("-> Make: {0}, PetName: {1}, Color: {2}.",
myDataReader["mark"].ToString().Trim(),
myDataReader["petname"].ToString().Trim(),
myDataReader["color"].ToString().Trim());
s += "\n";
}
label1.Text = s;
}
oCon.Close();
}
```

Данные примеры показывают, что выполнение запросов к базе данных происходит одинаково, независимо от поставщика данных. Далее в примерах будем использовать только Npgsql.

Пример выполнения метода ExecuteScalar (), подсчитывающего количество записей в БД:

```
try
 {
    // создание объекта подключения
    Connection();
    // создание объекта команды
    string strSQL = "SELECT Count(*) FROM inventory";
    myCommand = new NpgsqlCommand(strSQL, oCon);
    // Получение объекта чтения данных
    Int64 n = (Int64)myCommand.ExecuteScalar();
    label1.Text = "Количество записей ="+ n.ToString();
 } catch (NpgsqlException ex)
 {
    label1.Text = "Ошибка команды";
 }finally
 {
    oCon.Close();
 }
```

Метод ExecuteNonQuery() используется для выполнения команд, которые не возвращают результирующих наборов данных. Результатом работы метода ExecuteNonQuery() является количество обработанных записей. В следующем примере демонстрируется возможность удаления товара из таблицы inventory (рис. 3.6).

```
int n=0;
try
{
    // создание объекта подключения
    Connection();
    // создание объекта команды
    string strSQL = "DELETE FROM Inventory WHERE carid=12";
    myCommand = new NpgsqlCommand(strSQL, oCon);
```

```
    // Получение результата
    n=myCommand.ExecuteNonQuery();
    label1.Text="Удалено: "+n+" записей";
} catch(NpgsqlException ex)
{
    label1.Text = "Ошибка удаления";
 } finally
{
    oCon.Close();
 }
```

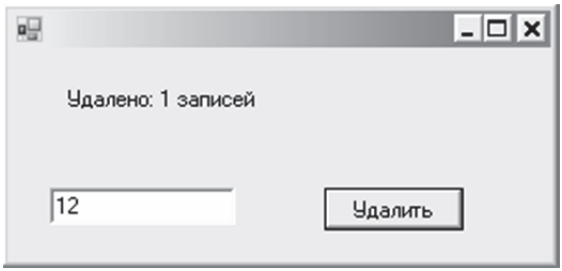


Рис. 3.6. Удаление записи из таблицы базы данных

В реальных ситуациях практически всегда пользователю требуется ввести некоторое условие, в соответствии с которым будет выполняться запрос SQL на удаление. Таким условием, например, может быть ввод наименования объекта хранения или идентификационного номера записи, подлежащей удалению. Для реализации такой возможности в предыдущем примере добавим элемент управления TextBox, предназначенный для ввода идентификационного номера записи, которую необходимо удалить, а также изменим текст строки, формирующей команду, следующим образом:

```
NpgsqlCommand myCommand = new NpgsqlCommand ("DELETE FROM inventory WHERE carid ='"+ textBox1.Text +"'", oCon);
```

Как видно, в данном случае строка SQL-запроса удаления записи из таблицы inventory формируется динамически в зависимости от введенного значения в элемент textBox1. Такая практика динамического

формирования запросов в реальных приложениях допустима, однако при этом могут возникать проблемы, связанные с безопасностью приложения, например, атаки внедрением SQL. Пользователь может ввести в поле ввода параметра текст, отличный от того, который от него ожидает приложение. Это может приводить к тому, что текст SQL-запроса фактически изменяется и выполняет не то действие, на которое рассчитывал программист при его реализации. Например, если в предыдущем примере в элемент textBox1 ввести текст «1 OR 1=1» (рис. 3.7), то в результате будут удалены все записи из таблицы inventory. После формирования строки SQL-запрос будет выглядеть так:

"DELETE FROM inventory WHERE carid = 1 OR 1=1"

и условие удаления будет истинно для всех строк таблицы.

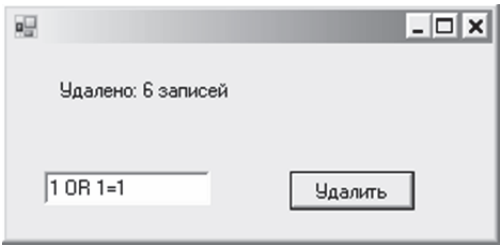


Рис. 3.7. Недопустимая команда пользователя

Решений данной проблемы может быть несколько. Во-первых, можно постараться проанализировать текст, введенный пользователем, до того как он будет подставлен в текст SQL-запроса. При этом можно анализировать длину введенного текста (если длина превышает стандартную длину вводимых данных, ожидаемых от пользователя, следует предпринять определенные действия и, возможно, заблокировать выполнение запроса), можно придумать и другие алгоритмы анализа вводимых данных, но все равно они не могут гарантировать стопроцентной безопасности приложения.

Лучшей практикой защиты от атаки внедрением SQL является использование параметризованных команд. Кроме того, параметризованные команды SQL более просты при формировании и применении.

3.6. Параметризованные SQL-команды

Для задания параметров строки SQL-запроса используется класс NpgsqlParameter, являющийся наследником DbParameter (табл. 3.3).

Таблица 3.3

Основные методы класса NpgsqlParameter

DbType	Выдает или устанавливает тип данных параметра, представляемый в виде типа CLR
Direction	Выдает или устанавливает вид параметра: только для ввода, только для вывода, для ввода и для вывода или параметр для возврата значения
IsNullable	Выдает или устанавливает, может ли параметр принимать пустые значения
ParameterName	Выдает или устанавливает имя параметра
Size	Выдает или устанавливает максимальный размер данных для параметра (полезно только для текстовых данных)
Value	Выдает или устанавливает значение параметра

Пример создания параметризованной команды:

```
 string sql = "INSERT INTO inventory
VALUES(@Mark,@Color,@PetName)";
 // У этой команды будут внутренние параметры.
 using (NpgsqlCommand cmd = new NpgsqlCommand(sql, oCon))
 {
   // Заполнение коллекции параметров.
   NpgsqlParameter param = new NpgsqlParameter();
   param.ParameterName = "@Mark";
   param.Value = this.TextBox2.Text;
   param. NpgsqlDbType = NpgsqlTypes.NpgsqlDbType.Varchar;
   cmd.Parameters.Add(param);
   //остальные параметры добавляются аналогично
   //но можно и так
    cmd.Parameters.Add("@Mark",
NpgsqlTypes.NpgsqlDbType.Varchar).Value = this.TextBox2.Text;
```

```
        cmd.Parameters.Add("@Color",
    NpgsqlTypes.NpgsqlDbType.Varchar).Value = this.TextBox3.Text;
        cmd.Parameters.Add("@PetName",
    NpgsqlTypes.NpgsqlDbType.Varchar).Value = this.TextBox4.Text;
      cmd.ExecuteNonQuery();
    }
```

3.7. Транзакции

Транзакция – это набор операций в базе данных, которые должны быть либо все выполнены, либо все не выполнены. Транзакции применяются для обеспечения безопасности, верности и непротиворечивости данных в таблице.

Транзакции очень важны тогда, когда при работе с базой данных требуется взаимодействие с несколькими таблицами или несколькими хранимыми процедурами (или с сочетанием неделимых объектов базы данных).

Все поставщики данных имеют класс, наследованный от DBTransaction, который реализует интерфейс IDBTransaction. Основные методы, наследуемые от интерфейса:

- Commit() – сохранить выполненные действия.
- RollBack() – отменить выполненные действия.

```
NpgsqlTransaction tx = null;
 try
 {
    Connection();
     // Создание объекта команды на языке SQL.
     string strSQL = string.Format("Delete from inventory where
     carid={0}", id);
     NpgsqlCommand myCommand = new NpgsqlCommand(strSQL,
     oCon);
     tx = oCon.BeginTransaction(); //создание транзакции
     myCommand.Transaction = tx;  // включение команды в транзакцию
     myCommand.ExecuteNonQuery();    //выполнение команды
     tx.Commit();    // фиксирование результатов транзакции
```

```
    } catch (NpgsqlException) // обработка исключения записи в базу
      данных
{
        tx.Rollback();  // откат транзакции
        label5.Text = "Ошибка записи в БД";
} finally
{
        oCon.Close();       // закрытие подключения
}
```

3.8. ADO.NET – автономный уровень

ADO.NET позволяет смоделировать в памяти компьютера данные из базы данных с помощью многочисленных членов пространства имен System.Data (DataSet, DataTable, DataRow, DataColumn, DataView и DataRelation). При этом возникает иллюзия, что компьютер постоянно подключен к внешнему источнику данных, хотя на самом деле все операции выполняются с локальной копией реляционных данных (рис. 3.8).

Рис. 3.8. Работа с данными через ADO.NET на автономном уровне

Тип DataSet представляет собой контейнер для любого количества объектов DataTable, каждый из которых содержит коллекцию объектов DataRow и DataColumn.

Объекты адаптеров данных выполняют связующую роль между клиентским уровнем и реляционной базой данных. С их помощью можно получить объекты DataSet, поработать с их содержимым и отправить измененные строки обратно для дальнейшей обработки. В ре-

зультате получается широко масштабируемое .NET-приложение обработки данных.

Объекты адаптеров пересылают данные между вызывающим процессом и источником данных с помощью объектов DataSet. Объект адаптера автоматически обслуживает подключение к базе данных. Для повышения масштабируемости адаптеры данных держат подключение открытым минимально возможное время. Как только вызывающий процесс получит объект DataSet, вызывающий уровень полностью отключается от базы данных и остается с локальной копией удаленных данных. Физическая база данных не обновляется, пока вызывающий процесс явно не передаст DataSet адаптеру данных для обновления.

Рассмотрим создание DataSet и NpgsqlDataAdapter на примере:

