

Приложение «Книжный магазин» Приложение 2.0



Авторские права

© Postgres Professional, 2017–2024

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

edu@postgrespro.ru

Отказ от ответственности

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Обзор приложения «Книжный магазин 2.0»

Схема данных

Интерфейс с клиентской частью

Разграничение доступа

Интернет-магазин для покупателей

- поиск книг

- детальная информация о выбранной книге

- корзина для зарегистрированных пользователей

«Админка» для сотрудников

- заказ книг у поставщика

- установка розничной цены

- фоновые задания: запуск, просмотр результатов

В этом курсе, как и в DEV1, мы будем использовать приложение «Книжный магазин», но новой версии 2.0. Клиентская часть полностью готова, а серверную мы будем улучшать по мере прохождения курса.

Как и раньше, клиентская часть состоит из интернет-магазина и «админки». Однако со времен DEV1 магазин вырос и число покупателей увеличилось.


Теперь, чтобы приобрести книги, покупатели должны регистрироваться на сайте и входить под своим именем. В магазине теперь есть корзина, а книги имеют цену.


Админка позволяет заказывать книги и устанавливать им цену. Добавление книг и авторов из админки исключено из приложения 2.0.

Зато добавилась возможность запускать фоновые задания (например, отчеты) и просматривать их результаты после завершения.

Приложение предназначено исключительно для демонстрации концепций, излагаемых в этом курсе. Оно умышленно сделано крайне упрощенным и не может служить образцом проектирования реальных систем.

Приложение





Учебное приложение DEV2

Книжный магазин

Админка

Вход

Регистрация







SQL

Найти

Сортировать по

рейтингу

↓

	Основы технологий ба... Новиков Б. А., Горшков...	★★★★★	240 с.	70x100/16	640 Р
	PostgreSQL: Основы яз... Моргунов Е. П.	★★★★★	336 с.	70x100/16	660 Р
	SQL и реляционная те... Дейт К.	★★★★★	480 с.	70x100/16	1420 Р
	Рефакторинг SQL-прил... Фаро С., Лерми П.	★★★★★	336 с.	70x100/16	980 Р
	Oracle. Основы стоимо... Льюис Д.	★★★★★	528 с.	70x100/16	810 Р
	Oracle. Оптимизация п... Миллс К., Холт Д.	★★★★★	464 с.	70x100/16	1920 Р

пул соединений (6432) ▾

☐ Trace

```
select to_json(f.*) from
webapi.get_image
(book_id=>$1) f
[7]
```

SELECT 1 83ms/251ms

web@localhost:6432 (pid 457211)

```
select to_json(f.*) from
webapi.get_image
(book_id=>$1) f
[8]
```

SELECT 1 30ms/99ms

web@localhost:6432 (pid 457211)

```
select to_json(f.*) from
webapi.get_image
(book_id=>$1) f
[11]
```

SELECT 1 25ms/152ms

4

Приложение состоит из двух частей, представленных вкладками.

- «Книжный магазин» — это интерфейс веб-пользователя, в котором он может просматривать и покупать книги.
- «Админка» — интерфейс сотрудников магазина, в котором они могут заказывать книги и устанавливать их цену, а также выполнять фоновые задания.

В учебных целях вся функциональность представлена на одной общей веб-странице. Если какая-то часть функциональности недоступна из-за того, что на сервере нет подходящего объекта, приложение сообщит об этом. Также приложение выводит текст запросов, которые оно посылает на сервер.

Приложение позволяет выбрать сервер для подключения. В основном мы будем использовать значение по умолчанию.

Исходный код приложения не является темой курса, но может быть получен в git-репозитории <https://pubgit.postgrespro.ru/pub/dev2app.git>

Демонстрация приложения

В этой демонстрации мы показываем приложение «Книжный магазин 2.0» в том виде, в котором оно будет после завершения всех практических заданий. Приложение доступно в браузере виртуальной машины курса по адресу <http://localhost/>

Открываем файл <http://localhost/>...

```
student$ xdg-open http://localhost
```

Основные сущности



Основные сущности нашей базы данных практически не изменились со времен курса DEV1. Это:

- **Книга.** К книгам добавились атрибуты.
- **Автор.** Книги и авторы связаны отношением многие-ко-многим (авторство).
- **Операции** с книгами: покупки в магазине и поступления на склад. К атрибутам добавилась цена книги.

