RedDatabaseLab

Выпуск 0.2

Роман Симаков

апр. 01, 2024

Оглавление

Глава 1

Лабораторные работы

1.1 Проектирование БД

1.1.1 Цель

Научится проектировать БД используя RedExpert и СУБД Ред База Данных.

1.1.2 Порядок выполнения

1. Установить СУБД Ред База Данных (https://youtu.be/6xPJtHTNVak)
2. Установить RedExpert
3. Создать базу данных
4. Создать БД телефонный справочник, которая включает в себя несколько таблицы со столбцами различных типов данных, первичные и внешний ключи.
5. Заполнить каждую таблицу тестовыми данными.

1.1.3 Справочная информация

1. В учебной виртуальной машине RedExpert и СУБД Ред База Данных уже установлены.
2. Перед созданием БД подготовьте папку, в которой будет размещаться файл БД. Владельцем папки необходимо сделать пользователя *firebird* (или *reddatabase* для СУБД Ред База Данных 5 и выше). Например следующий образом в терминале от пользователя *root*:

```
mkdir /db
chown reddatabase. /db
```

3. Создание БД осуществляется в пункте “База Данных”, “Создать”. Из необходимых параметров указать:

a. Драйвер - *Jaybird*

b. Файл базы данных в созданной папке, например, */db/phones.fdb*

c. Пользователь - *sysdba*

d. Пароль - *masterkey*

e. Кодировка - *UTF8*

4. Для создания объектов необходимо выполнить подключение к созданной БД и развернуть дерево объектов. Далее нажимая правой кнопкой мыши в контекстном меню выбирать нужные пункты (например, Создать таблицу).

5. Внешние ключи указываются на вкладке “Ограничения”.

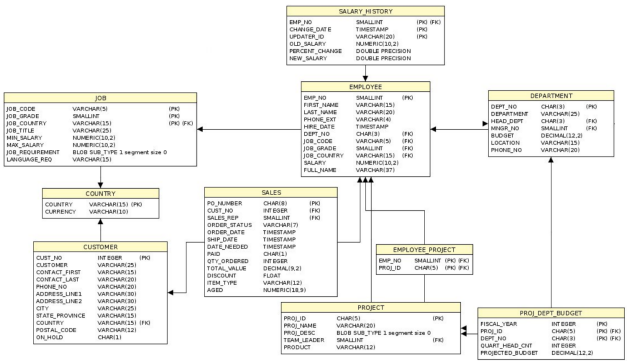
1.2 Выборка, фильтрация и сортировка данных

1.2.1 Цель

Изучить основы языка SQL. Научится делать выборку данных, фильтрацию и сортировку.

1.2.2 Порядок выполнения

1. Изучить схему БД *employee.fdb*.



2. Выбрать все страны и их валюты.

3. Отобразить всех клиентов из одного города. Город выбрать самостоятельно.

4. Выбрать всех сотрудников, принятых на работу в 1991 году. Отобразить полное имя и дату приема на работу. Отсортировать данные по дате приема на работу в порядке убывания.

5. Выбрать всех сотрудников из страны «USA» с зарплатой от 70000 до 100000. Отобразить полное имя, страну и зарплату. Отсортировать данные в порядке убывания зарплаты.

6. Выбрать первые 3 самых высокооплачиваемых сотрудника.

7. Выбрать все заказы, по которым была оплата, но заказ не доставлен. Отобразить номер заказа, дату заказа, статус оплаты, статус заказа, количество дней просрочки доставки заказа (с даты заказа по текущую дату). Отсортировать данные в порядке уменьшения количества дней просрочки.

8. Выбрать коды всех сотрудников у которых в 1993 году было снижение зарплаты. Отобразить код сотрудника, дату изменения зарплаты, предыдущую зарплату, процент изменения, новую зарплату. Отсортировать данные по проценту изменения в порядке убывания.

1.2.3 Справочная информация

1. Пример. Необходимо извлечь полное имя *FULL_NAME* и дату приема *HIRE_DATE* из таблицы сотрудников *EMPLOYEE*.

```
SELECT E.FULL_NAME, E.HIRE_DATE FROM EMPLOYEE E
```

Для упрощения написания запроса для таблицы *EMPLOYEE* задан псевдоним *E*, который используется при перечислении выбираемых полей.

2. Пример. Из таблицы *EMPLOYEE* необходимо выбрать всех сотрудников из списка стран *{Canada, USA}* (таблица *JOB_COUNTRY*), у которых заработная плата *SALARY* превышает 100000.

```
SELECT * FROM EMPLOYEE E WHERE UPPER(E.JOB_COUNTRY) IN ('CANADA', 'USA')  AND E.SALARY >␣
↪100000
```

В запрос добавлено условие, позволяющее отфильтровать записи с недопустимыми значениями поля *JOB_COUNTRY*. При описании условия использована функция *UPPER*, переводящая все символы строкового поля к верхнему регистру для учета всех возможных вариантов записи названия стран.

1.3 Соединение и агрегирование данных

1.3.1 Цель

Изучить основы языка SQL. Научится выполнять соединение и агрегирование наборов данных.

1.3.2 Порядок выполнения

1. Для выполнения работы используется БД *employee.fdb*.
2. Найти среднюю зарплату всех сотрудников.
3. Найти среднюю зарплату по департаменту и вывести с названием департамента.
4. Найти сотрудника, который работает дольше всех.
5. Найти средний стаж сотрудников по странам, и упорядочить по возрастанию.
6. Выбрать участников и проекты, в которых они участвуют.
7. Посчитать сколько участников в каждом проекте. Упорядочить по убыванию. Вывести с названием проекта.
8. Выбрать сотрудников департамента с самым большим бюджетом.
9. Выбрать сотрудников, у которых изменялась зарплата с начала указанного года.
10. Выбрать сотрудников, с зарплатой выше средней.
11. Выбрать покупателя, совершившего более 10 покупок за указанный год.
12. Выбрать покупателей, совершивших самые дорогие и самые дешевые покупки в каждом месяце указанного года. Вывести их фамилию, стоимость покупки и количество покупок вообще.

1.4 Оконные функции

1.4.1 Цель

Изучить оконные функции языка SQL.

1.4.2 Порядок выполнения

1. Для выполнения работы используется БД *employee.fdb* и **оконные функции**.
2. Вычислить процент от общей зарплаты для каждого сотрудника
3. Вычислить процент от общей зарплаты департамента для каждого сотрудника
4. Выбрать покупателей и сумму их покупок
5. Выбрать проекты и количество сотрудников в них
6. Выбрать продажи, их даты и общую сумму выручки к моменту продажи
7. Выбрать сотрудников и сумму их продаж
8. Вычислить во сколько раз увеличилась зарплата каждого сотрудника за время работы.
9. Самостоятельно придумать 3 запроса на использование других оконных функций

1.5 Разработка приложения Qt для работы с БД

1.5.1 Цель

Научиться использовать БД при разработке приложений с помощью библиотеки Qt.

1.5.2 Порядок выполнения

1. Изучить *Разработка приложения на Qt* для создания приложения.
2. Протестировать полученное приложение и исправить или улучшить неправильное поведение.

1.6 Разработка web-приложения для работы с БД

1.6.1 Цель

Научиться использовать БД при разработке приложений с помощью фреймворка Flask и Python.

1.6.2 Порядок выполнения

1. Изучить *Разработка web-приложения на Flask* для создания приложения.
2. Реализовать приложение и при этом русифицировать интерфейс.

1.7 Хранимые процедуры

1.7.1 Цель

Изучить язык разработки хранимых процедур.

1.7.2 Порядок выполнения

Для выполнения работы используется БД *employee.fdb* и язык *PSQL*.

1. Создать хранимую процедуру для добавления нового сотрудника в базу данных.
2. Создать хранимую процедуру для изменения департамента сотрудника.
3. Создать хранимую процедуру для удаления проекта из базы данных.
4. Создать хранимую процедуру для увеличения/уменьшения зарплаты сотрудника на определённый процент.
5. С помощью профайлера исследовать производительность хранимых процедур при выполнении сложных запросов к базе данных.

1.7.3 Справочная информация

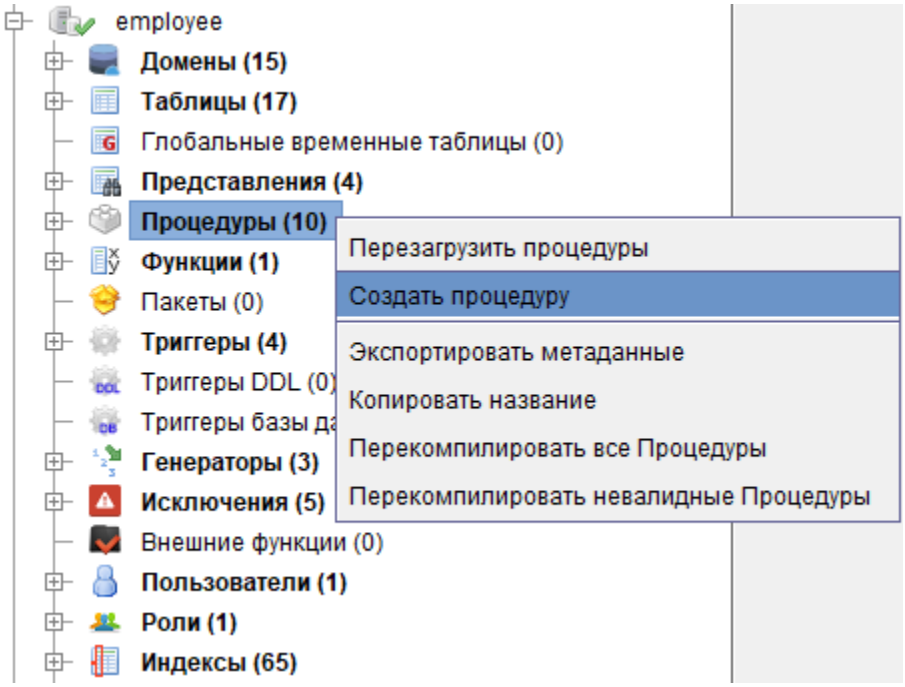
Хранимая процедура, является программой, хранящейся в области метаданных базы данных и выполняющейся на стороне сервера. К хранимой процедуре могут обращаться хранимые процедуры, триггеры и клиентские программы. Допустима рекурсия — хранимая процедура может обращаться сама к себе.

