Using Databases in Laravel

- The essentials
  - Architecture of a web app with a database
  - Framework concerns
  - Laravel and Databases
  - Database Configuration
  - Migrations
    - Creating migrations
    - Defining table structure in a migration
    - Defining columns
    - Running migrations
  - Using the query builder
    - Basics
    - Ordering
    - Retrieving a single row
- Digging deeper
  - Digging deeper into column definitions
  - Other datatypes
  - Other column modifiers
  - Understanding the migration order
  - Rolling back migrations
- Further reading

The essentials

In this course, students will discover that databases serve as the foundational backbone for storing and managing data in the majority of web applications. Understanding databases is crucial as they underpin the storage, retrieval, and manipulation of information that powers dynamic web applications across various domains. Storing data in a database offers several advantages for web applications:

1. **Structured Storage:** Databases provide structured storage, meaning they organize data into tables, rows, and columns. This structure allows for efficient storage and retrieval of information, making it easier to manage and query data.
2. **Data Integrity:** Databases often enforce data integrity constraints, ensuring that the data stored follows specified rules. This helps maintain accuracy and consistency within the data, reducing errors that could occur with manual file-based storage.
3. **Concurrency Control:** Web applications often have multiple users accessing and modifying data simultaneously. Databases offer mechanisms for handling concurrency, allowing multiple users to interact with the data without causing conflicts or inconsistencies.
4. **Security:** Databases provide various security features such as access control, authentication, and encryption, which help protect sensitive information from unauthorized access or tampering.
5. **Scalability:** Databases can scale to handle large volumes of data and increased user loads. They offer techniques like sharding, replication, and clustering to distribute data and workload across multiple servers, ensuring optimal performance as the application grows.

An alternative to traditional databases could be using file storage or flat files. While this method might work for smaller applications, it lacks many of the advantages databases offer. Flat files are less structured, making it harder to manage and query data efficiently. They also don't typically provide built-in mechanisms for enforcing data integrity or handling concurrent access, which can lead to issues in larger or more complex applications.

Architecture of a web app with a database

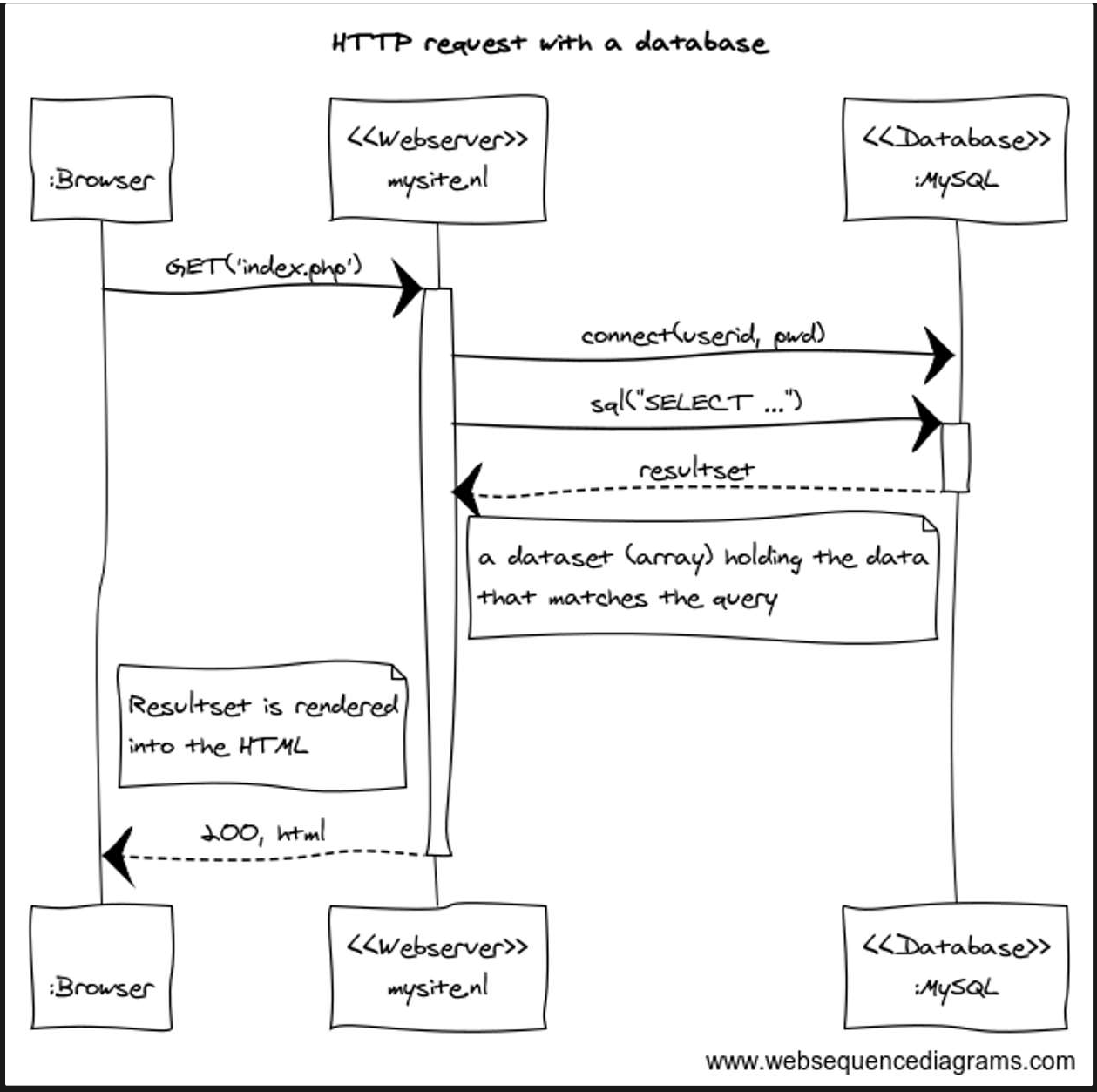
Typically web applications that work with a database consist of:

- a Web server, running a server side scripting language like PHP - exactly as described before
- a Database (server) - a software application that hosts databases

These applications collaborate as follows:

1. A client browser sends an HTTP request to the web server
2. The web server starts to process the request. It discovers it contains PHP, so it runs it
3. Using PHP, the script makes the server establishes a connection to the database server
4. Another PHP statement tells the server to execute a specific SQL statement on the database server.
5. The database server processes the query, and returns a *resultset*, the data that matches the query, back to the webserver
6. The webserver (PHP) renders the resultset into the HTML. PHP code tells the server exactly how to render it
7. The webserver finally returns a 200 HTTP response containing the rendered HTML

The following sequence diagram shows this:



Framework concerns

Web frameworks, when interfacing with databases, encounter several critical concerns:

1. **Adaptability and Vendor Lock-In:** Avoiding vendor lock-in is crucial, as it prevents dependency on specific database vendors or technologies. A good framework provides abstraction layers or ORM (Object-Relational Mapping) tools, enabling developers to work with different databases without significant code changes. This flexibility ensures adaptability to changing project requirements or migrations to different database systems.

2. **Maintainability:** Frameworks must ensure that database interactions remain maintainable. This involves employing clear, standardized methods to connect, query, and update the database. A well-structured framework simplifies maintenance by organizing database-related code logically, facilitating future updates and modifications without compromising functionality.
3. **Security:** Web frameworks should prioritize security when interacting with databases. This includes implementing measures to prevent common vulnerabilities such as SQL injection, enforcing proper authentication and authorization mechanisms, and ensuring encrypted communication channels between the application and the database. Additionally, frameworks should regularly update and patch database connectors or libraries to mitigate emerging security risks.

Frameworks that address these concerns effectively provide a solid foundation for web applications, ensuring maintainability, adaptability across various database systems, and robust security measures to protect sensitive data.

Laravel and Databases

Laravel addresses these concerns through its robust features and practices:

1. **Adaptability and Vendor Lock-In:** Laravel minimizes vendor lock-in by offering support for multiple database systems like MySQL, PostgreSQL, SQLite, and SQL Server through its database abstraction layer by:
    a. storing the database configuration info in a single file: `.env`
    b. abstracting vendor specific SQL syntax with it's *query builder*
    c. allowing developers to define database schemas and changes in a database-agnostic way using *migration* files. Enabling seamless transitions between different database systems without extensive code alterations.
2. **Maintainability:** Laravel promotes maintainability by utilizing a clear and expressive syntax called *Eloquent ORM* for interacting with databases. Eloquent provides an intuitive and consistent way to define database models and relationships, keeping database-related code organized within the application. This promotes maintainability by making it easier to read, write, and modify database queries.

