

[КАК СТАТЬ АВТОРОМ](#)[Финальный питч-дек Битвы](#)[Офер за неделю для бэкендеров](#)0
Рейтинг**Evrone**[Подписаться](#)**Evrone**

22 мар в 19:40

Курс по Ruby+Rails. Часть 7. Модели и ActiveRecord

🕒 14 мин 👁 2.6K

Блог компании Evrone, Ruby*, Ruby on Rails*

[Тutorial](#)

Модели и ActiveRecord

evrone
→ ruby course

На одной из прошлых лекций вы познакомились с паттерном MVC – Model, View, Controller. И сегодня пришло время подробнее разобраться в том, что прячется за первой буквой этой аббревиатуры.

М – это модель. В паттерне MVC этот слой отвечает за работу с данными и бизнес логику.



+3



10



1



Если вы планируете хранить какие-то данные в вашем приложении, вам не обойтись без базы данных. А для удобного взаимодействия с данными в ней не обойтись без моделей.

Проще говоря, модель – это сущность для работы с данными. Она представляет из себя специальный класс, который связывается с определенной таблицей в базе данных. Каждый раз, когда вы захотите получить или изменить данные в базе, вы будете делать это с помощью модели. Для того, чтобы лучше понять, что она из себя представляет и для чего нужна, необходимо познакомиться ещё с парой терминов, а именно – ORM и Active Record.

Object Relational Mapping (ORM, объектно-реляционное отображение) – это техника, которая позволяет связывать объекты приложения с таблицами базы данных, то есть позволяет совмещать объектно-ориентированный стиль программирования и механику реляционных баз данных. В результате модель создает единую сущность из таблицы в базе и объекта, созданного Ruby. Основная задача – упростить работу с источником данных, программист работает с объектами, не занимаясь написанием тривиальных запросов (в реальности, конечно, так не всегда бывает :).

Существуют много разных подходов к реализации ORM, в Rails используется ActiveRecord, являющийся реализацией одноименного паттерна программирования.

ActiveRecord – это паттерн, описанный в книге Patterns of Enterprise Application Architecture Мартина Фаулера. Идея ActiveRecord заключается в объединении в одном объекте логики предметной области и логики работы с базой данных. Таким образом, модель хранит в себе сразу и сами данные, и логику для работы с ними. ActiveRecord в Rails предоставляет механизмы, для представления моделей, их взаимосвязей, операций CRUD (Create, Read, Update, Delete), поиска, валидаций и многого другого.

Как правило, каждой таблице соответствует свой класс, экземпляр этого класса представляет собой строку в таблице, а его структура (атрибуты) соответствует колонкам.

Работа с данными происходит как с помощью методов класса (создания объектов, получение коллекций, обновления множества строк), так и методов экземпляра класса

(манипуляция с отдельными строками). Это обеспечивает удобную среду для доступа к базе данных и сохранения в ней объектов.

В результате, ActiveRecord позволяет нам работать со строкой таблицы, как с обычным объектом, обращаясь к полям таблицы через, как правило, одноименные, методы этого объекта. Кроме того, в нашем распоряжении оказывается большое разнообразие методов для построения запросов к базе на получение различных выборок без необходимости использовать SQL. Ещё модель может выполнять валидацию данных, а поддержка связей позволяет быстро находить связанные записи в базе.

Теперь мы можем попробовать ещё раз сформулировать, что же такое модель. Получается, что модель – это та сущность, которая связывает определенную таблицу базы с определенным классом в нашем коде. При этом мы можем работать с таблицей в целом через методы этого класса (например, выполнять запросы), и работать с конкретной строкой таблицы как с объектом класса. Таким образом, разработчик может взаимодействовать с данными в базе, как с любыми другими объектами, при этом практически без необходимости использовать SQL.

Давайте знакомиться с моделями ближе.

Рассмотрим процесс создания новой модели. Прежде всего, нужно определиться с именем и необходимыми атрибутами. Предположим, у нас есть таблица `users`, в которой хранятся такие сведения о пользователях, как никнейм, e-mail, дата рождения и флаг, является ли пользователь модератором. Для работы с этой таблицей нам понадобится модель.