Существует два вида хранимых процедур — выполняемые хранимые процедуры (executed stored procedures) и хранимые процедуры выбора (selected stored procedures).

Выполняемые хранимые процедуры осуществляют обработку данных, находящихся в базе данных, или вовсе не связанных с базой данных. Эти процедуры могут получать входные параметры и возвращать выходные параметры.

Хранимые процедуры выбора как правило осуществляют выборку данных из базы данных, возвращая произвольное количество полученных строк. Процедуры выбора также могут получать входные параметры. Значение каждой очередной прочитанной строки возвращается вызвавшей программе в выходных параметрах.

Для создания хранимой процедуры с помощью RedExpert необходимо открыть дерево объектов текущего подключения, найти группу «Процедуры» и в контекстном меню выбрать пункт «Создать процедуру».

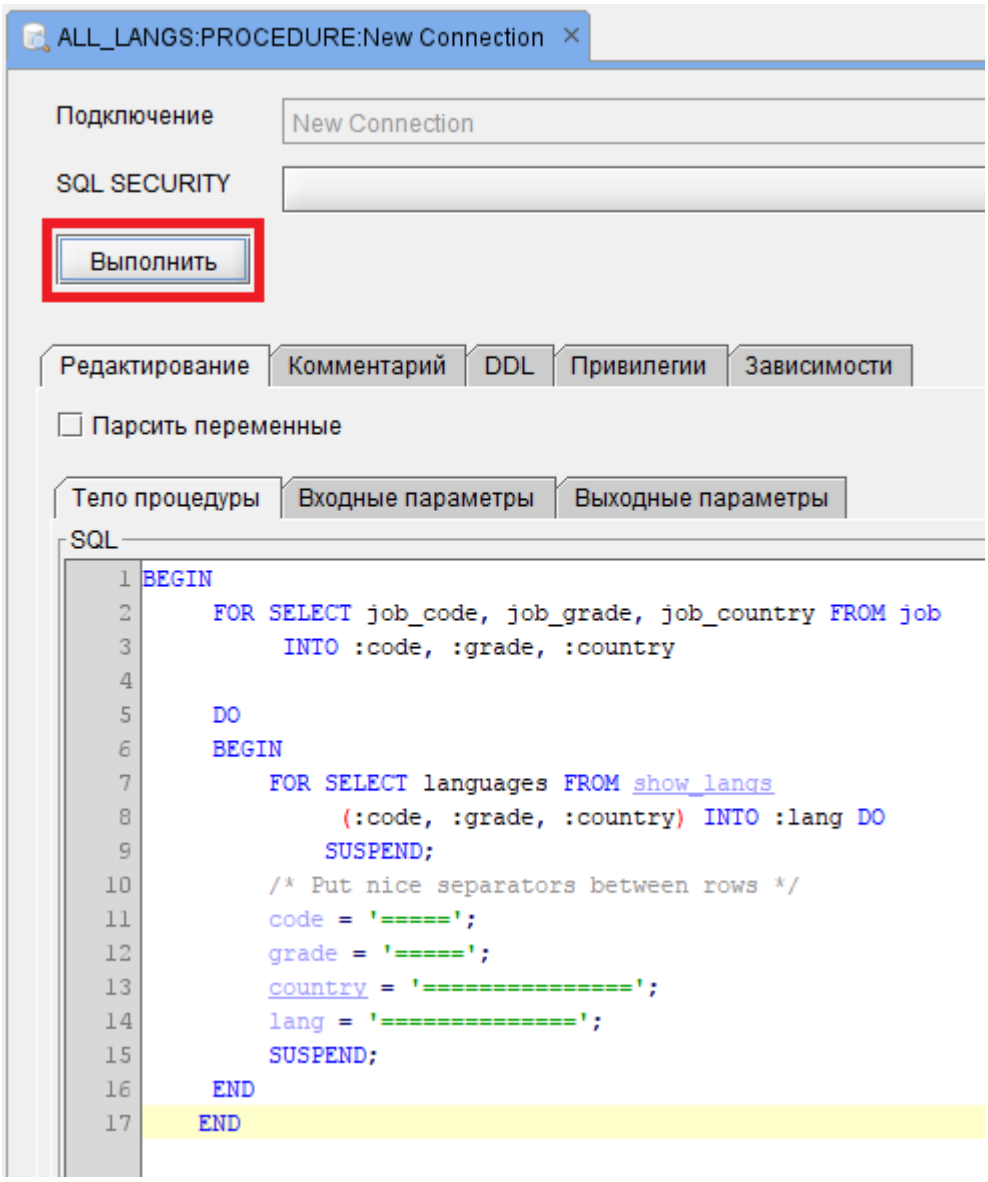


В открывшемся окне создать процедуру можно двумя способами: написать самостоятельно код PSQL во вкладке DDL, либо использовать специальные вкладки «тело процедуры», «входные параметры», «выходные параметры».

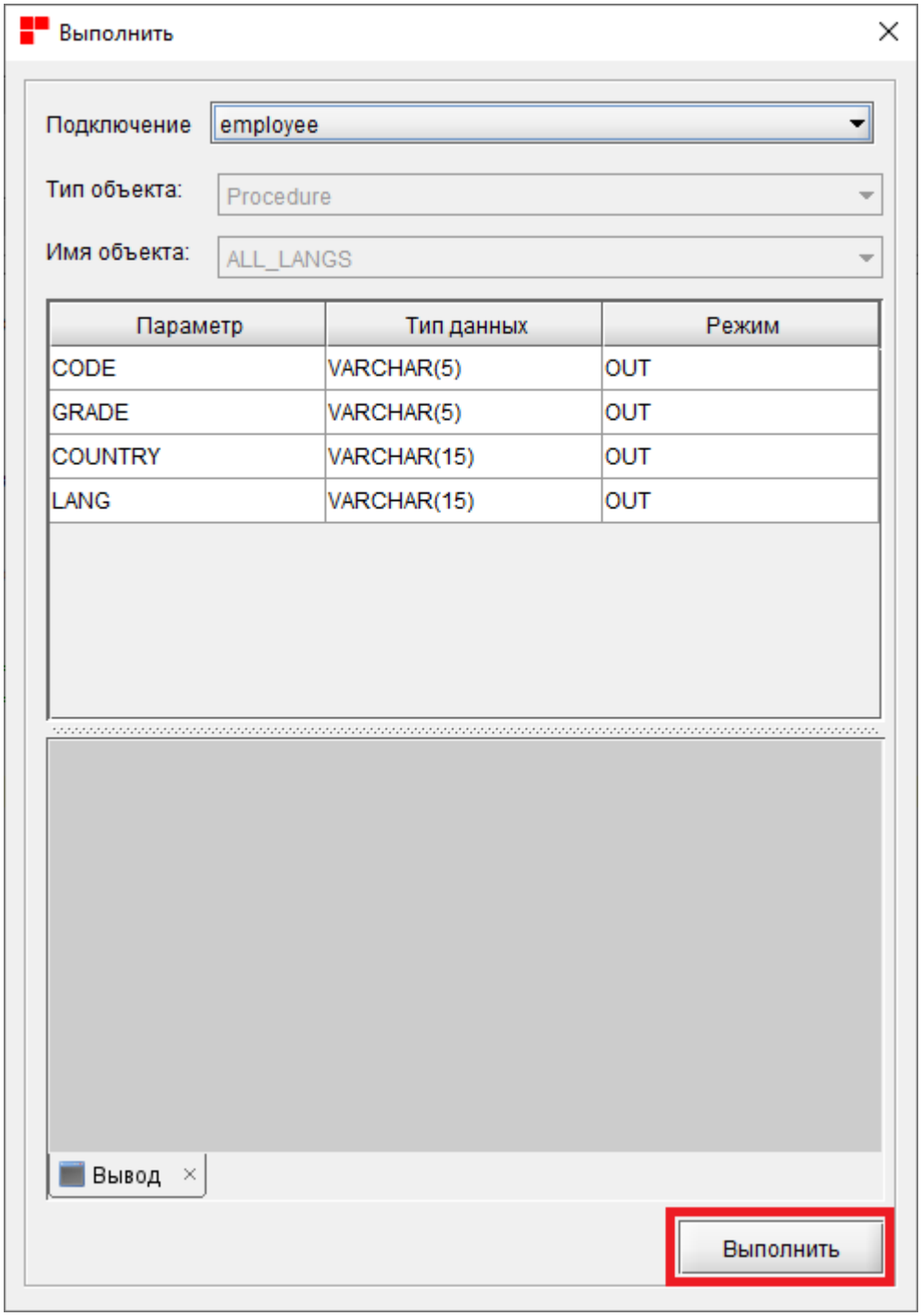
Тело процедуры состоит из необязательных объявлений локальных переменных, подпрограмм и именованных курсоров, и одного или нескольких операторов, или блоков операторов, заключённых во внешнем блоке, который начинается с ключевого слова BEGIN, и завершается ключевым словом END.

