IPL
escola superior
de tecnologia e gestão
instituto politécnico
de leiria

Laravel

Routes, Controllers, Request, Validation, Sessions

Marco Monteiro

Contributors

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

Summary

1. Routes
2. Controllers
3. Request
4. Validation
5. Sessions

1 – ROUTES

Routes

- Routes registration file for Web Applications:

`routes/web.php`

All Web Application routes are defined (registered) on this file, which is automatically loaded by the framework

Other types of application's routes can be defined on other files (e.g. "routes/api.php" for stateless API routes)

Routes

- Routes can be handled by a Closure function:

```
Route::get('/test', function () {
  return view('pages.index');
});
```

- or by a method of a controller (**action**).
  ***This is the preferred alternative*:**

```
Route::get('/test',
            [Controller::class, 'action']);
```

Http method

URL Pattern

URL Patterns can have parameters:

```
Route::get('/users/{id}', '…');
```

Route Methods

- Routes that correspond to HTTP Methods (or HTTP Verbs):

```
Route::get(…);
Route::post(…);
Route::put(…);
Route::patch(…);
Route::delete(…);
Route::options(…);
```

- Other routes:

```
Route::redirect('/here', '/there');
Route::view('url', 'view.name');
```

  - Both **redirect** and **view** use the HTTP GET method

Routes

- Which HTTP Method to choose?

HTTP Method (Route Method)	Use it for:
`get`	**Show** data / forms When data (usually on the database) is not changed
`post`	To **create** new data (insert on database) – when user submits data from a create form For **login** form or any other context that requires a form or business operation that is not covered by other HTTP methods
`put`	To **modify** data (update on database) – when user submits data from an update form. *E.g. to update all user data*
`patch`	To **modify partial** data – the same as put, but when updating only a part of the resource. *E.g. to activate/deactivate a user; to change user's password*
`delete`	To **delete** data (delete or *softdelete* on database)

Named Routes

- Named routes allow the convenient generation of URLs or redirects for specific routes.

```
Route::get('photos', …)->name('photos.index');
Route::get('photos/{photo}', …)->name('photos.show');
```

- Generating URLs to named routes (examples):

```
$url = route('photos.index');
```

```
return redirect()->route('photos.index');
```

```
<form action="{{ route('photos.index') }}" …>
```

- Generating URLs to named routes with parameters (examples):

```
$url = route('photos.show', ['photo' => 'value']);
```

```
return redirect()-> route('photos.show',
                          ['photo' => 'value']);
```

Route Parameters – Order of Routes

- **Order** of route definition is **important**

```
Route::get('cars/{car}', [Ctr::class, 'index']);
Route::get('cars/my', [Ctr::class, 'mycar']);
```

- "mycar" method would never be executed. Why?
  - The URL "cars/my" – would execute **index** method, with parameter {**car**} value equal to "my"
- To solve this, just change the order or route definition

```
Route::get('cars/my', [Ctr::class, 'mycar']);
Route::get('cars/{car}', [Ctr::class, 'index']);
```

  - *Use "artisan route:list" command to check the order*

Route Model Binding

- Some route parameters may represent a specific model instance

```
Route::get('products/{product}', [Ctr::class, 'show']);
```

  - URL example: `http://meu.site/products/365`

- Typical controller code starts by loading the model from DB:

```
public function show($id)  {
    $prod = Product::findOrFail($id);
    return view('x')->with('product', $prod);
}
```

However, if we specify the **model class** type on the controller function argument and use the **same name** as the route parameter, then Laravel injects the Model instance (already loaded) directly to the controller function.

```
public function show(Product $product)  {
  return view('x') ->with('product', $product);
}
```

Route Middleware

- It is possible to attach middlewares to routes
- Example: attaching the middleware "**auth**" to a route, makes it only accessible to authenticated users.

```
Route::get('accounts', [Ctr::class, 'show'])
                        ->middleware('auth');
```

- Laravel already includes "auth" middleware (among several others), but it is possible to add our own custom middleware and attach it to any route
- Middleware code is executed automatically before the route handler (action method of the controller).
- With "auth" middleware, the code checks if the user is authenticated. If user in not authenticated, the execution is interrupted (controller method is not executed) and the user is redirected to the login page

Route Groups

- Route groups allow you to share route attributes, such as middleware or namespaces, across a large number of routes without needing to define those attributes on each individual route.