```
// Создание строки подключения
string ConnectionString =
"Server=localhost;Port=5432;User=postgres;Password=123;Database=
AutoLot;";
// Создание объекта команды на языке SQL.
 string strSQL = "SELECT * FROM inventory";
// Создание объекта NpgsqlDataAdapter
// параметры: строка запроса и строка подключения
NpgsqlDataAdapter da = new
NpgsqlDataAdapter(strSQL,ConnectionString);
// Создание объекта DataSet и заполнение таблицы Inventory данными
DataSet ds = new DataSet();
// Можно задать соответствие имен столбцов базы данных и понятных названий.
 DataTableMapping custMap = da.TableMappings.Add("inventory",
"В наличии");
 custMap.ColumnMappings.Add("carid", "Номер");
 custMap.ColumnMappings.Add("mark", "Марка");
 custMap.ColumnMappings.Add("petname", "Название");
 custMap.ColumnMappings.Add("color", "Цвет");
 da.Fill(ds, "inventory");
 // Отображение таблицы через таблицу DataGridView
 dataGridView1.DataSource = ds.Tables["В наличии"].DefaultView;
```

Результат заполнения объекта DataSet через NpgsqlDataAdapter представлен на рис. 3.9.

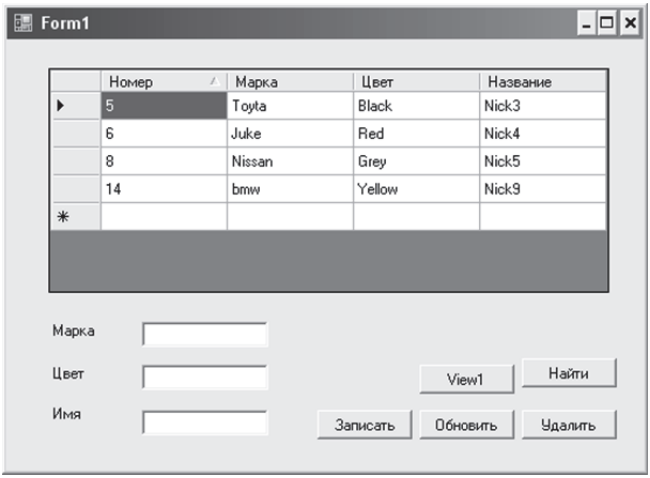


Рис. 3.9. Вывод данных из таблицы «В наличии» DataSet

В библиотеках поставщиков наследники базового класса DbDataAdapter содержат свойства и методы, представленные в табл. 3.5.

Таблица 3.5

Методы и свойства класса базового класса DbDataAdapter

Fill()	Выполняет команду SQL SELECT (указанную в свойстве SelectCommand) для запроса к базе данных и загрузки этих данных в объект DataTable
SelectCommand InsertCommand UpdateCommand DeleteCommand	Содержат SQL-команды, отправляемые в хранилище данных при вызовах методов Fill () и Update ()
Update ()	Выполняет команды SQL INSERT, UPDATE и DELETE (указанных в свойствах InsertCommand, UpdateCommand и DeleteCommand) для сохранения в базе данных изменений, выполненных в DataTable

Особенность работы с объектом адаптера данных состоит в том, что при его использовании не нужно открывать или закрывать подключение к базе данных. Все это делается автоматически. Однако

адаптеру данных нужно передать объект подключения или строку подключения (на основании которой все равно будет создан объект подключения), чтобы сообщить адаптеру данных, с какой базой данных вы хотите взаимодействовать.

3.9. Компонент отображения данных DataGridView

В качестве альтернативного варианта подключения к базе данных и отображения можно использовать компонент DataGridView, у которого в свойстве DataSource задаются источник данных, поставщик данных, строка подключения и параметры подключения. Для этого используется мастер настройки источника данных.

После перетаскивания компонента на страницу в его свойствах нужно выбрать позицию DataSource и Добавить источник данных проекта. Далее нужно выбрать База данных → Набор данных → Создать подключение → Выбрать источник данных (PostgreSQL) (рис. 3.10, 3.11, 3.12).

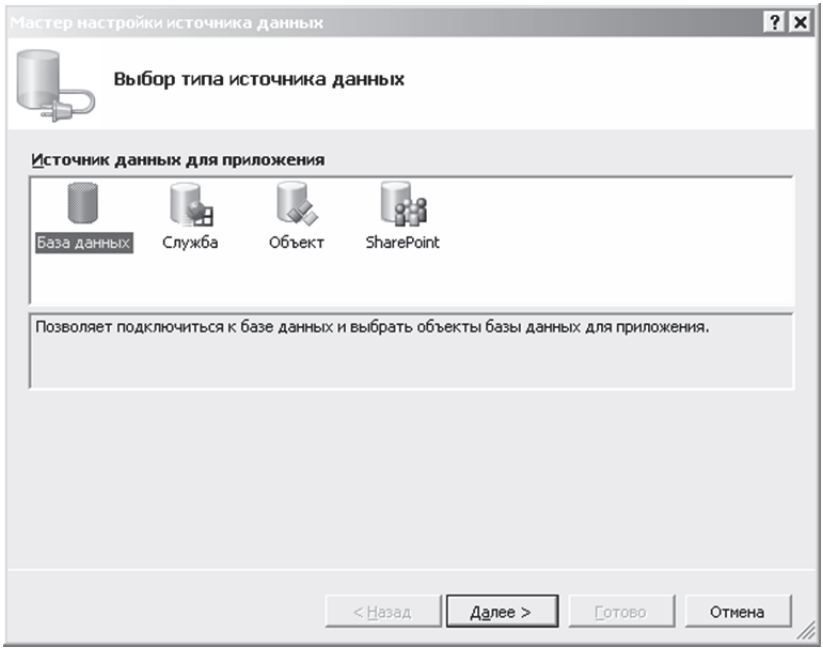


Рис. 3.10. Выбор источника данных

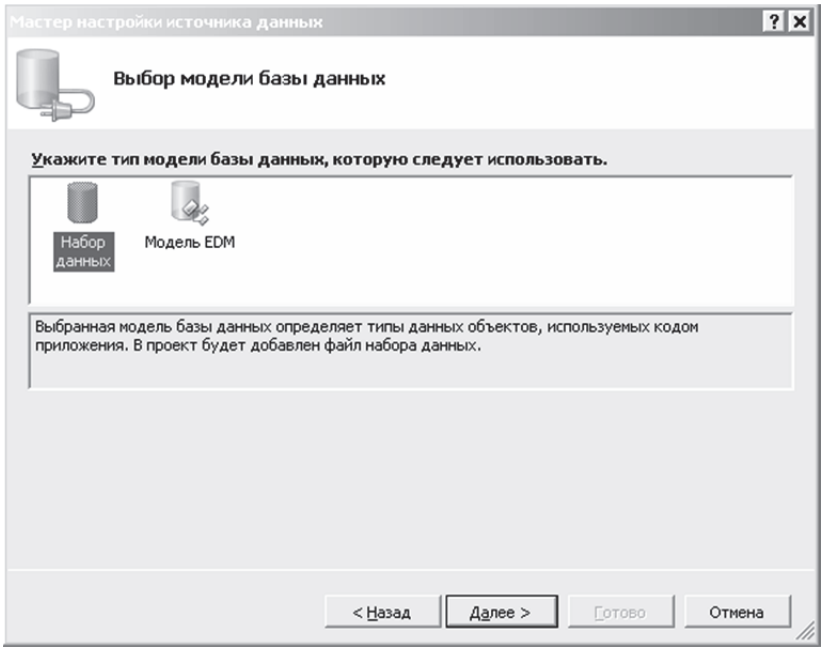


Рис. 3.11. Выбор модели данных

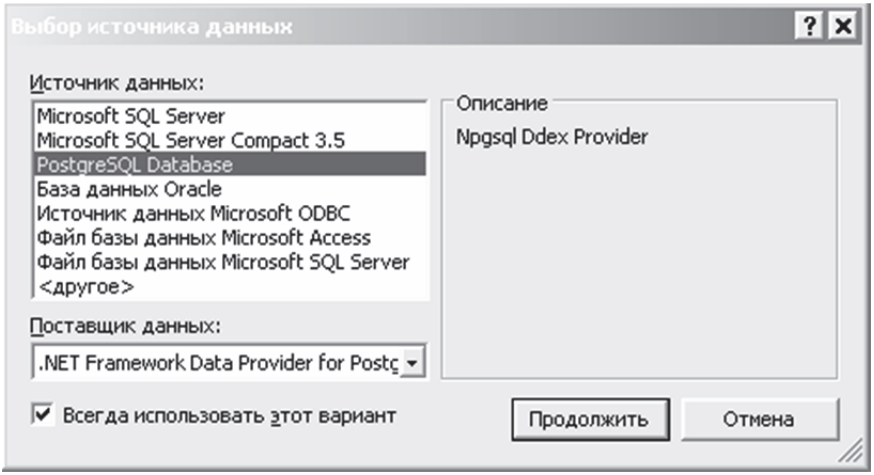
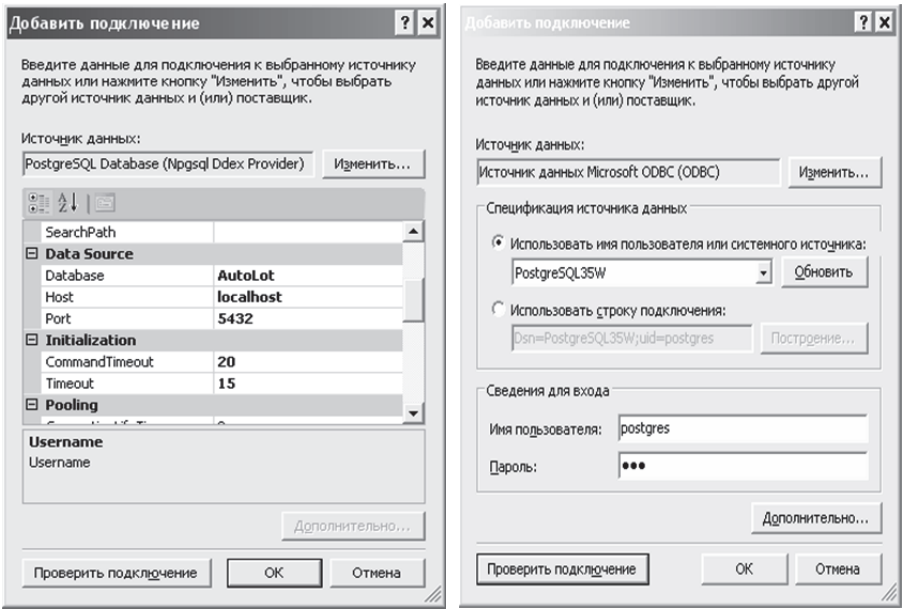


Рис. 3.12. Выбор поставщика данных

При выборе поставщика ODBC указывается созданный ранее объект ODBC и задаются параметры подключения (рис. 3.13).



а *б*

Рис. 3.13. Настройка параметров подключения:

а – для поставщика Npgsql; *б* – для поставщика ODBC

В проекте создаются объекты DataSet, InventoryBindingSource и InventoryTableAdapter (рис. 3.14).

При просмотре объекта InventoryBindingSource данные таблицы отображаются (рис. 3.15).

Стоит заметить, что последний способ подключения через компонент DataSet, по сути, является «оберткой» для подключения через ODBC или Npgsql. Подключение через компонент DataSet используется встроенными компонентами, например такими, как DataGridView.

Еще один немаловажный вопрос – это необходимость использовать источник данных. Его использование, безусловно, является большим плюсом, так как позволяет разработчику полностью абстрагироваться от типа СУБД.

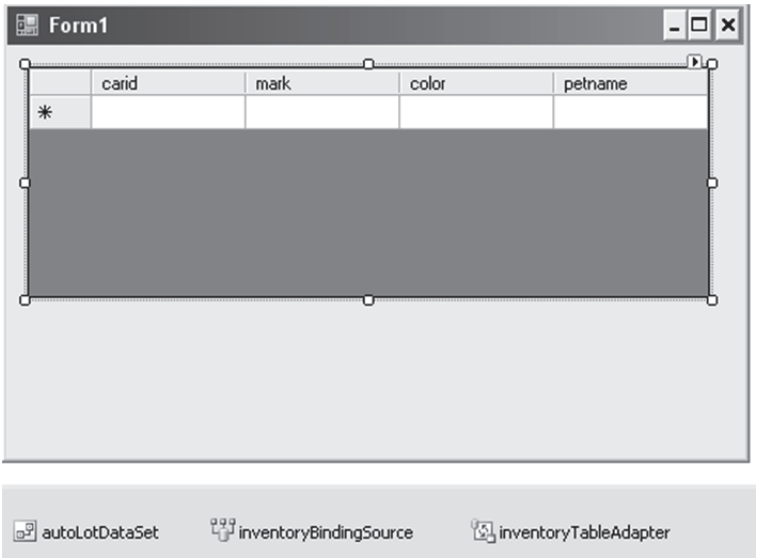


Рис. 3.14. Создание DataSet для объекта DataGridView

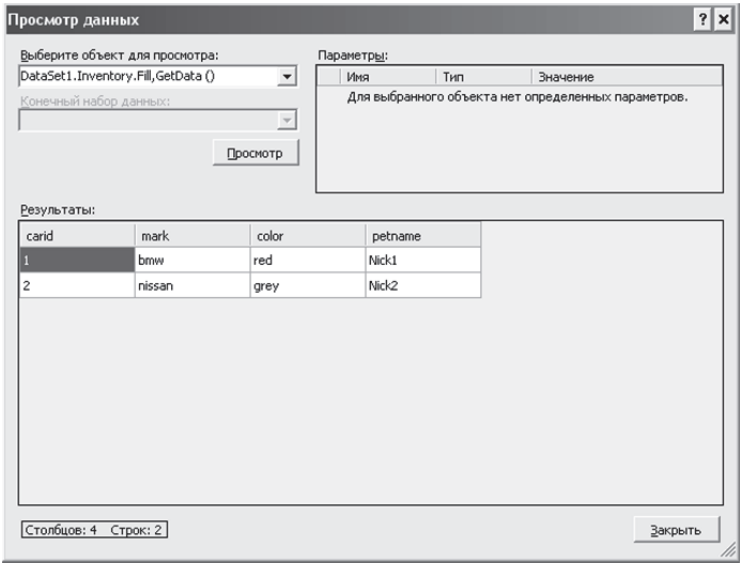


Рис. 3.15. Проверка правильности получения данных таблицы

Для объекта dataGridView можно указать, будут ли редактироваться, добавляться и удаляться строки таблицы. Также возможна сортировка по любому столбцу, выбор конкретной строки. Также имеется возможность переименования столбцов и редактирования шаблона таблицы.

3.10. Выполнение SQL-запросов к базе данных

После того как соединение с базой данных установлено, можно приступать к выполнению запросов. Стоит отметить, что запросы к базе данных можно разделить на две основные группы: запросы выборки и запросы модификации данных. Для каждого из этих типов существуют свои классы исполнения запросов.

Примеры выборки данных уже были рассмотрены в предыдущих примерах для двух поставщиков: ODBC и Npgsql.

Модификация данных через dataGridView осуществляется довольно просто, так как при создании объекта InventoryTableAdapter автоматически были сформированы параметризованные SQL-команды для удаления, обновления и добавления записей в таблицу. Достаточно добавить кнопку на форму Обновить и в обработчике события Save_Click вызвать метод Update (рис. 3.16).