Её можно создать вручную, как подкласс `ApplicationRecord`, а можно использовать генератор. Т.к. в Rails принят подход «convention over configuration» (соглашения над конфигурацией), существуют следующие соглашения:

- Классы моделей называются в единственном числе и записываются в CamelCase (`User` , `Project` , `Book` , `ProjectManager` , `Person`)
- Соответствующие им таблицы – во множественном в snake_case (`users` , `projects` , `books` , `project_managers` , `people`)
- Механизмы образования множественного числа Rails очень мощные, они способны образовывать множественное (и единственное) число как для правильных, так и для неправильных форм слов. Поэтому мы можем смело использовать названия вроде `Person-people`, и Rails нас поймет.

Следующие соглашения касаются имен столбцов:

- Имена столбцов в таблице указываются в snake_case (например, `due_date`)

- При использовании миграций для создания таблиц по умолчанию создается столбец `id`, который используется как первичный ключ. Вы можете задать другое имя для первичного ключа, но обратите внимание, что ActiveRecord не поддерживает использование столбцов с именем `id`, не являющихся первичными ключами.
- Внешние ключи должны носить имя связанной таблицы в форме единственного числа с суффиксом `id`. Например, если в таблице `profiles` вы планируете хранить внешний ключ для связанной таблицы `users`, то он должен называться `user_id`.
- В столбце с именем `created_at` автоматически будет установлена дата создания записи.
- В столбце `updated_at` будет устанавливаться текущая дата каждый раз при обновлении записи.
- Имя столбца `type` зарезервировано для случаев использования STI (один из механизмов отображения наследования на реляционную базу данных), поэтому не стоит использовать его, если вы не планируете хранить объекты разных классов в одной таблице.

Приняв во внимание эти соглашения, мы можем выполнить команду `rails g model` (или `rails generate model`), указав имя модели, атрибуты и опциональные ключи. Это команда создаст класс модели в папке `app/models`, и миграцию в папке `db/migrate`. Миграции – это инструмент изменения структуры базы данных, и им будет посвящена отдельная лекция. Если кратко – это специальные классы, которые позволяют создавать или удалять таблицы, и менять их структуру, то есть они описывают изменения схемы базы данных. Таким образом, модели работают с данными в базе, а миграции со структурой самой базы.

```
class Project < ApplicationRecord
end
```

Атрибуты указываются в виде пар `имя:тип`. К возможным типам относятся `integer`, `float`, `time`, `date`, `text` и ряд других. Если тип не указывать, по умолчанию будет использоваться `string`.

Возможные типы:

- `integer`
- `primary_key`
- `decimal`
- `float`

- `boolean`
- `binary`
- `string`
- `text`
- `date`
- `time`
- `datetime`

Также есть такой тип, как `references`. Он используется для указания связей в моделях. Подробно о связях мы поговорим в одной из следующих лекций, но сейчас в качестве простого примера представим, что вы хотите создать модель `Profile`, связанную с моделью `User`. Для этого при выполнении команды `rails g` при создании модели `Profile` укажем атрибут `user` с типом `references`. В результате в таблице `profiles` создастся колонка `user_id` для хранения `id` (внешних ключей) связанных с профилем пользователей.

```
rails g model Profile address:string user:references
```

Вы также можете добавить атрибутам суффикс `uniq` или `index`. Первый используется, если вы хотите, чтобы в столбце не было повторяющихся значений (например, `e-mail` должен быть уникальным), а второй для добавления индекса на столбец.

```
rails g model User email:string:uniq
```

Кроме того, при создании модели можно указать дополнительные опции. Например, ключ `--no-migration` отменит создание миграции. С помощью ключа `--no-indexes` можно отключить автоматическое создание индекса для колонок с типом `references`. А ключ `--no-timestamps` отключает создание колонок `created_at` и `updated_at`, которые создаются по умолчанию автоматически.