- **Route::group** method creates a group of routes
- It is possible to create **Nested** Route Groups
  - Using the Route::group method inside a group closure function

Route Groups - Middleware


```
Route::middleware(['auth', 'b'])->group(function () {
   Route::get('users',          [Ctr::class, 'index']);
   Route::get('users/profile', [Ctr::class, 'profile']);
   Route::get('users/{user}',  [Ctr::class, 'edit']);
   Route::middleware('c')->group(function () {
        Route::get('photos/{user}',  [Ctr::class, photos']);
   });
});
```

- Attaches the middlewares '`auth`' and '`b`' to all 4 routes inside the *"main"* group (including the route inside the nested group).

- Attaches the middleware '`b`' only to the route of the nested group (photos/{user}

Route Groups – Prefix

- **prefix** method adds a prefix to all URLs in the group

```
Route::prefix('admin')->group(function () {
    Route::get('/users',          [Ctr::class, 'index']);
    Route::get('/users/profile', [Ctr::class, 'profile']);
    Route::get('/users/{user}',  [Ctr::class, 'edit']);
});
```

- The following URLs will be considered valid:

  /admin/users

  /admin/users/profile

  /admin/users/123

Route Groups – Route Name Prefix

- **name** adds a prefix to all route names in the group

```
Route::name('admin.')->group(function () {
    Route::get('/users',          [Ctr::class, 'index'])
        ->name('users.index');
    Route::get('/users/profile', [Ctr::class, 'profile']);
        ->name('users.profile');
    Route::get('/users/{user}',  [Ctr::class, 'edit']);
        ->name('users.edit');
});
```

- Route names for the routes in the group (this just affects the route names, not the route URLs:

admin.users.index

admin.users.profile

admin.users.edit

Route Groups – Multiple Attributes

- A group can be associated with multiple attributes. Example:

```
Route::middleware('auth')
      ->prefix('admin')
      ->name('admin.')
      ->group(function () {
   . . .
   Route::get('/users',          [Ctr::class, 'index'])
      ->name('users.index');
   Route::get('/users/profile', [Ctr::class, 'profile']);
      ->name('users.profile');
   Route::get('/users/{user}',  [Ctr::class, 'edit']);
      ->name('users.edit');
   . . .
});
```

Route Class Facade

- To access the current route use the **Route** class facade:

```
$route  = Route::current();
$name   = Route::currentRouteName();
$action = Route::currentRouteAction();
```

- Usage example:

```
<… @class([
 'bg-blue' => Route::currentRouteName() ==
                'products.index'
])
```

2 – CONTROLLERS

Controllers

- Controllers extends the abstract controller class included with Laravel: `App\Http\Controllers\Controller`

```
namespace App\Http\Controllers;

use App\Models\User;

class UserController extends Controller
{
    public function show($user) {…}
}
```

action

Controller's method that "handles" a route

Resourceful controllers

- A Resourceful Controller handles all **typical routes** for **CRUD** (Create, Read,Update, Delete) operations on a resource (e.g., for resource "photos"):

Verb	URI	Action	Route Name
GET	/photos	index	photos.index
GET	/photos/create	create	photos.create
POST	/photos	store	photos.store
GET	/photos/{photo}	show	photos.show
GET	/photos/{photo}/edit	edit	photos.edit
PUT/PATCH	/photos/{photo}	update	photos.update
DELETE	/photos/{photo}	destroy	photos.destroy

Always try to follow the conventions used by a Resourceful controller.

