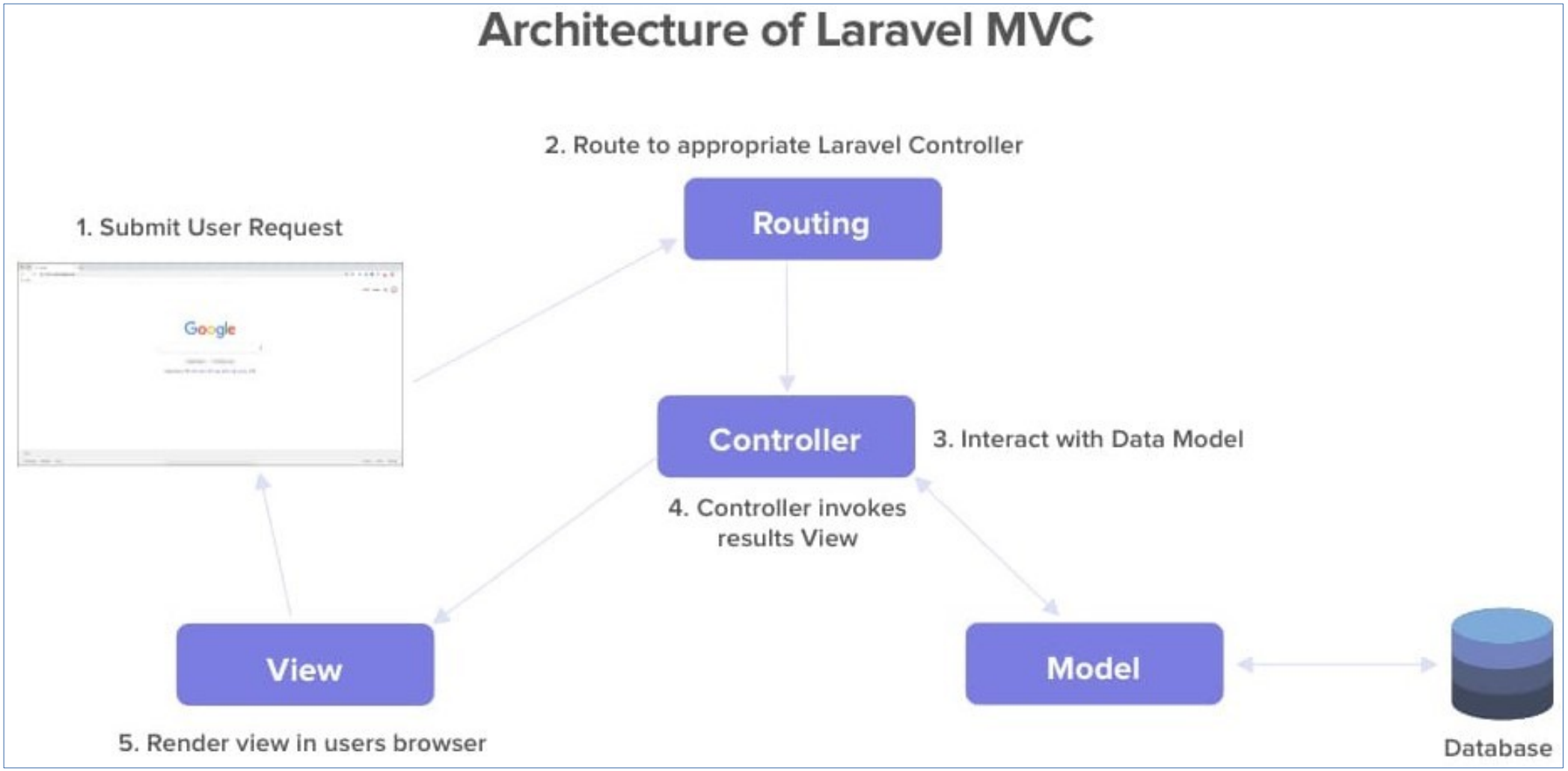
UECS3294 ADVANCED WEB APPLICATION DEVELOPMENT
CHAPTER 3 : DATABASES AND OBJECT-RELATIONAL MAPPING (ORM)

LOO YIM LING
ylloo@utar.edu.my

Previously - Laravel Framework Architecture



Information available on https://www.netsolutions.com/insights/laravel-framework-benefits/

Databases and Object-Relational Mapping (ORM)

1) Raw SQL queries.
2) Models and Active record.
3) CRUD.
4) Pagination.
5) Data migration.
6) Data Seeding.
7) Mass Assignment.
8) Relationships.
9) Query builder.