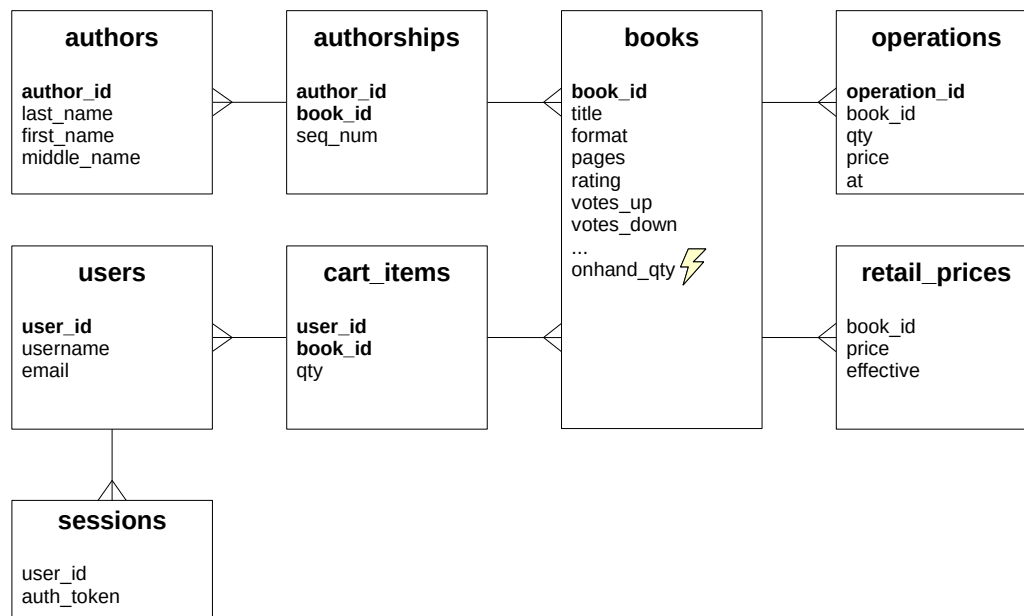
Новые сущности:

- **Розничная цена.** Мы рассматриваем цену как отдельную сущность, а не атрибут книги, поскольку она может меняться со временем, то есть имеет диапазон дат действия.
- **Пользователь** веб-магазина. Определяется именем (логином) и имеет почтовый адрес.

Пользователь может класть книги в корзину, и поэтому связан с книгами отношением многие-ко-многим. (Отдельной сущности для корзины мы не предусматриваем.)

- **Сеанс** работы пользователя с магазином. Сеанс связан с вопросами аутентификации и разграничения доступа.

Основные таблицы



7

Основные таблицы базы данных представлены на слайде.

В качестве идентификаторов используются суррогатные ключи, генерируемые с помощью последовательностей.

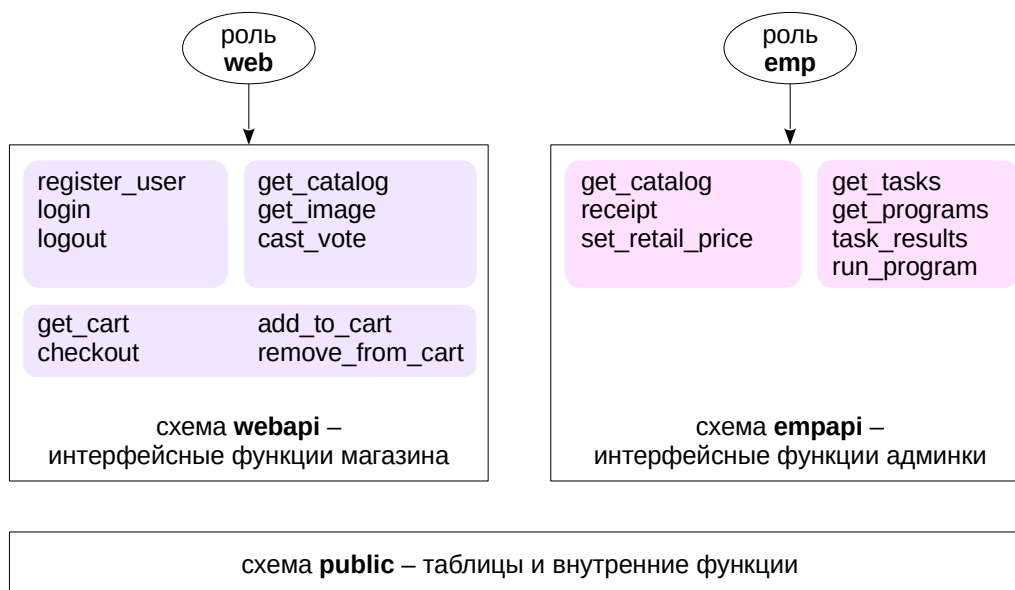
Связи «многие ко многим» представлены дополнительными таблицами:

- **authorships** — авторство;
- **cart_items** — книги в корзине.

Отметим, что в таблице книг **books** имеется дополнительный столбец `onhand_qty`, содержащий текущее количество книг на складе магазина, и обновляемый триггером по таблице операций.

С расчетом рейтинга книг (`rating`) связаны еще два столбца: голоса пользователей «за» (`votes_up`) и «против» (`votes_down`).

Интерфейс с клиентом



8

Клиентский API серверной части полностью построен на функциях.

Интерфейсные функции веб-магазина доступны только пользователю **web** и размещены в схеме **webapi**:

- аутентификация и разграничение доступа;
- работа с каталогом книг;
- операции с корзиной.

Интерфейсные функции админки доступны только пользователю **emp** и размещены в схеме **empapi**:

- работа с каталогом книг;
- фоновые задания.