Для того чтобы выполнить хранимую процедуру с помощью RedExpert необходимо:

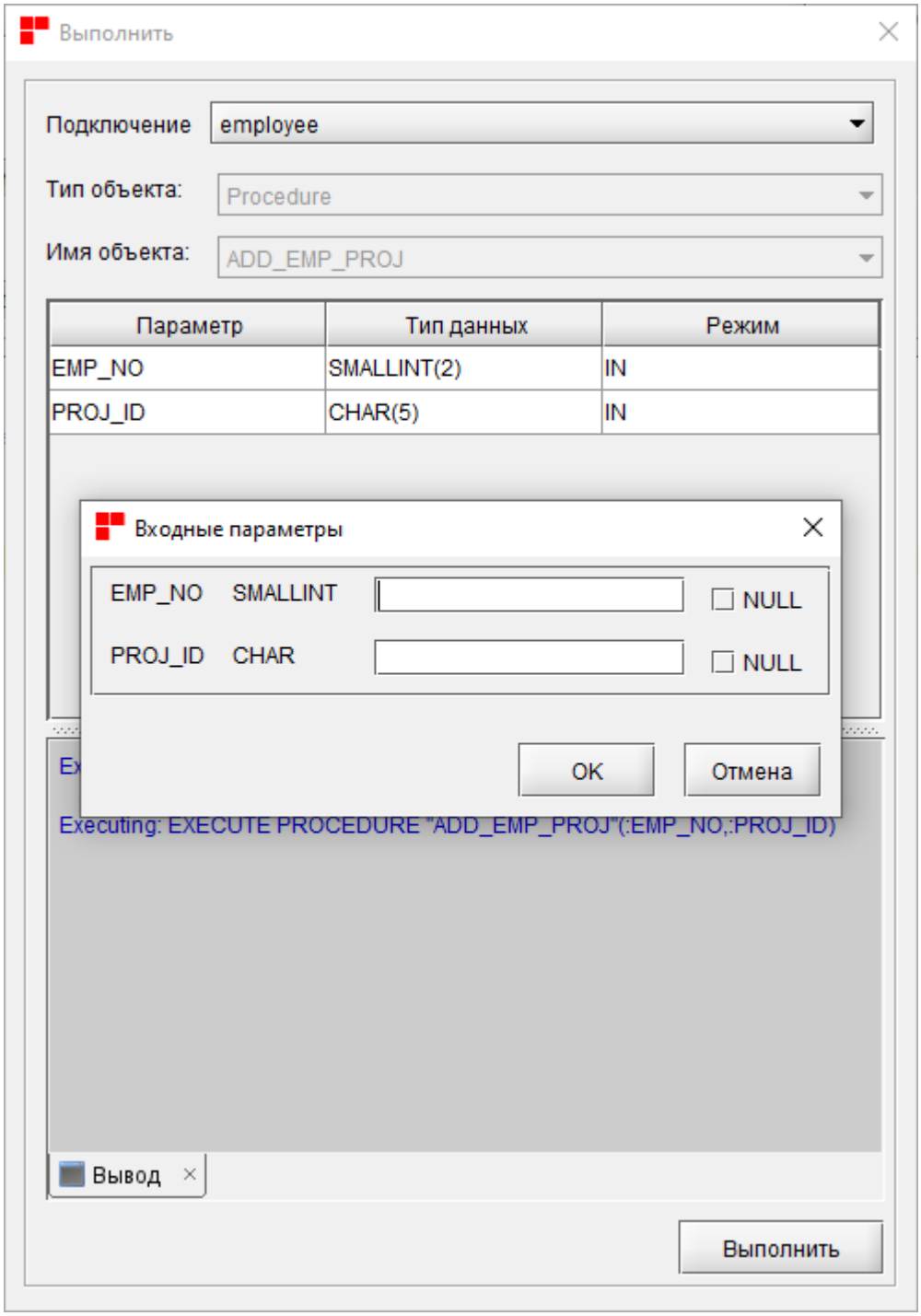
1. Двойным кликом мыши нажать на имя процедуры в группе «Процедуры» дерева объектов текущего подключения
2. Нажать кнопку «Выполнить»



3. В появившемся окне снова нажать кнопку «Выполнить»



4. (Опционально) Если у процедуры имеются входные параметры, необходимо ввести их значения в появившемся окне и нажать кнопку «OK»



Для получения дополнительной информации обратитесь к Руководству по SQL

Глава 2

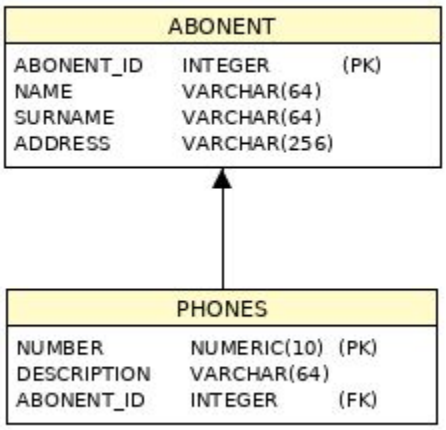
Приложения

2.1 Разработка приложения на Qt

В данной лабораторной работе рассматривается пример создания простейшего приложения в среде QtCreator, работающего с БД.

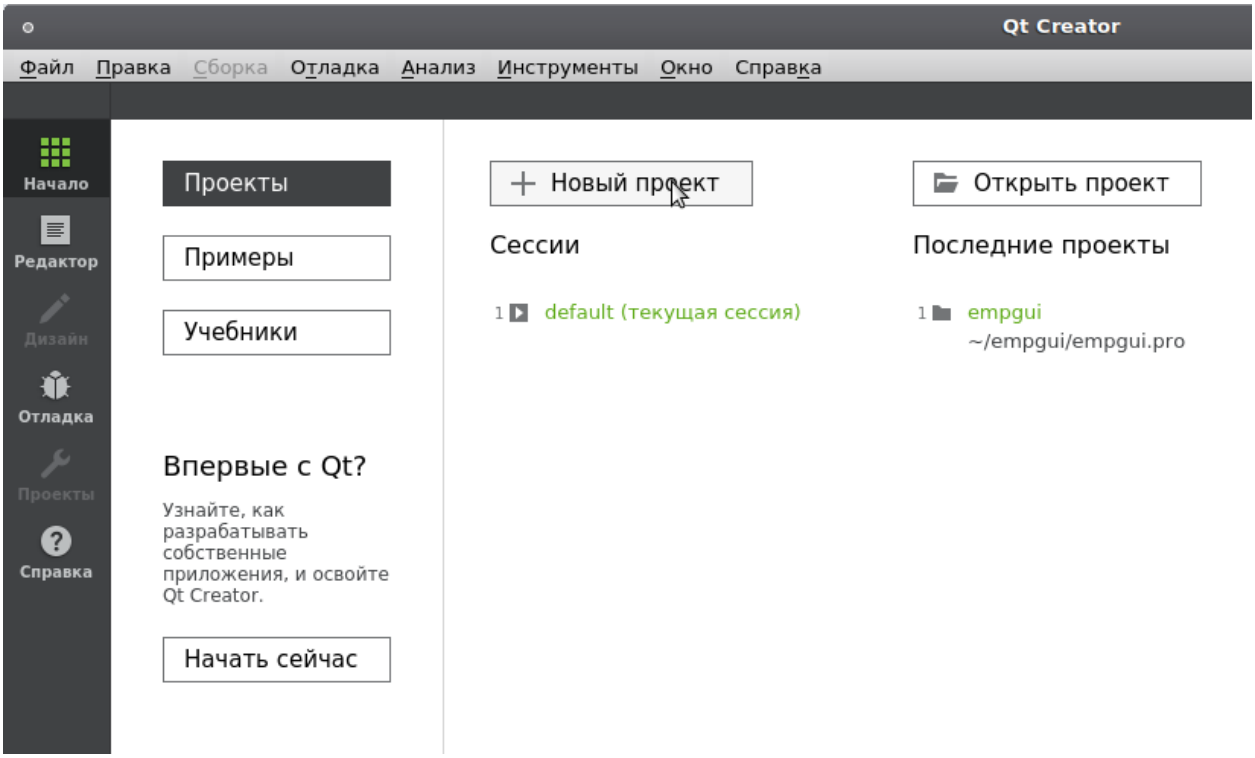
2.1.1 Структура БД

Для разработки приложения необходимо создать БД следующей структуры:

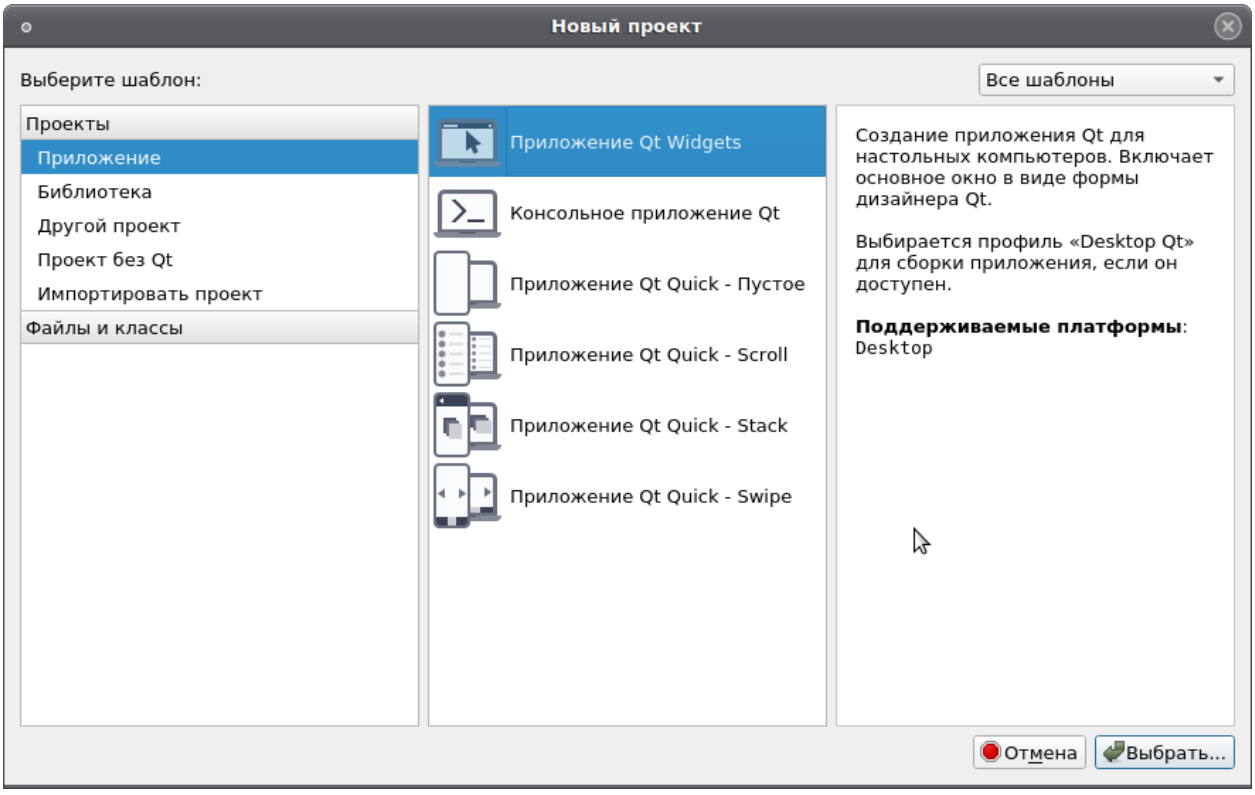


2.1.2 Создание проекта и интерфейса

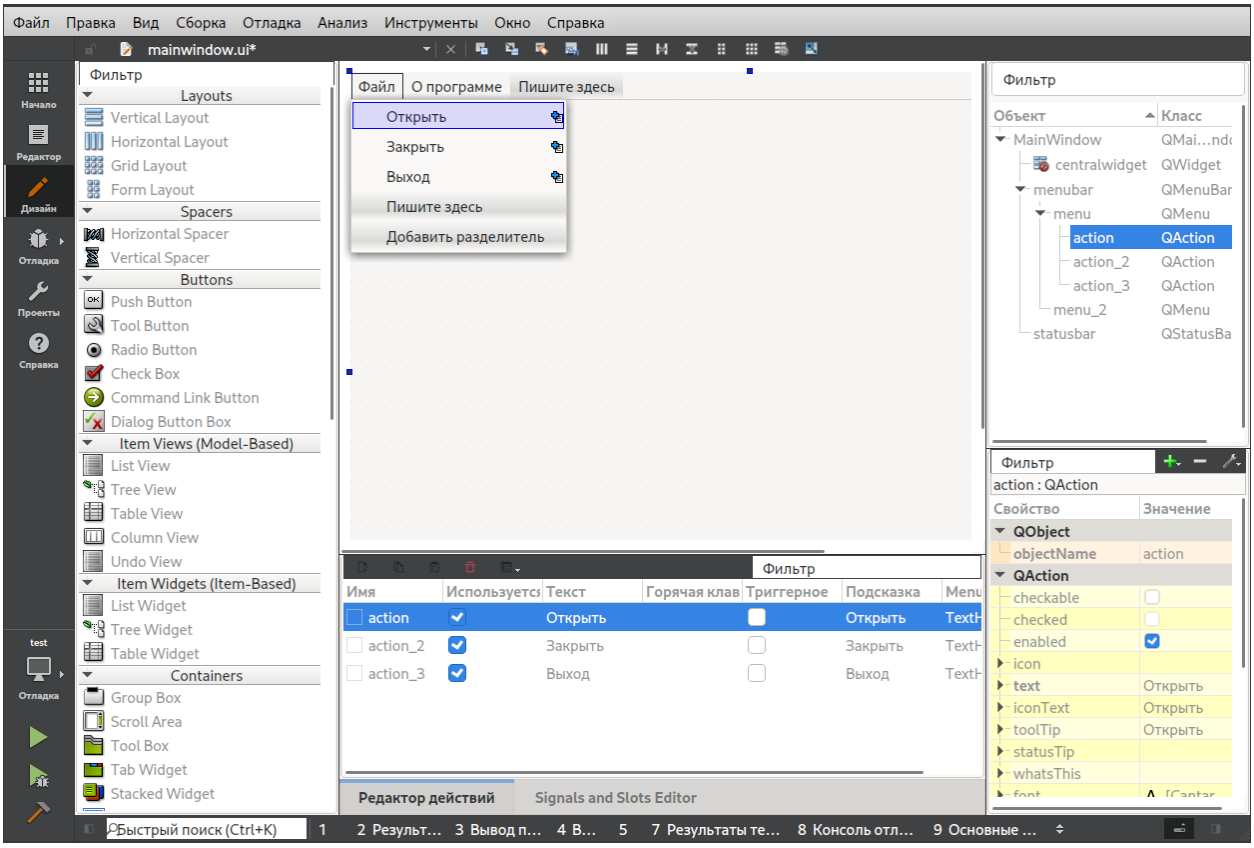
Создаем новый проект



В качестве шаблона выбираем “Приложение/Приложение Qt Widgets”



Далее выбираем все опции по умолчанию, указав имя проекта (`phones`) и его расположение. В дереве объектов дважды кликаем по форме (`mainwindow.ui`) и приступаем к размещению элементов на форме. Для этого используются `Push Button`, меню и `QTreeView`. В меню создаем ряд пунктов для открытия, закрытия БД, выхода и показа справки.



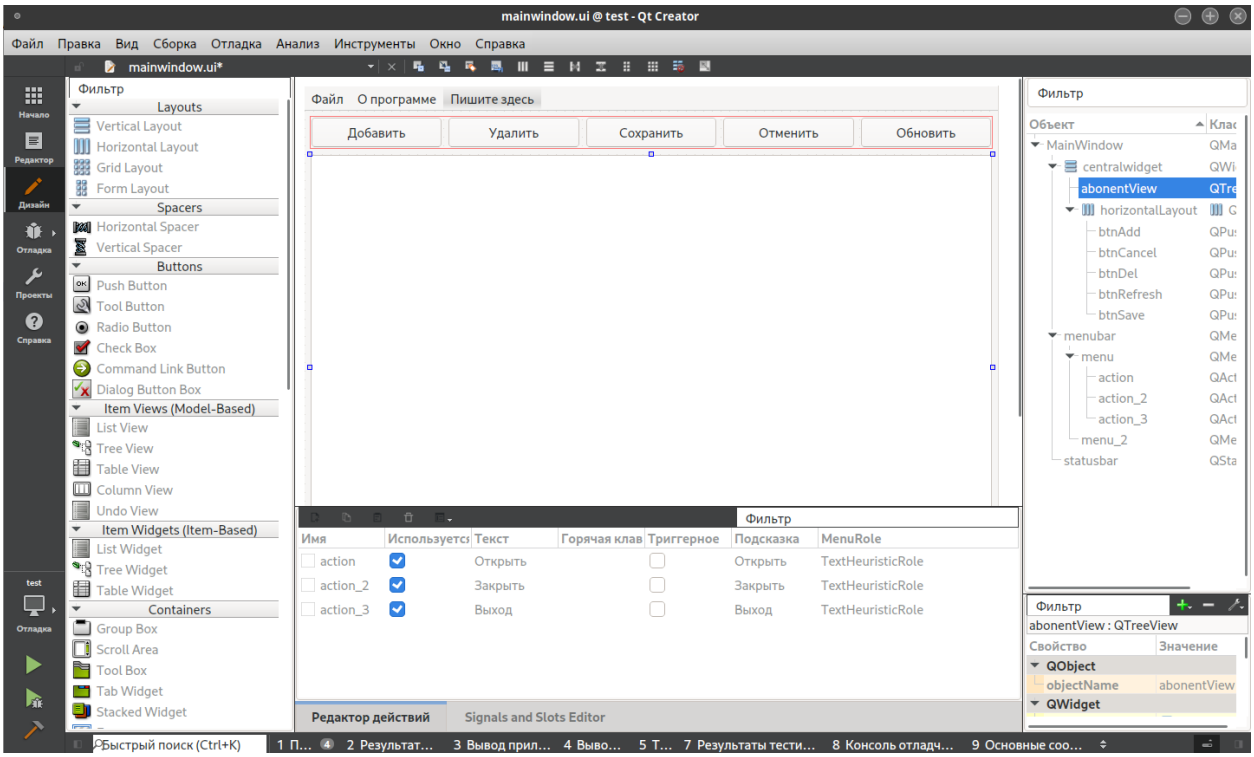
Остальные компоненты размещаем как на рисунке. Первой кнопке устанавливаем свойства: objectName = «btnAdd» text = «Добавить» Свойство text можно выставить двойным кликом по компоненту.

Аналогичным образом выставляем свойства оставшихся кнопок, так что в итоге должно получиться (слева направо):

1. btnAdd, «Добавить»
2. btnDel, «Удалить»
3. btnSave, «Сохранить»
4. btnCancel, «Отменить»
5. btnRefresh, «Обновить»

Чтобы выровнять их необходимо все выделить и скомпоновать по горизонтали (ПКМ (правая кнопка мыши), Компоновка, По горизонтали).

Далее перетаскиваем компонент `Tree View` и чтобы все выровнять, ПКМ по свободному месту на форме, Компоновка, По вертикали. Меняем у компонента свойство objectName=»abonentView». У нас получится форма, как на рисунке, с выровненными компонентами, способными адаптироваться под изменения размеров окна. Подробнее о компоновке изучите отдельно.



2.1.3 Написание кода

Перед написанием добавим в файл проекта `CMakeLists.txt` компонент Sql и дописать в компоненты, линкуемые к приложению `phones`.

```
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets Sql)
...
target_link_libraries(phones PRIVATE Qt${QT_VERSION_MAJOR}::Widgets Qt${QT_VERSION_MAJOR}
↪::Sql)
```

Перед добавлением кода обработчиков, добавить нужные заголовочные файлы для работы с базами данных и объявить поле для модели данных. Это делается в классе `MainWindow` файла `mainwindow.h` следующим образом:

```
#include <QMainWindow>
#include <QtSql/QSqlQuery>
#include <QtSql/QSqlTableModel>

namespace Ui {
class MainWindow;
}

// ранее сгенерированный код

private:
    Ui::MainWindow *ui;

    QSqlTableModel *abonentModel = nullptr;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
};
```