3. **Security:** Laravel prioritizes security through various measures. It includes protection against SQL injection by utilizing parameter binding in queries by default. Additionally, Laravel's authentication and authorization features provide simple yet powerful tools for implementing secure user authentication and access control. Laravel's community actively monitors and addresses security vulnerabilities, ensuring prompt updates and patches to address emerging threats.

Overall, Laravel's emphasis on a clean, expressive ORM, support for multiple databases, and built-in security measures helps developers build maintainable, adaptable, and secure web applications without being tightly bound to specific database technologies.

Database Configuration

When a database is configured, Laravel automatically connects to it on each request. It is important to know that there are 2 files that play a role in the database and that in normal circumstances, you only need to configure one file. These files are:

- `config/database.php` - typically holds generic configuration details for different types of database servers like MySQL, SQLite and PostgreSQL
- `.env` - holds application environment specific details, the connection details to the database server your application needs to connect to

> ! Never add any connection details like usernames and passwords to files that are committed to your project GitHub repository. Once committed, they are visible to anyone that has access to your repository. Typically, novice developers tend to add these details to `config/database.php` or `env.example`. Make sure you avoid this mistake in all your projects.

In this course, we always work with a MySQL database, which is typically configured in `.env` as follows:

```
DB_CONNECTION=mysql DB_HOST=127.0.0.1 DB_PORT=3306 DB_DATABASE=mydatabase
DB_USERNAME=root DB_PASSWORD=
```

Explanation:

- `DB_CONNECTION` tells Laravel which type of database server to connect to. Typically, this must be `mysql`, but it can also be `sqlite`, `pgsql` or `sqlsrv`.
- `DB_HOST` - URL of the database server to connect to

- `DB_PORT` - port number of the database server. For MySQL, it's 3306 by default
- `DB_DATABASE` - name of the database to connect to, while MySQL can host multiple databases that are fully separated from each other
- `DB_USERNAME` - the username of the user account that is used to connect to the database server
- `DB_PASSWORD` - the password of that account

Migrations

As mentioned before, Laravel uses migrations to keep your database structure maintainable and adaptable. Migrations serve as a version control system for your database, enabling your team to collectively establish and distribute the blueprint of your application's database structure, so nobody needs to manually add tables or columns on your environment, your team mates environments and the production environment.

Creating migrations

Migrations are PHP files that reside in the `database/migrations` directory. Each file holds a class definition that has 2 methods: `up` and `down`. The `up` method describes how the database structure needs to change. Typically adding new tables, columns and/or indexes. The `down` method should reverse this change. The Digging Deeper chapter explains more about this.

A standard Laravel installation already has some migration files.

> ⚠️ Do not modify or remove these predefined migrations, they are part of the built-in functionality of the framework.

Migrations can be created with an Artisan command like:

```
php artisan make:migration create_tasks_table
```

Laravel tries to help you as much as possible. In order to do this, you should try to hold on to specific migration naming conventions. The rules here are:

- When a migration needs to create a table, use `create_yourTableNameHere_table`. So the `create_tasks_table` will generate a migration that already has code that creates a new table called `tasks`. You then only need to specify which columns you want in your table

- When a migration needs to update an existing table, use `update_yourTableNameHere_table`.
- When a migration needs to drop an existing table, use `drop_yourTableNameHere_table`.

When the migration is created, Laravel prepends the name with a timestamp. This timestamp is important in the process of running these migrations.

