Module 21- Laravel Migrations Introduction

Laravel migrations are like version control for your database, allowing you to manage and modify database schema over time. They are typically used to create or update database tables programmatically, making your schema changes trackable and reversible.

In software development, **migration** refers to moving or transferring data, systems, or configurations from one environment to another. In the context of web development (like Laravel or other frameworks), **database migration** is commonly used to manage and version the database schema efficiently.

Why We Use Migration

1. **Version Control for Databases**
    - It ensures that database changes (like adding a table or column) are version-controlled, just like code.
    - You can roll back or forward to specific versions easily.
2. **Team Collaboration**
    - Multiple developers can work on the database without conflicts since migrations define the structure in code.
    - It avoids issues caused by manually modifying the database.
3. **Consistency Across Environments**
    - Helps in syncing the database structure across different environments (development, staging, production).
    - Guarantees the database setup is the same everywhere.

4. **Easier Deployment**
    - Migrations automate database changes during deployment, making updates faster and less error-prone.
5. **Reproducibility**
    - You can recreate the entire database schema from scratch using migrations, which is helpful for onboarding or setting up new environments.

Benefits of Migration

1. **Trackable Changes**
    - Each migration has a record of what was changed, making it easy to audit and understand changes over time.
2. **Rollback Capabilities**
    - If a migration introduces an issue, you can easily revert the changes.
3. **Programmatic Management**
    - Migrations use code to define database structure, making it easier to manage and document than manual SQL scripts.
4. **Time-Saving**
    - Reduces repetitive tasks when modifying the database.
    - Automatically applies changes to all environments without manual intervention.
5. **Error Prevention**
    - Reduces human error by automating database updates.
6. **Seamless Integration with Frameworks**

- Many frameworks like Laravel and Django provide built-in migration tools, making it straightforward to integrate into development workflows.

Rules for Using Migration

1. **One Migration for One Purpose**
   - Each migration file should address a single purpose, like creating a table, adding a column, or modifying a constraint. Avoid combining unrelated changes in one migration.
2. **Keep Migrations Sequential**
   - Ensure migrations are created in a logical sequence so they can be run in the correct order. Frameworks usually timestamp migration files to maintain this order automatically.
3. **Use Framework Commands**
   - Always use the framework's CLI commands (e.g., php `artisan make:migration` in Laravel) instead of creating migration files manually. This ensures proper formatting and reduces errors.
4. **Use Descriptive Names**
   - Name migrations clearly to indicate their purpose. For example:
     - ✅ `create_users_table`
     - ✅ `add_email_to_users`
     - ❌ `migration_1`
5. **Avoid Directly Modifying Live Databases**
   - Make changes only through migrations to keep database schema consistent across environments.
6. **Test Migrations Locally**

   - Run migrations locally before deploying to production to catch errors early.
7. **Use Rollbacks Wisely**
   - Ensure every migration has a rollback (down) method to reverse changes if needed. This is critical for handling mistakes or failed deployments.
8. **Avoid Data Manipulation**
   - Migrations should focus on structural changes, like creating or altering tables. If data needs to be inserted or updated, use seeders or a separate script.
9. **Backup Databases Before Major Changes**
   - Always back up the database before running migrations in production to prevent accidental data loss.
10. **Document Dependencies**
    - If a migration depends on another, ensure the dependent migration is run first. Most frameworks handle this through timestamps or order.
11. **Keep Production Data Safe**
    - Be cautious with destructive operations (like dropping tables or columns) in migrations. Confirm that such changes won't cause data loss unless intended.
12. **Use Constraints and Indexes Properly**
    - Define primary keys, foreign keys, and indexes during table creation to optimize database performance.

Example in Laravel

Creating a migration to create a `users` table:

bash

Copy code

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

Generated migration file:

php

Copy code

```
public function up()

{

    Schema::create('users', function (Blueprint
$table) {

        $table->id();              // Primary
key

        $table->string('name');    // Name
column

        $table->string('email')->unique(); //
Email column with unique constraint

        $table->timestamps();      // Created
and updated timestamps

    });

}
```

```
public function down()

{

    Schema::dropIfExists('users'); // Rollback
logic

}
```

Steps to Apply Laravel Migration

1. **Create a Migration:** Use the artisan command to generate a migration file.

php artisan make:migration create_table_name

This creates a file in the `database/migrations` directory.

Define the Schema: Open the generated file and define the schema inside the `up()` method for creating the table or making schema changes. Use the `down()` method for rollback.

Run the Migration: Use the artisan command to execute migrations.

```
php artisan migrate
```

1.

Rollback (if needed): To revert the last batch of migrations:
bash
Copy code
```
php artisan migrate:rollback
```

2.

Example: Implementing Migrations in a Project

Scenario: You are creating a "users" table with columns for `id`, `name`, `email`, and `password`.

Step 1: Create the Migration File

Run the following command:

bash
Copy code
```
php artisan make:migration create_users_table
```

Step 2: Define the Schema

Edit the generated migration file located in `database/migrations`.

php
Copy code
```
<?php
```

```
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function
(Blueprint $table) {
            $table->id(); // Auto-incrementing
primary key
            $table->string('name'); // Name
column
            $table->string('email')->unique();
// Email column, must be unique
            $table->string('password'); //
Password column
            $table->timestamps(); // Adds
created_at and updated_at columns
        });
    }

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

Step 3: Run the Migration

Execute the migration to create the `users` table in the database.

bash
Copy code
```
php artisan migrate
```

Explanation:

- **`Schema::create`**: Creates a new table named `users`.
- **Columns**:
  - `id`: Automatically generates an integer primary key.
  - `name`: Stores the user's name.
  - `email`: Unique column for the user's email.

  - `password`: Stores the user's password (hash it for security).
  - `timestamps`: Automatically manages created_at and updated_at columns.
- **`Schema::dropIfExists`**: Deletes the table when rolling back the migration.

Example Usage in a Project

Project: Railway Food Delivery Website

Suppose you need a `restaurants` table to store details about restaurants:

Generate Migration:
bash
Copy code
```
php artisan make:migration
create_restaurants_table
```

1.

Define Schema:
php
Copy code
```
public function up()
{
    Schema::create('restaurants', function
(Blueprint $table) {
```

sondagar divya

```
        $table->id(); // Primary key
        $table->string('name'); // Restaurant
name
        $table->string('location'); // Location
of the restaurant
        $table->string('cuisine_type'); // Type
of cuisine offered
        $table->timestamps(); // Created and
updated timestamps
    });
}

public function down()
{
    Schema::dropIfExists('restaurants');
}
```

2.

Run Migration:
bash
Copy code

```
php artisan migrate
```

3.

Insert Data (Optional, via Seeder): Create a seeder to populate the table.
bash

Copy code

```
php artisan make:seeder RestaurantsTableSeeder
```

Inside `database/seeders/RestaurantsTableSeeder.php`:

php

Copy code

```
DB::table('restaurants')->insert([
    ['name' => 'Spicy Treats', 'location' =>
'Delhi', 'cuisine_type' => 'Indian'],
    ['name' => 'Burger World', 'location' =>
'Mumbai', 'cuisine_type' => 'Fast Food'],
]);
```

4.

Run Seeder:

bash

Copy code

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

5. php artisan db:seed --class=RestaurantsTableSeeder