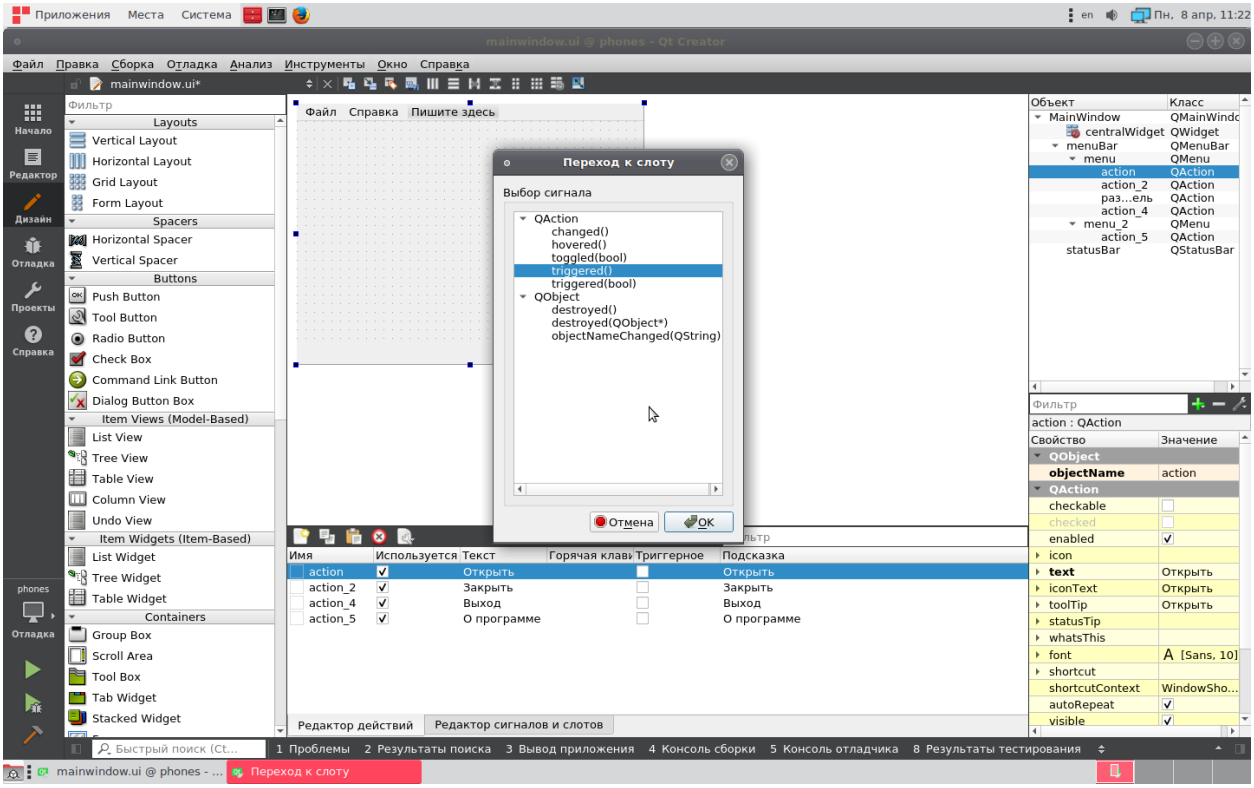
Также добавим заголовочные файлы, которые нам понадобятся в файле `mainwindow.cpp`:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

// новые файлы
#include <QMessageBox>
#include <QtSql/QSqlDatabase>
#include <QtSql/QSqlError>
#include <QFileDialog>
#include <QDebug>
```

Обработчик нажатия пунктов меню пишется следующим образом:

1. Выделяем нужный пункт меню на форме.
2. Внизу также будет список пунктов. На нужном кликаем правой кнопкой мыши и выбираем “Переход к слоту”. Далее выбираем “triggered“.
3. Открывается редактор кода, в котором мы увидим заготовку метода, который будет вызван при нажатии на пункт меню.



Добавляем код соединения с БД, т.е. обработчик пункта меню “Файл/Открыть”

```
// Connection is already open
if (abonentModel)  // Field of MainWindow
    return;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
QSqlDatabase db = QSqlDatabase::addDatabase("QIBASE", "phones");
db.setHostName("localhost");
db.setDatabaseName(QFileDialog::getOpenFileName(this, "Open Database", "/var/rdb",
↪"Database files (*.fdb);;All files (*)"));

const bool ok = db.open("SYSDBA", "masterkey");

if (!ok)
{
    ShowMessage("Ошибка подключения");
    return;
}

abonentModel = new QSqlTableModel(this, db);
abonentModel->setTable("ABONENT");
abonentModel->setEditStrategy(QSqlTableModel::OnManualSubmit);
abonentModel->select();
ui->abonentView->setModel(abonentModel);
```

Код для закрытия БД

```
delete abonentModel;
abonentModel = nullptr;
QSqlDatabase::database("phones").close();
```

Можно протестировать. БД должна открываться и данные должны появляться в таблице. Функции `ShowMessage` нужна для вывода сообщений на экран. Она определяется в начале файла `mainwindow.cpp` следующим образом.

```
void ShowMessage(const QString text)
{
    QMessageBox msg;
    msg.setText(text);
    msg.exec();
}
```

Для написания обработчика нажатия кнопки кликните на ней на форме правой кнопкой мыши и в “Перейти к слоту” выберете `clicked`. Ниже все обработчики кнопок. Говорящие названия обработчиков описывают действия.

```
void MainWindow::on_btnAdd_clicked()
{
    if (!abonentModel->insertRow(0))
        ShowMessage(abonentModel->lastError().text());
}

void MainWindow::on_btnSave_clicked()
{
    if (!abonentModel->submitAll())
        ShowMessage(abonentModel->lastError().text());
}
```

(continues on next page)

(продолжение с предыдущей страницы)

```
void MainWindow::on_btnDel_clicked()
{
    QModelIndex i = ui->abonentView->selectionModel()->currentIndex();
    if (!abonentModel->removeRow(i.row()))
        ShowMessage(abonentModel->lastError().text());
}

void MainWindow::on_btnCancel_clicked()
{
    abonentModel->revertAll();
}

void MainWindow::on_btnRefresh_clicked()
{
    abonentModel->select();
}
```

Важно: На этом разработка первого приложения завершена. Необходимо его тщательно протестировать.

2.2 Разработка web-приложения на Flask

В данной лабораторной работе рассматривается пример создания простейшего веб-приложения, называемого **Flaskr**. Пользователи могут регистрироваться, входить в систему, создавать посты, редактировать и удалять свои посты.

За основу взята пошаговая инструкция с официального сайта Flask, однако в качестве СУБД используется российская СУБД Ред База Данных.

2.2.1 Создание приложения

Прежде всего необходимо создать каталог, в котором будет располагаться проект.

```
$ mkdir redflaskr
$ cd redflaskr
```

Далее будем предполагать, что вся работа выполняется в каталоге `redflaskr`. Все пути к файлам будем указывать относительно него.

Теперь необходимо настроить виртуальное окружение для Python (Python virtual environment) и установить Flask и драйвер СУБД Ред База Данных.

Виртуальное окружение используется для управления зависимостями проекта как при разработке так и при разворачивании на продуктиве. Чем больше проектов есть у разработчика, тем более вероятно что они требуют различных версий библиотек Python или даже разных версий самого Python. Новые версии одних библиотек могут разрушить совместимости в другом проекте.

Виртуальное окружение (virtual environment)
: Независимая группа библиотек для каждого проекта. Пакеты установленные для одного проекта не влияют на другие проекты или пакеты операционной системы.

Python поставляется с модулем `venv` для создания виртуального окружения.

Для создания виртуального окружения выполните команду

```
$ python3 -m venv venv
```

Перед началом работы над проектом активируйте соответствующее окружение:

```
$ . venv/bin/activate
```

В активированном окружении установите Flask и драйвер для СУБД Ред База Данных:

```
$ pip install Flask
$ pip install fdb
```

Проекты на Python используют пакеты для организации кода и мы этим воспользуемся.

Каталог проекта будет содержать:

- **flaskr**
  Пакет Python, содержащий код приложения и другие файлы.
- **venv**
  Виртуальное окружение, в котором будет установлен Flask, драйвер СУБД fdb и другие зависимости.

Приложение Flask это объект (instance) класса Flask. Все, что связано с приложением (настройки, URL адреса, прочее) будет настраиваться в этом объекте.

Наиболее простой путь создания приложения - это создать глобальный объект непосредственно в начале программы, однако по мере роста проекта это может принести проблемы.

Вместо создания объекта глобально, мы будем создавать его внутри функции. Такая функция называется *фабрикой приложения* (application factory). Все настройки, регистрации и т.п. будут происходить внутри функции, после чего объект приложения будет возвращен.

Фабрика приложения

Создайте каталог `flaskr`, а внутри файл `__init__.py`, который содержит *фабрику приложения* и говорит Python, что катало `flaskr` должен рассматриваться как пакет.

```
$ mkdir flaskr
```

flaskr/__init__.py

```
import os

from flask import Flask

def create_app(test_config=None):
    # create and configure the app
    app = Flask(__name__, instance_relative_config=True)
    app.config.from_mapping(
        SECRET_KEY='dev',
        DATABASE='localhost:/var/rdb/flaskr.fdb', # /var/rdb/ dir must exist like in lab1
        USER='sysdba',
        PASSWORD='masterkey',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        LIBRARY='/opt/RedDatabase/lib/libfbclient.so'
    )

    if test_config is None:
        # load the instance config, if it exists, when not testing
        app.config.from_pyfile('config.py', silent=True)
    else:
        # load the test config if passed in
        app.config.from_mapping(test_config)

    # ensure the instance folder exists
    try:
        os.makedirs(app.instance_path)
    except OSError:
        pass

    # a simple page that says hello
    @app.route('/hello')
    def hello():
        return 'Hello, World!'

    return app
```

`create_app` это функция *фабрика приложения*. Позже она будет дополнена, но и сейчас она многое делает:

1. `app = Flask(__name__, instance_relative_config=True)` **создает объект приложения Flask.**

   __name__
   имя текущего модуля Python. Приложению необходимо знать где оно располагается, чтобы установить некоторые пути.

   instance_relative_config
   говорит приложению, что файлы конфигурации размещаются относительно каталога `instance`. Он размещается вне каталога `flaskr` и содержит локальные данные: конфигурационные файлы, БД.

2. `app.config.from_mapping` **устанавливает значения параметров конфигурации по умолчанию.**

   SECRET_KEY
   используется классом Flask и расширениями для обеспечения безопасности хранимых данных. Значение `dev` позволяет удобно разрабатывать приложения, но должно быть заменено случайным значением при поставке приложения заказчику.

   DATABASE
   путь к файлу БД. БД размещается в каталоге `/var/rdb/`. В зависимости от нужд приложения, может быть любым, в том числе псевдонимом БД на удаленном сервере.

   USER
   Имя пользователя, от которого будет производиться соединение.

   PASSWORD
   Пароль. Для встроенного сервера игнорируется.

LIBRARY
путь до клиентской библиотеки `libfbclient.so`.

3. `app.config.from_pyfile` перезаписывает значения параметров конфигурации значениями из файла `config.py` каталога `instance`, если он существует. Например, при поставке приложения в нем можно указать реальное значение `SECRET_KEY`.
4. `os.makedirs()` гарантирует существование каталога `app.instance_path`. Flask не создает каталог автоматически.
5. `@app.route()` создает простой маршрут, чтобы убедиться что приложение работает, прежде чем продолжить его разрабатывать. Это связывает URL `/hello` и функцию, которая сформирует ответ. В данном случае строку „Hello, World!“.

Запуск приложения

Теперь можно запустить приложение, используя команду `flask`. Укажите Flask где искать приложение и запустите его в режиме разработчика.

> **Предупреждение:** Вы должны быть в каталоге redflaskr, но не в его подкаталогах.

Режим разработчика показывает интерактивный отладчик когда страница выбрасывает исключение и перезапускает сервер, когда вы делаете изменения в коде. Его можно оставить запущенным и просто обновлять страницу в браузере по мере разработки.