Defining table structure in a migration

A migration that creates a table typically looks like this:

```
return new class extends Migration { /** * Run the migrations. */ public function up(): void { Schema::create('foo_bars', function (Blueprint $table) { $table->id(); $table->timestamps(); }); } /** * Reverse the migrations. */ public function down(): void { Schema::dropIfExists('foobars'); } };
```

If we look at the `up` method, it consists of one `Schema::create` statement. This statement will create the given table in the database. The second argument is a function that is called on creation. It passes the `$table` variable, that is used to define the structure ('blueprint') of that table. The body of that function defines the actual table structure. When generated, it holds:

- `$table->id()` - Defines a column `id` that Laravel likes to have as the primary key
- `$table->timestamps()` - Defines two columns: `created_at` and `updated_at` that Laravel likes to have. They will hold timestamps when a row is created and last updated

> ! Although the primary key and the use of these timestamps are fully configurable, this course will always work with these exactly as Laravel is configured by default. Therefore you should **never remove these wo statements**. If you ever feel the need to do this in this course, it will probably be a lack of understanding of your assignment. You should contact a teacher or student assistant for help

Defining columns

Additional columns can be defined like so:

```
$table->id(); $table->string('fibble_floob'); $table->integer('snazzle_dazzle')->nullable(); $table->timestamps();
```

Each additional statement defines a columns name (the string argument) and a datatype (method name). Additional column properties like default value or whether or not `null` values are allowed can be defined by chaining extra method calls.

> 💡 It is a good practice to consider the order if the columns carefully as they will appear in exactly the same order in most DB managers when you want to browse the data. Therefore you want the most important columns to appear first, and the least important last. At least make sure that the `$table->id()` statement is the first statement and the `$table->timestamps()` is the final one.

Essential Datatypes

Essential datatypes you can use here are:

- `string()` - stores strings. In MySQL, it will be a varchar of 255 characters. This can be used for most text based columns
- `text()` - stores 'text blobs'; large amounts of text. Use this when you know that these columns will contain large strings (>255 characters)
- `integer()` - for integer numbers
- `boolean()` - for boolean values
- `timestamp()` - holds date/time values

Essential additional column properties

Laravel calls these *column modifiers*. The most important ones for this course are:

- `nullable()` - whether or not a `null` value is allowed. If not defined, `null` values are not allowed
- `default()` - specify a default value for a column

Running migrations

Migrations must be executed in order to update the database structure on a specific environment. To run migrations, developers use the Artisan command `php artisan migrate`. This command reads the migration files that haven't been executed and applies their changes to the database. In order to do this, Laravel maintains a `migrations` table in the database to track which migrations have been executed, preventing duplications. Laravel runs migrations in batches, typically executing each migration file in sequence. The file name is used to define the sequence order. This is why the timestamp is important: the earliest timestamp is run first.

> ⚠️ When you made a mistake somewhere, and the database becomes corrupt or your migrations do not work anymore, you can use the `php artisan migrate:fresh` command. But be aware that this command drops the entire database content (all the tables, including its data) and runs all the migrations as if it was a new environment. So, it might fix your situation, but you lose all of your application data in the process.

Using the query builder

> 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.

Basics

A typical Laravel query looks like so:

```
// Build a query $query = DB::table('users')->where('id', '>', 10); //
Execute the query and return the resultset $resultset = $query->get();
```

Notice that you can add different SQL clauses to the query by chaining more methods

When executed, this will make Laravel to execute (about) the following SQL query:

```
SELECT * FROM users WHERE id > 10;
```

When the query is executed, a collection of objects is returned. Each object represents one table row. You can access each column's value by accessing the column as a property of the object:

```
$users = DB::table('users')->get(); foreach ($users as $user) { echo $user->name; // echoes the value of the 'name' column of that row }
```

Ordering

Ordering query results is essential too. You can chain the `orderBy` method to a query to sort the results like so:

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

The first argument is the column name to sort by and the second one specifies the sorting order. This can be either `asc` or `desc`.

Retrieving a single row

Sometimes you just need a single row from a database table. To do this, you may use the `first` method instead of the `get`. For example:

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

The `first` method does not return a collection but a single object. Because Laravel works by default with a primary key column `id`, it also has the convenient `find` method that returns the object whose `id` value matches the methods argument, like so:

```
$user = DB::table('users')->find(3);
```

⚠️ Note that, although the query builder is a feature in Laravel that allows you to da any database interaction, you should prefer to use **Eloquent** instead. Eloquent has the query builder built in and also allows for more expressive and maintainable code.