```
private void Save_Click(object sender, EventArgs e)
 {
    this.inventoryTableAdapter.Update(this.dataSet2.inventory);
 }
```

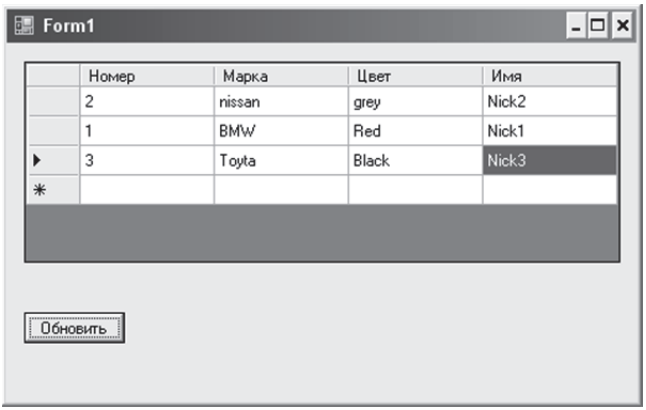


Рис. 3.16. Обновление данных в таблице inventory

Запрос на ввод данных. Добавим на форму поля для ввода данных (рис. 3.17).

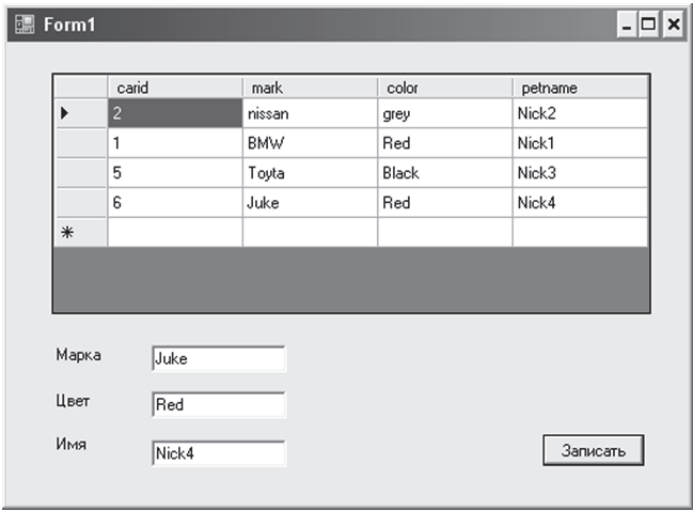


Рис. 3.17. Добавление данных в таблицу базы данных

Для записи данных в таблицу напишем обработчик для кнопки Записать:

```
private void Save_Click(object sender, EventArgs e)
{
    try
        {
            // создание подключения
            Connection();
            // Создание объекта команды запроса
            string strSQL = string.Format("INSERT INTO inventory
        (mark,color,petname) VALUES ('{0}','{1}','{2}')", textBox1.Text,
        textBox2.Text, textBox3.Text);
    NpgsqlCommand myCommand = new NpgsqlCommand(strSQL,
oCon);
    myCommand.ExecuteNonQuery();    //выполнение запроса
    oCon.Close();                   // закрытие подключения
```

```
        UpdateCrid();                    // обновление таблицы
        }catch (NpgsqlException) // обработка исключения записи в базу данных
    {
        label5.Text = "Ошибка записи в БД";
    }
}
void UpdateCrid()
{
    string ConnectionString =
   "Server=localhost;Port=5432;User=postgres;Password=123;Database=AutoLot;";
    string strSQL = "Select * From inventory";
    // Создание объекта NpgsqlDataAdapter
    // задаем строку запроса и строку подключения
    NpgsqlDataAdapter da = new NpgsqlDataAdapter(strSQL, ConnectionString);
    // Создание объекта DataSet и заполнение таблицы Inventory данными
    DataSet ds = new DataSet();
    da.Fill(ds, "inventory");
    // Отображение таблицы через таблицу DataGridView
    dataGridView1.DataSource = ds.Tables["inventory"].DefaultView;
    }
```

Запрос на изменение данных. Для выполнения функции изменения данных в таблице нужно выбрать запись. Данные выбранной строки отображаются в текстовых полях, предназначенных для изменения (рис. 3.18). Заполнение полей происходит в обработчике события объект DataGridView dataGridView1_SelectionChanged.

```
private void dataGridView1_SelectionChanged(object sender, EventArgs e)
    {
      // определение номера выделенной строки
```

```
DataGridViewSelectedRowCollection t =
dataGridView1.SelectedRows;
if (t.Count > 0)
{
   DataGridViewRow row = t[0];
   id = Convert.ToInt32(row.Cells[0].Value);
   textBox1.Text = Convert.ToString(row.Cells[1].Value).Trim();
   textBox2.Text = Convert.ToString(row.Cells[2].Value).Trim();
   textBox3.Text = Convert.ToString(row.Cells[3].Value).Trim();
   }
}
```

Ввести необходимые изменения в текстовые поля и нажать кнопку обновить (рис. 3.18). В обработчике события Update_Click происходит соединение с базой данных, выполнение SQL-запроса и обновление объекта DataGridView.

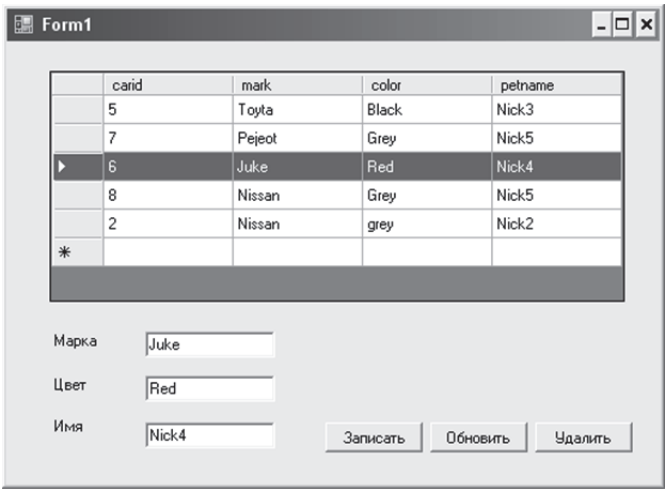


Рис. 3.18. Выбор записи для редактирования

```
private void Update_Click(object sender, EventArgs e)
{
   try
```

```
    {
    Connection();  // соединение с базой данных
    // Создание объекта команды на языке SQL.
    string strSQL = string.Format("UPDATE inventory SET mark='{0}',color='{1}',petname='{2}' where carid={3}", textBox1.Text, textBox2.Text, textBox3.Text, id);
    NpgsqlCommand myCommand = new NpgsqlCommand(strSQL, oCon);
    myCommand.ExecuteNonQuery();
    oCon.Close();       // закрытие подключения
    UpdateCrid();       // обновление таблицы
    textBox1.Text = "";   // очистка текстовых полей
    textBox2.Text = "";
    textBox3.Text = "";
    id=0;
    }catch (NpgsqlException) // обработка исключения записи в базу данных
  {
    label5.Text = "Ошибка записи в БД";
  }
  }
```

Запрос на удаление записи. Для удаления выбранной записи необходимо обработать событие Delete_Click:

```
  private void Delete_Click(object sender, EventArgs e)
  {
    try
    {
    Connection();  // соединение с базой данных
    // Создание объекта команды на языке SQL.
    string strSQL = string.Format("DELETE FROM inventory WHERE carid={0}", id);
    NpgsqlCommand myCommand = new NpgsqlCommand(strSQL, oCon);
```

```
myCommand.ExecuteNonQuery();
oCon.Close();       // закрытие подключения
UpdateCrid();       // обновление таблицы
textBox1.Text = "";   // очистка текстовых полей
textBox2.Text = "";
textBox3.Text = "";
} catch (NpgsqlException) // обработка исключения записи в базу данных
  {
    label5.Text = "Ошибка записи в БД";
  }
  }
```

Запрос на поиск данных. Для поиска нужно задать искомое значение в поле ввода. В обработчике события Search_Click анализируется указанное значение и формируется SQL-запрос по условию. В результате обновления объекта DataGridView в таблице отображаются все найденные соответствия (рис. 3.19).

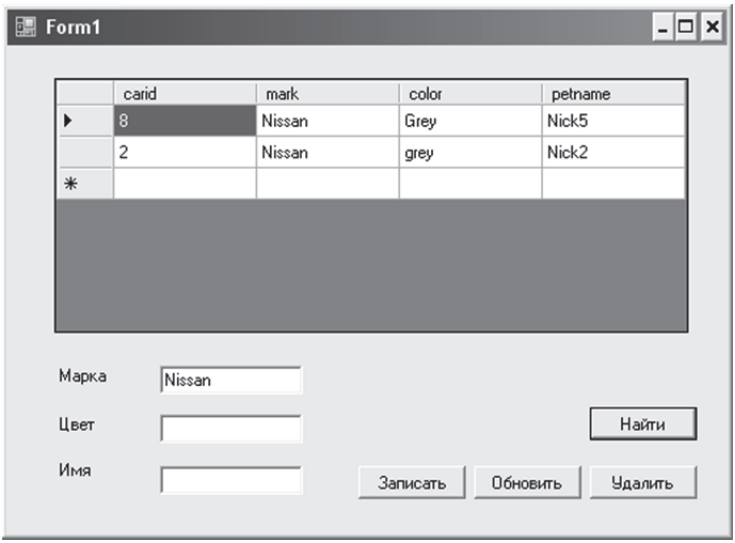


Рис. 3.19. Поиск по заданному значению и полю

```
 private void Search_Click(object sender, EventArgs e)
 {
     string strSQL = "";
     if (textBox1.Text != "") strSQL = string.Format("SELECT * FROM
inventory WHERE mark='{0}'", textBox1.Text);
     else if (textBox2.Text != "") strSQL = string.Format("SELECT *
FROM inventory WHERE color='{0}'", textBox2.Text);
     else if (textBox3.Text != "") strSQL = string.Format("SELECT *
FROM inventory WHERE retname='{0}'", textBox3.Text);
     if (strSQL != "")
  {
     string ConnectionString =

"Server=localhost;Port=5432;User=postgres;Password=123;Database=A
utoLot;";
     // Создание объекта NpgsqlDataAdapter
     // задаем строку запроса и строку подключения
     NpgsqlDataAdapter da = new NpgsqlDataAdapter(strSQL,
ConnectionString);
     // Создание объекта DataSet и заполнение таблицы inventory
данными
     DataSet ds = new DataSet();
   da.Fill(ds, "inventory");
     // Отображение таблицы через таблицу DataGridView
     dataGridView1.DataSource = ds.Tables["inventory"].DefaultView;
     }
  }
```

3.11. Работа с представлениями

Для создания представления используется объект DataView. В представлении можно объединить несколько таблиц одним SQL-запросом. Также можно построить несколько представлений в программе и при необходимости отображать их пользователю.

Добавим в базу данных еще одну таблицу, содержащую сведения о владельцах автомобилей.

