

Laravel

- Основы работы с базами данных. **Eloquent ORM**

Массовое заполнение

При создании новой модели её конструктору передаётся массив атрибутов. Эти атрибуты затем присваиваются модели через массовое заполнение. Это удобно, но в то же время представляет серьёзную проблему с безопасностью, когда вы передаёте ввод от клиента в модель без проверок — в этом случае пользователь может изменить **любое** поле вашей модели. По этой причине по умолчанию Eloquent защищает от массового заполнения.

Необходимо определить в классе модели свойство `$fillable` или `$guarded`.

Указание доступных к заполнению атрибутов

Свойство `$fillable` указывает, какие поля должны быть доступны при массовом заполнении. Их можно указать на уровне класса или объекта.

```
class User extends Model {  
    protected $fillable = ['first_name', 'last_name', 'email'];  
}
```

В этом примере только три перечисленных поля будут доступны массовому заполнению.

Laravel

- Основы работы с базами данных. **Eloquent ORM**

Указание охраняемых (*guarded*) атрибутов модели

Противоположность `$fillable` — свойство `$guarded`, которое содержит список запрещённых к заполнению полей:

```
class User extends Model {  
    protected $guarded = ['id', 'password'];  
}
```

Защита всех атрибутов от массового заполнения

В примере выше атрибуты `id` и `password` **не могут** быть присвоены через массовое заполнение. Все остальные атрибуты — могут. Вы также можете запретить все атрибуты для заполнения, используя символ `*`:

```
protected $guarded = ['*'];
```

Laravel

- Основы работы с базами данных. **Eloquent ORM**

Вставка, обновление, удаление

Для создания новой записи в БД необходимо создать и проинициализировать экземпляр модели и вызвать метод `save()`.

Сохранение новой модели

```
$user = new User;  
$user->name = 'John';  
$user->save();
```

Внимание: Модели Eloquent содержат автоинкрементные идентификаторы (*autoincrementing*). Однако если необходимо использовать собственные идентификаторы, нужно установить свойство `$incrementing` класса модели в значение `false`.

Также можно использовать метод `create()` для создания и сохранения модели одной строкой. Метод вернёт добавленную модель. Однако перед этим нужно определить либо свойство `$fillable`, либо `$guarded` в классе модели, так как изначально все модели Eloquent защищены от массового заполнения.

Laravel

- Основы работы с базами данных. **Eloquent ORM**

Создание модели

// Создание нового пользователя в БД...

```
$user = User::create(['name' => 'John']);
```

// Получение пользователя по свойствам, или его создание, если такого не существует...

```
$user = User::firstOrCreate(['name' => 'John']);
```

// Получение пользователя по свойствам, или создание нового экземпляра...

```
$user = User::firstOrCreate(['name' => 'John']);
```

Laravel

- Основы работы с базами данных. **Eloquent ORM**

Обновление полученной модели

Для обновления модели вам нужно получить её, изменить атрибут и вызвать метод `save()`:

```
$user = User::find(1);  
$user->email = 'john@foo.com';  
$user->save();
```

Сохранение модели и её отношений

Иногда вам может быть нужно сохранить не только модель, но и все её отношения. Для этого используется метод `push()`:

```
$user->push();
```

Вы также можете выполнять обновления в виде запросов к набору моделей:

```
$affectedRows = User::where('votes', '>', 100)->update(['status' => 2]);
```

Laravel

- Основы работы с базами данных. **Eloquent ORM**

Удаление существующей модели

Для удаления модели вызовите метод `delete()` на её объекте:

```
$user = User::find(1);  
$user->delete();
```

Удаление модели по ключу

```
User::destroy(1);  
User::destroy([1, 2, 3]);  
User::destroy(1, 2, 3);
```

Конечно, можно выполнять удаление на наборе моделей:

```
$affectedRows = User::where('votes', '>', 100)->delete();
```

Laravel

- Основы работы с базами данных. **Eloquent ORM**

Мягкое удаление

При «мягком» удалении модели, она остаётся в базе данных, но устанавливается её поле `deleted_at`. Для включения мягких удалений на модели необходимо использовать `SoftDeletes`:

```
use Illuminate\Database\Eloquent\SoftDeletes;
class User extends Model {
    use SoftDeletes;
    protected $dates = ['deleted_at'];
}
```

