IPL
escola superior
de tecnologia e gestão
instituto politécnico
de leiria

Laravel

Database & Models

Marco Monteiro

Contributors

- Author(s):
  - Marco Monteiro (marco.monteiro@ipleiria.pt)

Summary

1. Database - DB
2. Models - Eloquent ORM
3. Model Relationships
4. Collections

5. *Migrations & Seeds*

2 – DATABASE - DB

Database

- Laravel supports (currently and by default) 5 different relational database systems:
  - MySQL
  - MariaDB
  - PostgreSQL
  - SQLite
  - SQL Server
- Under the hood, Laravel uses **PDO** (PHP Data Objects) to create and run all kind of queries in a safe manner protecting your application against SQL injection attacks
- Interaction with database can be done with **raw SQL**, the **fluent query builder**, or the **Eloquent ORM**

Configuration

- "config/database.php" *(example with a subset of settings)*

```
'default' => env('DB_CONNECTION', 'mysql'),
...
'mysql' => [  . . .
            'host' => env('DB_HOST', '127.0.0.1'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
              . . . ],
```

- ".env" file *(example with a subset of settings)*

```
DB_HOST=127.0.0.1
DB_DATABASE=work
DB_USERNAME=root
DB_PASSWORD=root_password
```

- Value defined on ".env" file is used. If no value is defined on ".env" file, then default value of config file is used

DB facade

- DB facade allows to execute **raw SQL**
- Has a static method for each kind of DML statement

```
use Illuminate\Support\Facades\DB;
. . .
$results= DB::select('select * from users where user = ?', [23]);
$results= DB::select('select * from users where user = :id',
           ['id' => 23]);

DB::insert('insert into users (id, name) values (?, ?)',
           [24, 'John Doe']);

DB::update('update users set votes = 100 where name = ?',
           ['John Doe']);

DB::delete('delete from users where id = ?', [23]);

// Generic statement
DB::statement('drop table users');
```

DB facade

- **Transactions**
  - All operations are completed, or all operations are cancelled
  - Operations of the transaction can include DB class or Eloquent Models

```php
// As a callback closure
DB::transaction(function () {
    DB::update('update users set votes = 1');
    DB::delete('delete from posts');
});
```

```php
// As a try / catch block
try {
    DB::beginTransaction();
    DB::update('update users set votes = 1');
    DB::delete('delete from posts');
// Do something else that could raise an exception (...)
    DB::commit();
} catch (\Exception $e) {
    DB::rollback();
}
```

DB facade: Query Builder

- Query builder is a **fluent interface** to create and run queries in a safe and error-prone way
- Retrieve all rows from a table

```
$users = DB::table('users')->get();
```

  - Returns a **Collection** containing the results where each row is an instance of the PHP **StdClass** object.
    It does not return an Eloquent Model
  - You may access each column's value by accessing the column as an attribute of the object:

```
foreach ($users as $user) {
    echo $user->name;
}
```

DB facade: Query Builder

- Retrieve data with query builder – other examples

```
// Single row
$user = DB::table('users')->where('name', 'John')->first();

// Where operator:
$users = DB::table('users')->where('age', '>', 100)->get();
$users = DB::table('users')->where('age', '>', 18)
                           ->where('country', 'pt')
                           ->orWhere('name', 'like', 'J%')
                           ->get();

// With select
$users = DB::table('users')
         ->select('name', 'email as user_email')->get();

// Distinct operator
$users = DB::table('users')->distinct()->get();

// Raw methods
$orders = DB::table('orders')
   ->selectRaw('price * ? as price_with_tax', [1.0825])->get();
```

DB facade: Query Builder

- Retrieve data with query builder – other examples

```
// Order by, group by and having
$users = DB::table('users')
                    ->orderBy('name', 'desc')
                    ->groupBy('count')
                    ->having('count', '>', 100)
                    ->get();
// Joins
$users = DB::table('users')
        ->join('contacts', 'users.id', '=', 'contacts.user_id')
        ->join('orders', 'users.id', '=', 'orders.user_id')
        ->select('users.*', 'contacts.phone', 'orders.price')
        ->get();

$users = DB::table('users')
        ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
        ->get();
```