```
CREATE TABLE owners (
 ownerid SERIAL,
 carid INTEGER,
 name CHAR(30),
 CONSTRAINT "OwnerID" PRIMARY KEY(ownerid)
);
```

Построим представление, содержащее сведения фамилия владельца → марка автомобиля.

```
    string ConnectionString =
    "Server=localhost;Port=5432;User=postgres;Password=123;Database
=AutoLot;";
 // Создание объекта NpgsqlDataAdapter
NpgsqlDataAdapter da = new NpgsqlDataAdapter("SELECT
owners.name, inventory.mark FROM inventory,owners WHERE
owners.carid=inventory.carid", ConnectionString);
 // Создание объекта DataTable и заполнение виртуальной таблицы
данными
 DataTable table = new DataTable();
 da.Fill(table);
 DataColumnCollection dd = table.Columns;
dd[0].ColumnName="Владелец";
dd[1].ColumnName = "Марка";
//Создание объекта DataView
DataView view = new DataView(table);
// view.RowFilter = "Владелец='Петров'"; – задать фильтрацию
данных
// вывод данных в BataGridView
 dataGridView1.DataSource = view;
```

Для отображения результата пользователю полученное представление свяжем через DataSource с элементом dataGridView1 (рис. 3.20).

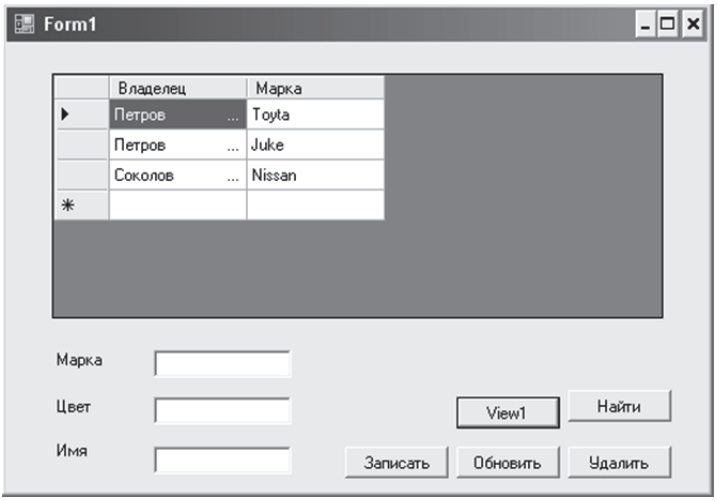


Рис. 3.20. Создание представления и вывод в DataGridView

3.12. Entity Framework

Платформа ADO.NET Entity Framework (EF) – это программная модель, которая связывает конструкции базы данных с объектно-ориентированными конструкциями. Используя EF, можно взаимодействовать с реляционными базами данных, не имея дело с кодом SQL (при желании).

Другой возможный подход состоит в том, чтобы вместо обновления базы данных посредством нахождения строки, обновления и отправки ее обратно на обработку в пакете запросов SQL, просто изменять свойства объекта и сохранять его состояние. И в этом случае исполняющая среда EF обновляет базу данных автоматически. Существенно выиграть от использования EF могут крупные приложения.

Общим подходом к разработке приложения или службы, обрабатывающие данные, хранимые в базе данных, представляет собой его разделение на три части: модель домена, логическую модель и физическую модель. Модель домена определяет сущности и связи в моделируемой системе. Логическая модель для реляционной базы данных обеспечивает нормализацию сущностей и связей в целях создания таблиц с ограничениями внешнего ключа. В физической модели учитываются возможности конкретной системы обработки данных путем

определения зависящих от ядра базы данных подробных сведений о хранении данных.

В EF каждый из этих трех уровней описывается в XML-файле *.edmx . Этот файл содержит XML- описания сущностей, физической базы данных и инструкции относительно того, как отображать эту информацию между концептуальной и физической моделями.

Ниже представлен пример модели базы данных AutoLot, содержащей таблицу inventory в XML-файле *.edmx:

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="2.0"
xmlns:edmx="http://schemas.microsoft.com/ado/2008/10/edmx">
<!-- EF Runtime content -->
<edmx:Runtime>
<!-- SSDL content -->
<edmx:StorageModels> – описание физической таблицы
                      inventory
<Schema Namespace="AutoLotModel.Store" Alias="Self"
Provider="Npgsql" ProviderManifestToken="9.3.5"
xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator"
xmlns="http://schemas.microsoft.com/ado/2009/02/edm/ssdl">
<EntityContainer Name="AutoLotModelStoreContainer">
<EntitySet Name="inventory"
EntityType="AutoLotModel.Store.inventory" store:Type="Tables"
Schema="public" />
</EntityContainer>
<EntityType Name="inventory">
<Key>
<PropertyRef Name="carid" />
</Key>
<Property Name="carid" Type="int4" Nullable="false"
StoreGeneratedPattern="Identity" />
<Property Name="mark" Type="bpchar" MaxLength="20" />
```

<Property Name="color" Type="bpchar" MaxLength="15" />

<Property Name="petname" Type="bpchar" MaxLength="20" />

</EntityType>

</Schema>

</edmx:StorageModels>

В этой части файла дано описание модели базы данных и ее таблиц в Entity Framework.

<!-- CSDL content -->

<edmx:ConceptualModels> – **описание концептуальной модели**

<Schema Namespace="AutoLotModel" Alias="Self" xmlns:annotation="http://schemas.microsoft.com/ado/2009/02/edm/annotation" xmlns="http://schemas.microsoft.com/ado/2008/09/edm">

<EntityContainer Name="AutoLotEntities" annotation:LazyLoadingEnabled="true">

<EntitySet Name="Cars" EntityType="AutoLotModel.Car" />

</EntityContainer>

<EntityType Name="Car">

<Key>

<PropertyRef Name="carid" />

</Key>

<Property Name="carid" Type="Int32" Nullable="false" annotation:StoreGeneratedPattern="Identity" />

<Property Name="mark" Type="String" MaxLength="20" Unicode="true" FixedLength="true" />

<Property Name="color" Type="String" MaxLength="15" Unicode="true" FixedLength="true" />

<Property Name="carnickname" Type="String" MaxLength="20" Unicode="true" FixedLength="true" />

</EntityType>

</Schema>

</edmx:ConceptualModels>

В этой части файла дано описание концептуальной модели базы данных и ее таблиц. Имена таблиц и полей в ней могут отличаться от

физической модели. Также типы данных соответствуют типам языка программирования.

<!-- C-S mapping content -->

<edmx:Mappings> – **описание отображения**

<Mapping Space="C-S"
xmlns="http://schemas.microsoft.com/ado/2008/09/mapping/cs">

<EntityContainerMapping StorageEntityContainer=
"AutoLotModelStoreContainer" CdmEntityContainer="AutoLotEntities">

<EntitySetMapping Name="Cars"><EntityTypeMapping
TypeName="AutoLotModel.Car"><MappingFragment
StoreEntitySet="inventory">

<ScalarProperty Name="carid" ColumnName="carid" />

<ScalarProperty Name="mark" ColumnName="mark" />

<ScalarProperty Name="color" ColumnName="color" />

<ScalarProperty Name="petname" ColumnName="petname" />

</MappingFragment></EntityTypeMapping></EntitySetMapping>

</EntityContainerMapping>

</Mapping>

</edmx:Mappings>

В этой части файла дано описание отображения концептуальной модели базы данных и ее таблиц на физическую.

</edmx:Runtime>

. . . . . .

</edmx:Edmx>

По завершении построения модели создаются классы:

- public partial class AutoLotEntities: ObjectContext{} – является точкой входа в программную модель EF.
- public partial class Inventory : EntityObject{} – сущностный класс, отображения на таблицу базы данных. Содержит определения полей таблицы и методы-свойства для доступа к ним.

Entity Framework направлен на то, чтобы дать приложениям возможность чтения и изменения данных, представленных в виде сущностей и связей в концептуальной модели. EF использует данные в модели и файлах сопоставления для преобразования запросов объектов к типам сущностей, представленным в концептуальной модели, в запро-

сы, зависящие от источника данных. Результаты запросов преобразуются в объекты, которыми управляют EF. Также платформа EF реализует следующие способы выполнения запросов к концептуальной модели и возврата объектов.

- LINQ to Entities. Обеспечивает поддержку запросов LINQ для выполнения запросов к типам сущности, которые определены в концептуальной модели.
- Entity SQL. Независимый от хранилища диалект SQL, который работает непосредственно с сущностями в концептуальной модели и поддерживает основные понятия EDM (модель данных с использованием сущностей). Entity SQL используется и с запросами объектов, и с запросами, выполняемыми с помощью поставщика EntityClient.

На рис. 3.21 показана архитектура доступа к данным в Entity Framework:

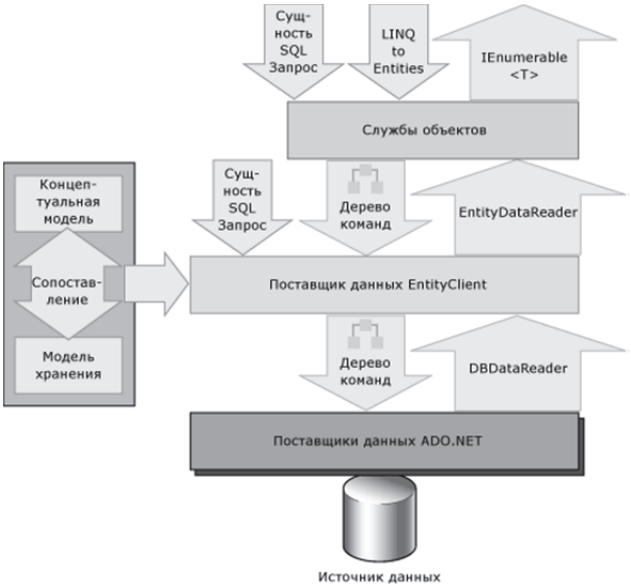


Рис. 3.21. Схема доступа к данным в Entity Framework

Для создания модели EDM (Entity Data Model – моделью сущностных данных) в Visual Studio.Net есть мастер модели EDM, который можно вызвать, кликнув правой кнопкой мыши по проекту в Обозревателе решений и выбрав Модель ADO.NET EDM. На следующей

странице мастера выбирается способ получения модели либо созданием из существующей базы данных, либо пустая модель (рис. 3.22).