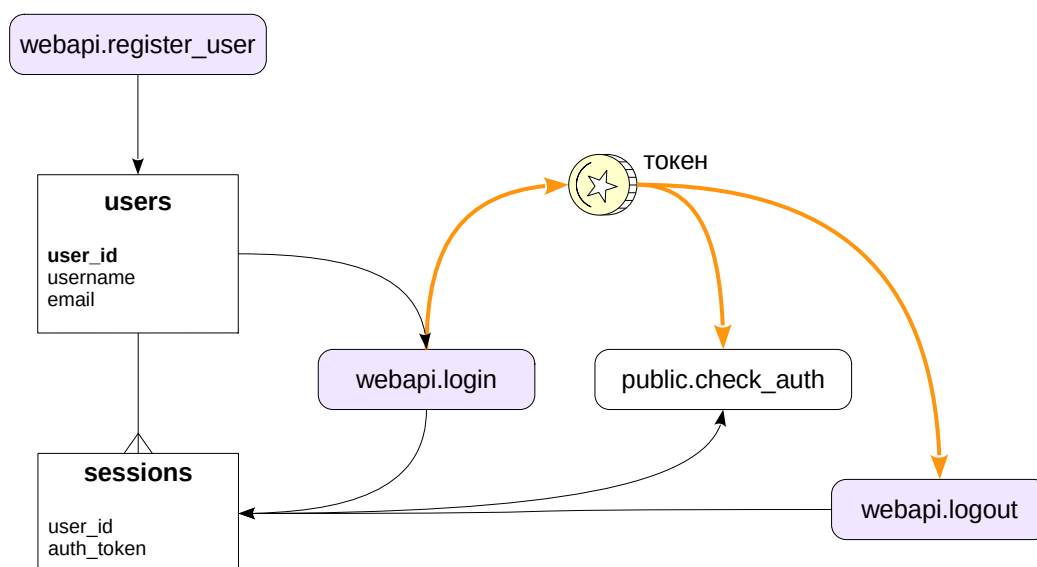
Аутентификация в админке не предусмотрена.

Все таблицы, а также внутренние функции, не открытые напрямую для клиента, размещены в схеме **public**.

Интерфейсные функции объявлены как SECURITY DEFINER, чтобы иметь доступ к объектам базы данных. С помощью механизма привилегий по умолчанию (ALTER DEFAULT PRIVILEGES) доступ к функциям в схеме **webapi** автоматически выдается только пользователю **web**, в схеме **empapi** — пользователю **emp**, а доступ к функциям в схеме **public** отбирается у роли public.

(Вопросы разграничения доступа рассматриваются в курсе DEV1.)

Сейчас мы рассмотрим интерфейс подробнее, но обратите внимание, что информационная панель веб-клиента показывает, какие вызовы он отправляет на сервер.



Аутентификация пользователей интернет-магазина происходит на клиенте. С точки зрения базы данных клиент всегда представлен одной ролью **web**.

Новый пользователь регистрируется вызовом **register_user**. Для простоты мы не используем пароли (но если бы использовали, то хеш пароля хранился бы в таблице **users**).

Чтобы иметь возможность совершать покупки, пользователь должен войти в систему. Это выполняет функция **login**. Она создает сеанс пользователя, который определяется *токеном* (UUID).

Токен возвращается клиенту и дальше клиент передает его как параметр во все функции, связанные с покупками. Каждая такая функция первым делом проверяет правильность токена с помощью вызова **check_auth**, который определяет по токену имя пользователя.

Наконец, вызов **logout** завершает сеанс.

Функционал истечения срока сеанса не реализован, но может быть легко добавлен.

Аутентификация

```
=> \c bookstore2
```

You are now connected to database "bookstore2" as user "student".

У нас уже есть два зарегистрированных пользователя:

```
=> SELECT * FROM users;
```

user_id	username	email
1	alice	alice@localhost
2	bob	bob@localhost

(2 rows)

Зарегистрируем еще одного. Почтовый адрес можно указывать любой — мы будем отправлять пользователям письма, но все они попадут в локальный почтовый ящик пользователя student.

```
=> SELECT webapi.register_user('charlie', 'charlie@localhost');
```

```
register_user
-----
```

(1 row)

Пользователь входит в систему и получает токен:

```
=> SELECT webapi.login('charlie');
```

```
login
-----
```

```
3d748d1a-d957-4ef0-89e0-0f3f97493d12
```

(1 row)

При этом в базе появляется сеанс:

```
=> SELECT * FROM sessions;
```

auth_token	user_id
ce40771d-734a-45a4-b92a-8291c0812c1d	1
3d748d1a-d957-4ef0-89e0-0f3f97493d12	3

(2 rows)

Токен можно проверить функцией, закрытой для клиента:

```
=> SELECT username
FROM users
WHERE user_id = check_auth('00000000-0000-0000-0000-000000000000');
```