DB facade: Query Builder

- Insert data – examples:

```
// Insert one row
DB::table('users')->insert(
    ['email' => 'john@example.com', 'votes' => 0]
);

// Insert one row and retrieves auto-incrementing id
$id = DB::table('users')->insertGetId(
    ['email' => 'john@example.com', 'votes' => 0]
);

// Insert multiple rows
DB::table('users')->insert([
    ['email' => 'taylor@example.com', 'votes' => 0],
    ['email' => 'dayle@example.com', 'votes' => 0]
]);
```

DB facade: Query Builder

- Update and Delete data – examples:

```
// Updating records
DB::table('users')
            ->where('id', 1)
            ->update(['votes' => 1]);

// Incrementing the value of a column
DB::table('users')->increment('votes');
DB::table('users')->increment('votes', 5);

// Decrementing the value of a column
DB::table('users')->decrement('votes', 5);

// Deleting records
DB::table('users')->where('votes', '<', 100)->delete();

// Deleting all records
DB::table('users')->delete();
```

DB facade: Query Builder

- Query builder includes more features, such as:
  - Chunking results
  - List selection
  - Dynamic "where" clauses
  - Advanced "where" clauses
  - Aggregates
  - Locking
  - Pagination
  - Etc.

- Official documentation:
  https://laravel.com/docs/queries

3 – MODELS - ELOQUENT ORM

Eloquent ORM

- **ORM – Object Relational Mapping**
- Each database table has a corresponding "Model"
- Models are defined by extending the class Model

  (`Illuminate\Database\Eloquent\Model`*)*

```
use Illuminate\Database\Eloquent\Model;
. . .
class Product extends Model   { . . . }
```

- To create a model, we can use CLI artisan:

```
php artisan make:model Product
```

- By default, models are created on:
  - `Folder:` `app/Models`
  - `Namespace:` `App\Models`

Eloquent ORM – Model Instances

- Database **tables** correspond to **model classes**
- Each **instance** (object) of the Eloquent class model corresponds to a **row** of the database table.
- An instance (object) is called a **Model**
- Database table **columns** correspond to model (instance / object) **attributes**

```
$prod = Product::find(123);
$prod->name = 'New Name of Product';
$prod->qty = 20;
$prod->price = 23.99;
$prod->save();
// name, qty and price are also table columns
```

Eloquent ORM - Conventions

- Each model class assumes **by convention**:
  - An underlying table with the snake case plural name of the class (model class should be a singular)
    - Model="**Product**" ➔ Table Name = "**products**"
  - A primary key column called "**id**" (auto increment column)
  - Two timestamp columns: ("**created_at**", "**updated_at**")

Eloquent ORM - Conventions

- Defaults can be overridden using instance attributes

```
class Product extends Model
{
    // Overrides table name
    protected $table = 'shop_products';

    // Overrides primary key name
    protected $primaryKey = 'product_code';

    // Disables auto increment primary key
    public $incrementing = false;

    // overrides primary key type
    protected $keyType = 'string';

    // Disables auto timestamps
    public $timestamps = false;
}
```

Eloquent ORM – Queries

- The Eloquent model base class also serves as a query builder, which means that DB facade queries builder operations are also supported by Eloquent Models

Example with DB facade class:

```
$user = DB::table('users')->where('name', 'John')->get();
```

Same example with Eloquent Model class:

```
// Eloquent Model Class = User (table 'users')
$user = User::where('name', 'John')->get();
```

- Main difference is that DB class returns a collection of **StdClass objects**, and User Eloquent model class returns a collection of User models (**Model objects**)

Eloquent ORM – Queries

- Eloquent query examples

```
// Retrieving all records
$prods = Product::all();

// Retrieving by PK
$prod = Product::find(123);
$prod = Product::find([2, 7, 123, 24]);
$prod = Product::findOrFail(7);
    //throws ModelNotFoundException if product does not exist

// Filtering
$lowStock = Product::where('qty', '<', 5)->get();

// Order by and limit
$lowStock = Product::where('qty', '<', 5)
          ->orderBy('qty', 'desc')
          ->orderBy('name')
          ->take(10)                    // only the first 10 rows
          ->get();
```

Query Builder – multiple operations

- It is possible to combine **multiple** query builder operations (DB class or Eloquent) on 1 PHP expression or multiple expressions

```
$p = Product::where('qty', '<', 5)
           ->where('type', 12)
           ->orderBy('qty', 'desc')
           ->orderBy('name')
           ->take(10)
           ->get();
```

```
$prods = Product::query();    //An empty query builder
if ($filterByQtd) {
    $prods->where('qty', '<', $filterByQtd);
}
if ($filterByType) {
   $prods->where('type', $filterByType)
}
$dados= $prods->orderBy('qty', 'desc')
              ->orderBy('name')
              ->take(10)
              ->get();
```