Resourceful controllers

- To create a resourceful controller (with all methods pre-built):

```
php artisan make:controller PhotoController --resource
```

- To create **all** routes, just add one line to the routes file:

```
Route::resource('photos', PhotoController::class);
```

- To create **partial** resource routes (a subset of the routes):

```
Route::resource('photos', PhotoController::class)
                ->only(['index', 'show']);
```

```
Route::resource('photos', PhotoController::class)
                ->except(['create', 'store',
                          'update', 'destroy'
                         ]);
```

Supplementing Resource Controllers

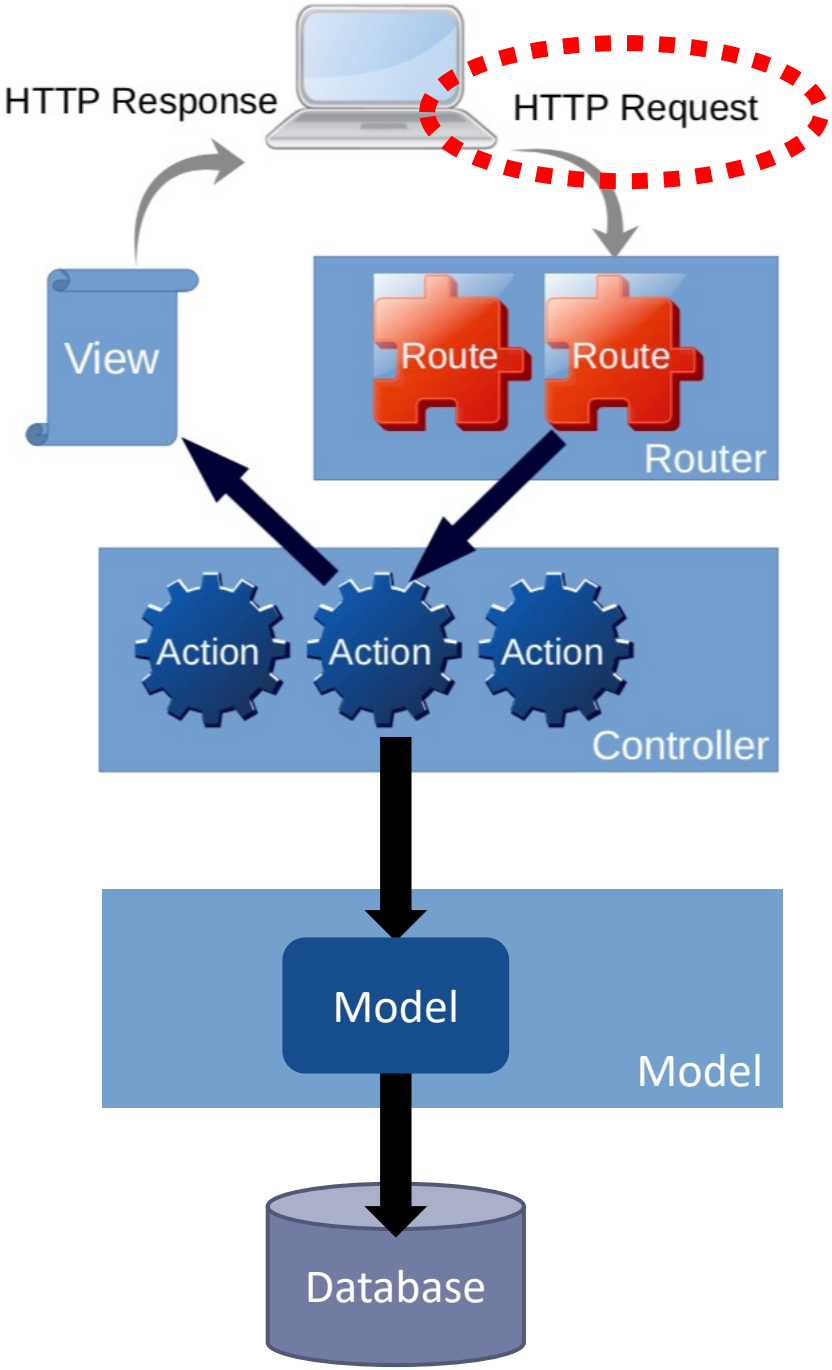
- To add additional routes to a resource controller, define these additional routes **before** the Route::resource(…)
- Why?
  - The order of the routes is important.
  - If Route::resource(…) is called first, the routes defined by the resource method may unintentionally take precedence over your supplemental routes:

```
Route::get('photos/popular',
            [PhotoController::class, 'getPopular']);
Route::resource('photos', PhotoController::class);
```

3 – REQUEST

HTTP Requests

- All incoming HTTP Requests pass through the routing mechanism and are passed on to the controller
- Laravel creates an instance of a Request class (`Illuminate\Http\Request`) with information about the **current HTTP Request**, namely: URL query string parameters, form field values, uploaded files, etc…
- Good practice of the MVC pattern: use the Request instance on the **controller only**.