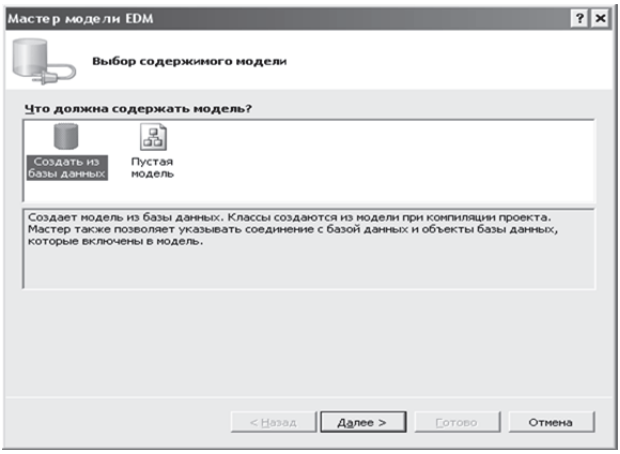


Рис. 3.22. Выбор способа получения модели

Далее создается подключение к базе данных (рис. 3.23):

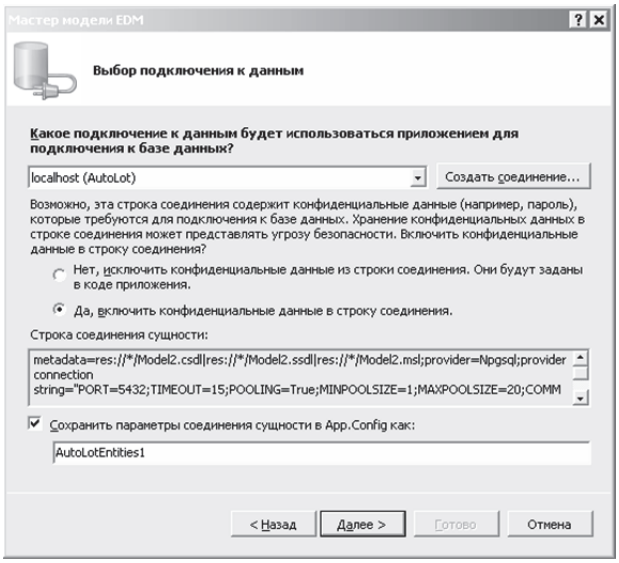


Рис. 3.23. Выбор подключения к данным

Затем выбираются все таблицы, хранимые процедуры и представления (рис. 3.24), которые должны войти в модель.

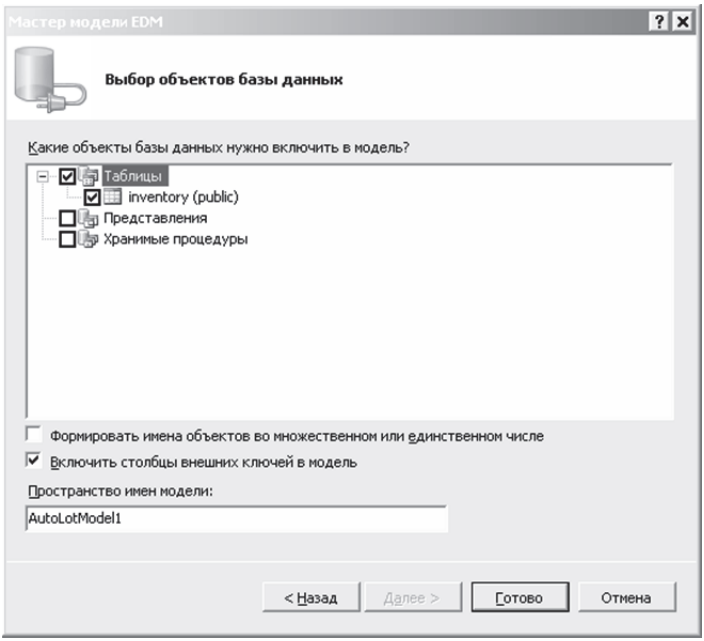


Рис. 3.24. Выбор объектов базы данных

Платформа EF включает в себя поставщик данных EntityClient. Поставщик управляет соединениями, переводит запросы сущностей в запросы, зависящие от источника данных, и возвращает модуль чтения данных, который используется EF для материализации данных сущности в виде объектов. Если материализация объектов не требуется, то поставщик EntityClient может также работать в качестве стандартного поставщика данных ADO.NET, позволяющий приложениям выполнять запросы Entity SQL и получать данные только для чтения, возвращаемые модулем чтения данных.

Сущности. Строго типизированные классы называются сущностями (entities). Сущности – это концептуальная модель физической базы данных, которая отображается на предметную область. Модель EDM представляет собой набор классов клиентской стороны, которые отображаются на физическую базу данных. Однако сущности не обяза-

тельно должны точно отображаться на схему базы данных. Сущностные классы можно реструктурировать, и исполняющая среда отобразит эти уникальные имена на схему базы данных. Пример создания сущности Inventory в Visual Studio.NET приведен на рис. 3.25. Сущность inventory можно переименовать, также можно определить уникальные имена свойств объекта, которые будут отображаться на нужные столбцы таблицы inventory.

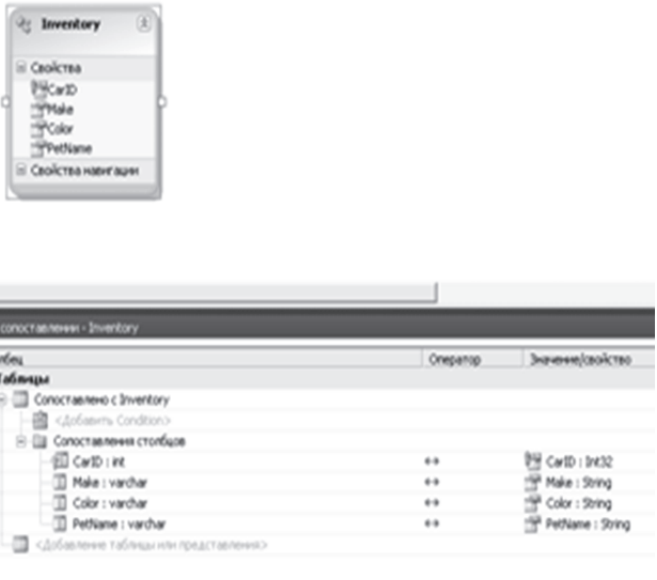


Рис. 3.25. Сущность Inventory и ее отображение на таблицу базы данных inventory

Используя созданные в примере сущностный класс Inventory и связанный с ним AutoLotEntities, можно добавить новые данные в таблицу базы данных. Класс AutoLotEntities называется контекстом объектов, и его назначение – обеспечение взаимодействия с физической базой данных.

```
public static void Main(string[] args)
{
// строка соединения считывается из сгенерированного конфигурационного файла
```

```
using (AutoLotEntities contex = new AutoLotEntities())
{
    contex. Inventories.AddObject(new Inventory() { CarID = 2345,
Color = "Red", Make = "Pinto", PetName = "Pepe" });
    contex.SaveChanges();
}
}
```

Поставщик данных имеет инфраструктуру, которая позволяет использование сборки System.Data.Entity.dll. Две основные части API-интерфейса EF – это службы объектов (object services) и клиент сущности (entity client).

Службы объектов. Под службами объектов подразумевается часть EF, которая управляет сущностями клиентской стороны при работе с ними в коде. Службы объектов отслеживают изменения, внесенные в сущность (например, смена цвета автомобиля с зеленого на синий), управляют отношениями между сущностями (скажем, просмотр всех заказов для клиента с заданным именем), а также обеспечивают возможности сохранения изменений в базе данных и сохранение состояния сущности с помощью сериализации (XML или двоичной). Уровень службы объектов управляет любым классом, расширяющим базовый класс **EntityObject.**

Клиент сущности. Эта часть API-интерфейса EF отвечает за работу с поставщиком данных ADO.NET для установки соединений с базой данных, генерации необходимых SQL-операторов на основе состояния сущностей и запросов LINQ, отображения извлеченных данных на корректные формы сущностей, а также управления прочими деталями, которые обычно приходится делать вручную, если не используется Entity Framework.

Функциональность уровня клиента сущности определена в пространстве имен System.Data.EntityClient. Это пространство имен включает набор классов, которые отображают концепции EF (такие, как запросы LINQ to Entity) на лежащий в основе поставщик данных ADO.NET. Эти классы (т. е. EntityCommand и EntityConnection) очень похожи на классы, которые можно найти в составе поставщика данных ADO.NET. Уровень клиента сущности обычно работает «за кулисами», но может взаимодействовать с клиентом сущности напрямую, если нужен полный контроль над его действиями (особенно SQL-запросами).

Если требуется более тонкий контроль над тем, как сущностный клиент строит SQL-оператор на основе входящего запроса LINQ, можно использовать Entity SQL. Это независимый от базы данных диалект SQL, который работает непосредственно с сущностями. Если требуется более высокая степень контроля над извлеченными данными, то можно вручную обрабатывать записи с помощью класса EntityDataReader.

Классы ObjectContext и ObjectSet<T>. Генерация файла *.edmx дает в результате сущностные классы, которые отображаются на таблицы базы данных, и класс, расширяющий ObjectContext.

ObjectContext предлагает набор базовых служб для дочерних классов, включая возможность сохранения всех изменений (которые в конечном итоге превращаются в обновление базы данных), настройку строки соединения, удаление объектов, вызов хранимых процедур и т. д. (табл. 3.6). Класс, унаследованный от ObjectContext, служит контейнером, управляющим сущностными объектами, которые сохраняются в коллекции типа ObjectSet<T> (табл. 3.7).

Таблица 3.6

Основные методы класса ObjectContext

AcceptAllChanges()	Принимает все изменения, проведенные в сущностных объектах внутри контекста объектов
AddObject()	Добавляет объект к контексту объектов
DeleteObject()	Помечает объект для удаления
ExecuteFunction<T>()	Выполняет хранимую процедуру в базе данных
ExecuteStoreCommand()	Позволяет отправлять команду SQL прямо в хранилище данных
GetObjectByKey()	Находит объект внутри контекста объектов по его ключу
SaveChanges()	Отправляет все обновления в хранилище данных
CommandTimeout	Свойство получает или устанавливает значение таймаута в секундах для всех операций контекста объектов
Connection	Свойство возвращает строку соединения, используемую текущим контекстом объектов
SavingChanges	Событие инициируется, когда контекст объектов сохраняет изменения в хранилище данных

Таблица 3.7

Основные методы класса ObjectSet<T>

AddObject()	Позволяет вставить объект в коллекцию
CreateObject<T>	Создает новый экземпляр указанного сущностного типа
DeleteObject()	Помечает объект для удаления

Для таблицы Inventory базы данных AutoLot получается класс с именем (по умолчанию) AutoLotEntities. Этот класс поддерживает свойство по имени Inventories, которое инкапсулирует член данных ObjectSet<Inventory>.

3.13. Реализация операций с таблицей через EDM

Выбор всех записей:

```
private static void PrintAll()
{
        using (AutoLotEntities contex = new AutoLotEntities())
    {
     foreach(Inventory avt in context.Inventories)
        Console.WriteLine(avt);
 }
}
```

Удаление записи:

```
private static void Remove()
{
    using (AutoLotEntities contex = new AutoLotEntities())
     {
 EntityKey = new EntityKey("AutoLotEntitis.Inventories", "carid",2);
// проверка существования записи. Возвращает ссылку на объект коллекции ObjectSet<T>
        Inventory avt = (Inventory) contex.GetObjectByKey(key);
        if(avt != null)
 {
```

```
            Contex.DeleteObject(avt);
            contex.SaveChanges();
        }
    }
}
```

Обновление записи:

```
private static void Update()
{
    using (AutoLotEntities contex = new AutoLotEntities())
 {
        EntityKey = new EntityKey("AutoLotEntitis.Inventories",
"carid",2);
        // проверка существования записи
        Inventory avt = (Inventory) contex.GetObjectByKey(key);
        if(avt != null)
 {
            avt.color = "Black";
            contex.SaveChanges();
        }
    }
}
```

Контрольные вопросы

1. Каким образом осуществляется доступ к данным в ADO.Net?
2. Что такое поставщик данных? Зачем введено это понятие?
3. Каких вы знаете поставщиков данных для работы с СУБД PostgreSQL?
4. Как организован доступ к данным базы данных?
5. Что значит работа с подсоединенными объектами? С отсоединенными объектами?
6. Как подключиться к базе данных? Какие существуют способы подключения?