Query Builder

- All previous examples (that combine **multiple** operations) expressions, end with a method that is responsible for returning data – **get()**
- **query()** method returns an instance of a **Query Builder**
- All other methods (where; orderBy; take), except get, are only affecting the query builder.
  - They are not manipulating anything on the database, they are just manipulating a PHP object that is building the SQL command.
- Only when the get() method is invoked, and only then, the Eloquent Model sends an SQL command to the database, which then returns the data.

Query Builder

Invalid

```
$prods = Product::where('qty', '<', 5);
```

- $prod has no data – it is an instance of the query builder

Valid

```
$prods = Product::where('qty', '<', 5)->get();
```

- $prod has data – get() method returns data from DB

Queries – 1 model

- Methods that returns **1 model** (1 row):
- **find** – return a model instance from its primary key

```
$product = Product::find(23);
```

- **findOrFail** – return a model instance from its primary key. If it does not exist, throws a "model is not found" exception. If exception is not handled a "404 HTTP Response" is sent to the client

```
$product = Product::findOrFail(23);
```

- **first** – return the first model instance (first row) from a query

```
$product = Product::where('qty', '<', 5)->first()
```

Queries – 1 aggregated value

- Methods that only returns **1 (aggregated) value**:

count / max / min / avg / sum

- Examples:

```
$users = Users::count();
```

```
$total_computers =
   Product::where('type', 'computer')->count();
```

```
$more_expensive = Product::max('price');
```

```
$average_price_of_computer =
   Product::where('type', 'computer')->avg('price');
```

Queries – collection of models

- Methods that returns **collections of models**:
- **all** – return all models (all rows in the table). It does not work with query build methods (where, take, etc..)

```
$product = Product::all();
```

- **get** – return a collection of models.

```
$products = Product::where('qty', '<', 5)->get();
```

- **paginate** – return a collection of models, but with pagination (only a "page" is returned at once)

```
$p = Product::where('qty', '<', 5)->paginate(10);
```

- **chunk** – return a collection of models, but handles all data in chunks (example: handling 1 million models, but only 100 at a time)

Queries - arrays

- Pluck method – returns an array with one field
- **pluck**– returns an array with a field value relative to a set of rows

```
$allIds = Product::pluck('id');
```

Array with all ids of the Product table

```
$names = Product::where('category', 'ssd drive')
                    ->pluck('name');
```

Array with all names of all "ssd drive" (product category)

```
$names = Product::where('category', 'laptop')
                    ->pluck('name', 'id');
```

Array with all names of all "laptops" (product category).
*- Array **keys** will be the "**id**" of the products*
*- Array **values** will be the "**name**" of the products*

Query Builder – Callbacks


```
$users = User::where('cat', '=', 'worker')->where('votes', '>', 100)
            ->orWhere('name', '=', 'John Special')->get();
```

```
select * from users where cat = 'worker'
     and votes > 100 or name = 'John Special'
```

- It is possible to group several expressions within a **callback**
  - Callback function receives and modifies the query instance
  - Example that adds "parenthesis":

```
$users = DB::table('users')->where ('cat', '=', 'worker')
    ->where(function ($query) {
        $query->where('votes', '>', 100)
              ->orWhere('name', '=', 'John Special');
    })
   ->get();
```

```
select * from users where cat = 'worker'
    and ( votes > 100 or name = 'John Special' )
```

Compare resulting SQL of this example with previous example

Pagination

- Retrieve data from the DB with "**paginate**" method:

```
$prods = Product::where('qty', '<', 5)->where('type', 12)
          ->orderBy('qty', 'desc')
          ->orderBy('name')
          ->paginate(10);      // Each page has 10 rows
```

- On the Blade view use one of the following code sections to generate the pagination links:

```
{{ $prods->links() }}
```

```
{{ $prods->withQueryString()->links() }}
```

‹ 1 2 3 4 5 6 7 8 ... 237 238 ›

- Laravel will automatically coordinate pagination links and pagination results

Eloquent ORM – insert, update, delete

- Insert, update or delete can be done in two ways:
  - Model instance method
  - Model Eloquent class method (static methods)
- Using **model instance** methods:

```
// Inserting
$user = new User
$user->name = 'John Doe';
$user->save();
$insertedId = $user->id; // Accessing the auto-inc column id

// Updating
$user = User::find(2);
$user->name = 'New name here';
$user->save();

// Deleting
$user = User::find(3);
$user->delete();
```