ERROR: query returned no rows

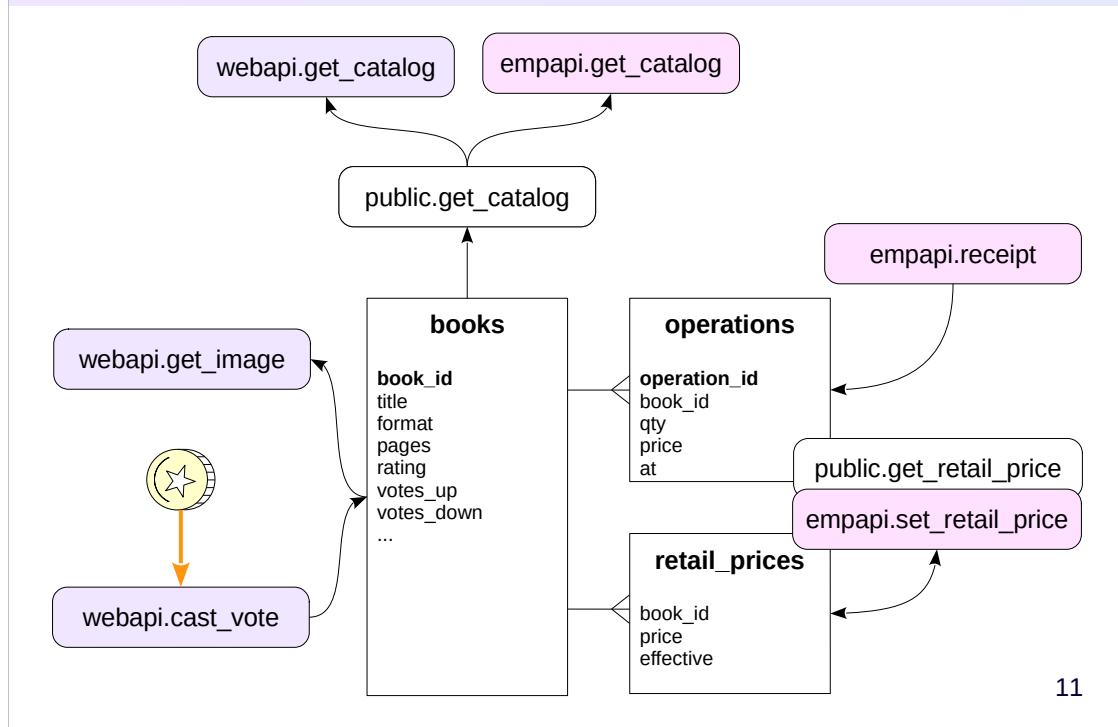
CONTEXT: PL/pgSQL function check_auth(uuid) line 5 at SQL statement

```
=> SELECT username
FROM users
WHERE user_id = check_auth('3d748d1a-d957-4ef0-89e0-0f3f97493d12');
```

```
username
-----
```

```
charlie
```

(1 row)



Список книг (конечно, вместе с авторами, хотя эти таблицы не показаны на слайде для краткости) нужен как магазину, так и админке — но с разным набором полей. Всю возможную информацию выбирает закрытая функция **public.get_catalog**, получая на вход параметры поиска. Функции **webapi.get_catalog** и **empapi.get_catalog** реализованы как обертки вокруг **public.get_catalog**.

Получение обложки книги выполняет функция **webapi.get_image**. Обложки не входят в **get_catalog**, чтобы клиент как можно быстрее отобразил результаты поиска, а обложки подгружал в фоновом режиме.

Голосование «за» или «против» книги выполняется функцией **webapi.cast_vote**. Функция допускает голосование несколько раз, но работает только для зарегистрированных пользователей.

Заказ книги у поставщика в админке выполняет функция **empapi.receive**. Она создает соответствующую операцию с книгой.

За установку розничной цены на книгу отвечает функция **empapi.set_retail_price**. Текущую цену возвращает функция **public.get_retail_price**, которую клиент никогда не вызывает напрямую, но она используется во многих интерфейсных функциях.

Каталог книг

Информацию о книгах клиент получает функцией `get_catalog`. Например, для интернет-магазина:

```
=> SELECT book_id, title, authors_list, format, rating, price
FROM webapi.get_catalog('рефакторинг','rating','asc') \gx

-[ RECORD 1 ]+-----
book_id      | 6
title        | Рефакторинг SQL-приложений
authors_list | {(7,Фаро,Стефан,\"\"), (8,Лерми,Паскаль,\"\" )}
format       | 70x100/16
rating       | 0
price        | 980
-[ RECORD 2 ]+-----
book_id      | 14
title        | Чистый код: создание, анализ и рефакторинг
authors_list | {(18,Мартин,Роберт,\"\" )}
format       | 70x100/16
rating       | 0
price        | 1350
```

Установим розничную цену для одной книги:

```
=> SELECT empapi.set_retail_price(6, 1000.00, now());

      set_retail_price
-----
(6,1000.00,["2024-08-14 11:04:21.847599+03",""])
(1 row)
```

```
=> SELECT book_id, price
FROM webapi.get_catalog('рефакторинг','rating','asc');

 book_id | price
-----+-----
        6 | 1000.00
       14 | 1350
(2 rows)
```

Поступление 50 книг по 100 Р на склад:

```
=> SELECT empapi.receipt(6, 50, 100.00);

 receipt
-----
(1 row)
```

Этот вызов создает соответствующую операцию:

```
=> SELECT * FROM operations
WHERE book_id = 6
ORDER BY operation_id DESC
LIMIT 1 \gx

-[ RECORD 1 ]+-----
operation_id | 34675
book_id      | 6
qty          | 50
price        | 100.00
at           | 2024-08-14 11:04:21.967527+03
```