Database with Laravel

1) Laravel makes interacting with databases simple We can access databases in Laravel using:
   - Raw SQL
   - Query builder
   - Eloquent Object-Relational Mapping (ORM)
2) Currently, Laravel supports four databases:
   - MySQL
   - PostgreSQL SQLite
   - SQL Server
3) Support for Oracle is possible with third-party packages.
4) In our syllabus, we will only focus on MySQL databases. Tutorials & guides for using other databases can be found in Laravel documentation.

Eloquent Object-Relational Mapping (ORM)

1) Eloquent ORM provides a beautiful, simple ActiveRecord implementation for working with databases.
2) Each table has a corresponding model which is used to interact with that table.
3) If you have completed Practical 3, you would have tried generating a model class using Artisan CLI.
4) A database can also be created with migration when generating the model class using the `--migration` or `-m` option

```
php artisan make:model User -migration
php artisan make:model User -m
```

Eloquent ORM: Table Names

1) The model named photo in App\Models\Photo corresponds to database table named photos
2) By convention, plural name of the class will be used as the table name unless another name is explicitly specified.
3) Eloquent will assume the App\Models\Photo model records in the photos table.
4) We may specify a custom table by defining a table property on the model:

```
protected $table = 't_photo';
```

Eloquent ORM: Primary Key

1) Eloquent assumes that each table has a primary key column named id
2) To override, define a `protected $primaryKey` property in the model.
3) Eloquent also assumes that the primary key is an incrementing integer value, which means that by default the primary key will be cast to an int automatically.

Not recommended but can be done to override the defaults:

```
protected $primaryKey = 'key_Name';
protected $incrementing = 'false';
protected $keyType = 'string';
```

Eloquent ORM: Timestamps

1) By default, Eloquent expects the timestamp columns `created_at` and `updated_at` to exist in the tables.
2) To have these columns NOT automatically managed by Eloquent, set the `$timestamps` property to false
3) To customize the name of the timestamp columns, set the `CREATED_AT` and `UPDATED_AT` constants in the model:

```
public $timestamps = 'false';
const CREATED_AT = 'creation_date';
const UPDATED_AT = 'last_update';
```

Eloquent ORM: Database Connections

1) By default, all Eloquent models use the default database connection configured for your application.
2) To specify a different connection for the model, set the `$connection` property:

```
protected $connection = 'mysql2nd';
```

Eloquent ORM: Database Connections

3) Then, in config\database.php, the connection must have been defined:

```
'mysql2nd' => [
'driver' => 'mysql',
'host' => 'another.database.com',
'port' => '3306',
'database' => 'database_name',
'username' => 'username',
'password' => 'password',
'charset' => 'utf8',
'collation' => 'utf8_general_ci',
'prefix' => '',
'strict' => true,
'engine' => null,
],
```

Eloquent ORM: Retrieving Models

1) With models defined, we can fluently query the database table associated with the respective models.
2) In the example, using the `all()` method, a query that will return all of the results in the model's table was executed.

```
$photos = Photo::all();
foreach ($photos as $photo) {
   echo $photo->title;
}
```

Eloquent ORM: Retrieving Models (Additional Constraints)

1) We may add constraints to queries using query builder, and then use the `get()` method to retrieve the results.
   - The get method returns an Illuminate\Support\Collection instance containing the results where each result is an instance of the PHP `StdClass` object.

```
$photos = Photo::where('status', 1)
   ->orderBy('title', 'asc')
   ->take(10)
   ->get();
```