Для отображения всех моделей, в том числе удалённых, используйте метод `withTrashed()`:

```
$users = User::withTrashed()->where('account_id', 1)->get();
```

Метод `withTrashed` может быть использован в отношениях:

```
$user->posts()->withTrashed()->get();
```

Если нужно получить **только** удалённые модели, вызовите метод `onlyTrashed()`:

```
$users = User::onlyTrashed()->where('account_id', 1)->get();
```

Для восстановления мягко удалённой модели в активное состояние используется метод `restore()`:

```
$user->restore();
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Если необходимо полностью удалить модель из БД, используется метод `forceDelete()`:

```
$user->forceDelete();
```

Он также работает с отношениями:

```
$user->posts()->forceDelete();
```

Для того, чтобы узнать, удалена ли модель, можно использовать метод `trashed()`:

```
if ($user->trashed())  
{  
    //  
}
```

Поля времени

По умолчанию Eloquent автоматически поддерживает поля `created_at` и `updated_at`. Следует только добавить эти *timestamp*-поля к таблице, и Eloquent позаботится об остальном. Если в этом нет необходимости:

```
class User extends Model {  
    protected $table = 'users';  
    public $timestamps = false;  
}
```


Laravel

Основы работы с базами данных. **Eloquent ORM**

Создание заготовки запроса

Заготовки позволяют вам повторно использовать логику запросов в моделях. Для создания заготовки просто начните имя метода со scope:

```
class User extends Model {  
    public function scopePopular($query)  
    {  
        return $query->where('votes', '>', 100);  
    }  
    public function scopeWomen($query)  
    {  
        return $query->whereGender('W');  
    }  
}
```

Использование заготовки

```
$users = User::popular()->women()->orderBy('created_at')->get();
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Динамические заготовки

Иногда вам может потребоваться определить заготовку, которая принимает параметры. Для этого просто добавьте эти параметры к методу заготовки:

```
class User extends Model {  
    public function scopeOfType($query, $type)  
    {  
        return $query->whereType($type);  
    }  
}
```

А затем передайте их при вызове метода заготовки:

```
$users = User::ofType('member')->get();
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Отношения

В большинстве случаев таблицы связаны с другими таблицами БД. Eloquent упрощает работу и управление такими отношениями. Laravel поддерживает многие типы связей:

[Один к одному](#)

[Один ко многим](#)

[Многие ко многим](#)

[Ко многим через](#)

[Полиморфические связи](#)

[Полиморфические связи «многие ко многим»](#)

Laravel

Основы работы с базами данных. **Eloquent ORM**

Один к одному

Создание связи «один к одному»

Связь вида «один к одному» можно определить в Eloquent:

```
class User extends Model {  
    public function phone()  
    {  
        return $this->hasOne('App\Phone');  
    }  
}
```

Первый параметр, передаваемый `hasOne()`, — имя связанной модели. Как только отношение установлено, вы можете получить к нему доступ через динамические свойства Eloquent:

```
$phone = User::find(1)->phone;
```

Сгенерированный SQL имеет такой вид:

```
select * from users where id = 1
```

```
select * from phones where user_id = 1
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Eloquent считает, что поле в таблице называется по имени модели плюс `_id`. В данном случае предполагается, что это `user_id`. Если вы хотите перекрыть стандартное имя, передайте второй параметр методу `hasOne()`. Кроме того, возможно передать в метод третий аргумент, чтобы указать, какие локальные столбцы следует использовать для объединения:

```
return $this->hasOne('App\Phone', 'foreign_key');  
return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

Создание обратного отношения

Для создания обратного отношения в модели `Phone` используется метод `belongsTo()` («принадлежит к»):

```
class Phone extends Model {  
    public function user()  
    {  
        return $this->belongsTo('App\User');  
    }  
}
```

В примере выше Eloquent будет искать поле `user_id` в таблице `phones`.

Laravel

Основы работы с базами данных. **Eloquent ORM**

Если необходимо назвать внешний ключ по другому, следует передать имя вторым параметром в метод `belongsTo()`:

```
class Phone extends Model {  
    public function user()  
    {  
        return $this->belongsTo('App\User', 'local_key');  
    }  
}
```

Кроме того, третий параметр определяет имя связанного столбца в родительской таблице:

```
class Phone extends Model {  
    public function user()  
    {  
        return $this->belongsTo('App\User', 'local_key', 'parent_key');  
    }  
}
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Один ко многим

Примером отношения «один ко многим» является пост в блоге, который имеет комментарии:

```
class Post extends Model {  
    public function comments()  
    {  
        return $this->hasMany('App\Comment');  
    }  
}
```

Теперь мы можем получить все комментарии с помощью динамического свойства:

```
$comments = Post::find(1)->comments;
```

Если нужно добавить ограничения на получаемые комментарии, можно вызвать метод `comments()` и продолжить добавлять условия:

```
$comments = Post::find(1)->comments()->where('title', '=', 'foo')->first();
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Второй параметр метода `hasMany()` может быть использован для перекрытия стандартного имени ключа. И как и для отношения «hasOne» также может быть указан локальный столбец:

```
return $this->hasMany('App\Comment', 'foreign_key');  
return $this->hasMany('App\Comment', 'foreign_key', 'local_key');
```

Определение обратного отношения

Для определения обратного отношения так же используется метод `belongsTo()`:

```
class Comment extends Model {  
    public function post()  
    {  
        return $this->belongsTo('App\Post');  
    }  
}
```


Laravel

Основы работы с базами данных. **Eloquent ORM**

Многие ко многим

Отношения типа «многие ко многим» — более сложные, чем остальные виды отношений. Примером может служить пользователь, имеющий много ролей, где роли также относятся ко многим пользователям. Нужны три таблицы для этой связи: `users`, `roles` и `role_user`. Название таблицы `role_user` происходит от упорядоченных по алфавиту имён связанных моделей, она должна иметь поля `user_id` и `role_id`.

Вы можете определить отношение «многие ко многим» через метод `belongsToMany()`:

```
class User extends Model {  
    public function roles() {  
        return $this->belongsToMany('App\Role');  
    }  
}
```

Теперь мы можем получить роли через модель `User`:

```
$roles = User::find(1)->roles;
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Вы можете передать второй параметр к методу `belongsToMany()` с указанием имени связующей (*pivot*) таблицы вместо стандартной:

```
return $this->belongsToMany('App\Role', 'user_roles');
```

Вы также можете перекрыть имена ключей по умолчанию:

```
return $this->belongsToMany('App\Role', 'user_roles', 'user_id', 'foo_id');
```

Конечно, вы можете определить и обратное отношение на модели `Role`:

```
class Role extends Model {  
    public function users()  
    {  
        return $this->belongsToMany('App\User');  
    }  
}
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Ко многим через

Связь «ко многим через» обеспечивает удобный короткий путь для доступа к удалённым отношениям через промежуточные. Например, модель Country может иметь много Post через модель User. Таблицы для этих отношений будут выглядеть так:

countries

id - integer

name - string

users

id - integer

country_id - integer

name - string

posts

id - integer

user_id - integer

title – string

Laravel

Основы работы с базами данных. **Eloquent ORM**

Несмотря на то, что таблица `posts` не содержит столбца `country_id`, отношение «`hasManyThrough`» позволит нам получить доступ к `posts` через `country` с помощью `$country->posts`. Отношения:

```
class Country extends Model {  
    public function posts()  
    {  
        return $this->hasManyThrough('App\Post', 'App\User');  
    }  
}
```

Если вы хотите указать ключи отношений вручную, вы можете передать их в качестве третьего и четвертого аргументов метода:

```
class Country extends Model {  
    public function posts()  
    {  
        return $this->hasManyThrough('App\Post', 'App\User', 'country_id', 'user_id');  
    }  
}
```

Laravel

Основы работы с базами данных. Eloquent ORM

Полиморфические отношения

Полиморфические отношения позволяют модели быть связанной с более, чем одной моделью. Например, может быть модель **Photo**, содержащая записи, принадлежащие к моделям **Staff** и **Order**. Мы можем создать такое отношение таким образом:

```
class Photo extends Model {
    public function imageable()
    {
        return $this->morphTo();
    }
}
class Staff extends Model {
    public function photos()
    {
        return $this->morphMany('App\Photo', 'imageable');
    }
}
class Order extends Model {
    public function photos()
    {
        return $this->morphMany('App\Photo', 'imageable');
    }
}
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Чтение полиморфической связи

Теперь мы можем получить фотографии и для сотрудника, и для заказа:

```
$staff = Staff::find(1);  
foreach ($staff->photos as $photo)  
{  
    //  
}
```

При чтении связи на модели **Photo**:

```
$photo = Photo::find(1);  
$imageable = $photo->imageable;
```

Отношение **imageable** модели **Photo** вернёт либо объект **Staff**, либо объект **Order** в зависимости от типа модели, которой принадлежит фотография.

Laravel

Основы работы с базами данных. **Eloquent ORM**

Структура таблиц полиморфической связи

Рассмотрим структуру БД для полиморфического отношения

staff

id - integer

name - string

orders

id - integer

price - integer

photos

id - integer

path - string

imageable_id – integer

imageable_type – string

Главные поля, на которые нужно обратить внимание: **imageable_id** и **imageable_type** в таблице photos. Первое содержит ID владельца, в нашем случае — заказа или персонала, а второе — имя класса-модели владельца. Это позволяет ORM определить, какой класс модели должен быть возвращён при использовании отношения [imageable](#).

Laravel

Основы работы с базами данных. **Eloquent ORM**

Полиморфические связи многие ко многим

Структура таблиц полиморфической связи многие ко многим

В дополнение к традиционным полиморфическим связям вы можете также задать полиморфические связи многие ко многим. Например, модели блогов `Post` и `Video` могут разделять полиморфическую связь с моделью `Tag`. В-первых, рассмотрим структуру таблиц:

posts(id – integer, name – string)

videos(id – integer, name – string)

tags(id – integer, name – string)

taggables(tag_id – integer, taggable_id – integer, taggable_type – string)

Далее, мы готовы к установке связи с моделью. Обе модели `Post` и `Video` будут иметь связь «morphToMany» через метод `tags`:

```
class Post extends Model {  
    public function tags()  
    {  
        return $this->morphToMany('App\Tag', 'taggable');  
    }  
}
```


Laravel

Основы работы с базами данных. **Eloquent ORM**

Модель Tag может определить метод для каждого из своих отношений:

```
class Tag extends Model {  
    public function posts()  
    {  
        return $this->morphedByMany('App\Post', 'taggable');  
    }  
    public function videos()  
    {  
        return $this->morphedByMany('App\Video', 'taggable');  
    }  
}
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Запросы к отношениям

Проверка связей при выборке

При чтении отношений модели может быть необходимо ограничить результаты в зависимости от существования связи. Например, вы хотите получить все статьи в блоге, имеющие хотя бы один комментарий. Для этого можно использовать метод `has()`:

```
$posts = Post::has('comments')->get();
```

Можно конструировать вложенные операторы `has` с помощью точечной нотации:

```
$posts = Post::has('comments.votes')->get();
```

Можно использовать методы `whereHas` и `orWhereHas`, чтобы поместить условия "where" в запросы `has`:

```
$posts = Post::whereHas('comments', function($q)
{
    $q->where('content', 'like', 'foo%');
})->get();
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Динамические свойства

Eloquent позволяет читать отношения через динамические свойства. Eloquent автоматически определит используемую связь и вызовет `get()` для связей «один ко многим» и `first()` — для связей «один к одному». Эта связь будет доступна через динамическое свойство с тем же именем. К примеру, для следующей модели `$phone`:

```
class Phone extends Model {  
    public function user()  
    {  
        return $this->belongsTo('App\User');  
    }  
}  
$phone = Phone::find(1);
```

Вместо того, чтобы получить e-mail пользователя так:

```
echo $phone->user()->first()->email;
```

...вызов может быть сокращён до такого:

```
echo $phone->user->email;
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Активная загрузка

Активная загрузка (*eager loading*) призвана устранить проблему запросов $N + 1$. Например, представьте, что у нас есть модель **Book** со связью к модели **Author**. Отношение определено как:

```
class Book extends Model {  
    public function author()  
    {  
        return $this->belongsTo('App\Author');  
    }  
}
```

Теперь предположим, у нас есть цикл:

```
foreach (Book::all() as $book)  
{  
    echo $book->author->name;  
}
```

Цикл выполнит один запрос для получения всех книг в таблице, а затем будет выполнять по одному запросу на каждую книгу для получения автора. Таким образом, если у нас 25 книг, то потребуется 26 запросов.

Laravel

Основы работы с базами данных. **Eloquent ORM**

Возможно использовать активную загрузку для кардинального уменьшения числа запросов. Отношение будет активно загружено, если оно было указано при вызове метода `with()`:

```
foreach (Book::with('author')->get() as $book)
{
    echo $book->author->name;
}
```

В цикле выше будут выполнены всего два запроса:

```
select * from books
```

```
select * from authors where id in (1, 2, 3, 4, 5, ...)
```

Разумное использование активной загрузки поможет сильно повысить производительность приложения.

Можно загрузить несколько отношений одновременно:

```
$books = Book::with('author', 'publisher')->get();
```

или вложенные отношения:

```
$books = Book::with('author.contacts')->get();
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Ограничения активной загрузки

Иногда вам может быть нужно не только активно загрузить отношение, но также указать условие для его загрузки:

```
$users = User::with(['posts' => function($query)
{
    $query->where('title', 'like', '%первое%');

}])->get();
```

В этом примере мы загружаем сообщения пользователя, но только те, заголовок которых содержит подстроку «первое».

Функции-замыкания активной загрузки не ограничиваются только условиями. Вы также можете применить упорядочивание:

```
$users = User::with(['posts' => function($query)
{
    $query->orderBy('created_at', 'desc');

}])->get();
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Ленивая активная загрузка

Можно активно загрузить связанные модели напрямую из уже созданного набора объектов моделей. Это может быть полезно при определении во время выполнения, требуется ли такая загрузка или нет, или в комбинации с кэшированием.

```
$books = Book::all();
```

```
$books->load('author', 'publisher');
```

Вы можете передать замыкание, чтобы задать ограничения для запроса:

```
$books->load(['author' => function($query)
{
    $query->orderBy('published_date', 'asc');
}]);
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Связанные модели

Часто вам нужно будет добавить связанную модель. Например, вы можете создать новый комментарий к сообщению. Вместо явного указания значения для поля `post_id` вы можете вставить модель напрямую через её родителя — модели `Post`:

```
$comment = new Comment(['message' => 'A new comment.']);  
$post = Post::find(1);  
$comment = $post->comments()->save($comment);
```

В этом примере поле `post_id` вставленного комментария автоматически получит значение ID своей статьи.

Сохранить несколько связанных моделей можно так:

```
$comments = [  
    new Comment(['message' => 'A new comment.']),  
    new Comment(['message' => 'Another comment.']),  
    new Comment(['message' => 'The latest comment.'])  
];  
  
$post = Post::find(1);  
$post->comments()->saveMany($comments);
```


Laravel

Основы работы с базами данных. **Eloquent ORM**

Связывание моделей (*belongs to*)

При обновлении связей `belongsTo` можно использовать метод `associate()`. Он установит внешний ключ на дочерней модели:

```
$account = Account::find(10);
```

```
$user->account()->associate($account);
```

```
$user->save();
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Связывание моделей «многие ко многим»

```
$user = User::find(1);
```

```
$user->roles()->attach(1);
```

Вы также можете передать массив атрибутов, которые должны быть сохранены в связующей (*pivot*) таблице для этого отношения:

```
$user->roles()->attach(1, ['expires' => $expires]);
```

Конечно, существует противоположность `attach()` — `detach()`:

```
$user->roles()->detach(1);
```

Оба метода `attach()` и `detach()` также принимают в качестве параметров массивы ID:

```
$user = User::find(1);
```

```
$user->roles()->detach([1, 2, 3]);
```

```
$user->roles()->attach([1 => ['attribute1' => 'value1'], 2, 3]);
```

Laravel

Основы работы с базами данных. Eloquent ORM

Использование `sync()` для привязки моделей «многие ко многим»

Вы также можете использовать метод `sync()` для привязки связанных моделей. Этот метод принимает массив ID, которые должны быть сохранены в связующей таблице. Когда операция завершится, переданные ID будут существовать в промежуточной таблице для данной модели:

```
$user->roles()->sync([1, 2, 3]);
```

Добавление данных для связующей таблицы при синхронизации

Вы также можете связать другие связующие таблицы с нужными ID:

```
$user->roles()->sync([1 => ['expires' => true]]);
```

Иногда вам может быть нужно создать новую связанную модель и добавить её одной командой. Для этого вы можете использовать метод `save()`:

```
$role = new Role(['name' => 'Editor']);  
User::find(1)->roles()->save($role);
```

В этом примере новая модель `Role` будет сохранена и привязана к модели `User`. Вы можете также передать массив атрибутов для помещения в связующую таблицу:

```
User::find(1)->roles()->save($role, ['expires' => $expires]);
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Работа со связующими таблицами

Работа отношения многие ко многим требует наличия промежуточной таблицы. Например, предположим, что наш **User** имеет множество связанных объектов **Role**. После чтения отношения мы можем прочитать таблицу **pivot** на обеих моделях:

```
$user = User::find(1);  
foreach ($user->roles as $role)  
{  
    echo $role->pivot->created_at;  
}
```

Заметьте, что каждая модель **Role** автоматически получила атрибут **pivot**. Этот атрибут содержит модель, представляющую промежуточную таблицу, и она может быть использована как любая другая модель Eloquent.

Laravel

Основы работы с базами данных. **Eloquent ORM**

По умолчанию, только ключи будут представлены в объекте `pivot`. Если ваша связующая таблица содержит другие поля, вы можете указать их при создании отношения:

```
return $this->belongsToMany('App\Role')->withPivot('foo', 'bar');
```

Теперь атрибуты `foo` и `bar` будут также доступны на объекте `pivot` модели `Role`.

Если вы хотите автоматически поддерживать поля `created_at` и `updated_at` актуальными, используйте метод `withTimestamps()` при создании отношения:

```
return $this->belongsToMany('App\Role')->withTimestamps();
```

Удаление всех связующих записей

Для удаления всех записей в связующей таблице можно использовать метод `detach()`:

```
User::find(1)->roles()->detach();
```

Заметьте, что эта операция не удаляет записи из таблицы `roles`, а только из связующей таблицы.

Laravel

Основы работы с базами данных. **Eloquent ORM**

Коллекции

Все методы Eloquent, возвращающие набор моделей — либо через `get()`, либо через отношения — возвращают объект-коллекцию. Этот объект реализует стандартный интерфейс PHP `IteratorAggregate`, что позволяет ему быть использованным в циклах как массив. Однако этот объект также имеет набор других полезных методов для работы с результатом запроса.

Проверка на существование ключа в коллекции

Например, мы можем выяснить, содержит ли результат запись с определённым первичным ключом, методом `contains()`:

```
$roles = User::find(1)->roles;  
if ($roles->contains(2))  
{  
    //  
}
```

Коллекции также могут быть преобразованы в массив или строку [JSON](#):

```
$roles = User::find(1)->roles->toArray();  
$roles = User::find(1)->roles->toJson();
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Проход по элементам коллекции

Коллекции Eloquent имеют несколько полезных методов для прохода и фильтрации содержащихся в них элементов:

```
$roles = $user->roles->each(function($role)
{
    //
});
```

Фильтрация элементов коллекции

При фильтрации коллекций передаваемая функция будет использована как функция обратного вызова для [array filter](#).