Eloquent ORM – insert, update, delete

- Using Eloquent **class methods** (static methods)

```php
// Creates a new user
$user = User::create(['name' => 'John Doe']);

// Fetch user by the attr, or create it if it doesn't exist
$user = User::firstOrCreate(['name' => 'John Doe']);

// Updates the user by properties
$user = User::update(['id' => 2, 'name' => 'New name here']);

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

Eloquent ORM – insert, update, delete

- Mass updates – example:

```
Flight::where('active', 1)
      ->where('destination', 'San Diego')
      ->update(['delayed' => 1]);
```

- Mass Assignment – examples:

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

```
$user->fill(['name' => 'John Doe']);
```

- A set of attributes (table columns) that can be filled at once (with an array).
- To prevent mass-assignment vulnerability, models (by default) don't support mass assignment. Model's attributes **$fillable** (white list) and **$guarded** (black list) specify which attributes can be mass assignable

```
class User extends Model
{
    protected $fillable = ['name', 'age'];
    protected $guarded = ['password', 'email'];
}
```

Eloquent ORM – Soft Deletes

- **Soft deletes**
  - Requires **deleted_at** attribute (value != null ➔ soft deleted)
  - Model must use `SoftDeletes` trait

```
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model
{
    use SoftDeletes;
    //protected $dates = ['deleted_at'];
}
```

- Calling **delete** method on the model, will set deleted_at with the current date and time – it will not delete the model (row)
- When querying a model that uses soft deletes, the soft deleted instances (rows) will be automatically excluded from query results.

Eloquent ORM – Soft Deletes


```
// Determine if a model was soft deleted
if ($flight->trashed()) { . . . }

// Including soft deletes on a query
$flights = Flight::withTrashed()
                ->where('account_id', 1)->get();

// Retrieving only the soft deleted models
$flights = Flight::onlyTrashed()
                ->where('airline_id', 1)->get();

// Restoring (un-delete) a model
$flight->restore();

// Restoring (un-delete) several models
Flight::withTrashed()
        ->where('airline_id', 1)
        ->restore();

// Permanently Deleting Models
$flight->forceDelete();
```

4 – MODEL RELATIONSHIPS

Eloquent Relationships

- Database table are often related to each other using relationships (foreign keys)
- Eloquent ORM can represent table relationships as methods.
- These methods can be abstracted as **collection** attributes or as **references** attributes
  - Example with relationships: get all posts of a specific user (id= 13)

```
$posts = User::find(13)->posts;
```

  - Previous example (with relationships) is equivalent to next example (without relationships):

```
$posts = Post::where('user_id', 13)->get();
```

Eloquent Relationships – 1:1

- One to One - Definition

```
class User extends Model
{
    // A user may have or not a phone
    public function phone()
    {
        return $this->hasOne(Phone::class);
    }
}
```

```
class Phone extends Model
{
    // A phone always belongs to a user
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

Eloquent Relationships – 1:1

- One to One – Using the models

```
$phone = User::find(123)->phone;
$phone->number = '99293283';
```

```
$user = Phone::find(324)->user;
$str = "I'm calling " . $user->name;
```

Eloquent Relationships

- hasOne (non default columns)

```
// Will look for a column called user_id on Phone model's table
return $this->hasOne(Phone::class);

// Overrides available
return $this->hasOne(Phone::class, 'foreign_key');
return $this->hasOne(Phone::class, 'foreign_key', 'local_key');
```

- belongsTo (non default columns)

```
// Will look for a column called user_id
return $this->belongsTo(User::class);

// Overrides available
return $this->belongsTo(User::class, 'foreign_key');
return $this->belongsTo(User::class, 'foreign_key', 'owner_key');
```

Eloquent Relationships – 1:N

- One to Many - Definition

```
class User extends Model
{
    // A user may have 0 or more posts
    public function posts()
    {
        return $this->hasMany(Post::class);
    }
}
```

```
class Post extends Model
{
    // A post always belongs to a user
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

Eloquent Relationships – 1:N

- One to Many – Using the models

```
$posts = User::find(123)->posts;
foreach ($posts as $post) {
    // . . . E.g. $post->title
}
```