Итак, мы разобрались с параметрами команды `rails g model` и теперь готовы создать свою первую модель для таблицы пользователей. Напомню, что имя модели будет совпадать с названием таблицы, но в форме единственного числа. И нам нужны атрибуты для имени, электронного адреса, даты рождения и флага для отметки модератор этот пользователь или нет. Итоговая команда будет выглядеть следующим образом:

```
rails g model User name email:uniq birthday:date moderator:boolean
```

Обратите внимание, что мы опустили типы для атрибутов `name` и `email`, и для них применится тип по умолчанию, то есть `string`. И указали суффикс `uniq` для атрибута `email`, чтобы добавить проверку на уникальность значений в этой колонке на уровне базы данных.

```
~/les08$ rails g model User name email:uniq birthday:date moderator:boolean
invoke active_record
create db/migrate/20211219123021_create_users.rb
create app/models/user.rb
invoke test_unit
create test/models/user_test.rb
create test/fixtures/users.yml
```

После выполнения команды создадутся несколько файлов, а именно модель, миграция и тесты. В данном случае нас интересует модель, поэтому давайте посмотрим на неё в `app/models/user.rb`

Мы видим класс `User`, наследуемый от `ApplicationRecord`.

```
class User < ApplicationRecord
end
```

А вот, собственно, и всё. Всё, что наделяет этот класс магией моделей, находится в `ApplicationRecord`, поэтому часто модели просто создаются вручную. Гораздо больше интересного можно увидеть в миграции в `db/migrate`. Там появился файл, название которого начинается с даты создания миграции. Это временная метка, которая идентифицирует миграцию, и она же помещается в таблицу `schema_migrations` вашей базы, в которой фиксируются все примененные миграции. Далее идет имя класса миграции, но записанное в `snake_case`. Обратите внимание, что имя класса миграции должно описывать, что делает миграция. В нашем случае, оно выглядит как `CreateUsers`, потому что мы создаем таблицу `users`.

Миграция содержит метод `change`, в котором описаны изменения, вносимые в базу. Сейчас мы создаем таблицу `users` с атрибутом `name` с типом `string`, атрибутом `email` также с типом `string`, атрибутом `birthday` с типом `date` и атрибутом `moderator` с типом `boolean`. `t.timestamps` создает колонки `created_at` и

`updated_at` , где буду фиксироваться даты создания и соответственно обновления пользователей. Строка, начинающаяся с `add_index` устанавливает индекс для столбца `email` нашей таблицы, и добавляет проверку на уникальность значений.

```
class CreateUsers < ActiveRecord::Migration[5.2]
  def change
    create_table :users do |t|
      t.string :name
      t.string :email
      t.date :birthday
      t.boolean :moderator
      t.timestamps
    end
    add_index :users, :email, unique: true
  end
end
```

Миграции так же, как и модели, можно создавать с помощью генератора или вручную. Для создания только миграции без модели есть отдельный генератор, который мы разберем в следующей лекции.

Если вдруг после создания модели вы захотите её удалить, сделать это можно из консоли, выполнив команду `rails d model` , передав ей имя удаляемой модели. В нашем случае команда будет выглядеть, как `rails d model User` . `d` является алиасом для `destroy` .

```
~/les08$ rails d model User
  invoke  active_record
  remove  db/migrate/20211219123021_create_users.rb
  remove  app/models/user.rb
  invoke  test_unit
  remove  test/models/user_test.rb
  remove  test/fixtures/users.yml
```

Это действие приведет к удалению модели вместе с соответствующей миграцией.

Базовые действия CRUD

Теперь, когда мы умеем создавать модели, давайте разберемся с базовыми операциями, а именно с созданием новых объектов, их чтением, обновлением и удалением. Для этого

создадим модель `Project` с атрибутами `name` и `due_date`. Несмотря на то, что класс модели выглядит пустым, он наследует от `ApplicationRecord` всё необходимое, для работы с базой. А после выполнения миграции с помощью команды `rails db:migrate` будет создана таблица `projects`, и мы сможем начать работать с данными в ней.

```
class Project < ApplicationRecord
end
```

Create

Рассмотрим базовые операции с данными, которые доступны без написания дополнительного кода. В первую очередь – создание записей. Для этого можно использовать методы `create` или `new`. Разница между ними заключается в том, что `create` создаст объект и сразу сохранит его в базе, а `new` только инициализирует его, а для сохранения позже придется дополнительно вызвать метод `save`. Оба эти метода (`create` и `new`) принимают хэш атрибутов. Или, при необходимости, вы можете указать блок, в который будет передан инициализированный объект.

```
Project.create(name: 'New Project')
# #<Project id: 1, name: "New Project", due_date: nil>
# INSERT INTO "projects" ("name", "created_at", "updated_at") VALUES (?, ?, ?) [["name", "N
project = Project.new do |pr|
  pr.name = 'Second project'
end

#<Project id: nil, name: "Second project", due_date: nil>
project.save
# INSERT INTO "projects" ("name", "created_at", "updated_at") VALUES (?, ?, ?) [["name", "S
```