Eloquent ORM: Retrieving Models (Single & Multiple Row)

1) Use the `find` method to query a row by `id` (primary key)
2) Use the `first` method to retrieve the first result
3) Use the `find` method with an array of primary keys, which will return a collection of the matching records (multiple rows):

```
$photo = Photo::find(2977);
$photo = Photo::where('status', 1)
->first();
$photos = Photo::find([11, 12, 15]);
```

Eloquent ORM: Retrieving Models (Not Found Exceptions)

1) An exception need to be thrown if a model is not found.
2) This is particularly useful in routes or controllers.
3) The `findOrFail` and `firstOrFail` query methods will retrieve the models. However, if no result is found, `Illuminate\Database\Eloquent\ModelNotFoundException` will be thrown.
4) If the exception is not caught, a 404 HTTP response is automatically sent back to the user.

```
$photo = Photo::findOrFail(2351);
$photo = Photo::where('status', 1)
->firstOrFail();
```

Eloquent ORM: Retrieving Models (Selecting Columns)

1) Extract a single column value from a record using the `value` method. This method will return the value of the column directly.
2) To retrieve a collection containing the values of a single column, use `pluck` method.

```
$title = Photo::find(2977)>value('title');

$titles = Photo::where('status', 1)
->pluck('title');
foreach ($titles as $title)
{ echo $title; }
```

Eloquent ORM: Retrieving Models (Chunking Results)

1) In the event to process thousands of Eloquent records, use `chunk` method.
2) The `chunk` method retrieves a chunk of Eloquent models, feeding them to a given `Closure` for processing.
3) Using the `chunk` method will conserve memory when working with large result sets.

```
Photo::chunk(100, function ($photos)
{ foreach ($photos as $photo) {}
}
```

Eloquent ORM: Retrieving Models (Cursor)

1) The `cursor` method enables iteration through the records using a cursor, which will only execute a single query.
2) When processing large amounts of data, the `cursor` method may be used to greatly reduce memory usage

```
foreach (Photo::where('status', 1)
->cursor() as $photo) {}
```

Eloquent ORM: Inserting & Updating Models

1) To create a new record in the database, create a new model instance, set attributes on the model, then call the `save` method.
2) When the `save` method is called, a record will be inserted into the database table.

```
public function store(Request $request) {
$photo = new Photo;
$photo->title = $request->title;
//mass assignable
$photo->fill($request->all());
$photo->save(); }
```

Eloquent ORM: Inserting & Updating Models

1) The `save` method may also be used to update models that already exist in the database.
2) To update a model, it needs to be retrieveed, set attributes for update, and then call the `save` method.

```
public function update(Request $request) {
$photo = new Photo;
$photo->title = $request->title;

//mass-assignable:
$photo->fill($request->all());
$photo->save(); }
```

Eloquent ORM: Inserting & Updating Models (Mass Assignment)

1) The `create` method can be used to save a new model in a single line. The inserted model instance will be returned from the method.
2) Before doing so, `fillable` and `guarded` attribute on the model need to be specified, as all Eloquent models protect against mass-assignment by default.

Eloquent ORM: Inserting & Updating Models (Mass Assignment)

1) A mass-assignment vulnerability occurs when a user passes an unexpected HTTP parameter through a request, and that parameter changes a column in the database that we did not expect.
   - E.g., a malicious user might send an `is_admin` parameter through an HTTP request, which is then passed into our model's `create` method, allowing the user to escalate themselves to an administrator.