7. Для чего используется источник данных ODBC? Как его создать средствами ОС Windows?

8. Как задать команду для выполнения работы с данными базы данных?

9. Какие есть методы выполнения команды и что они возвращают?

10. Как отключиться от базы данных?

11. Какие объекты используются при работе в автономном режиме?

12. Как организуется обновление физической базы данных в автономном режиме?

13. Зачем используется платформа ADO.NET Entity Framework? В чем отличие между применением этой платформы для доступа к данным и работы с классами поставщика данных?

14. Что такое модель EDM?

15. Что называют сущностями (entities)?

16. Какие существуют более тонко организованные способы формирования запросов к базе данных в EF?

4. БЕЗОПАСНОСТЬ БАЗ ДАННЫХ

Базы данных (БД) обычно используют для хранения различных данных, что налагает на использование баз определенные ограничения. Отдельные записи или таблицы могут быть видны только некоторым пользователям, и даже к таблицам, видимым всем, есть разные уровни допуска: не всем разрешено добавлять новые данные или изменять уже существующие. У каждого пользователя существуют определенные права доступа к таблицам и другим объектам БД, таким как схемы или функции.

Предпочтительнее не присваивать роли непосредственно пользователям, а использовать промежуточную роль (ROLE), для которой назначаются права. Затем она присваивается подходящим пользователям. Например, роль clerk может иметь права для вставки и обновления данных в таблице user_account, но только права для вставки данных в таблицу audit_log.

С другой стороны, требуется, чтобы только определенные люди могли иметь доступ к БД, чтобы пользователь не мог видеть, что делают в БД остальные (если, конечно, он не администратор или аудитор), чтобы пользователи не могли передавать права, которые были им даны.

Следует учитывать и еще один немаловажный аспект: сервер должен находиться в безопасном месте, и процедуры доступа к нему должны быть безопасными. Однако вопрос общей защиты БД, серверной машины и сети настолько обширен, что его сложно осветить в настоящем учебном пособии.

4.1. Управление правами пользователей

На работающем сервере все пользователи делятся как минимум на две группы: администраторы и конечные пользователи. Причем администраторы могут делать все (являются суперпользователями), а конечные пользователи могут совсем немного: как правило, изменять данные в нескольких таблицах и читать еще несколько таблиц.

Не очень разумно давать простым пользователям право создавать или изменять определения объектов БД, а значит, они не должны иметь права create для всех схем, включая public.

Для конечных пользователей существуют и другие роли. Аналитики, например, могут только делать выборку данных из одной таблицы или представления или выполнять несколько функций. Менеджер уполномочен только давать или отнимать права.

Для того чтобы забрать у пользователя права доступа к таблице, текущий пользователь должен быть суперпользователем, владельцем таблицы или иметь доступ grant для этой таблицы. Вы можете отнять права и у пользователя, который является суперпользователем.

Чтобы отнять все права на таблицу mysecrettabie у пользователя userwhoshouldnotseeit, необходимо выполнить следующую SQL- команду:

```
REVOKE ALL ON mysecrettabie FROM userwhoshouldnotseeit;
```

Однако таблица все еще остается открытой для пользователей через роль PUBLIC, поэтому следует также записать:

```
REVOKE ALL ON mysecrettabie FROM PUBLIC;
```

По умолчанию у всех пользователей есть права (SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES и TRIGGER) на все вновь созданные таблицы посредством специальной роли PUBLIC. Чтобы определенный пользователь больше не мог получить доступ к таблице, права на нее необходимо отнять и у этого пользователя, и у роли PUBLIC.

На рабочих системах лучше всего всегда включать выражения GRANT и REVOKE в скрипт создания БД, и тогда вы будете уверены, что доступ к этой базе получат только нужные пользователи. Если делать это вручную, можно что-нибудь забыть. Кроме того, таким образом вы удостоверитесь, что одни и те же роли используются для разработки и тестирования окружения, так что на стадии развертывания с этой стороны неожиданностей не будет.

Вот пример части скрипта для создания базы данных:

```
CREATE TABLE table1(…
) ;
REVOKE ALL ON table1 FROM GROUP PUBLIC;
GRANT SELECT ON table1 TO GROUP webreaders;
```

GRANT SELECT, INSERT, UPDATE, DELETE ON table1 TO editors;

GRANT ALL ON table1 TO admins;

При отъеме или назначении прав рекомендуется использовать полное имя, иначе может неожиданно оказаться, что вы работали не с той таблицей.

Чтобы посмотреть эффективный путь БД, выполните:

pguser=# show search_path ;

search_path

"$user".public (1 row)

Чтобы увидеть, на какую таблицу будет оказано влияние, если вы опустите имя схемы, запустите в psql:

pguser=# \d x

Table “public.x"

Column | Type | Modifiers

В результате будет получено полное имя таблицы public.x, включая схему.

Чтобы выполнять какие-либо действия над таблицей, пользователь должен иметь права доступа к ней. По умолчанию PostgreSQL дает всем полные права посредством роли PUBLIC, но, если настройки были изменены для усиления безопасности, при создании таблицы права на нее у роли PUBLIC могут быть отозваны.

Указанные ниже команды сначала дают роли полный доступ к схеме, права на просмотр (SELECT) и изменение (INSERT, UPDATE, DELETE), а затем назначают эту роль двум пользователям БД:

GRANT ALL ON someschema TO somerole;

GRANT SELECT, INSERT, UPDATE, DELETE ON someschema. sometable TO somegroup;

GRANT somerole TO someuser, otheruser;

В PostgreSQL нет требования одних прав для получения других: у пользователя могут быть таблицы только для записи, куда нужно вставлять данные, но нельзя их просматривать. Администратор обычно создает по этому принципу очередь сообщений: сотрудники отправляют данные одному из менеджеров, но не видят, что отправили другие.

С другой стороны, вы не сможете изменить или удалить сделанную вами запись. Это необходимо для аудита таблиц журнала, в котором фиксируются все изменения и который нельзя модифицировать.

Также нужен доступ к схеме: чтобы получить доступ к таблице, пользователь должен иметь доступ к схеме, содержащей эту таблицу:

```
GRANT USAGE ON SCHEMA someschema TO someuser;
```

Часто приходится давать группам пользователей схожие права на какие-либо объекты БД. Для этого сначала необходимо предоставить все права промежуточной роли (или разрешенной группе), а затем принять в эту группу избранных пользователей:

```
CREATE GROUP webreaders;
GRANT SELECT ON pages TO webreaders;
GRANT INSERT ON viewlog TO webreaders;
GRANT webreaders TO tim, bob;
```

Теперь и tim, и bob имеют права select на таблицу pages и insert на таблицу viewlog. Можно также добавлять права роли уже после принятия в нее пользователей. Таким образом, после

```
GRANT INSERT, UPDATE, DELETE ON comments TO webreaders;
```

у пользователей bob и tim появятся все эти права на таблицу comments.

В более ранних версиях PostgreSQL не было простого способа предоставить права для работы с двумя или несколькими объектами, кроме перечисления их в команде grant и revoke.

В версии 9.0 команды grant или revoke можно распространять на все однотипные объекты схемы:

```
GRANT SELECT ON ALL TABLES IN SCHEMA staging TO bob;
```

Но все еще нужно дополнительно отдельной командой давать права доступа к схеме.

Чтобы создавать новых пользователей, вы должны быть суперпользователем или иметь права createrole или createuser.

В командной строке запустите команду createuser и ответьте на несколько вопросов:

```
pguser@vhost:~$ createuser bob
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) y
Shall the new role be allowed to create more new roles? (y/n) n
```

pguser@vhost:-$ createuser tim

Shall the new role be a superuser? (y/n) y

Программа createuser – это просто обертка для выполнения команд SQL над кластером БД. Она подключается к базе данных postgres, задает вопрос, а затем выполняет команды SQL для создания пользователя. То же самое можно сделать, запустив SQL-команду CREATE USER:

CREATE ROLE bob WITH NOSUPERUSER INHERIT NOCREATEROLE CREATEDB LOGIN;

CREATE ROLE tim WITH SUPERUSER;

Проверка ролей пользователя:

pguser=# \du tim

List of roles Role name | Attributes | Member of

Tim | Superuser | {}

Начиная с версий 8.x, команды CREATE USER и CREATE GROUP фактически являются вариациями команды create role.

Выражение CREATE USER u; эквивалентно CREATE ROLE u LOGIN; а выражение CREATE GROUP g; равносильно CREATE ROLE g NOLOGIN;

Иногда необходимо временно отозвать у пользователя права на подключение, при этом не удаляя пользователя и не меняя его пароль.

Чтобы менять права доступа пользователей, вы должны быть суперпользователем или иметь право revoke (в этом случае вы не сможете менять права суперпользователей).

Временно запретить пользователю вход в систему можно так:

pguser=# alter user bob nologin;

ALTER ROLE

Чтобы вернуть пользователю возможность устанавливать соединение, выполните

pguser=# alter user bob login;

ALTER ROLE

В системном каталоге PostgreSQL ставится флажок, запрещающий пользователю вход в систему. При этом текущие соединения не разрываются.

Есть и другой способ не дать пользователю войти в систему. Вы можете установить число возможных соединений для этого пользователя (connection limit) равным 0:

```
pguser=# alter user bob connection limit 0;
ALTER ROLE
```

Если вы хотите разрешить пользователю bob устанавливать до 10 соединений одновременно, выполните:

```
pguser=# alter user bob connection limit 10;
ALTER ROLE
```

Кроме того, можно полностью снять ограничение числа возможных соединений:

```
pguser=# alter user bob connection limit -1;
ALTER ROLE
```

Если вы отозвали права на установление соединений у пользователей и хотите сейчас же отключить их, выполните следующее (для этого у вас должны быть права суперпользователя):

```
SELECT pg_terminate_backend(procpid)
        FROM from pg_stat_activity a
        JOIN pg_roles r ON a.usename = r.rolname AND not rolcanlogin;
```

В ранних версиях PostgreSQL, где нет функции pg_terminate_ backend (), вы можете в командной строке на сервере набрать в качестве пользователя postgres:

```
postgres@vhost: ~$ psql -t -c "\
select 'kill ' || procpid from pg_stat_activity a \
join pg_roles r on a.usename = r.rolname and not rolcanlogin;"\
| bash
```

и получить тот же результат.

В этом случае из запроса формируются команды kill, которые направляются затем оболочке для выполнения.

При попытке сбросить пользователя, работающего с таблицами или другими объектами БД, вы получите следующее сообщение об ошибке:

```
testdb=# drop user bob;
ERROR: role "bob" cannot be dropped because some objects depend on it
DETAIL: owner of table bobstable owner of sequence bobstable_id_seq
```

Проще всего не сбрасывать пользователя, а запретить пользователю устанавливать соединения:

pguser=# alter user bob nologin;

ALTER ROLE

Еще один плюс состоит в том, что при последующем аудите и тестировании вы будете знать, кто создал таблицу или откуда она взялась.

