Laravel Application Development, Continued

Fall 2018 – Introduction to Full-Stack Development

Instructor: Matthew Fritter

matt@frit.me

Stepping Back: Lab 2

- The due date for Lab 2 will be extended until Friday at midnight, as it was a more difficult lab than I was intending
- The SimpleChat site that we developed in Lab 2 is pretty typical of a simple web application, and includes many of the concepts that you should be familiar with in Full-Stack Web Development
  - GET and POST requests
  - Dealing with AJAX calls, API handling
  - Reading/Writing to a database
- Good news: Lab 3 will be extending Lab 2, so once you've got the basics down, it should be smoother sailing

SimpleChat: Creating a New Room

- The front page of the site has a form that accepts a chatroom name and submits it via POST route to the create() function in the ChatController
- In the create function, we take that name and generate a fairly unique hash based on it by appending the current time to the name and hashing it
  - Review: Hashing is a one-way function that maps a variable length input (a string of potentially any length) to a fixed length output (a string of fixed length – for MD5, 128 bits, or 32 characters).
  - Why do we do this? To prevent overlaps if two people create rooms with the same name. There's an edge case where two people make rooms with the same name at the exact same second – but we won't worry about that for now
- The create function then inserts the name and hash of the room into the database, and redirects us to a new URL based on that hash
  - Since we're using a redirect, we're going to hit the Router again

SimpleChat: Entering the Room

- When we go to the '/chat/<hash>' URL, we hit a GET route that calls the join() function; This is when a user "joins" the chatroom
- The GET route passes the hash as a parameter to identify which room we're loading, and gets the room name by querying the database for a matching hash
- We assign the user a random username and a random color (either HTML5 named color, or hex code)
- Then, we return the "chat" blade (view), passing along the data for the username, color, chatroom name, and chatroom hash

SimpleChat: Sending a Message

- When we send a message in the chatroom, JavaScript client-side is sending an asynchronous (doesn't wait for completion) POST request to the server. This request includes the username, color, message text, and the hash of the chatroom.
- The Router, using the API routes file, passes this request to the send() function.
- In the send function, we pull the data out of the request, sanitize it (make it safe), and then use it to build an HTML string with the color username and message.
- Then, we insert the message into the database along with the hash (which identifies which room the message belongs in), and the timestamp (when the message was processed by the server)
- Since this is an AJAX POST request, we don't need to return anything.