Eloquent ORM: Inserting & Updating Models (Mass Assignment: fillable, create)

1) Once we have made the attributes mass assignable, we can use the `create` method to insert a new record in the database.
2) The `create` method returns the saved model instance

```
class Flight extends Model {
protected $fillable = ['title']; }

$photo = Photo::create(['title' => 'The
Pinnacles, Western Australia']);

$photo->fill($request->all());
```

Eloquent ORM: Inserting & Updating Models (Mass Assignment: guarded)

1) While `$fillable` serves as a white list of attributes that should be mass assignable, `$guarded` property should contain an array of attributes not intended to be mass assignable.
2) Thus, `$guarded` functions like a black list.
3) We should only use either `$fillable` or `$guarded` , but not both.

```
class Photo extends Model {
protected $guarded = ['votes']; }
```

Eloquent ORM: Deleting Models

1) To delete a model, call the `delete` method on a model instance
2) If the primary key of the model is known, the model can be deleted without retrieving it using `destroy` method
3) A delete statement can be executed to delete a set of models

```
$photo = Photo::find(505);
$photo->delete();

Photo::destroy([505, 506, 508]);

$deletePhotos = Photo::where('status', 0)
->delete(); //delete all photos of status=0
```

Eloquent ORM: Deleting Models (Soft Delete)

1) In addition to actually removing records from the table, Eloquent can perform soft deletion of models.
2) When models are soft deleted, they are not actually removed from the table.
3) Instead, a `deleted_at` table attribute is set on the model and updated into the
4) If a model has a non-null `deleted_at` value, the model has been soft deleted.
5) To enable soft deletes for a model, use the Illuminate\Database\Eloquent\SoftDeletes on the model and add the `deleted_at` column to the `$dates` property

Eloquent ORM: Deleting Models (Soft Delete)

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

Eloquent ORM: Deleting Models (Retrieving and Restoring Soft Deleted Models)

1) The `onlyTrashed` method will retrieve only soft deleted models
2) To restore a soft deleted model into an active state, use the `restore` method on a model instance

```
$photos = Photo::onlyTrashed()
->where('status', 1) -> get();

$photo->restore();
```

Eloquent ORM: Deleting Models (Permanent Delete)

1) To permanently remove a model from the table, use the `forceDelete` method

```
// Force deleting a single model
$photo->forceDelete();
```

Eloquent ORM: Managing Relationships One-to-One

1) A `User` model model might be associated with one `Phone`
2) Place a `phone` method on the `User` model. The `phone` method should call the `hasOne` method and return its result

```
namespace App;
use Illuminate\Database\Eloquent\Model;
class User extends Model {
public function phone() {
return $this->hasOne(Phone::class); } }
```

Eloquent ORM: Managing Relationships One-to-One

1) The relationship also need to be defined inversely using the `belongsTo` method

```
namespace App; use Illuminate\Database\
Eloquent\Model;
class Phone extends Model {
public function user() {
return $this->belongsTo(User::class); } }
```

Eloquent ORM: Managing Relationships One-to-One

1) Once the relationship is defined, the related record maybe retrieved using Eloquent's dynamic properties.
2) Dynamic properties enable access to relationship methods as if they were properties defined on the model

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

Eloquent ORM: Managing Relationships One-to-Many

1) For instance, an album may have many photos

```
namespace App; use Illuminate\Database\
Eloquent\Model;
class Album extends Model {

public function photos() {
return $this->hasMany(Photo::class); } }
```

Eloquent ORM: Managing Relationships One-to-Many

1) The relationship also need to be defined inversely using the `belongsTo` method as the inverse for a one-to-many relationship is always a one-to-one relationship

```
namespace App; use Illuminate\Database\
Eloquent\Model;
class Photo extends Model {

public function album() {
return $this->belongsTo(Album::class); } }
```

Eloquent ORM: Managing Relationships One-to-Many

1) Once the relationship method has been defined, the collection of related photos by accessing the photos property.

```
use App\Models\Album;
$photos = Album::find(1)->photos;
foreach ($photos as $photo) {
}

**Further contraints
$photo = Album::find(1)->photos()
                       ->where('title', 'foo')
                       ->first();
```

Eloquent ORM: Managing Relationships Many-to-Many

1) For instance, a user has many roles, where the roles are also shared by other users.
   - Many users may have the role of "Admin".
   - To define this relationship, three database tables are needed; `users`, roles amd `role_user`
   - The `role_user` table is the pivot table and its name is derived from the alphabetical order of the related model names, and contains the `user_id` and `role_id` columns