```
$ export FLASK_APP=flaskr
$ export FLASK_ENV=development
$ flask run
```

Вы увидите вывод, подобный этому:

```
* Serving Flask app "flaskr"
* Environment: development
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 855-212-761
```

Перейдите по адресу http://127.0.0.1:5000/hello в браузере и вы увидите сообщение «Hello, World!».

2.2.2 Работа с БД

Подключение к БД

При работе с БД первое что необходимо сделать, создать подключение. Все запросы выполняются через подключение, которое закрывается когда работа выполнена.

В web-приложениях подключение обычно привязывается к запросу. В какой-то момент обработки запроса оно создается, а перед отправкой ответа закрывается.

Кроме этого необходимо написать код для инициализации БД.

`flaskr/db.py`

```
import fdb

import click
from flask import current_app, g
from flask.cli import with_appcontext


def init_db():
    try:
        conn = fdb.connect(
            dsn=current_app.config['DATABASE'],
            user=current_app.config['USER'],
            password=current_app.config['PASSWORD'],
            fb_library_name=current_app.config['LIBRARY']
        )
        conn.drop_database()
    except Exception as e:
        print(e)


    conn = fdb.create_database(
        dsn=current_app.config['DATABASE'],
        user=current_app.config['USER'],
        password=current_app.config['PASSWORD'],
        fb_library_name=current_app.config['LIBRARY']
    )

    metadata = [
        '''
        RECREATE TABLE users (
            id integer generated by default as identity primary key,
            username varchar(256) UNIQUE NOT NULL,
            password varchar(256) NOT NULL
        )
        ''',
        '''
        RECREATE TABLE posts (
            id integer generated by default as identity primary key,
            author_id INTEGER NOT NULL REFERENCES users (id),
            created TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
            title varchar(120) NOT NULL,
            body varchar(5000) NOT NULL
        )
        '''
    ]

    cursor = conn.cursor()

    for query in metadata:
        cursor.execute(query)

    conn.commit()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def get_db():
    if 'db' not in g:
        g.db = fdb.connect(
            dsn=current_app.config['DATABASE'],
            user=current_app.config['USER'],
            password=current_app.config['PASSWORD'],
            fb_library_name=current_app.config['LIBRARY']
        )

    return g.db


def close_db(e=None):
    db = g.pop('db', None)

    if db is not None:
        db.close()


@click.command('init-db')
@with_appcontext
def init_db_command():
    """Clear the existing data and create new tables."""
    init_db()
    click.echo('Initialized the database.')


def init_app(app):
    app.teardown_appcontext(close_db)
    app.cli.add_command(init_db_command)
```

`get_db` создает подключение к БД.

g
: специальный объект, уникальный для каждого запроса. Он используется для хранения данных, которые могут использоваться множеством функции во время обработки запросы. Подключение создается и повторно используется, если `get_db` вызывается не первый раз.

current_app
: другой специальный объект, который указывает на приложение Flask, обрабатывающее запрос.

fdb.connect
: устанавливает подключение к БД используя параметры конфигурации. Файл БД создается в функции `init_db`.

`close_db` закрывает подключение к БД, если `g.db` установлен.

`init_db` создает БД и необходимые объекты.

metadata
: список SQL команд для создания объектов БД: таблицы пользователей `users` и таблицы постов `posts`.

Вначале функция пытается установить соединение с имеющейся БД для того, чтобы ее удалить. В случае неудачи печатается исключение для целей отладки. Далее в любом случае создается новая БД, используя все те же параметры конфигурации.

conn.cursor()
: создает объект *курсор*, с помощью которого можно выполнять все запросы к СУБД.

В цикле выполняются все запросы из списка метаданных и в завершении производиться завершение транзакции и применение всех изменений.

`click.command()` определяет команду командной строки «init-db», которая вызывает функцию `init_db` и сообщает об успешности выполнения инициализации пользователю.

`init_app(app)` регистрирует созданные функции.

Функции `close_db` и `init_db_command` должны быть зарегистрированы в объекте приложения, иначе они не будут использоваться.

app.teardown_appcontext()
: говорит Flask вызвать указанную функцию при очистке после отправки ответа.

app.cli.add_command()
: добавляет новую команду, которая может вызываться с командой `flask`.

Импортируйте и добавьте вызов этой функции в *фабрике приложения*.

`flaskr/__init__.py`

```
def create_app():
app = ...
# existing code omitted

from . import db
db.init_app(app)

return app
```

Инициализация БД

Теперь, когда команды `init-db` зарегистрирована, она может быть вызвана используя команду `flask`, аналогично команде `run`.

Предупреждение: Если у вас все еще запущен сервер с предыдущего этапа, необходимо его остановить или выполнить команду в другом терминале. Помните что необходимо находиться в каталоге `redflaskr` и активировать виртуальное окружение.

Запустите команду

```
$ flask init-db
Initialized the database.
```

В каталоге `instance` должен появится файл `flaskr.fdb`.

2.2.3 Эскизы (Blueprints) и представления (Views)

Функция-представление
(View function) функция, которая отвечает на запросы к приложению.

Flask использует шаблоны, чтобы сопоставить входящие URL запросов функциям-представлениям.

Эскиз
(Blueprint) способ организовать группы связанных представлений и другой код.

Вместо регистрации представление и другого кода непосредственно в приложении, они регистрируются в эскизе. Далее эскиз регистрируется в приложении, в фабрике приложения.

Создание эскиза

Flaskr будет иметь два эскиза. Один для аутентификации, другой для функций работы с постами. Код для каждого эскиза будет размещаться в отдельных модулях. В первую очередь сделаем модуль аутентификации.

`flaskr/auth.py`

```
import functools

from flask import (
    Blueprint, flash, g, redirect, render_template, request, session, url_for
)
from werkzeug.security import check_password_hash, generate_password_hash

from flaskr.db import get_db

bp = Blueprint('auth', __name__, url_prefix='/auth')
```

Этот код создает эскиз под названием „auth“. Как и объект приложение, эскиз должен знать где его создали. Для этого `__name__` передается вторым параметром. `url_prefix` будет предшествовать всем URL адресам, связанным с этим эскизом.

Импортируйте и зарегистрируйте эскиз в *фабрике приложения*, используя `app.register_blueprint()`. Добавьте следующий код в конец *фабрики приложения* перед возвращением объекта приложения.

`flaskr/__init__.py`

```
def create_app():
    app = ...
    # existing code omitted

    from . import auth
    app.register_blueprint(auth.bp)

    return app
```

Эскиз аутентификации будет иметь функции-представления для регистрации новых пользователей и входа/выхода существующих.

Первое представление: регистрация

Когда пользователь заходит на страницу `/auth/register` представление `register` вернет HTML код с формой для заполнения. При отправке данных формы она будет проверена и либо будет показано сообщение об ошибке, либо будет создан новый пользователь и осуществлен переход на страницу входа.

Сейчас просто допишем код функции-представления к имеющемуся коду файлы `flaskr/auth.py`. Далее будут написаны шаблоны для генерации HTML форм.

`flaskr/auth.py`

```
# existing code
#...

@bp.route('/register', methods=('GET', 'POST'))
def register():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        db = get_db()
        error = None

        if not username:
            error = 'Username is required.'
        elif not password:
            error = 'Password is required.'

        if error is None:
            try:
                db.cursor().execute(
                    "INSERT INTO users (username, password) VALUES (?, ?)",
                    (username, generate_password_hash(password)),
                )
                db.commit()
            except db.DatabaseError as e:
                if e.args[2] == 335544665:  #isc_unique_key_violation
                    error = f"User {username} is already registered."
            else:
                return redirect(url_for("auth.login"))

        flash(error)

    return render_template('auth/register.html')
```

Вот что делает эта функция:

1. **@bp.route**
   сопоставляет URL `/register` с функцией-представлением `register`. Когда Flask получит запрос на `/auth/register` будет вызвана функция `register` и ее результат будет отправлен пользователю.
2. Для отправки формы пользователем используется метод `POST`. Если это так, то проверяет входные параметры.
3. **request.form**
   специальный тип ассоциативного массива с параметрами формы, содержащий пары ключ-значение. Пользователь будет вводить туда `username` и `password`.

4. Проверяем что введенные значение не пустые.
5. Если проверка успешна, вставляем данные нового пользователя в базу данных.
   - **db.cursor().execute**
     создает объект курсор из подключения и вызывает его метод execute для выполнения запроса на вставку в таблицу. Метод принимает SQL динамического запроса, с ? вместо значений и список значений. Такой способ защищает от SQL инъекций и строго рекомендуется в отличие от конструирования запроса как статического в виде одной строки.
   - Для целей безопасности пароль не храниться в БД в открытом виде. Он хешируется функцией `generate_password_hash()`. Поскольку этот метод меняет данные, то транзакцию необходимо подтвердить с помощью `db.commit()`.
   - **db.DatabaseError**
     исключение, которое будет выброшено при ошибке выполнения запроса. Код ошибки `335544665` соответствует ошибке дубликата первичного ключа, т.е. возникнет когда такой пользователь уже есть.
6. После сохранения пользователя он перенаправляется на страницу входа. `url_for()` генерирует URL адрес представления на основе его имени. Это предпочтительнее прямой записи URL, т.к. позволяет менять адрес позднее, без смены остального кода. `regirect()` генерирует ответ для перенаправление на указанный (сгенерированный) URL адрес.
7. Если проверка параметров будет неудачной, то пользователю будет показана ошибка. `flash()` сохраняет сообщения и они будут показаны при следующем рендеринге шаблона.
8. При первом открытии страницы `/auth/register` или при ошибке проверки параметров, должна быть снова показана страница регистрации. `render_template()` рендерит шаблон, содержащий HTML, который мы вскоре напишем.

Представление: вход

Это представление пишется по той же схеме, что и представление `register` выше, также дописывается к существующему коду.

`flaskr/auth.py`

```
# existing code
#...