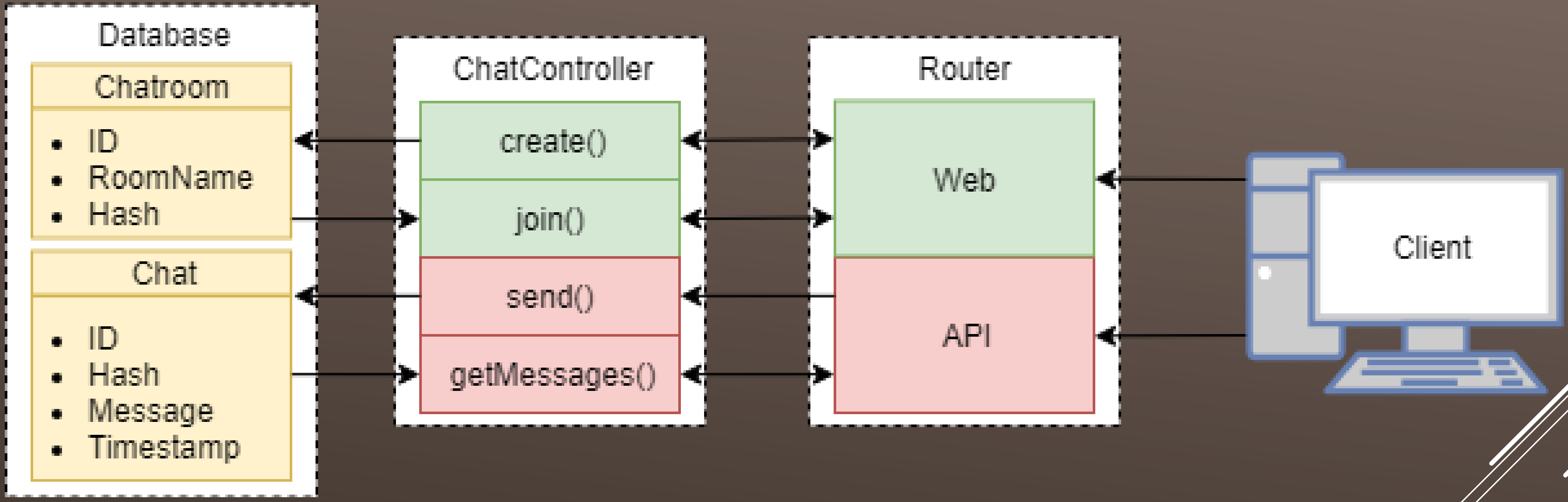
SimpleChat: Receiving Messages

- On the client side, the JavaScript makes an asynchronous GET call for new messages every five seconds
  - This is usually called polling, or specifically periodic polling
- Each message in the database has an auto-incremented integer ID associated with it – newer messages will always (or *almost* always) have a higher ID number than older messages
- The client-side JavaScript keeps track of the largest (maximum) message ID that the client has seen
- When the GET call is made, this maximum message ID, along with the room hash is passed as parameters through an API route to the getMessages() function
- getMessages() queries the database for all messages where the hash is the same (indicating same room) and where the ID is larger than the maximum parameter (indicating newer messages than the last message loaded in the client)
- This is then compacted into a JSON array and passed back to the JavaScript, which parses it and writes it out to the text box on the chat page

Simplicity

- From the outset, when you're not familiar with the code, this may seem like a complicated process
- However, keep in mind that this is a functional anonymous web chat, if lacking in features and rather slow in refresh rate (it's no Discord, for sure)
- In total, it's only:
  - Four Controller Functions
  - Two GET routes and two POST routes (not including /about and /contact pages, which are static)
  - Three blades (again, not counting the about and contact blades)
  - Some client side JavaScript (<100 lines)

SimpleChat: Diagram



Questions

- Are there any remaining questions about Lab 2?
- I'm available via email, if you want a quicker response you can also reach me on Discord at the OC Computer Science Discord – there's a channel dedicated to COSC 419
- Again, the due date has been pushed back to Friday

Quiz

- No Talking, no computers
- Should only take about 10-15 minutes
- Make sure that you put your name and student # on your quiz

The Laravel QueryBuilder Querying the DB

Laravel QueryBuilder Overview

- Laravel supports MySQL, Postgres, SQLite, and SQL Server out of the box
- SQLite databases can be quickly configured using the .env environment file, as we did in Lab 1, or you can use the database.php configuration file found in the config folder
- The DB Façade allows us to use the QueryBuilder to query databases in controllers or route functions
- The QueryBuilder also handles the sanitizing of SQL input, preventing SQL injection attacks (we'll discuss these later)

★ Setup and Selecting Tables

- Any controller or route file that uses the DB façade will require a use declaration, placed below the namespace declaration but before the class declaration in a Controller, or just below the php tag in a route file:

```
use Illuminate\Support\Facades\DB;
```

- The table method allows us to specify which table we want to query or update, and is the basis for most of the queries we'll be doing:

```
DB::table('chat')
```

★ The Get Function

- The get() and first() functions are what return actual results in the QueryBuilder, so you'll most likely use one of the two on every query you write
- The get() function will return a collection of objects that represent rows in the table:

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

  - In this case, the get() function is returning the entire contents of the chat table, which is stored in the $rows variable
  - To get individual column values, you can iterate over the $rows and then pull out variables based on the original column name in the table:

```
foreach($rows as $row) {
        array_push($nameArray, $row->name);
}
```

★ The First Function

- While the get() function is useful for returning sets of results (multiple rows) from a query, oftentimes you'll only be interested in a single row
  - For example, you might want to return the maximum value for a given column in the table, without caring about duplicate values
- Get()'s behaviour means that even if a query returns only a single row (or even no rows at all), Get will still return a collection object, meaning you can't directly pull values out of the result without first getting to the objects within the collection
- The First() function, on the other hand, returns the first row of the query and returns it as an object, not a collection, meaning you can instantly interface with it:

```
$row = DB::table('chat')->first();
            Return $row->name;
```

  - In this case, it just grabs the first row of the 'chat' table and returns the value of the 'name' column for that row

★ Additional Functions and Clauses

- In between our table method and our get() or first() function, we can place additional clauses that constrain or expand our data selection and sort our data:

```
      $rows = DB::table(‘chat’)->…..Additional Clauses…..->get();
```

- These functions represent many of the classic database functions that you see in SQL, including, but not limited to:
  - Select (which columns do you want to return?)
  - Where (Return only rows where some condition is true)
  - Join (Combine two or more tables together)
  - Ordering (Sort result rows based on a column value)