Eloquent ORM: Managing Relationships Many-to-Many

1) The relationship's table structure can be summarized as following:

users
- id - integer
- name - string

roles
- id - integer
- name - string

role_user
- user_id - integer
- role_id - integer

Eloquent ORM: Managing Relationships Many-to-Many

1) Many-to-many relationships are defined by writing a method that returns the result of the belongsToMany method

```
namespace App; use Illuminate\Database\
Eloquent\Model;
class User extends Model {

public function roles() {
return $this->belongsToMany(Role::class); }
}
```

Eloquent ORM: Managing Relationships Many-to-Many

1) For the inverse relation, we will also define a method that returns the result of the belongsToMany method

```
namespace App; use Illuminate\Database\
Eloquent\Model;
class Role extends Model {

public function users() {
return $this->belongsToMany(User::class); }
}
```

Eloquent ORM: Managing Relationships Many-to-Many

1) Once the relationship is defined, the user's roles can be accessed using the `roles` dynamic relationship property
2) Further constraints can be added to the relationship query by calling the `roles` method and continuing to chain conditions onto the query

```
use App\Models\User;
$user = User::find(1);
foreach ($user->roles as $role) {
   $roles = User::find(1)->
      roles()->orderBy('name')->get();
}
```

Eloquent ORM: Query Builder

1) Laravel's database query builder provides a convenient, fluent interface to creating and running database queries. It can be used to perform most database operations in your application and works perfectly with all of Laravel's supported database systems.
2) The Laravel query builder uses PDO parameter binding to protect web application against SQL injection attacks. There is no need to clean or sanitize strings passed to the query builder as query bindings

Eloquent ORM: Query Builder
Retrieving All Rows

1) Use `table` method provided by the `DB` facade to begin a query.
2) The `table` method returns a fluent query builder instance for the given table, enabling chain of more constraints onto the query and then finally retrieve the results of the query using the get method

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

Eloquent ORM: Query Builder
Retrieving Single Row

1) Use the `DB` facade's `first` method. This method will return a single `stdClass` object

```
//object
$user = User::where('name', 'John')
   ->first();
return $user->email;
//column value
$email = User::where('name', 'John')
   ->value('email');
//single row by id
$user = User::find(3);
```

Eloquent ORM: Query Builder
Chunking Results

1) For instance, retrieve the entire users table in chunks of 100 records at a time

```
User::orderBy('id')
   ->chunk(100, function ($users) {
     foreach ($users as $user) {
     }
```

Eloquent ORM: Query Builder
Select Statements

1) In case of not wanting to select all columns from a database table; specify a custom "select" clause for the query using the `select` method

```
$users = User::
        select('name', 'email as user_email')
        ->get();
```

Eloquent ORM: Query Builder
Joins

1) The query builder may also be used to add join clauses to queries.
2) The first argument passed to the `join` method is the name of the table need to join to, while the remaining arguments specify the column constraints for the join. Multiple tables may be joined in a single query.

```
$users = User::
   join('contacts', 'users.id', '=',
      'contacts.user_id')
  ->join('orders', 'users.id', '=', 'orders.user_id')
  ->select('users.*', 'contacts.phone', 'orders.price')
  ->get();
```

Eloquent ORM: Query Builder
Where Clauses

1) The most basic call to the `where` method requires three arguments. The first argument is the name of the column. The second argument is an operator, which can be any of the database's supported operators. The third argument is the value to compare against the column's value.

```
$users = User::where('votes', '=', 100)
              ->where('age', '>', 35)
              ->get();
```

Eloquent ORM: Query Builder
Ordering

1) The `orderBy` method allows sorting of results of the query by a given column. The first argument accepted by the `orderBy` method should be the column to sort by, while the second argument determines the direction of the sort and may be either `asc` or `desc`:

```
$users = User::orderBy('name', 'desc')
                ->get();
```