Пользователь может голосовать за книгу:

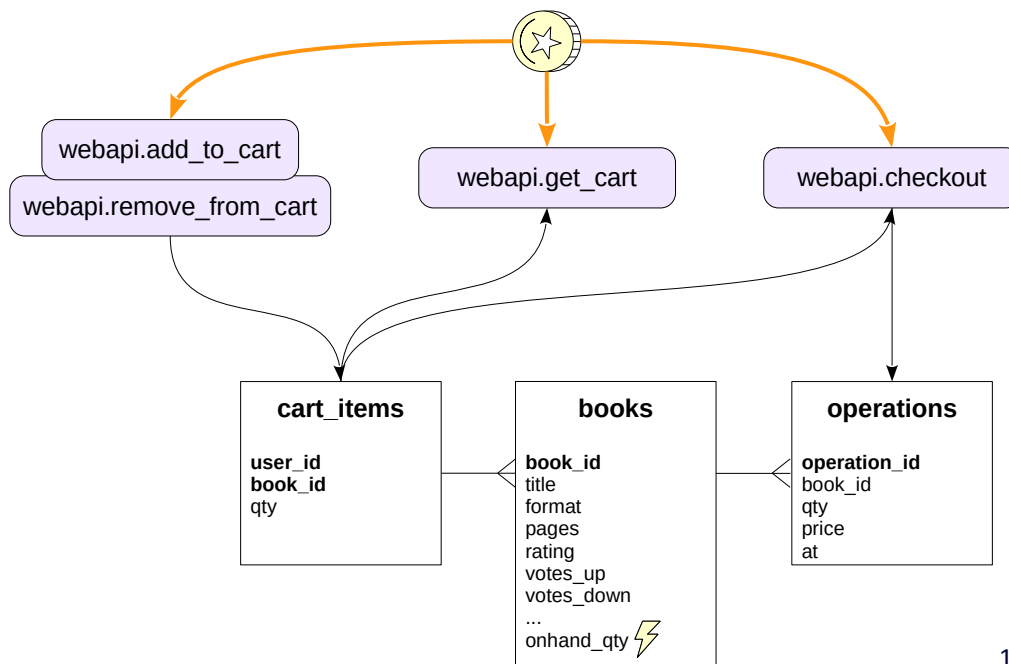
```
=> SELECT webapi.cast_vote('3d748d1a-d957-4ef0-89e0-0f3f97493d12',6,+1);

 cast_vote
-----
(1 row)
```

```
=> SELECT book_id, votes_up, votes_down  
FROM webapi.get_catalog('рефакторинг', 'rating', 'asc');
```

book_id	votes_up	votes_down
6	1	0
14	0	0

(2 rows)



Все функции, относящиеся к покупке, работают от имени конкретного пользователя магазина и поэтому требуют токена.

Функция `webapi.add_to_cart` добавляет в корзину одну книгу или убирает из корзины одну книгу. Функция `webapi.remove_from_cart` полностью удаляет всю позицию из корзины.

Функция `webapi.get_cart` возвращает содержимое корзины.

Функция `webapi.checkout` совершает покупку, убирая позиции из корзины и создавая соответствующие операции с книгами.

Весь процесс взаимодействия с пользователем во время покупки книг не может быть выполнен в рамках одной транзакции СУБД, поскольку этот процесс занимает неизвестное (большое) время. Вместо этого мы позволяем пользователю добавлять книги в корзину, не обращая внимания на наличие необходимого количества книг на складе. Каждое добавление происходит в отдельной (очень короткой) транзакции, так что в базе данных не возникает никаких долговременных блокировок. Фактическая проверка количества происходит только в транзакции при покупке: если на складе будет недостаточно книг, ограничение целостности в базе данных не позволит выполнить транзакцию.

(В реальной системе необходимо было бы предусмотреть резервирование товара и фиксацию цены на время, отведенное для оплаты онлайн. Конечно, и это действие тоже не должно приводить к длинным транзакциям.)

Корзина

Положим книги в корзину:

```
=> SELECT webapi.add_to_cart(
    auth_token => '3d748d1a-d957-4ef0-89e0-0f3f97493d12',
    book_id => 6,
    qty => +1 -- по умолчанию
);
```

```
add_to_cart
-----
(3,6,1)
(1 row)
```

```
=> SELECT webapi.add_to_cart(
    auth_token => '3d748d1a-d957-4ef0-89e0-0f3f97493d12',
    book_id => 6
);
```

```
add_to_cart
-----
(3,6,1)
(1 row)
```

```
=> SELECT webapi.add_to_cart(
    auth_token => '3d748d1a-d957-4ef0-89e0-0f3f97493d12',
    book_id => 3
);
```

```
add_to_cart
-----
(3,3,1)
(1 row)
```

```
=> SELECT webapi.add_to_cart(
    auth_token => '3d748d1a-d957-4ef0-89e0-0f3f97493d12',
    book_id => 1
);
```

```
add_to_cart
-----
(3,1,1)
(1 row)
```