★ The Select Function

- The select() function allows us to specify which columns we want to retrieve from the database.

```
$rows = DB::table(‘chat’)->select(‘message’, ‘name’)->get();
```

- The select function takes one or more string parameters, with the strings being the column names as defined within the database table
- Select also allows for aliasing column names in the output results using “as”, shown below:

```
$rows = DB::table(‘chat’)->select(‘name as user’)->get();
```

- There is also a raw version of the function, which allows raw SQL input. This should be used with caution, as it can allow SQL injection:

```
$rows = DB::table(‘chat’)->selectRaw(‘count(?) as total’, [‘name’])->get();
```

★ The Where Clause

- The where() clause allows us to place conditions on which rows we want to return:

```
$rows = DB::table(‘chat’)->where(‘name’, ‘=‘, ‘Bob’)->get();
```

- Where clauses take three parameters: the first parameter is a string representing the column name, the second is the comparator (‘>’, ‘<‘, ‘=‘, ‘like’, ‘!=‘, etc), and the third is the conditional value (what you’re comparing the column against).
- Multiple where clauses can be stacked to produce where AND where style logic:

```
$rows = DB::table(‘chat’)->where(‘name’, ‘=‘, ‘Bob’)->where(‘id’, ‘>’, 50)->get();
```

- If the comparator is ‘=‘, the comparator can be omitted for simplicity:

```
$rows = DB::table(‘chat’)->where(‘id’, 50)->get();
```

★ Other Where Clauses

- While stacking where() functions leads to where AND where behaviour, there is a separate function for defining where OR where behaviour:

```
$row = DB::table(‘chat’)->where(‘id’, 50)->orWhere(‘name’, ‘Bob’)->get()
```

- Like the select function, there is also a raw version of the where function. Like the select function, it is also potentially dangerous:

```
$row = DB::table(‘chat’)->whereRaw(‘id > ? OR id = ?’, [200, 1])->get();
```

- There are also a variety of additional where functions, some of which may come in handy. For example, whereBetween:

```
$row = DB::table(‘chat’)->whereBetween(‘id’, [10, 20])->get();
```

  - whereBetween() takes a column name and an array of two values, and returns all rows where the column value falls between them.
  - whereNotBetween() is identical in syntax, but selects rows where the column value is outside the range of the array values
- whereDate() is another handy function, taking a column name and a formatted date (i.e. ‘2018-09-24’) and selecting rows where the column has an equivalent date.

A Quick Exercise

What would be returned with the following QueryBuilder query?

```
DB::table('users')->select('username')->where('postCount', '>', 0)
->where('Id', '!=', 1)->get();
```

Id	username	postCount
1	admin	1032
2	frit	7
3	zer0	81
4	pikachu99	255
5	j_doe	0
6	Michael	928
7	frodo	304

A Quick Exercise

What would be returned with the following QueryBuilder query?

```
DB::table('users')->select('username')->where('postCount', '>', 0)
->where('Id', '!=', 1)->get();
```

Id	username	postCount
1	admin	1032
2	frit	7
3	zer0	81
4	pikachu99	255
5	j_doe	0
6	Michael	928
7	frodo	304

★ Joins

- The most likely join you'll be wanting to perform is an inner join, which outputs the overlap between two tables on a given column
- In QueryBuilder syntax, the inner join function looks like this:

```
DB::table('table1')->join('table2', 'table1.id', '=', 'table2.id')
```

- The first parameter in the join() is a string defining the name of the table you want to join on.
- The second parameter is a column name from the original table
- Third parameter is a comparator
- Fourth parameter is a column name from the table you're joining on

★ Left Joins

- One of the other most common joins to perform is a left join, which keeps the entirety of one table, and adding the overlapping results of the second table
- In QueryBuilder, this can be performed with the leftJoin() function, which maintains the same syntax as the join() function:

```
DB::table(‘table1’)->leftJoin(‘table2’, ‘table1.id’, ‘=’, ‘table2.id’)
```

- In the above example, all rows of table1 will be preserved, while only rows from table2 that have a match on the ‘id’ column will be appended
- Joins (of any type) are placed in the QueryBuilder ‘chain’ just like other clauses:

```
$row = DB::table(‘t1’)->join(‘t2’, ‘t1.id’, ‘=’, ‘t2.id’)
                      ->select(‘user’)->get();
```