```
$users = $users->filter(function($user)
{
    return $user->isAdmin();
});
```

Laravel

Основы работы с базами данных. **Eloquent ORM**

Применение функции к каждому объекту коллекции

```
$roles = User::find(1)->roles;
```

```
$roles->each(function($role)  
{  
    //  
});
```

Сортировка коллекции по значению

```
$roles = $roles->sortBy(function($role)  
{  
    return $role->created_at;  
});
```

```
$roles = $roles->sortByDesc(function($role)  
{  
    return $role->created_at;  
});
```


Laravel

Основы работы с базами данных. **Eloquent ORM**

Сортировка коллекции по значению

```
$roles = $roles->sortBy('created_at');
```

```
$roles = $roles->sortByDesc('created_at');
```

Использование произвольного класса коллекции

Иногда вам может быть нужно получить собственный объект `Collection` со своими методами. Вы можете указать его при определении модели Eloquent, перекрыв метод `newCollection()`:

```
class User extends Model {  
  
    public function newCollection(array $models = [])  
    {  
        return new CustomCollection($models);  
    }  
}
```

Laravel

Миграции

Миграции — по сути система контроля версий для базы данных. Они позволяют команде изменять её структуру, в то же время оставаясь в курсе изменений других участников.

Создание миграций

Для создания новой миграции вы можете использовать Artisan-команду:

```
php artisan make:migration create_users_table
```

Миграция будет помещена в папку database/migrations и будет содержать метку времени, которая позволяет фреймворку определять порядок применения миграций.

Можно также использовать параметры `--table` и `--create` для указания имени таблицы и того факта, что миграция будет создавать новую таблицу (а не изменять существующую):

```
php artisan make:migration add_votes_to_users_table --table=users
```

```
php artisan make:migration create_users_table --create=users
```

Laravel

Миграции

```
<?php

use Illuminate\Database\Migrations\Migration;

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Support\Facades\Schema;

class CreateFlightsTable extends Migration
{
    /**
     * Run the migrations.
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();

            $table->string('name');

            $table->string('airline');

            $table->timestamps();

        });
    }

    /**
     * Reverse the migrations.
     * @return void
     */
    public function down()
    {
        Schema::drop('flights');
    }
}
```

Laravel

Миграции

Применение всех неприменённых миграций

php artisan migrate

Внимание: Если при применении миграции вы получаете ошибку «class not found» («класс не найден»), попробуйте выполнить команду **composer** dump-autoload.

Принудительные миграции в продакшне

Некоторые операции миграций разрушительны, они могут привести к потере данных. Для предотвращения случайного запуска этих команд на боевой БД перед их выполнением запрашивается подтверждение. Для принудительного запуска команд без подтверждения используйте ключ `--force`:

php artisan migrate --force

Laravel

Миграции

Откат миграций

Отмена изменений последней миграции

`php artisan migrate:rollback`

Отмена изменений всех миграций

`php artisan migrate:reset`

Откат всех миграций и их повторное применение

`php artisan migrate:refresh`

Seed загрузка начальных данных в БД

Кроме миграций, описанных выше, Laravel также включает в себя механизм наполнения вашей БД начальными данными (*seeding*) с помощью специальных классов. Все такие классы хранятся в `database/seeds`. Они могут иметь любое имя, но вам, вероятно, следует придерживаться какой-то логики в их именовании — например, `UserTableSeeder` и т.д. По умолчанию определён класс `DatabaseSeeder`. Из этого класса вы можете вызывать метод `call()` для подключения других классов с данными, что позволит вам контролировать порядок их выполнения.

Laravel

Загрузки данных

Пример класса для загрузки начальных данных

```
class DatabaseSeeder extends Seeder {  
    public function run()  
    {  
        $this->call('UserTableSeeder');  
        $this->command->info('Таблица пользователей загружена данными!');  
    }  
}
```

```
class UserTableSeeder extends Seeder {  
    public function run()  
    {  
        DB::table('users')->delete();  
        User::create(['email' => 'foo@bar.com']);  
    }  
}
```

Laravel

Загрузки данных

Для добавления данных в БД используйте Artisan-команду db:seed:

```
php artisan db:seed
```

По умолчанию команда **db:seed** вызывает класс [DatabaseSeeder](#), который может быть использован для вызова других классов, заполняющих БД данными. Однако, вы можете использовать параметр **--class** для указания конкретного класса для вызова:

```
php artisan db:seed --class=UserTableSeeder
```

Вы также можете использовать для заполнения БД данными команду migrate:refresh, которая также откатит и заново применит все ваши миграции:

```
php artisan migrate:refresh --seed
```