Вот что у нас в корзине:

```
=> SELECT *
FROM webapi.get_cart('3d748d1a-d957-4ef0-89e0-0f3f97493d12') \gx
```

```
-[ RECORD 1 ]+-----
book_id      | 1
title        | Основы технологий баз данных
authors_list | {"(1,Новиков,Борис,Асенович)","(2,Горшкова,Екатерина,Александровна)"}
qty          | 1
onhand_qty   | 34
price        | 640
-[ RECORD 2 ]+-----
book_id      | 3
title        | PostgreSQL: Основы языка SQL
authors_list | {"(4,Моргунов,Евгений,Павлович)"}
qty          | 1
onhand_qty   | 80
price        | 660
-[ RECORD 3 ]+-----
book_id      | 6
title        | Рефакторинг SQL-приложений
authors_list | {"(7,Фаро,Стефан,\"\""), "(8,Лерми,Паскаль,\"\"")"}
qty          | 2
onhand_qty   | 90
price        | 1000.00
```

Уберем одну книгу:

```
=> SELECT webapi.remove_from_cart(
    auth_token => '3d748d1a-d957-4ef0-89e0-0f3f97493d12',
    book_id => 1
);

remove_from_cart
-----

(1 row)
```

И совершим покупку:

```
=> SELECT * FROM webapi.checkout('3d748d1a-d957-4ef0-89e0-0f3f97493d12');

checkout
-----

(1 row)
```

Что осталось в корзине?

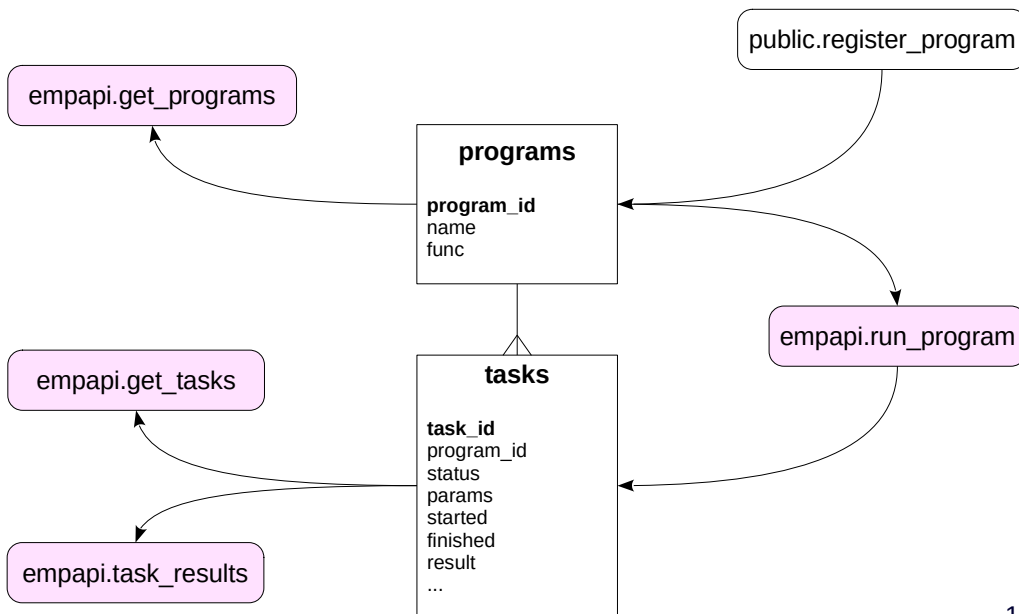
```
=> SELECT book_id, title, qty, onhand_qty, price
FROM webapi.get_cart('3d748d1a-d957-4ef0-89e0-0f3f97493d12') \gx

(0 rows)
```

Конечно, ничего. Зато появились операции покупки:

```
=> SELECT * FROM operations
ORDER BY operation_id DESC
LIMIT 2 \gx

-[ RECORD 1 ]+-----
operation_id | 34677
book_id      | 3
qty          | -1
price        | 660
at           | 2024-08-14 11:04:22.700245+03
-[ RECORD 2 ]+-----
operation_id | 34676
book_id      | 6
qty          | -2
price        | 1000.00
at           | 2024-08-14 11:04:22.700245+03
```

Для фоновых заданий используются еще две таблицы, не показанные на основной схеме:

- **programs** — программы, которые можно запускать.
У программы есть название и имя «исполняемой» функции.
- **tasks** — собственно задания, то есть экземпляры программ.
Задание имеет статус (запланировано к запуску, работает и т. п.), значения параметров, дату запуска и окончания работы, результат выполнения и другие.

Программа регистрируется функцией `public.register_program`. Клиент не вызывает эту функцию, поэтому она находится в схеме `public`.

Задание ставится на выполнение функцией `empapi.run_program`.

Несколько функций предназначены для получения информации:

- о списке программ — `empapi.get_programs`;
- о списке заданий — `empapi.get_tasks`;
- о результате выполнения — `empapi.task_results`.