```
$user = Post::find(1)->user;
$str = "Post owner is " . $user->name;
```

Eloquent Relationships – N:M

- Many to Many - Definition

```
class User extends Model
{
    // A user may have 0 or more roles
    public function roles()
    {
        return $this->belongsToMany(Role::class);
    }
}
```

```
class Role extends Model
{
    // A role can be owned by multiple users
    public function users()
    {
        return $this->belongsToMany(User::class);
    }
}
```

Eloquent Relationships – N:M

- Many to Many – **Pivot** Table
  - belongsToMany requires a **pivot table** to link the related tables. In the previous example the following tables must exist: `users, roles, role_user`
    - *The role_user table name is derived from the alphabetical order of the related model names, and should have user_id and role_id columns*

```
// Assumes the existence of a pivot table called user_role with
// two columns: user_id, role_id
return $this->belongsToMany(Role::class);

// Overriding pivot table and columns
return $this->belongsToMany(Role::class, 'pivot_table');
return $this->belongsToMany(Role::class,
                'pivot_table', 'col_table_1', 'col_table_2');
```

Eloquent Relationships – N:M

- Many to Many – access **Pivot** Table
  - Pivot table may include extra columns (besides foreign keys)
  - They are accessible through pivot attribute:

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

foreach ($user->roles as $role) {
    echo $role->pivot->created_at;
}
```

  - Filtering relationships via Pivot table columns

```
class User extends Model {
    public function roles()
    {
        return $this->belongsToMany(Role::class)
                    ->wherePivot('approved', 1);
    }
}
```

Eloquent Relationships – N:M

- Many to Many – **Pivot** Table as a Model
  - Although pivot table doesn't require a Model, it is possible to create a Model for the pivot table
  - Using the "pivot" model to define the relationship:

```
class User extends Model {  . . .
    public function roles()
    {
        return $this->belongsToMany(Role::class)
                    ->using(UserRole::class);
    }   . . . . }
```

  - The "pivot" model extends from **Pivot** class

```
class UserRole extends Pivot
{
    //
}
```

Eloquent Relationships – on queries

- All relationships can be accessed through its dynamic attribute:

```
// Retrieving the dynamic phone property
$phone = User::find(1)->phone;

// Retrieving all posts for a user
$posts = User::find(1)->posts;
```

- Relationships can also be used as an "expression" of a query:

```
// Retrieving all posts from a user with 10 or more likes
$posts = User::find(1)
              ->posts()
              ->where('like_count', '>=', 10)
              ->get();
```

Eloquent Relationships – null value

- If foreign key accepts null on the database, then the dynamic property might return null. We must take that into consideration:

```
$categoryName= Product::find(2)?->category->name;
```

or

```
$category= Product::find(2)->category;
// $category has an instance of Category model or null
$categoryName= $category ? $category->name : null;
```

  - "products" table has a foreign key that references the "categories" table, and accepts null – this means that the product might not have a category
  - When the foreign key value is null, the relationship dynamic attribute (category) also returns null
  - The following code will generate an **error**, because the category returns null, which does not have the "name" property

```
$categoryName= Product::find(2)->category->name;
```

Eloquent Relationships

- **Lazy loading** *(default)*
  - Relationship objects are only loaded when required.

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

For 25 books, 26 queries are executed. 1 query for the Book model (books table) and 25 additional queries to retrieve the author of each book

- **Eager loading**
  - Relationship object are loaded when the original model is loaded.

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

For the same 25 books, only 2 queries are executed. 1 query for the Book model (books table) and 1 query for all the authors of the 25 books

Eloquent Relationships

- Inserting related models

```
// belongsTo relationship
$phone = new Phone(['number' => 123123123]);
$user = User::find(1);
$user->phone()->associate($phone);
$user->save();
// hasMany relationship
$firstPost = new Post(['title' => 'Hello', 'body' => '...']);
$user->posts()->save($firstPost);
$posts = [
    new Post(['title' => 'A new post', 'body' => '...']),
    new Post(['title' => 'Another one', 'body' => '...']),
    new Post(['title' => 'That\'s all folks', 'body' => '...'])
];
$user->posts()->saveMany($posts);
// belongsToMany relationship
$user->roles()->attach(1); // attach role with id 1
$user->roles()->detach(1); // detach role with id 1 from user
```

Eloquent ORM

- Query builder includes more features, such as:
  - Timestamps
  - Query scopes
  - Polymorphic relations
  - Accessors and mutators
  - Events and observers
  - Eloquent collections
  - Etc.