Join Example

T1

id	username	postCount
1	admin	1032
2	frit	7
3	zer0	81

T2

id	post	date
1	Hello	10-21-2017
3	Hi	10-21-2017
4	Bonjour	10-22-2017

```
$rows = DB::table(‘T1’)->join(‘T2’, ‘T1.id’, ‘=‘, ‘T2.id’)->get();
```

id	username	postCount	post	date
1	admin	1032	Hello	10-21-2017
3	zer0	81	Hi	10-21-2017

★ Ordering Results

- Often, we'll want to sort our results by some metric. Laravel provides the orderBy() function, which is identical to the SQL equivalent:

  ```
  $row = DB::table('chat')->orderBy('id', 'desc')->get();
  ```

  - The first argument in the orderBy() function is the column name you want to sort by. The second argument is either 'asc' (ascending) or 'desc' (descending), which sets the sort order
- You usually want to place the orderBy() clause near the end of the chain, just before you perform your get() or first()
- Want to order by date easily? The latest() and oldest() functions will automatically sort in date order if your table contains a date column called 'created_at'
  - You may also pass a column name to latest() and oldest() if your date column is named differently:

  ```
  $rows = DB::table('chat')->latest('timestamp')->get();
  ```

Ordering Results (Continued)

- The orderBy() function can be stacked to sort on multiple columns in order of priority:

```
$row = DB::table(‘chat’)->orderBy(‘name’, ‘asc’)->orderBy(‘id’, ‘desc’)->get();
```

- The above is effectively the same as the SQL declaration:

```
SELECT * from chat ORDER BY ‘name’ asc, ‘id’ desc
```

- Another function provided by Laravel is inRandomOrder(), which takes no parameters and simply shuffles the rows in the results
  - From my personal experience, don’t expect too much randomness from the inRandomOrder() function.

★ Groups, Offsets, and Limits

- The groupBy() and having() functions can be used to group results on a column based on a condition

```
$row = DB::table('chat')->groupBy('timestamp')->having('id', '>', '50')->get();
```

- The groupBy() function takes a string column name to group on, and the having() function is identical in signature to the where() function – column name, comparator, and compared value.
- You'll usually want to use these with aggregate functions such as count, min, and avg.
- You can also choose how many records you want to return in a query. The limit() function takes an integer input, and will return that many records:

```
$row = DB::table('chat')->orderBy('id', 'asc')->limit(10)->get();
```

  - In the SimpleChat database, this would return the 10 newest messages
- If you want to skip a number of records from the beginning of the results, you can use the offset() function, which also takes an integer:

```
$row = DB::table('chat')->offset(20)->get();
```

  - This would ignore the first 20 rows in the table, and return the rest.

The Laravel QueryBuilder Modifying Records

★ Modifying the Database - Insertions

- Up until now, the QueryBuilder functions we've looked are strictly for performing queries – they don't change records in the database
- Luckily, the QueryBuilder makes it fairly simple to perform insertions, updates, and deletions on our databases
- Inserting a record into the database is as simple as passing an array of values to the insert() function:

```
        DB::table('chat')->insert(['message' => 'Hello!', 'timestamp' =>
time()]);
```

- The associative array that we pass to the insert() function should include all the columns that are considered **required** in our table, that is, they cannot contain null values
  - The associative array is surrounded by square brackets [], and has key-value pairs, with the key being the column name ('message'), and the value being what we want to insert into that column ('Hello!')

★ Insertions with Auto-Increment

- You may notice that in this code, we don’t return anything from the insertion:

```
DB::table(‘chat’)->insert([‘message’ => ‘Hello!’, ‘timestamp’ => time()]);
```

- This poses a bit of a problem. Say you have an auto-incrementing ID field in your table – how do you get the ID of the row you just inserted? You could query the table to get the newest ID, but that would be inefficient and might not get the correct row
- Solution: the insertGetId() function returns an int that is the ‘id’ of the inserted column. If your auto-increment field is named something other than ‘id’, you can pass the column name as a second parameter in the function. Otherwise, it is used exactly like the insert() function

```
            $rowId = DB::table(‘chat’)->insertGetId([‘message’ =>
‘Hello!’,           ‘timestamp’ => time()]);
```

  - The $rowId variable in this example now has the auto-incremented ‘id’ value of the inserted row

★ Updating Records

- You can easily update records using the where() clauses that we previously covered. Simply identify the row(s) that you want to update using the where() clauses, and then use the update() function:

```
DB::table(‘chat’)->where(‘timestamp’, ’10-21-2018’)->update([‘message’ => ‘REDACTED’]);
```