Если вам действительно нужно избавиться от пользователя, вы должны передать все, чем он владел, другому пользователю, так что выполните следующий запрос, который является дополнением PostgreSQL к стандарту SQL:

REASSIGN OWNED BY bob TO bobs_replacement;

При этом владение всеми объектами БД, которые принадлежали роли bob, будет передано роли bobs_replacement.

Однако для этого у вас должны быть права изменения обеих ролей, и необходимо повторить этот запрос во всех базах данных, где bob владеет какими-либо объектами, поскольку REASSIGN OWNED работает только в текущей базе данных.

Команда REASSIGN OWNED была добавлена в PostgreSQL в версии 8.2. Если ваша БД имеет более раннюю версию, вам придется поработать в командной строке Unix.

Для начала извлеките из дампа схемы назначения прав:

dbuser:~$ pg_dump -s mydatabase I grep -i "alter.* owner to bob"

ALTER FUNCTION public.somefunction() OWNER TO bob;

ALTER TABLE public.directory OWNER TO bob;

ALTER TABLE public.directory_seq OWNER TO bob;

ALTER TABLE public.document_id_seq OWNER TO bob;

ALTER TABLE public.documents OWNER TO bob;

Затем просто замените в полученном результате bob на нового пользователя и передайте команды обратно базе данных:

dbuser: ~$ pg_dump -s mydb I grep -i "owner to bob" > tmp. sql

dbuser:~$ sed -e 's/TO bob/TO bobs_replacement/' < tmp.sql

| psql mydb

И хорошо бы посмотреть сначала на измененные данные:

dbuser:~$ pg_dump -s mydb I grep -i "owner to bob" > tmp.sql dbuser:~$ sed -e 's/TO bob/TO bobs_replacement/' < tmp.sql >tmp2.sql

dbuser:~$ less tmp2.sql dbuser:~$ psql mydb < tmp2.sql

Как и в случае с командой REASSIGN OWNED, это работает каждый раз только для одной базы данных, так что следует все повторить для каждой БД, если в ней находятся объекты, принадлежащие роли, которую вы хотите удалить.

4.2. Проверка безопасности паролей пользователей

В PostgreSQL нет встроенной возможности проверки паролей. Максимум, что можно сделать, убедиться, что все пароли пользователей зашифрованы и что файл pg_hba.conf не позволяет подключаться с обычным паролем. То есть в качестве метода аутентификации пользователей всегда используйте MD5.

Для подключения клиентских приложений из доверенных частных сетей, реальных или виртуальных (VPN), можно использовать доступ по узлу (host based access), если вы уверены, что машину, на которой запущено приложение, не используют подозрительные личности. Для удаленного доступа из публичных сетей лучше применять клиентские сертификаты SSL.

Узнать, у каких пользователей незашифрованные пароли, можно с помощью запроса:

```
test2=# select usename, passwd from pg_shadow where passwd not like
'md5%' or length(passwd) <> 35;
usename | passwd
tim         weakpassword
asterisk   md5chicken
(2 rows)
```

Пользователей с зашифрованными паролями показывает следующий запрос:

```
test2=# select usename, passwd from pg_shadow where passwd like
'md5%' and length(passwd) = 35;
usename | passwd
bob2 |    md518cf038878cd04fa207e7f5602013a36
(1 row)
```

Шифрование паролей БД – это еще только половина дела. Гораздо хуже то, что неизвестно, действительно ли пользователи выбирают пароли, которые трудно угадать. Например, «password», «secret» или

«test», да и любые общеупотребительные слова, являются не очень хорошими паролями.

Если вы не верите, что пользователь может придумать хороший пароль, напишите приложение-оболочку, которое будет проверять качество пароля при его изменении.

У роли суперпользователя есть такие права, которые можно назначить и другим ролям, не являющимся суперпользователями.

Например, роли bob предоставляется возможность создавать новые базы данных:

```
ALTER ROLE BOB WITH CREATEDB;
```

Для создания новых пользователей понадобится назначить право CREATEUSER:

```
ALTER ROLE BOB WITH CREATEUSER;
```

Однако обычным пользователям можно давать и более узкие права и контролируемый доступ к некоторым действиям, зарезервированным для суперпользователей, с помощью функций SECURITY DEFINER. Аналогично можно передавать часть прав одного пользователя другому.

Для начала вам необходимо иметь права суперпользователя БД. Предположим, что вы – суперпользователь по умолчанию postgres.

Продемонстрируем два случая назначения обычному пользователю прав, доступных только суперпользователям.

База данных должна поддерживать внутренний язык PL/pgSQL. Начиная с версии 9.0 рекомендуется по умолчанию устанавливать поддержку PL/pgSQL во все новые базы данных, однако эту опцию могли изменить создатели пакетов или администраторы. Если поддержки нет, то как суперпользователь PostgreSQL выполните следующее:

```
test2=# CREATE LANGUAGE plpgsql;
CREATE LANGUAGE
```

Одна из возможностей суперпользователей, которой нет у обычных пользователей, дать указание postgres копировать табличные данные из файла:

```
pguser@vhost:~$ psql -U postgres test2
test2=# create table lines(line text);
CREATE TABLE
test2=# copy lines from '/home/bob/names. txt';
```

```
COPY 37
test2=# SET SESSION AUTHORIZATION bob;
SET
test2=> copy lines from ’/home/bob/names.txt';
ERROR: must be superuser to COPY to or from a file
HINT: Anyone can COPY to stdout or from stdin. psql's \copy
Command also works for anyone.
```

Чтобы пользователь bob мог копировать данные непосредственно из файла, суперпользователь может написать для него специальную функцию-обертку:

```
create or replace function copy_from(tablename text, filepath text)
returns void security definer as $$
declare
begin
execute 'copy ' || tablename || ' from ' ' ' || filepath || ' ' ' ';
end;
$$ language plpgsql;
```

И хорошо бы сделать так, чтобы эту функцию мог применять только определенный пользователь:

```
REVOKE ALL ON FUNCTION copy_from( text, text)
FROM PUBLIC;
GRANT EXECUTE ON FUNCTION copy_from( text, text) TO bob;
```

Кроме того, можно проверить, чтобы bob копировал файлы только из своей домашней директории.

Когда вызывается функция с security definer, Postgres меняет права сессий для пользователя, который определил ее при выполнении.

Таким образом, когда bob вызывает функцию copy_from(tablename, filepath), он фактически на время ее выполнения становится суперпользователем.

Это похоже на функцию setuid в Unix-системах, где можно сделать так, чтобы все, у кого есть право исполнения программы, запускали ее на правах ее владельца. Риски при этом тоже сходные.

Есть и другие операции, зарезервированные только для суперпользователей PostgreSQL, например, настройка некоторых параметров.

Некоторые параметры, контролирующие запись в журнал, могут определяться только суперпользователями.

Если вы хотите дать своим разработчикам право настраивать добавление записей в журнал, можно написать такую функцию:

```
create or replace function debugging_info_on()
returns void
security definer
as $$
begin
        set client_min_messages to 'DEBUG1';
        set log_min_messages to 'DEBUG1';
        set log_error_verbosity to 'VERBOSE';
        set log_min_duration_statement to 0;
end;
$$ language plpgsql;
revoke all on function debugging_info_on() from public;
grant execute on function debugging_info_on() to bob;
```

После этого можно вернуться к первоначальному варианту, присвоив всем переменным значения по умолчанию:

```
create or replace function debugging_info_reset ()
returns void
security definer
as $$
begin
    set client_min_messages to DEFAULT;
    set log_min_messages to DEFAULT;
    set log_error_verbosity to DEFAULT;
    set log_min_duration_statement to DEFAULT;
end;
$$ language plpgsql;
```

В данном случае нет необходимости манипулировать с GRANT и REVOKE, поскольку в возвращении значений по умолчанию нет ника-

кого риска. Вместо SET xxx to DEFAULT можно использовать более короткую версию RESET XXX. Или просто завершите сессию, так как эти параметры действуют только для текущей сессии.

4.3. Аудит изменений DDL

Ниже описывается, как собрать из журналов БД DDL (Data Definition Language), чтобы отследить изменения структуры базы данных.

В файле postgresql.conf укажите: log_statement = 'ddl'

Можно также указать значение mod или all. И не забудьте переподгрузить новую конфигурацию:

/etc/init-d/postgresql reload

Найдите в журнале все упоминания команд create, alter и drop.

postgres@vhost:~$ egrep -i "create | alter | drop" \
/var/log/postgresql/postgresql-9.4-main.log

Если имеется ротация журналов, то необходимо также применить grep (быстрый просмотр) для всех журналов.

Если журнал слишком старый и вы не сохранили предыдущие журналы где-то в другом месте, то вам не повезло.

По умолчанию в postgresql.conf по поводу ротации указано:

log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
log_rotation_age = 1d log_rotation_size = 10MB

Проверьте это, если думаете, что вам могут понадобиться журналы более чем недельной давности.

Вполне возможно, что некоторые изменения отражены в журнале, но их не видно в структуре базы данных. Большую часть DDL-команд PostgreSQL можно «откатить» (ROLLBACK), так что в журнале окажется просто список команд, выполненных PostgreSQL, но не зафиксированных. Журнал зависит не от транзакций, поэтому в нем хранятся и команды, к которым применили ROLLBACK.

Чтобы знать, какой пользователь совершил DDL-изменения, необходимо их также записывать в журнал.

Это можно обеспечить, включив в параметр log_line_prefix строку в формате %u. Минимальный рекомендованный формат log_line_prefix для отслеживания DDL - %t %u %d; при этом каждая строчка журнала будет начинаться с указания времени, пользователя и базы данных.

Если ведение журнала отключено или у вас есть не все журналы, то некоторую часть информации о том, кто и когда изменял схему БД, можно получить из системных таблиц, хотя это не самые надежные сведения.

Что вы действительно узнаете, так это владельца объекта БД (таблицы, последовательности, функции и т. п.), но владелец мог быть изменен с помощью ALTER TABLE ... SET OWNER to yyyy.

По идентификатору транзакции в столбце xmin в системных таблицах pg_class и pg_attrib можно определить примерное время создания объекта или время последнего изменения. Попробуйте найти близкие xmin в какой-нибудь таблице, в которой автоматически записывается время вставки; возможно, в каком-либо столбце определено default current_timestamp.

4.4. Аудит изменений данных

Сначала вам нужно решить некоторые вопросы.

- Нужно отслеживать все изменения или только некоторые?
- Какая информация об изменениях вам нужна? Возможно, только факт изменений?
- Если есть новое значение поля или кортежа, нужно ли знать старое?
- Достаточно ли записать пользователя, который произвел изменения, или необходима информация о его IP-адресе и других настройках соединения?
- Насколько безопасной (защищенной) должна быть информация о проведенном аудите? Например, есть ли необходимость хранить ее отдельно от исследуемой базы данных?

В зависимости от ответов можно выбрать метод, который подходит именно вам.

Чтобы получить информацию об изменении данных из журнала сервера, выполните следующее.

- укажите log_statement = 'mod' или 'all';
- выберите в журнале все команды INSERT, UPDATE, DELETE и TRUNCATE;
- или укажите способ хранения журналов на сервере БД или копирования на сторонний узел.

Чтобы собрать информацию с помощью триггеров, выполните следующие шаги.

1. Напишите функцию-триггер для сбора новых (а если нужно, то и старых) значений кортежей и сохранения их в специальных таблицах для аудита.

2. Добавьте эти триггеры к таблицам, которые необходимо отслеживать.