Digging deeper

The SHOULDs about this topic, described as links with some extra information to help understanding the content of the link

Digging deeper into column definitions

In the previous chapter, we discussed the essentials of column definitions in migrations: name, datatype and column modifiers. We listed the most essential types and modifiers, but there are a lot more of these. They are all described in the Laravel documentation

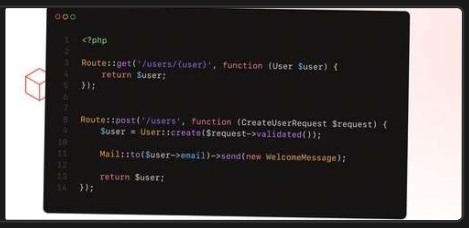
Other datatypes

You can find a complete list of available datatypes for columns here:

Laravel - The PHP Framework For Web Artisans

Laravel is a PHP web application framework with expressive, elegant syntax. We've already laid the foundation — freeing you

https://laravel.com/docs/migrations#available-column-types



Specifically noteworthy are types that represent dates, times, integer and real numbers.

Integer numbers

When you need to design your own database, and choose a datatype for an integer number, there are many options available. The most common ones are: `bigInteger`, `integer`, `mediumInteger`, `smallInteger`, `tinyInterger`, `unsignedInteger` and `unsignedTinyInteger`. What you choose depends on the following:

1. Signed/unsigned - If you are certain that negative values will not be stored here, use one of the unsigned types
2. Maximum value - the 'size' (big, small, tiny, etc.) determines the amount of bytes the number will use in storage. The larger the size, the more bytes it needs, but also the larger values you can store.

Type	Storage (Bytes)	Minimum Value Signed	Maximum Value Signed	Maximum Value Unsign
`TINYINT`	1	`-128`	`127`	`255`
`SMALLINT`	2	`-32768`	`32767`	`65535`
`MEDIUMINT`	3	`-8388608`	`8388607`	`16777215`
`INT`	4	`-2147483648`	`2147483647`	`4294967295`
`BIGINT`	8	`-9223372036854775808`	`9223372036854775807`	A lot ($2^{63} - 1$)

Real numbers

Representing real numbers has the same consideration:

Type	Storage (Bytes)	Minimum Value (Signed)	Maximum Value (Signed)
`FLOAT`	4	`-3.402823466E+38`	`-1.175494351E-38`
`DOUBLE`	8	`-1.7976931348623157E+308`	`-2.2250738585072014E-308`

When defining real number columns, you always need to add two parameters: the *precision* (total digits) and the *scale* (decimal digits)

Besides these, you also have the `decimal` option. This represents a database datatype that store *exact* numeric data values. These types are used when it is important to preserve exact precision, for example with monetary data. You can see this as integer numbers, but only has a decimal point at a fixed location.

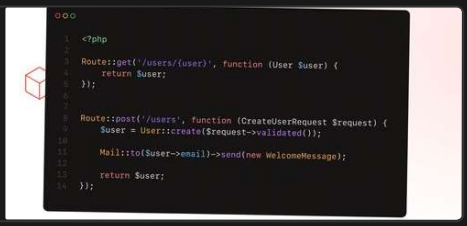
Other column modifiers

A complete list of the available column modifiers can be found here:

Laravel - The PHP Framework For Web Artisans

Laravel is a PHP web application framework with expressive, elegant syntax. We've already laid the foundation — freeing you

https://laravel.com/docs/migrations#column-modifiers



Particularly noteworthy modifiers are:

- `unsigned()` - specifies that the given numeric value should be unsigned
- `comment()` - specify a meaningful comment that is visible in a DB manager

- `useCurrect()` - for timestamp columns. It specifies that the current timestamp is used for the default value

Understanding the migration order

When Laravel executes the `php artisan migrate` command, the framework:

1. Lists all the files in `database/migration` directory in alphabetical ascending order
2. Lists all the records in a special database table called `migrations`. This table has a row for each migration filename that is already migrated
3. For each of the migration files in the list it:
    a. Checks if it already present in the `migrations` table
    b. If present, it skips it. If not present, it will:
        i. Call the `up` method of the migration
        ii. On success, adds the filename to the `migarions` table

This ensures that each migration is run only once in an environment.

Rolling back migrations

Migrations can be rolled back as well. This means that each migration can undo its changes bringing the database in a state as it was before the migration was