- Official documentation:
  https://laravel.com/docs/eloquent

5 – COLLECTIONS

Eloquent collections and base collections

Eloquent collections / Base collections

- All Eloquent methods that return more than one model will return instances of **Eloquent Collections**
  `Illuminate\Database\Eloquent\Collection`

- Eloquent collections extends Laravel's **base collection**
  `Illuminate\Support\Collection`

- Collections provides a fluent wrapper for working with arrays of data
  - Collections' data exists *"in memory"* only

- More info:
  - Eloquent collections: https://laravel.com/docs/eloquent-collections
  - Collections: https://laravel.com/docs/collections

Collections

- Collections include methods like:
  - `where`
  - `count`
  - `sortBy`
  - …
- These methods will filter, transform, transverse, etc … data **in memory**.
- For instance, if we apply the where method on a collection, it will filter data already in memory

```
. . .
$filtered = $collection->where('price', 100);

$filtered->all();
```

Collections vs Query Builder

- **Attention:**
  - These 2 sections of code will produce a similar result ($a var will have the same data) but are completely different:

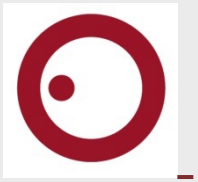
```
$a = Aluno::where('curso', 'TESP-TI')->get();
```

  - **where** is a method of the query builder
  - DB query executed:

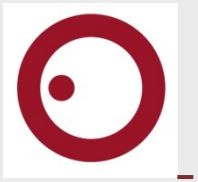
```
select * from alunos where curso = 'TESP-TI'
```

```
$a = Aluno::get()->where('curso', 'TESP-TI');
```

  - **where** is a method of the collection
  - DB query executed: `select * from alunos`
  - All “alunos” are loaded into the collection. Only then the filter is applied



5 – MIGRATIONS & SEEDS



Migrations

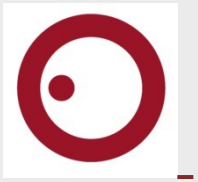
- Defines the **database structure** (database **schema**)

  https://laravel.com/docs/migrations

- Database **version control** - support multiple versions of the database schema – different developers or different installations of the application may have different versions of the database
- Typically paired with Laravel Schema Builder

  https://laravel.com/docs/schema

  - Database agnostic code to manipulate the DB structure

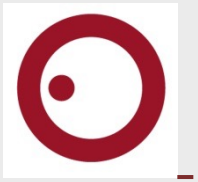


Migrations

- To create a migrate, execute:

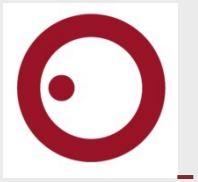
```
php artisan make:migration name_migration_file
```

- Each migration represents a version of the DB structure
- Each migration has a set of operations that upgrade the DB structure (*changes the DB structure from previous version to the new version*) and a set of operations that downgrade the DB structure (*changes the DB structure from the new version to the previous version*)
- Each migration is defined by a timestamped file (name of file includes a timestamp)– *version control is defined by the time*
- Database includes a table called "migrations" which is responsible for the synchronization between DB structure and migration files



Migrations

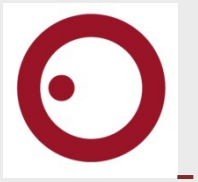
- Migration classes implement at least 2 methods:
  - **up()** – operations to upgrade the database to a new version
  - **down()** – operations to downgrade the database to the initial version. Down method reverts all up() operations
- Typically use Laravel Schema Builder - *examples will follow …*



Migration class : Schema Builder

- Examples of migrations classes (create “products” table):

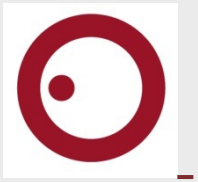
```
class CreateProductsTable extends Migration
{
    public function up()
    {
      Schema::create('products', function (Blueprint $table) {
          $table->bigIncrements('id');
          $table->string('name',30);
          $table->text('description')->nullable();
          $table->decimal('price', 13, 2);
          $table->decimal('discount', 13, 2);
          $table->timestamps();
        });
    }
    public function down()
    {   // reverts up() operations:
        Schema::dropIfExists('products');
    }
}
```