Ниже приведен несколько измененный пример выполнения функции «A PL/pgSQL Trigger Procedure For Auditing» из документации по PostgreSQL.

```
CREATE TABLE emp_audit(
text    NOT NULL,
timestamp NOT NULL,
text    NOT NULL,
text    NOT NULL,


CREATE OR REPLACE FUNCTION process_emp_audit () RETURNS
TRIGGER AS
$emp_audit$
BEGIN
  IF (TG_OP = 'DELETE') THEN
     INSERT INTO emp_audit SELECT 'DEL', now(), user, OLD.*;
  ELSIF (TG_OP = 'UPDATE') THEN
     -- save old and new values
     INSERT INTO emp_audit SELECT 'OLD', now(), user, OLD.*;
     INSERT INTO emp_audit SELECT 'NEW', now(), user, NEW.*;
  ELSIF (TG_OP = 'INSERT') THEN
     INSERT INTO emp_audit SELECT 'INS', now(), user, NEW.*;
ELSIF (TG_OP = 'TRUNCATE') THEN
     INSERT INTO emp_audit SELECT 'TRUNCATE', now(), user, -1;
END IF;
RETURN NULL; – result is ignored bacause this is an AFTER trigger
END;
$emp_audit$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER emp_audit
    AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit () ;
CREATE TRIGGER emp_audit_truncate
    AFTER TRUNCATE ON emp
  FOR EACH STATEMENT EXECUTE PROCEDURE process_emp_
  audit();
```

Для некоторых систем, требующих особой защиты, не подходит хранение журнала аудита на одной машине со всеми основными данными. В таком случае можно настроить удаленную запись изменений. Это можно сделать единственным способом: с помощью pl /proxy отправлять журнал изменений в другую БД.

Рассмотрим пример, как можно вести журнал для предыдущего примера в удаленной базе данных auditdb.

1. Создайте в удаленной БД таблицу для журнала emp_audit.

2. Создайте в удаленной БД функцию log_emp_audit ():

```
CREATE FUNCTION log_emp_audit(operation text, userid text,
empname text, salary integer ) RETURNS VOID AS $$
INSERT INTO emp_audit VALUES($1, now(), $2, $3, $4)
$$ LANGUAGE SQL;
```

3. Создайте прокси-функцию для log_emp_audit () в локальной БД (для этого необходимо сначала установить язык pl/proxy):

```
CREATE OR REPLACE FUNCTION log_emp_audit(
operation text, userid text, empname text, salary integer )
RETURNS VOID AS $$
CONNECT 'dbname=auditdb';
$$ LANGUAGE plproxy;
```

4. Создайте функции-триггеры, которые будут использовать прокси-функцию для сохранения данных во внешней БД:

```
CREATE OR REPLACE FUNCTION do_emp_audit () RETURNS
TRIGGER AS $$
BEGIN
IF (TG_OP = 'DELETE') THEN PERFORM log_emp_audit (' DEL' ,
user, OLD.empname, OLD.salary);
```

```
ELSIF (TG_OP = 'UPDATE') THEN -- save old and new values
PERFORM log_emp_audit ('OLD', user, OLD.empname, OLD.salary);
PERFORM log_emp_audit ('NEW', user, NEW.empname, NEW.salary);
ELSIF (TG_OP = 'INSERT') THEN
PERFORM log_emp_audit (' INS' , user, NEW.empname, NEW.salary);
END IF;
RETURN NULL; -- result is ignored since this is an AFTER trigger
END;
$$ LANGUAGE plpgsql;
```

5. Добавьте триггеры в таблицу emp:

```
CREATE TRIGGER emp_remote_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
FOR EACH ROW EXECUTE PROCEDURE do_emp_audit() ;
```

Убедитесь в том, что база данных аудита защищена. Проследите, чтобы единственным возможным действием пользователя audit, logger был вызов функции log_emp_audit ().

4.5. Шифрование секретных данных

Для начала узнайте, не запрещено ли шифрование в стране, где вы проживаете или где находится ваш сервер БД. Убедитесь в том, что на узле, где расположена ваша база данных, установлен pgcrypto. Например, в Ubuntu он поставляется с пакетом postgresql-contrib.

Установите его в БД, в которой будете его использовать.

```
psql mydb < /usr/share/postgresql/9.4/contrib/pgcrypto.sql
```

Кроме того, необходимо будет настроить GPG-ключи:

```
pguser@laptop:~$ gpg --gen-key
```

Здесь ответьте на несколько вопросов, выберите тип ключа DSA and Elgamal и введите пустой пароль.

Теперь экспортируйте ключи:

```
pguser@laptop:~$ gpg -a --export "PostgreSQL User (test key for PG Cookbook) <pguser@somewhere.net>" > public.key
pguser@laptop:~$ gpg -a --export-secret-keys * PostgreSQL User (test key for PG Cookbook) <pguser@somewhere.net>" > secret.key
```

Убедитесь в том, что доступ к секретному ключу имеете только вы и пользователи базы данных postgres.

pguser@laptop:-$ sudo chgrp postgres secret.key pguser@laptop:-$ chmod 440 secret.key pguser@laptop:~$ Is -l *.key

-rw-r--r-- 1 pguser pguser 1718 2010-03-26 13:53 public.key

-r--r----- 1 pguser postgres 1818 2010-03-26 13:54 secret.key

Чтобы секретные ключи не были видны в журналах БД, напишите функцию-обертку, которая будет вытаскивать их из файлов. Это необходимо делать на надежном внутреннем языке, например pl/pythonu, так как только на доверенном языке можно получить доступ к файловой системе. Для создания таких функций необходимо быть суперпользователем PostgreSQL.

```
create or replace function get_my_public_key () returns text as $$
return open(' /home/pguser/public.key').read()
$$
language plpythonu;
revoke all on function get_my_public_key () from public;
create or replace function get_my_secret_key() returns text as $$
return open( ' /home/pguser/secret .key').read()
$$
language plpythonu;
revoke all on function get_my_secret_key() from public;
```

Если вы не хотите, чтобы пользователи могли видеть фактические ключи, необходимо написать функции для шифрования и дешифровки, а затем дать к ним доступ конечным пользователям. Функция шифрования работает так:

```
create or replace function encrypt_using_my_public_key( cleartext text,
ciphertext out bytea)
AS $$
DECLARE
pubkey_bin bytea;
BEGIN
– text version of public key needs to be passed through function
dearmor() to get to raw key
```

```
pubkey_bin : = dearmor (get_my_public_key ()) ; ciphertext := pgp_pub_encrypt (cleartext, pubkey_bin);
END;
$$ language plpgsql security definer;
revoke all on function encrypt_using_my_public_key(text) from public;
grant execute on function encrypt_using_my_public_key(text) to bob;
```

Функция дешифровки выполняется так:

```
create or replace function decrypt_using_my_secret_key( ciphertext bytea, cleartext out text)
AS $$
DECLARE
secret_key_bin bytea;
BEGIN
-- text version of secret key needs to be passed through function dearmor() to get to raw binary key
secret_key_bin : = dearmor (get_my_secret_key ()) ;
cleartext := pgp_pub_decrypt (ciphertext, secret_key_bin);
END;
$$ language plpgsql security definer;
revoke all on function decrypt_using_my_secret_key(bytea)
from public;
grant execute on function decrypt_using_my_secret_ key(bytea) to bob;
```

Теперь протестируем шифрование:

```
test2=# select encrypt_using_my_public_key('X marks the spot!');
```

В результате получаем следующее:

```
encrypt_using_my_public_key | \301\301N\003\22
3o\215\2125\203\252;\020\007\376-z\233\211H..
```

Теперь проверим, как это работает в обоих направлениях:

```
test2=# select decrypt_using_my_secret_key (encrypt_using_ my_public_key('X marks the spot!'));
 decrypt_using_my_secret_key
```

X marks the spot!

(1 row)

Все правильно: получили первоначальную строку.

Что было сделано:

1) скрыты ключи от пользователей БД, которые не являются суперпользователями;

2) предоставлена оболочка для авторизованных пользователей, чтобы использовать шифрование и дешифровку.

Чтобы защитить секретные данные при передаче между клиентом и БД, необходимо соединяться с PostgreSQL только по SSL или с локального узла.

В некоторых случаях лучше не хранить пароль для дешифровки и зашифрованные данные на одной и той же машине. Целесообразней использовать шифрование с открытым/закрытым ключом и на стороне сервера делать только шифрование. Это означает, что на сервере будет храниться только ключ для шифрования, но не будет ключа для дешифровки. С другой стороны, вы можете встроить в вашу систему серверов отдельный сервер, который будет заниматься только шифрованием/дешифровкой.

Если информация является еще более секретной и с машины клиента нельзя передавать данные в незашифрованном виде, понадобится шифровать их перед отправкой БД. В таком случае PostgreSQL будет получать информацию сразу в зашифрованном виде, и в обычном он ее не увидит. Это также означает, что вы сможете использовать индексы ТОЛЬКО ДЛЯ выполнения WHERE encrypted_column = encrypted_data и для проверки уникальности. Однако даже WHERE = пригодится лишь в том случае, когда алгоритм шифрования каждый раз шифрует один и тот же текст одинаково, что ослабляет защиту.

Контрольные вопросы

1. Каким образом можно выполнить проверку ролей пользователя?
2. Как получить информацию об изменении данных из журнала сервера?
3. Какие роли можно определить для конечных пользователей?
4. С помощью какой команды можно отнять все права на таблицу у пользователя?
5. Кто имеет право создавать новых пользователей?

6. Каким образом можно временно запретить пользователю вход в систему?

7. С помощью каких команд можно узнать; у каких пользователей незашифрованные пароли? зашифрованные пароли?

8. Какие возможности предоставляет функция SECURITY DEFINER?

9. Каким образом можно предоставить разработчикам право настраивать добавление записей в журнал?

10. На какие группы можно разделить всех пользователей?

11. Кто изменяет права доступа пользователей?

12. Как можно настроить удаленную запись изменений?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Троелсен Э.* Язык программирования C# 5.0 и платформа .NET 4.5 / Э. Троелсен. – М.: ООО «И.Д. Вильямс», 2013. – 1312 с.

2. *Рихтер Дж*. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.0 на языке C# / Дж. Рихтер. – Спб.: Питер, 2012. – 928 с.

3. *Стэкер Мэтью А.* Разработка клиентских Windows – приложений на платформе Microsoft .NET Framework: учебный курс Microsoft / Мэтью Стэкер, Стивен Дж. Стэйн, Тони Нортроп; пер. с англ. – М.: Издательство «Русская редакция»; Спб.: Питер, 2008. – 624 стр.

4. *Павловская Т.А*. C#. Программирование на языке высокого уровня. учебник для вузов / Т.А. Павловская. – СПб.: Питер, 2007. – 432 с.: ил.

5. *Постолит А.В.* Visual Studio .NET: разработка приложений баз данных / А.В. Постолит. – СПб.: БХВ-Петербург, 2003. — 544 с.

6. Руководство по разработке для .NET Framework [Электронный ресурс]. URL: https://msdn.microsoft.com/ru-ru/library/hh156542%28v=vs.110%29.aspx (дата обращения: 20.04.2015).

7. Общие сведения об ADO.NET [Электронный ресурс]. URL: – https://msdn.microsoft.com/ru-ru/library/h43ks021%28v=vs.110%29.aspx (дата обращения: 20.04.2015).

8. *Уорсли Дж.* PostgreSQL. Для профессионалов / Дж. Уорсли, Дж. Дрейк. – СПб.: Питер, 2003. – 496 с.

9. *Саймон Ригс.* Администрирование PostgreSQL 9. Книга рецептов / Ригс Саймон, Кросинг Ханну; пер. с англ.: Самохвалова Е.В. – М.: ДМК Пресс, 2012. – 368 с.

10. *Смирнов С.Н*. Безопасность систем баз данных / С.Н. Смирнов. – М.: Гелиос АРВ, 2007. – 382 с.