Read

Для чтения записей из базы `ActiveRecord` предоставляет широкий инструментарий методов класса. Прежде всего, это поиск записи по первичному ключу – метод `find`, который позволяет найти запись по значению поля `id`. У метода `find` есть важная особенность: если запись с указанным `id` не найдется, это вызовет исключение. Вы можете передать в `find` массив первичных ключей – это вернет массив соответствующих элементов. Уже здесь становится очевидно, насколько использование `ActiveRecord` делает код лаконичней, чем использование чистого SQL, ведь короткое `Project.find(1)` заменяет собой обычную для SQL выборку с `select`, `from` и `where`.


```
Project.find(1)
# SELECT "projects".* FROM "projects" WHERE "projects"."id" = ? LIMIT ? [["id", 1], ["LIMIT", 1]]
=> #<Project id: 1, name: "New Project", due_date: nil>
```

```
Project.find([1, 2])
# SELECT "projects".* FROM "projects" WHERE "projects"."id" IN (?, ?) [["id", 1], ["id", 2], ["LIMIT", 1]]
=> [#<Project id: 1, name: "New Project", due_date: nil>, #<Project id: 2, name: "Second project", due_date: nil>]
```

Подобно методу `find` работает `find_by`, но он позволяет искать по любому атрибуту и возвращает первую запись, соответствующую условию. Например, запрос

`Project.find_by(name: "Second project")` вернет нам проект именно с этим именем. Кстати, в отличие от `find` в случае, если не будет найдено ни одной записи, `find_by` вместо вызова исключения просто вернет нам `nil`.

```
Project.find_by(name: "Second project")
# SELECT "projects".* FROM "projects" WHERE "projects"."name" = ? LIMIT ? [["name", "Second project"], ["LIMIT", 1]]
=> #<Project id: 2, name: "Second project", due_date: nil>
```

Если нам нужно получить первую или последнюю запись в базе, то помогут в этом методы `first` и `last` соответственно. Все эти методы – `find`, `find_by`, `first` и `last` – возвращают экземпляр класса `Project`. Обращаясь к его атрибутам, мы получаем данные из соответствующей колонки таблицы. Например, Мы можем получить первую запись таблицы `projects` и значение поля `name` в ней выполнив `Project.first.name`

```
Project.first
# SELECT "projects".* FROM "projects" ORDER BY "projects"."id" ASC LIMIT ? [["LIMIT", 1]]
=> #<Project id: 1, name: "New Project", due_date: nil>
```

```
Project.last
# SELECT "projects".* FROM "projects" ORDER BY "projects"."id" DESC LIMIT ? [["LIMIT", 1]]
=> #<Project id: 2, name: "Second project", due_date: nil>
```

```
Project.first.name
# SELECT "projects"."name" FROM "projects" ORDER BY "projects"."id" ASC LIMIT ? [["LIMIT", 1]]
=> "New Project"
```

Бывает, вам нужно получить все записи какой-то таблицы. Для этого существует метод `all`. Например, `Project.all` вернет все проекты, которые есть в базе. Вызвав на полученном отношении метод `each` можно пройтись по каждому элементу, но у такого метода есть существенный недостаток – `all` извлекает всю таблицу за раз, то есть в случае с большим количеством записей это может негативно отразиться на работе приложения, особенно при недостаточном количестве оперативной памяти.

```
Project.all
# SELECT "projects".* FROM "projects" LIMIT ? [["LIMIT", 11]]
=> #<ActiveRecord::Relation [#<Project id: 1, name: "New Project", due_date: nil
```

Избежать проблемы с памятью можно, если воспользоваться методом `find_each`. Он также получит все записи из базы, но пакетами, размер которых вы можете определить самостоятельно. После этого `find_each` передаст записи из пакетов в блок по одной, т.е. внутри блока вам будет доступна одна запись за раз. В нашем примере, мы с помощью `find_each` можем получить проекты и передать их по одному какому-нибудь методу.

```
Project.find_each do |project|
  check(project)
end
```

Подобным же образом работает метод `find_in_batches`. Его отличие от предыдущего заключается в том, что он получает за раз пачку записей и передает в блок всю пачку за раз (а не одну запись, как было до этого).

```
Project.find_in_batches |projects|
  check(projects)
end
```