- In this case, any records with a matching timestamp of ‘10-21-2018’ in the ‘chat’ database would have their message value overwritten with ‘REDACTED’
- The update function takes an associative array of key-value pairs as input, similar to the insertion function
- Just like performing a query, you can use multiple where(), orWhere(), whereBetween(), etc functions to identify the rows that you want to update

★ Deleting Records

- Deleting records is very similar to updating them: use where functions to define which rows you want to delete, and then call the delete function. The delete function takes no parameters:

```
DB::table(‘chat’)->where(‘timestamp’, ’10-21-2018’)->delete();
```

  - This would delete any records in the chat table that have a timestamp of ‘10-21-2018’
- If you want to completely empty out a table, you can use the truncate() function, which will delete **all** rows:

```
DB::table(‘chat’)->truncate();
```

Laravel Sessions

★ HTTP Sessions

- A session allows us to identify and store information pertaining to a user between HTTP requests. This is because the HTTP protocol is stateless – information is not kept between client-server transactions
- Recall that in the SimpleChat application, refreshing the page causes the user to be assigned a new username and color – this is because we have no way of identifying individual users across two HTTP requests
- A session allows us to store information across multiple HTTP requests without having to store the information on the user's system and re-send it
  - This also avoids the problem of a user modifying data stored on the client side – for example, the bonus mark in Lab 2

★ The Sessions Driver

- PHP itself supports the use of session variables; Laravel extends this implementation to make them easier to use
- In the session.php configuration file, located in the config directory, you can configure how Laravel stores session data. The options are:
  - File : session data is stored within files in the storage directory
  - Cookie : session data is stored client-side as an encrypted cookie
  - Database : session data is stored within a specified database
  - Memcache/Redis : specialized fast caching solution
- By default, Laravel uses the file option; this is fine for most applications

★ Session Variables and Helper

- There are several ways to access session variables, but we'll be using the *Global Session Helper*. The alternative would be to access the session variables via a Request object
- The Global Session Helper doesn't require any explicit **use** statement and is available in both Controllers and Route
- The Global Session Helper is simply a function called session(). We can call it with additional functions to retrieve, store, and delete data from the session
- Data is stored in the session as key-value pairs, with named keys referring to a specific value
- If you try to retrieve a key that doesn't exist in the session, it will return a NULL value

★ Accessing Session Variables

- Accessing data from the session is easy! To get the value associated with a specific key, simply call session(<key>):

```
$data = session(‘user’);
```

  - In this case, the $data variable will be equal to whatever value is associate with the key ‘user’ in the session – although if ‘user’ isn’t set yet, $data will = NULL

- You could use the PHP is_null function to test if a key exists in the session, but there’s an easier way – you can use the has() function:

```
if(session()->has(‘user’)){
        //Do something
}
```

- You can also get all the data for the session using the all() function:

```
$data = session()→all();
```

★ Writing Session Variables

- To write to a session variable, we can use the put() function:

```
session()->put('user', 'frit');
```

  - In this case, we'd be writing the value 'frit' to the session, with a key of 'user'
- Alternatively, you can write to the session using an associative array:

```
session(['user' => 'frit']);
```

- There may also be times that you want to write data to the session, but only for the next HTTP request that is called. Laravel allows you to do this using the flash() function:

```
session()->flash('user', 'frit');
```

  - In this case, the user:frit key value pair would be available when we try to read it in the next HTTP request the user makes, but would then be deleted
  - Flash is good for data that you need to persist between pages, but you don't want to persist permanently (for example, information about the previous page (back button), or a status message to display on the new page)

★ Deleting Session Variables

- To delete session variables on a key-by-key basis, you can use the forget() function:

```
session()->forget('user');
```

- If you want to wipe the entire session clean (delete **all** session variables, you can use the flush() command:

```
session()->flush();
```

- Sometimes, you may need to delete get a session variable for something and then delete it from the session. Laravel provides the pull() method, which will return the value and delete it from the session:

```
$username = session()->pull('user', 'default value');
```

  - In this case, $username will contain the value of the 'user' session variable, which will be deleted. If the 'user' variable doesn't exist, $username will instead hold 'default value'

A Word about Sessions

- For the most part, you don’t need to manually start or initialize a session – this is done automatically by the Laravel session middleware
- Recall from the previous lecture, however, that I noted that some of the differences between web routes and API routes in Laravel is the middleware – including the session middleware
- As a result, session won’t work properly when called via API routes – session variables, if assigned, won’t persist across pages, and sessions will return empty (unless they’ve been previously written while processing that page)
- However, sessions do persist across web routes
- Session data is kept for an adjustable lifetime, which can be set in the .env or in the session.php config files. By default, Laravel keeps session data for 120 minutes (2 hours)