@bp.route('/login', methods=('GET', 'POST'))
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        db = get_db()
        error = None
        user = db.cursor().execute('SELECT * FROM users WHERE username = ?', (username,)␣
↪).fetchonemap()

        if user is None:
            error = 'Incorrect username.'
        elif not check_password_hash(user['password'], password):
            error = 'Incorrect password.'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        if error is None:
            session.clear()
            session['user_id'] = user['id']
            return redirect(url_for('index'))

        flash(error)

    return render_template('auth/login.html')
```

Есть несколько отличий от представления `register`:

1. Для последующего использования в переменной сохраняется пользователь из таблицы `users`. `fetchonemap()` возвращает одну строку из запроса. Результат имеет тип *словарь*. Если запрос не вернул ни одной строки, результат будет `None`.
2. `check_password_hash()` хеширует пароль из формы и сравнивает его с сохраненным. Если равны хеши паролей, то и пароли совпадают.
3. `session` переменная типа *словарь*, которая хранит данные между запросами. Если аутентификация прошла успешно, тогда `id` пользователя сохраняется для новой сессии. Данные сохраняются в *cookie* которые отправляются в браузер. Браузер пришлет их обратно в запросах. Flask безопасно подписывает данные, так что они не могут подменены.

Теперь, когда идентификатор пользователя сохранен в сессии, он будет доступен последующим запросам. В начале каждого запроса, если пользователь прошел аутентификацию, информация о нем должна быть загружена и стать доступной другим представлениям.

`flaskr/auth.py`

```
# existing code
#...

@bp.before_app_request
def load_logged_in_user():
    user_id = session.get('user_id')

    if user_id is None:
        g.user = None
    else:
        g.user = get_db().cursor().execute('SELECT * FROM users WHERE id = ?', (user_id,
↪)).fetchonemap()
```

`@bp.before_app_request()` регистрирует функцию, которая выполняется перед функциями-представлениями, безотносительно какой URL был запрошен. `load_logged_in_user()` проверяет сохранен ли в `session` и если да, то читает информацию о нем из БД, сохраняя в `g.user`, который существует на протяжении всей обработки запроса. В противном случае `g.user` будет установлен в `None`.

Представления: выход

Для выхода необходимо удалить `user_id` из `session`. Тогда `load_logged_in_user` не сможет загружать пользователя в последующих запросах.

`flaskr/auth.py`

```
# existing code
#...

@bp.route('/logout')
def logout():
    session.clear()
    return redirect(url_for('index'))
```

Требования входа в других представлениях

Создание, редактирование и удаление постов блога требует чтобы пользователь был авторизован. *Декоратор* используется для проверки каждого представления, к которому он примеряется.

`flaskr/auth.py`

```
# existing code
#...

def login_required(view):
    @functools.wraps(view)
    def wrapped_view(**kwargs):
        if g.user is None:
            return redirect(url_for('auth.login'))

        return view(**kwargs)

    return wrapped_view
```

Этот *декоратор* возвращает новую функцию-представление, которая *оборачивает* оригинальное представление. Новая функция проверяет загружен ли пользователь и если нет перенаправляет на страницу входа. Если пользователь загружен, то вызывается оригинальная функция-представление и выполнение продолжается как обычно. Этот *декоратор* будет использован далее, при написании представлений блога.

2.2.4 Шаблоны (Templates)

Функции-представления написаны, но если мы перейдем по любому адресу, мы получим ошибку `TemplateNotFound`. Это потому что используется функция `render_template`, но для нее еще не написаны шаблоны. Шаблоны располагаются в каталоге `templates` внутри пакета `flaskr`.

Шаблоны - это файлы, содержащие статические данные вместе с *подстановками* (placeholders) для динамических данных. Шаблоны рендерятся с конкретными данными для получения финального документа. Flask использует библиотеку шаблонов Jinja.

Jinja похожа на Python. Специальные разделители используются для отличия синтаксиса Jinja от статических данных шаблона. Содержимое между `{{` и `}}` является выражением, результат которого

будет выведен в финальный документ. {% и %} выделяют управляющие потоки типа if и for. В отличие от Python блоки выделяются в начале и конце тегами.

Базовый шаблон

Каждая страница приложения будет иметь одинаковую структуру с различным заполнением. Вместо написания HTML структуры для каждого шаблона, каждый шаблон будет *расширять* базовый и перезаписывать отдельные секции.

flaskr/templates/base.html

```
<!doctype html>
<title>{% block title %}{% endblock %} - Flaskr</title>
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
<nav>
<h1>Flaskr</h1>
<ul>
    {% if g.user %}
    <li><span>{{ g.user['username'] }}</span>
    <li><a href="{{ url_for('auth.logout') }}">Log Out</a>
    {% else %}
    <li><a href="{{ url_for('auth.register') }}">Register</a>
    <li><a href="{{ url_for('auth.login') }}">Log In</a>
    {% endif %}
</ul>
</nav>
<section class="content">
<header>
    {% block header %}{% endblock %}
</header>
{% for message in get_flashed_messages() %}
    <div class="flash">{{ message }}</div>
{% endfor %}
{% block content %}{% endblock %}
</section>
```

g автоматически доступна в шаблонах. В зависимости от g.user показываются либо имя пользователя и ссылка на выход, либо ссылка на регистрацию и ссылка на вход. url_for() тоже автоматически доступна и используется для генерации URL адресов на представления вместо ручного их написания.

После заголовка страницы и перед основным содержимым шаблон в цикле проходит по всем сообщениям, возвращаемым get_flashed_messaged(). Функция flask() использовалась в представлениям для показа сообщений об ошибках и этот код их показывает.

Здесь определены 3 блока, которые будут переписаны другими шаблонами:

1. {% block title %} будет изменять отображаемый в браузере заголовок вкладки.
2. {% block header %} подобен title, но будет изменять заголовок, отображаемый на странице.
3. {% block content %} будет отображать содержимое каждой страницы, например форму входа или пост блога.

Этот базовый шаблон размещается прямо в каталоге templates. Для организованного хранения остальных, разместим их в каталогах по названиям эскизов.

Регистрация

flaskr/templates/auth/register.html

```
{% extends 'base.html' %}

{% block header %}
<h1>{% block title %}Register{% endblock %}</h1>
{% endblock %}

{% block content %}
<form method="post">
    <label for="username">Username</label>
    <input name="username" id="username" required>
    <label for="password">Password</label>
    <input type="password" name="password" id="password" required>
    <input type="submit" value="Register">
</form>
{% endblock %}
```

`{% extends 'base.html' %}` говорит Jinja что этот шаблон заменяет блоки базового шаблона. Все отображаемое содержимое должно содержаться внутри блока `{% block %}`, которые перезаписывают блоки базового шаблона.

Здесь используется полезный паттерн размещения блока `{% block title %}` внутри блока `{% block header %}`. Это установит `title block` и затем итоговое значение будет использовано в `header block`. Так что оба заголовка используют одно и тоже название, без необходимости писать его дважды.

Вход

Этот шаблон аналогичен шаблону регистрации, за исключением заголовка и кнопки submit.

flaskr/templates/auth/login.html

```
{% extends 'base.html' %}

{% block header %}
<h1>{% block title %}Log In{% endblock %}</h1>
{% endblock %}

{% block content %}
<form method="post">
    <label for="username">Username</label>
    <input name="username" id="username" required>
    <label for="password">Password</label>
    <input type="password" name="password" id="password" required>
    <input type="submit" value="Log In">
</form>
{% endblock %}
```

Регистрация

Теперь когда все шаблоны аутентификации написаны, можно зарегистрировать пользователя. Убедитесь что сервер запущен и перейдите по адресу http://127.0.0.1:5000/auth/register.

Попробуйте нажать на кнопку «Register» без заполнения формы и посмотрите на ошибки.

При успешном заполнении имени пользователя и пароля, вы будете перенаправлены на страницу логина. Попробуйте ввести некорректный логин или пароль.

Если вы войдете в систему, то все равно должны увидеть ошибку, т.к. еще нет представления для `index`.

2.2.5 Статические файлы

Представления аутентификации и шаблоны выглядят слишком примитивными. Чтобы немного стилизовать HTML добавим CSS. Таблицы стилей являются статическими файлами.

Flask автоматически добавляет представление `static`, которое принимает путь относительно каталога `flaskr/static` и обрабатывает его. Базовый шаблон `base.html` уже имеет статическую ссылку на файл `style.css`:

```
{{ url_for('static', filename='style.css') }}
```

Кроме CSS могут быть и другие типы статических файлов: JavaScript, изображения и т.п. Все они располагаются в каталог `flaskr/static` и для ссылки используется `url_for('static', filename='...')`.

Здесь мы не делаем упор на изучение CSS, так что просто скопируем содержимое файла в:

`flaskr/static/style.css`

```
html { font-family: sans-serif; background: #eee; padding: 1rem; }
body { max-width: 960px; margin: 0 auto; background: white; }
h1 { font-family: serif; color: #377ba8; margin: 1rem 0; }
a { color: #377ba8; }
hr { border: none; border-top: 1px solid lightgray; }
nav { background: lightgray; display: flex; align-items: center; padding: 0 0.5rem; }
nav h1 { flex: auto; margin: 0; }
nav h1 a { text-decoration: none; padding: 0.25rem 0.5rem; }
nav ul  { display: flex; list-style: none; margin: 0; padding: 0; }
nav ul li a, nav ul li span, header .action { display: block; padding: 0.5rem; }
.content { padding: 0 1rem 1rem; }
.content > header { border-bottom: 1px solid lightgray; display: flex; align-items: flex-
↪end; }
.content > header h1 { flex: auto; margin: 1rem 0 0.25rem 0; }
.flash { margin: 1em 0; padding: 1em; background: #cae6f6; border: 1px solid #377ba8; }
.post > header { display: flex; align-items: flex-end; font-size: 0.85em; }
.post > header > div:first-of-type { flex: auto; }
.post > header h1 { font-size: 1.5em; margin-bottom: 0; }
.post .about { color: slategray; font-style: italic; }
.post .body { white-space: pre-line; }
.content:last-child { margin-bottom: 0; }
.content form { margin: 1em 0; display: flex; flex-direction: column; }
.content label { font-weight: bold; margin-bottom: 0.5em; }
.content input, .content textarea { margin-bottom: 1em; }
```

(continues on next page)

(продолжение с предыдущей страницы)