Eloquent ORM: Query Builder
Grouping

1)The `groupBy` and `having` methods may be used to group the query results. The having method's signature is similar to that of the `where` method:

```
$users = User::groupBy('account_id')
             ->having('account_id', '>', 100)
             ->get();
```

Eloquent ORM: Query Builder Conditional When Statements

1) In case of certain query clauses are needed to apply to a query based on another condition.
2) For instance, one may only want to apply a where statement if a given input value is present on the incoming HTTP request.

```
$role = $request->input('role');
$users = User::
    when($role, function ($query, $role) {
     return $query->where('role_id', $role);
    })
    ->get();
```

Eloquent ORM: Query Builder
Insert Statements

1) The query builder also provides an `insert` method that may be used to insert records into the database table.
2) The `insert` method accepts an array of column names and values.

```
User::insert([
    'email' => 'mina@example.com',
    'votes' => 0
]);
```

Eloquent ORM: Query Builder
Update Statements

1) In addition to inserting records into the database, the query builder can also update existing records using the `update` method. The `update` method, like the `insert` method, accepts an array of column and value pairs indicating the columns to be updated. One may constrain the `update` query using `where` clauses.

```
$affected = User::where('id', 1)
                ->update(['votes' => 1]);
```

Eloquent ORM: Query Builder
Delete Statements

1)The query builder's delete method may be used to delete records from the table. One may constrain delete statements by adding "where" clauses before calling the delete method.

```
User::delete();

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

Eloquent ORM: Pagination

1) There are several ways to paginate items. The simplest is by using the paginate method on the query builder or an Eloquent query.
2) The paginate method automatically takes care of setting the query's "limit" and "offset" based on the current page being viewed by the user.
3) By default, the current page is detected by the value of the page query string argument on the HTTP request. This value is automatically detected by Laravel, and is also automatically inserted into links generated by the paginator.

Eloquent ORM: Pagination

1) Simple example of paginating 15 records:

```
//Pagination logic
return view('user.index', [
                'users' => User::paginate(15)

//Displaying pagination results
{{ $users->links() }}
```

Database Migrations

1) Migrations are like version control for your database, allowing your team to define and share the application's database schema definition.
2) If one have ever had to tell a teammate to manually add a column to their local database schema after pulling in changes from source control, one has faced the problem that database migrations solve.

Database Migrations

1) use the `make:migration` Artisan command to generate a database migration.
2) The new migration will be placed in `database/migrations` directory.
3) Each migration filename contains a timestamp that allows Laravel to determine the order of the migrations

```
php artisan make:migration
create_flights_table

php artisan migrate
```

Database Migrations

1) Automation of database table creation using `create` method on the `Schema` facade.
2) The `create` method accepts two arguments: the first is the name of the table, while the second is a closure which receives a `Blueprint` object that may be used to define the new table

```
Schema::create('users', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->string('email');
    $table->timestamps();
});
```

Database Seeding

1) Laravel includes the ability to seed database with test data using seed classes.
2) All seed classes are stored in the `database/seeders` directory.
3) By default, a `DatabaseSeeder` class is defined. From this class, use the `call` method to run other seed classes, allowing control of the seeding order.

Database Seeding

1) To generate a seeder, execute the `make:seeder` Artisan command.
2) All seeders generated by the framework will be placed in the `database/seeders` directory.

```
php artisan make:seeder UserSeeder
```

Database Seeding

1) Query builder to manually insert data
2) Eloquent model factories.

```
public function run()
{  DB::table('users')->insert([
     'name' => Str::random(10),
     'email' => Str::random(10).'@gmail.com',
     'password' => Hash::make('password'),
     ]);}

public function run()
{    User::factory()
           ->count(50)
           ->hasPosts(1)
           ->create(); }
```

Database Seeding

1) Execute the `db:seed` Artisan command to seed database.
2) By default, the `db:seed` command runs the `Database\Seeders\DatabaseSeeder` class, which may in turn invoke other seed classes.
3) However, `--class` option may be used to specify a specific seeder class to run individually

```
php artisan db:seed

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

END OF LECTURE 04