Sessions in Lab 3

- In Lab 3, we'll be adding session variables to our SimpleChat application
- Specifically, we'll make username and color persistent, so that if a user refreshes the chat room, or closes the tab and comes back later, their username and color will remain the same
- To do this, we'll be checking to see if the username and color session variables have been set, and only randomly picking a color if they're not – otherwise, we'll read the username and color from the session variable
- We'll also need to refresh the session variables when a message is received, to ensure that a name change is reflected in the session variable

Laravel Validation

★ Validating Input

- It’s often a good idea to first validate user input, before we try doing anything with it. This can avoid many potential issues:
  - An input is too long and won’t fit in the database
  - An input is malformed
  - An input is missing that is required for part of the back-end logic
- For this reason, Laravel includes a system of input validation that allows us to validate input and will automatically redirect users to the previous page and return error messages if something is wrong
- This shouldn’t be confused with HTML5 form validation, which is performed on the client-side. Client-side validation is good for ensuring that forms are filled out, but won’t prevent someone from making a POST request that doesn’t meet the validation requirements

★ Validation Function

- Validation is done within the controller, by performing the validate() function on a Request object:

```
$goodData = $request->validate([‘name’ => ‘required’]);
```

- If the validation succeeds, the validation function will return the validated data. If it fails, it will automatically redirect and pass the error messages as a variable
- The input parameter for the validate() function is an associative array of validation rules
    - These rules can include whether a field is required, maximum length, type of input (email, IP address, etc), or whether a field is unique within a database

★ Validation Rules

- Laravel supports many different validation rules, and they can be combined to validate multiple characteristics of a different field
- Some rules that are commonly used:
    - ‘required’ – Notifies that this field must exist, not be empty, and not be null
    - ‘max:<int>’ – Notifies that the maximum length of the field is <int> characters
    - ‘email’ – Notifies that the field must be formatted as an email address
    - ‘accepted’ – Notifies the field must have a value of yes, on, 1, or true
    - ‘integer’ – Notifies the field must be an integer value
    - ‘alpha_num’ – The field must only contain alpha-numeric characters

★ Validation Rules (Continued)

- Rules can be combined together using the pipe ‘|’ character
- For example, the following validation line requires that an input me alpha-numeric, with a maximum length of 255 characters, and it must be filled out:

```
[‘name’ => ‘required|max:255|alpha_num’]
```

- When validation is performed, it will continue, even if a validation rule is broken. It will then return **all** errors in validation that were detected. If instead you only want to validate until the first error is encountered (and subsequently, only return the first error), you can use the ‘bail’ rule:

```
[‘name’ => ‘bail|required|max:255|alpha_num’]
```

Example Validation Function

```
Public function createUser(Request $request){
        $goodData = $request->validate([
                ‘username’ => ‘required|alpha_num|max:255’,
                ‘email’ => ‘required|email’,
                ‘terms’ => ‘required|accepted’]);
        //Continue on with backend logic using $goodData
        return view(‘welome’);
}
```

★ When Things are Wrong

- As previously mentioned, when validation fails, the user is redirected to the previous page
- Laravel automatically uses flash() to store the error messages in the session
- Laravel also automatically pushes error messages from session variables to the view (blade) template via the $errors variable
- Recall that in blades, we can check to see if a variable exists, and if so change the content of the page. Using this in combination with a foreach loop allows us to print out error messages as HTML to the page, notifying the user of why their submission failed
- Because this relies on the session variables and a special middleware, ShareErrorsFromSession, validation only works by default on web routes, not API routes – although by specifying middleware, you can use validation for API routes

An Example of Echoing Errors

```
<html>
<head>
…
</head>
<body>
@if ($errors->any())
        @foreach ($errors->all() as $error)
                <p class="error-msg">{{ $error }}</p>
        @endforeach
@endif
…
…
</body>
</html>
```

Validation and Lab 3

- This is just a basic overview of validation – there are many more validation rules and things that can be done with validation in Laravel
- We'll talk more about validation as part of Lab 3, where we'll be providing some validation on messages passed through the SimpleChat app
- Key takeaway: define validation rules and pass them to your validate() function; if something goes wrong, make sure you have something in place to echo those errors out of the $errors variable on your form page
    - Nothing is more frustrating than being redirected back to the form, but not being told *why*!

That’s All Folks