Оба этих метода (`find_each` и `find_in_batches`) имеют дополнительные опции. Прежде всего, это `batch_size`. Эта опция определяет размер пакета, то есть количество записей, которые будут получены за раз. Например, запись `Project.find_each(batch_size: 100)` говорит о том, что все проекты будут получены пачками не превышающими 100 записей. Опция `start` указывает `id` первой записи, с которой начнется выборка, а опция `finish` подобно опции `start` указывает `id`, но

на этот раз той записи, которая должна стать последней в выборке. Таким образом можно настроить количество получаемых записей и снизить нагрузку на базу данных.

```
Project.find_each(batch_size: 100, start: 200) do |project|
  check(project)
end
```

Ещё одним способом получения нескольких записей является использование метода `where`, который позволяет задавать условия для выборки. Запросам будет посвящена отдельная лекция, но простейший способ использования метода `where` – это поиск по нескольким атрибутам. Например, запрос `Project.where(name: 'New Project', due_date: nil)` вернет нам все проекты с именем `New Project` и неуказанной датой `due_date`. Метод `where` также может принять строку с SQL условием.

```
Project.where(name: 'New Project', due_date: nil)
# SELECT "projects".* FROM "projects" WHERE "projects"."name" = ? AND "projects"."due_date" = ?
=> #<ActiveRecord::Relation [#<Project id: 1, name: "New Project", due_date: nil>]
Update
```

Update

Когда мы получили записи из базы, с ними нужно дальше что-то делать. Например, обновить. Тут всё просто – для обновления одной записи есть метод `update`, которому мы просто передаем модифицируемые атрибуты с их новыми значениями. Допустим, мы решили переименовать проект. Получаем нужную запись, вызываем метод `update` и задаем новое значение для названия. Поскольку метод `update` позволяет за раз обновить несколько атрибутов, мы можем заодно обновить значение и для `due_date`.

```
Project.first.update(name: 'New name', due_date: '2025-12-31')
# SELECT "projects".* FROM "projects" ORDER BY "projects"."id" ASC LIMIT ? [["LIMIT", 1]]
# UPDATE "projects" SET "due_date" = ? WHERE "projects"."id" = ? [["due_date", "2025-12-31", 1]]
```

Этот вариант является сокращенным для случая, когда мы получаем нужную запись, сохраняем её в переменную, задаем новые значения для атрибутов и затем сохраняем изменения вызвав метод `save`.

```
project = Project.last
project.name = 'Last project'
project.due_date = '2025-01-01'
project.save
# UPDATE "projects" SET "name" = ?, "due_date" = ? WHERE "projects"."id" = ? [{"name", "La
```

Иногда бывают ситуации, когда нужно за раз обновить несколько записей. Для этого используется метод `update_all`. Например, нам нужно обновить значение `due_date` для всех записей, у которых это значение раньше определенной даты. В этой ситуации мы можем использовать связку методов `where` и `update_all`, получив нужные записи с помощью условия и обновив их за раз.

```
Project.where("due_date < '2026-12-31'").update_all(due_date: '2026-12-31')
# UPDATE "projects" SET "due_date" = '2026-12-31' WHERE (due_date < '2026-12-31')
Destroy
```

Destroy

Последнее, с чем нам необходимо разобраться в рамках базовых операций CRUD – уничтожение записей. И тут всё предельно просто. Если нам необходимо удалить одну конкретную запись, находим её и вызываем метод `destroy`. Всё, запись удалена. Ещё один метод для удаления записей – `destroy_all`. Его можно вызвать как на самом классе, что приведет к удалению всех записей в связанной таблице, так и на результате запроса для удаления записей, удовлетворяющих каким-то условиям. А начиная с Rails версии 7 добавился метод `destroy_by`. Его можно использовать, если вы хотите удалить ряд записей по значению какого-то атрибута – он работает по аналогии с `find_by`, только вместо того, чтобы вернуть запись, уничтожает её.

```
Project.first.destroy
# DELETE FROM "projects" WHERE "projects"."id" = ? [{"id", 1}]
Project.destroy_all
# DELETE FROM "projects" WHERE "projects"."id" = ? [{"id", 2}]
# DELETE FROM "projects" WHERE "projects"."id" = ? [{"id", 3}]
# DELETE FROM "projects" WHERE "projects"."id" = ? [{"id", 4}]
...
```