Фоновые задания

Список зарегистрированных программ, которые можно выполнять как фоновые задания:

```
=> SELECT empapi.get_programs();

      get_programs
-----
(2,"Отправка письма",sendmail_task)
(3,"Отчет по складским остаткам",stock_task)
(1,Приветствие,greeting_task)
(3 rows)
```

Список фоновых заданий:

```
=> SELECT * FROM empapi.get_tasks() \gx

-[ RECORD 1 ]-----
task_id      | 2
program_id   | 2
name         | Отправка письма
host         |
port         |
started      | 2024-08-14 11:04:21.7617+03
finished     | 2024-08-14 11:04:22.764797+03
status       | finished
-[ RECORD 2 ]-----
task_id      | 1
program_id   | 1
name         | Приветствие
host         |
port         |
started      | 2024-08-14 11:03:53.538517+03
finished     | 2024-08-14 11:03:54.546073+03
status       | finished
```

Поставим в очередь на выполнение еще одно задание «Приветствие»:

```
=> SELECT empapi.run_program(1);

      run_program
-----
              3
(1 row)
```

В ответ получаем номер задания. Немного подождем...

Проверим статус задания:

```
=> SELECT status FROM empapi.get_tasks() WHERE task_id = 3;

      status
-----
      finished
(1 row)
```

Задание завершено. Получим результат:

```
=> SELECT * FROM empapi.task_results(3);

      task_results
-----
num  greeting +
---
1    Hello, world!+
2    Hello, world!+
3    Hello, world!+
(1 row)
```

Мы вернемся к фоновым заданиям позже в теме «Асинхронная обработка».

Часть рассмотренного функционала еще только предстоит реализовать

Прежде чем вносить изменения в приложение, необходимо разобраться в том, как оно устроено



Здесь и далее все практические задания, связанные с приложением, выполняются в базе данных bookstore2.

1. Сеанс удаляется, если пользователь выходит из системы, но если просто закрыть вкладку браузера, сеансы будут накапливаться. Реализуйте автоматическое удаление прошлых сеансов пользователя при повторном входе.
2. Реализуйте отсутствующую интерфейсную функцию `webapi.add_to_cart`. Функция должна работать только для пользователей, вошедших в систему. Если книга присутствует в корзине, количество ее экземпляров не может быть меньше одного. Проверьте результат в приложении. Какое обновление таблицы `cart_items` выполняется при изменении количества книг — обычное или HOT?

18

База данных bookstore2 уже создана. При необходимости ее можно пересоздать с помощью скрипта `bookstore2.sql` в домашнем каталоге.

1. Внесите изменение в функцию `webapi.login`: перед тем как создавать новый сеанс, закройте существующие сеансы этого пользователя, используя функцию `webapi.logout`.

Не забудьте выбрать подключение к основному серверу на порту 5432. (Это значение не выбирается по умолчанию, поскольку в следующих темах мы будем работать с пулом соединений.)

2. Функция `webapi.add_to_cart` присутствует в базе, но пуста. Сохраните ее сигнатуру без изменений. Функция принимает параметры:

- `auth_token` — токен;
- `book_id` — идентификатор книги;
- `qty` — количество (может быть +1 или -1, другие значения считаются некорректными).

Если `qty = +1`, надо либо добавить экземпляр книги в корзину (если такой книги еще нет в корзине), либо увеличить количество экземпляров на единицу.

Если `qty = -1`, надо уменьшить количество экземпляров книги в корзине на единицу. При этом количество не должно быть меньше 1 (чтобы полностью удалить книгу из корзины, используется другая функция — `webapi.remove_from_cart`).

Тип обновления можно узнать в таблице `pg_stat_all_tables` (вспомните тему «Архитектура. Многоверсионность»).

1. Удаление сеансов

Добавим в функцию входа удаление существующих сеансов:

```
=> CREATE OR REPLACE FUNCTION webapi.login(username text) RETURNS uuid
AS $$
DECLARE
    auth_token uuid;
    sessions record;
BEGIN
    -- сначала завершим все открытые сеансы
    FOR sessions IN
        SELECT s.auth_token
        FROM sessions s
        JOIN users u ON u.user_id = s.user_id
        WHERE u.username = login.username
    LOOP
        PERFORM webapi.logout(sessions.auth_token);
    END LOOP;
    -- новый сеанс
    INSERT INTO sessions AS s(auth_token, user_id)
        SELECT gen_random_uuid(), u.user_id
        FROM users u
        WHERE u.username = login.username
    RETURNING s.auth_token
        INTO STRICT auth_token; -- ошибка, если пользователя нет
    RETURN auth_token;
END;
$$ LANGUAGE plpgsql VOLATILE SECURITY DEFINER;

CREATE FUNCTION
```

2. Функция добавления в корзину

Во-первых, определим ограничение целостности на таблице cart_items, которое не даст количеству опуститься меньше единицы. Это надежнее и проще, чем реализовывать проверку в коде.

```
=> ALTER TABLE public.cart_items ADD CHECK (qty > 0);
```

```
ALTER TABLE
```