```
.content textarea { min-height: 12em; resize: vertical; }
input.danger { color: #cc2f2e; }
input[type=submit] { align-self: start; min-width: 10em; }
```

Откройте ссылку http://127.0.0.1:5000/auth/login и сейчас страница должна выглядеть как на картинке.

Больше о CSS можно узнать из документации https://developer.mozilla.org/docs/Web/CSS

2.2.6 Эскиз блога

Будут использованы те же подходы, что и для написания эскиза аутентификации. Блог показывает список всех постов, позволяя авторизованным пользователям создавать посты, а авторам менять и удалять посты.

Во время реализации каждого представления, оставляйте сервер запущенным (в режиме разработчика). После сохранения изменений, переходите по соответствующему URL в браузере и тестируйте его.

Объявим эскиз и добавим его в *фабрику приложения*.

`flaskr/blog.py`

```
from flask import (
    Blueprint, flash, g, redirect, render_template, request, url_for
)
from werkzeug.exceptions import abort

from flaskr.auth import login_required
from flaskr.db import get_db

bp = Blueprint('blog', __name__)
```

Импортируйте и зарегистрируйте эскиз.

flaskr/__init__.py

```
def create_app():
    app = ...
    # existing code omitted

    from . import blog
    app.register_blueprint(blog.bp)
    app.add_url_rule('/', endpoint='index')

    return app
```

В отличие от эскиза аутентификации, эскиз блога не имеет `url_prefix`. Таким образом представление `index` будет размещаться в корне `/`, представление `create` по адресу `/create` и т.д. Блог - основная функция Flaskr и логично сделать `index` основным представлением.

Однако, точка входа для `index` будет `blog.index`. Некоторые представления аутентификации ссылаются на простой `index`. `app.add_url_rule()` связывает точку входа `index` с адресом `/`, так что `url_for('index')` и `url_for('blog.index')` будут работать одинаково, генерируя одинаковый адрес URL.

В других приложениях вам может потребоваться дать другой `url_prefix` для эскиза и определить другой `index` для приложения, подобный представлению `hello`.

Представление Index

Индекс будет показывать все посты, начиная с последних. SQL запрос использует JOIN для получения аутентификационной информации из таблицы `users`.

flaskr/blog.py

```
@bp.route('/')
def index():
    db = get_db()
    posts = db.cursor().execute(
        'SELECT p.id, title, body, created, author_id, username'
        ' FROM posts p JOIN users u ON p.author_id = u.id'
        ' ORDER BY created DESC'
    ).fetchallmap()
    return render_template('blog/index.html', posts=posts)
```

flaskr/templates/blog/index.html

```
{% extends 'base.html' %}

{% block header %}
<h1>{% block title %}Posts{% endblock %}</h1>
{% if g.user %}
    <a class="action" href="{{ url_for('blog.create') }}">New</a>
{% endif %}
{% endblock %}

{% block content %}
```

(continues on next page)

(продолжение с предыдущей страницы)

```
{% for post in posts %}
    <article class="post">
    <header>
        <div>
        <h1>{{ post['title'] }}</h1>
        <div class="about">by {{ post['username'] }} on {{ post['created'].strftime('%Y-
↪%m-%d') }}</div>
        </div>
        {% if g.user['id'] == post['author_id'] %}
        <a class="action" href="{{ url_for('blog.update', id=post['id']) }}">Edit</a>
        {% endif %}
    </header>
    <p class="body">{{ post['body'] }}</p>
    </article>
    {% if not loop.last %}
    <hr>
    {% endif %}
{% endfor %}
{% endblock %}
```

Когда пользователь авторизован, блок `header` добавляет ссылку на представление `create`. Когда пользователь автор поста, он увидит ссылку «Edit» связанную с представлением `update` для поста. `loop.last` - это специальная переменная, доступная внутри циклов Jinja. Она используется для исключения печати разделительной линии для последнего поста.

Представление Create

Представление `create` походе на представление `register`. Либо отображается форма для заполнения данных, либо введенные данные проверяются и пост добавляется в базу данных или показывается ошибка.

Декоратор `login_required`, который мы написали ранее, будет использован для представлений блога. Пользователь должен быть авторизован, чтобы открывать эти представления, иначе он будет перенаправлен на страницу входа.

`flaskr/blog.py`

```
@bp.route('/create', methods=('GET', 'POST'))
@login_required
def create():
    if request.method == 'POST':
        title = request.form['title']
        body = request.form['body']
        error = None

        if not title:
            error = 'Title is required.'

        if error is not None:
            flash(error)
        else:
            db = get_db()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            db.cursor().execute(
                'INSERT INTO posts (title, body, author_id)'
                ' VALUES (?, ?, ?)',
                (title, body, g.user['id'])
            )
            db.commit()
            return redirect(url_for('blog.index'))

    return render_template('blog/create.html')
```

flaskr/templates/blog/create.html

```
{% extends 'base.html' %}

{% block header %}
<h1>{% block title %}New Post{% endblock %}</h1>
{% endblock %}

{% block content %}
<form method="post">
    <label for="title">Title</label>
    <input name="title" id="title" value="{{ request.form['title'] }}" required>
    <label for="body">Body</label>
    <textarea name="body" id="body">{{ request.form['body'] }}</textarea>
    <input type="submit" value="Save">
</form>
{% endblock %}
```

Представление Update

Представлениям `update` и `delete` необходимо извлечь пост по его идентификатору и сравнить автора с зарегистрированным пользователем. Чтобы избежать дублирования кода, напишем функцию получения поста и в дальнейшем используем ее в обоих представлениях.

flaskr/blog.py

```
def get_post(id, check_author=True):
    post = get_db().cursor().execute(
        'SELECT p.id, title, body, created, author_id, username'
        ' FROM posts p JOIN users u ON p.author_id = u.id'
        ' WHERE p.id = ?',
        (id,)
    ).fetchonemap()

    if post is None:
        abort(404, f"Post id {id} doesn't exist.")

    if check_author and post['author_id'] != g.user['id']:
        abort(403)

    return post
```

`abort()` выбрасывает специальное исключение, которое возвращает код статуса HTTP. Она принимает текст ошибки. Код 404 означает что страница не найдена (Not Found), а код 403 означает что доступ к странице запрещен (Forbidden).

Аргумент `check_author` может быть полезен в будущем.

Код представления `update`.

`flaskr/blog.py`

```
@bp.route('/<int:id>/update', methods=('GET', 'POST'))
@login_required
def update(id):
    post = get_post(id)

    if request.method == 'POST':
        title = request.form['title']
        body = request.form['body']
        error = None

        if not title:
            error = 'Title is required.'

        if error is not None:
            flash(error)
        else:
            db = get_db()
            db.cursor().execute(
                'UPDATE posts SET title = ?, body = ?'
                ' WHERE id = ?',
                (title, body, id)
            )
            db.commit()
            return redirect(url_for('blog.index'))

    return render_template('blog/update.html', post=post)
```

В отличие от представлений, что мы писали до этого, `update` принимает аргумент, `id`. На это указывает `<int:id>`. Реальный адрес будет выглядеть, например, `/1/update`. Flask извлечет 1 из адреса, убедиться что это значение типа `int` и передаст его как аргумент функции. Если не указать `int:` то аргумент будет `string`. Чтобы генерировать адреса для страницы обновления, `url_for()` необходимо передать `id` для заполнения: `url_for('blog.update', id=post['id'])`.

Представления `create` и `update` похожи друг на друга. Основное отличие в том, что `update` использует объект `post` и запрос `UPDATE` вместо `INSERT`. Теоретически можно придумать единый шаблон для этих двух представлений, но для наших целей мы не будем усложнять.

`flaskr/templates/blog/update.html`

```
{% extends 'base.html' %}

{% block header %}
<h1>{% block title %}Edit "{{ post['title'] }}"{% endblock %}</h1>
{% endblock %}

{% block content %}
```

(continues on next page)

(продолжение с предыдущей страницы)

```
<form method="post">
    <label for="title">Title</label>
    <input name="title" id="title"
    value="{{ request.form['title'] or post['title'] }}" required>
    <label for="body">Body</label>
    <textarea name="body" id="body">{{ request.form['body'] or post['body'] }}</textarea>
    <input type="submit" value="Save">
</form>
<hr>
<form action="{{ url_for('blog.delete', id=post['id']) }}" method="post">
    <input class="danger" type="submit" value="Delete" onclick="return confirm('Are you␣
↪sure?');">
</form>
{% endblock %}
```

Шаблон имеет две формы. Первая отправляет отредактированные данные на текущую страницу `/<id>/update`. Вторая форма содержит единственную кнопку и определяет атрибут `action` ссылающийся на представление `delete`. Кнопка использует JavaScript для запроса подтверждения действия.

Паттерн `{{ request.form['title'] or post['title'] }}` используется в зависимости от данных, которые надо показать в форме. Когда форма не отправлена, показываются данные оригинального поста, но если в форму переданы некорректные данные, необходимо показать это чтобы пользователь исправил ошибку. Для этого используется `request.form`. `request` - это еще одна переменная, автоматически доступная в шаблонах.

Представление Delete

Это представление не имеет своего шаблона, а кнопка удаления является частью шаблона `update.html`. Таким образом, достаточно написать только функцию представление и обработать только метод `POST`, а потом перенаправить на представление `index`.

`flaskr/blog.py`

```
@bp.route('/<int:id>/delete', methods=('POST',))
@login_required
def delete(id):
    get_post(id)
    db = get_db()
    db.cursor().execute('DELETE FROM posts WHERE id = ?', (id,))
    db.commit()
    return redirect(url_for('blog.index'))
```

Важно: На этом разработка первого приложения завершена. Необходимо его тщательно протестировать.

Для изучения того, как тестировать и распространять подобные приложения рекомендуем изучить материалы по Flask - https://flask.palletsprojects.com/en/2.0.x/tutorial