Migration class : Schema Builder

- Examples of migrations classes (create “categories” table):

```
class CreateCategoriesTable
 extends Migration
{
    public function up()
    {
    Schema::create('categories', function (Blueprint $table) {
          $table->bigIncrements('id');
          $table->string('name',20);
        });
    }
    public function down()
    {   // reverts up() operations:
        Schema::dropIfExists('categories');
    }
}
```

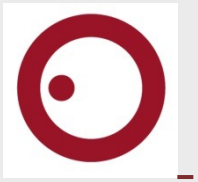


Migration class : Schema Builder

- Examples of migrations classes (modify “products” table – add a foreign key to "categories" table):

```
class AddCategoryToProductsTable extends Migration
{
public function up()
{
Schema::table('products', function (Blueprint $table) {
  $table->unsignedBigInteger('category_id');
  $table->foreign('category_id')->references('id')->on('categories');
  });
}

public function down()
{   // reverts up() operations:
  Schema::table('products', function (Blueprint $table) {
    $table->dropForeign('products_category_id_foreign');
  }
}
```



Migration class : Schema Builder

- Schema builder (**Schema** class) has commands to:
  - Create or drop tables, columns, relations, indexes, keys, etc…
- Example for columns:

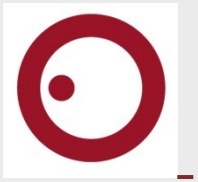
```
Schema::table('client', function (Blueprint $table) {
    $table->string('nome');
    $table->boolean('vip')->default(false);
    $table->string('cardnumber',10)->nullable();
}
```

Column Data Type

Column Name

Data type parameter (Size)

Column Modifier



Migration class : Schema Builder

- Example for Foreign Key:

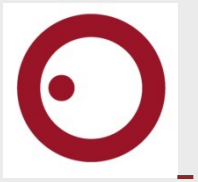
```
Schema::table('posts', function (Blueprint $table) {
    $table->unsignedInteger('user_id');
    $table->foreign('user_id')->references('id')->on('users');
});
```

Adds a foreign key

Name of Foreign Key Column
(created elseware)

Relationship Column at the
Foreign Table
(usually the primary key)

Foreign Table



Migration commands

- Artisan tool has a set of commands that use migrations to define the initial DB structure, upgrade or downgrade that DB structure
- Example of migration related commands:
  - Run pending migrations (upgrade)

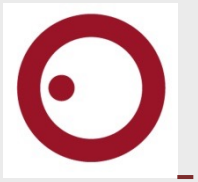
```
php artisan migrate
```

  - Rollback (downgrade) last migration

```
php artisan migrate:rollback
```

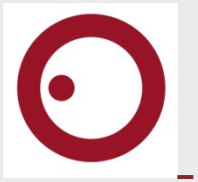
  - Drop all tables and re-run all migrations

```
php artisan migrate:fresh
```



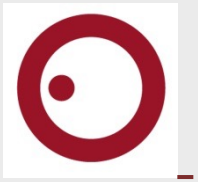
Migrations

- What happens when artisan command: “artisan migrate” is executed?
  1. Database structure is modified according to the schema builder commands
  2. The migration file name is added to the “migrations” table. This allows to control the history of all migrations applied
  3. When upgrading again, the migration system will execute all migration files that are not in the migrations table.
     - Order of execution is defined by timestamp (on the file name)
  4. When downgrading, migration system will execute the latest migration, which is defined on the “migrations” table.
     - Order of execution is defined by batch column value and by file name timestamp
- Please refer to the official documentation for a more in-depth explanation (`https://laravel.com/docs/migrations`)



Seeds

- Laravel also provides tools to seed (populate) your database with data using seed classes
- They are placed inside the "`database/seeds`" folder
- Using third-party libs (*pre-installed on Laravel*) such as the “fzaninotto/Faker” library it's relatively easy to populate the database with fake data for testing
- **Recommendation** for all your projects
  - Always populate the database with data that simulates a real word usage (at least the size of data). Many performance problems are detected early when running against a large database (*problems are detected during development time, not during production*)



Seeds

- Some examples (to execute the seeds)
  - Runs the default seeder – class DatabaseSeeder

```
php artisan db:seed
```

  - Runs the seeder UserTableSeeder

```
php artisan db:seed –class="UserTableSeeder"
```

- Please refer to the official documentation for a more in-depth explanation (`https://laravel.com/docs/seeding`)