Давайте вспомним, что сегодня узнали. Мы близко познакомились с М из паттерна MVC, узнали, для чего нужны модели и как с ними работать. Научились создавать модели с помощью генератора и вручную, поговорили о существующих в Rails соглашениях по именованию моделей, классов и столбцов. Разобрали методы, предоставляемые ActiveRecord для основных операций CRUD (создания, чтения, обновления и удаления), а именно `find` – позволяющий искать по `id`; `find_by`, который ищет по любому атрибуту; `where`, использующий условия для поиска записей; узнали об отличии методов `create` и `new`, и познакомились с методами `update` и `destroy`.

Теги: `ruby`, `ruby on rails`, курсы программирования

Хабы: Блог компании Evrone, Ruby, Ruby on Rails

**Evrone**

Компания

[Подписаться](#)[Сайт](#) [Сайт](#) [Сайт](#) [Сайт](#)**24**

Карма

0

Рейтинг

Evrone @Evrone

Пользователь

[Подписаться](#)

Комментарии 1

Публикации

[ЛУЧШИЕ ЗА СУТКИ](#) [ПОХОЖИЕ](#)**mr-pickles**

18 часов назад

16-, 8- и 4-битные форматы чисел с плавающей запятой



Средний



15 мин



6.7K

 +68 88 28

ru_vds

15 часов назад

Искусство создания понятных графиков



Средний



7 мин



3.5K

Тutorial

Перевод

 +38 79 2

dlinyj

20 часов назад

Измерение скорости чтения-записи носителей с помощью утилиты dd



Средний



11 мин



4K

Кейс

 +38 48 29

SLY_6

10 часов назад

Известная, но очень странная кошачья повадка: кошки, приносящие игрушку – это эволюционная загадка



5 мин



7.6K

Перевод

 +30 22 38

alizar

19 часов назад

Автономия разработчиков. Как устроены компании нового типа



Простой



7 мин



4K

Мнение

 +25 20 21

Petr0v1

14 часов назад

Марс всё ближе: несмотря на проблемы, запуск Starship можно считать успешным

🕒 4 мин 👁 4K

💎 +16

🔖 10

💬 11



Doctor_IT

14 часов назад

Сможет ли високосная минута решить проблему синхронизации часов?

🕒 6 мин 👁 2.7K

💎 +15

🔖 15

💬 14



ne_volkov

19 часов назад

Как в Ozon следят за чувствительной информацией в логах и при чем тут Толкиен?

🟢 Простой 🕒 10 мин 👁 3.2K

Кейс

💎 +15

🔖 22

💬 3



badcasedaily1

12 часов назад

Garbage Collection и JVM

🟢 Простой 🕒 17 мин 👁 2.1K

Обзор

💎 +13

🔖 67

💬 3



SLY_6

16 часов назад

Дайджест научпоп-новостей за неделю, о которых мы ничего не писали

🕒 8 мин 👁 1.6K

Дайджест

[Показать еще](#)

ИНФОРМАЦИЯ

Сайт	evrone.ru
Дата регистрации	2 августа 2022
Дата основания	2008
Численность	101–200 человек
Местоположение	Россия

БЛОГ НА ХАБРЕ

19 мая в 19:04

Курс по Ruby+Rails. Часть 8. Модели и первые шаги

1.8K 0

25 апр в 14:29

Что нового в Proxmox 7.4

6.9K 23

6 апр в 14:00

Как добавить сторонние драйверы в установочный образ VMware ESXi 8

4.9K 18

22 мар в 19:40

Курс по Ruby+Rails. Часть 7. Модели и ActiveRecord

2.6K 1

27 фев в 19:55

Подробный гайд по Docker на M1

13K 6

Ваш аккаунт	Разделы	Информация	Услуги
Профиль	Статьи	Устройство сайта	Корпоративный блог
Трекер	Новости	Для авторов	Медийная реклама
Диалоги	Хабы	Для компаний	Нативные проекты
Настройки	Компании	Документы	Образовательные
ППА	Авторы	Соглашение	программы
	Песочница	Конфиденциальность	Стартапам
			Спецпроекты



Настройка языка

Техническая поддержка

© 2006–2023, Habr