Затем определим функцию add_to_cart. Чтобы не проверять, существует ли книга в корзине, воспользуемся командой INSERT ON CONFLICT.

```
=> CREATE OR REPLACE FUNCTION webapi.add_to_cart(
    auth_token uuid,
    book_id bigint,
    qty integer DEFAULT 1
) RETURNS record
AS $$
<<local>>
DECLARE
    user_id bigint;
    res record;
BEGIN
    user_id := check_auth(auth_token);
    IF qty = 1 THEN
        INSERT INTO cart_items(
            user_id,
            book_id,
            qty
        )
        VALUES (
            user_id,
            book_id,
            1
        )
        ON CONFLICT ON CONSTRAINT cart_items_pkey
        DO UPDATE SET qty = cart_items.qty + 1
        RETURNING local.user_id, add_to_cart.book_id, add_to_cart.qty INTO res;
    ELSIF qty = -1 THEN
        UPDATE cart_items ci
        SET qty = ci.qty - 1
        WHERE ci.user_id = local.user_id
        AND ci.book_id = add_to_cart.book_id
        RETURNING local.user_id, add_to_cart.book_id, add_to_cart.qty INTO res;
    ELSE
        RAISE EXCEPTION 'qty = %, должно быть 1 или -1', qty;
    END IF;
    RETURN res;
END;
$$ LANGUAGE plpgsql VOLATILE SECURITY DEFINER;
```

CREATE FUNCTION

При изменении количества книг будут выполняться HOT-обновления, поскольку обновляемое поле (qty) не входит ни в один индекс.

```
=> SELECT n_tup_upd, n_tup_hot_upd
FROM pg_stat_all_tables
WHERE relid = 'cart_items'::regclass;
```

```

n_tup_upd | n_tup_hot_upd
-----+-----
0 | 0
(1 row)
```

```
=> SELECT webapi.login('alice');
```

```

login
-----
5e8ec4a5-2f8c-4044-8df4-3d83cb08a3b9
(1 row)
```

```
=> SELECT webapi.add_to_cart(
    auth_token => '5e8ec4a5-2f8c-4044-8df4-3d83cb08a3b9',
    book_id => 1
);
```

```

add_to_cart
-----
(1,1,1)
(1 row)
```

```
=> SELECT webapi.add_to_cart(
    auth_token => '5e8ec4a5-2f8c-4044-8df4-3d83cb08a3b9',
    book_id => 1
);
```

```
add_to_cart
-----
(1,1,1)
(1 row)
```

```
=> SELECT webapi.add_to_cart(
      auth_token => '5e8ec4a5-2f8c-4044-8df4-3d83cb08a3b9',
      book_id => 1,
      qty => -1
);
```

```
add_to_cart
-----
(1,1,-1)
(1 row)
```

```
=> SELECT n_tup_upd, n_tup_hot_upd
FROM pg_stat_all_tables
WHERE relid = 'cart_items'::regclass;
```

```
 n_tup_upd | n_tup_hot_upd
-----+-----
          2 |              2
(1 row)
```

1. Если для каждого пользователя магазина создать отдельную роль, аутентификацию можно поручить базе данных.
Хорошая ли это идея и почему?
2. Вместо того, чтобы создавать функции в базе данных, можно открыть доступ к таблицам и реализовать всю логику в приложении. А генерацию запросов переложить на ORM.
Хорошая ли это идея и почему?
3. Приложение реализует собственный механизм фоновых заданий. Вместо этого можно воспользоваться сторонним решением для очередей сообщений.
Хорошая ли это идея и почему?

1. Аутентификация пользователей приложения в базе данных

Это определенно плохая идея.

В приложении могут быть зарегистрированы тысячи и миллионы пользователей. С точки зрения СУБД все они абсолютно равноправны, так что создание для каждого отдельной роли не дает никаких преимуществ при разграничении доступа. А вот неприятностей не оберешься: динамическое и бесконтрольное выполнение команд CREATE ROLE и GRANT приложением ни к чему хорошему не приведет. Такие команды должен выполнять администратор базы данных.

2. Где размещать бизнес-логику?

Этот вопрос из разряда «религиозных», на него нет однозначного ответа. Все зависит от того, на какую часть системы делается акцент.

Подход, при котором бизнес-логика реализуется в приложении, а от базы данных отгораживаются ORM-ом (инструментом, помогающим связать объектную модель данных приложения с реляционной моделью данных базы), удобен для разработчиков, привыкших мыслить категориями процедурных и объектно-ориентированных языков программирования. Здесь во главу угла ставится приложение, а СУБД лишь обеспечивает надежное хранение данных. В качестве плюсов этого подхода часто называют простоту масштабирования и развертывания, независимость от конкретной СУБД.

Подход, при котором бизнес-логика реализуется в базе данных, а приложению предоставляется высокоуровневый интерфейс, удобен для разработчиков, глубоко знакомых с устройством конкретной СУБД и умеющих пользоваться ее возможностями. Здесь во главу угла ставится база и согласованность данных в ней, а приложений может быть и несколько. В качестве плюсов — все возможности языка SQL и реализованных в СУБД алгоритмов, тонкая оптимизация и настройка производительности, исключение пересылок по сети лишних данных.

В этом курсе, во всяком случае, мы всецело на стороне базы данных.

3. Собственная реализация очередей в базе данных

На этот вопрос также нет однозначного ответа.

Сравнение разных решений приведено в теме «Асинхронная обработка».