Access HTTP Request

- To access the HTTP Request, we can use dependency injection or request() helper function to get the **Request instance**

```
// with Dependency Injection
public function store(Request $request)
{
    $inputName= $request->input('name');
}
```

  - *When an action method of the controller has an argument with a "Request" type, Laravel injects the current request object in that argument*

```
// with Helper function
$inputName= request()->input('name');
```

- Request instance gives us access all information about the current request:

```
$path = $request->path();
$fullUrl = $request->fullUrl();
$method = $request->method();
// check https://laravel.com/docs/requests for more
```

Request – Input Values

- User sends information to the server through field values of the HTTP Request
  - **POST** (PUT/PATCH) message – values in the message body (**payload**)
  - **GET** message – values in the **query string** of the URL
- Field values are accessible through the **input** of the request
  - Retrieve all input data (as an array):

```
$inputArray = $request->all();
$inputArray = $request->input();
```

  - Retrieve one input (field) value

```
$name = $request->input('name');
```

  - Retrieve one input value with a default value

```
$name = $request->input('name', 'John');
```

- **input** includes both the POST (payload) & GET (query string) values
  - POST is considered first

Request – Input Values

- It is possible to access individual fields using dynamic properties:

```
$name = $request->name;
$type = $request->type ?? 'standard';
```

- To access only Get (query string) values:

```
$name = $request->query('name');
$type = $request->query('type', 'standard');
// All query string "parameters" as an associative array
$arrQueryStrings = $request->query();
```

- For array fields, use dot notation:

```
$name = $request->input('products.0.name');
$allNames =  $request->input('products.*.name');
```

- On the form (HTML):

```
<input type="text" name="products[0][name]">
<input type="text" name="products[1][name]">
<input type="text" name="products[2][name]">
```

4 – VALIDATION

Validation

- Validation guarantees that **HTTP request** data is correct
- Validation on the server
  - Executed when the HTTP request arrives on the server.
  - Validation on the server is fundamental, as it increases the application overall security (ensures that only "clean" data is considered)
  - Laravel handles validation on the server.
- Validation on the client
  - Executed before the HTTP request is submitted by the browser – uses JavaScript or HTML input properties (e.g. required)
  - Validation on the client improves user experience and application performance, as no HTTP request is submitted while data in not valid
  - Validation on the client does not increase application overall security, because any hacker can easily bypass validation on the client
  - Laravel does not implement validation on the client
- **Always** implement **validation on the server** (+ secure). Validation on the client is not mandatory, but can improve user experience and application performance

Laravel Validation Rules

- Laravel validation is based on **rules**
  - "**string**" syntax

```
'age'   => 'required|integer|between:1,120',
```

  - "**array**" syntax

```
'age'   => [ 'required',
             'integer',
             'between:1,120' ]
```

Rule arguments

  - "**Rule**" class

```
'nif'   => [
     'required', … ,
     Rule::unique('clients')->ignore($this->id)
]
```

  - "**callback**" function

```
'erasmusstudent' => ['nullable', 'in:0,1',
  function ($attribute, $value, $fail) {
    if (($this->specialstudent == "1") &&
      ($value == "1")) {
        $fail('Erasmus student cannot also
                  be a special student.');
    } } ]
```

Validation Rules

- Available validation rules:

https://laravel.com/docs/validation#available-validation-rules

- We can create our own validation rules

https://laravel.com/docs/validation#custom-validation-rules

Accepted
Accepted If
Active URL
After (Date)
After Or Equal (Date)
Alpha
Alpha Dash
Alpha Numeric
Array
Ascii
Bail
Before (Date)
Before Or Equal (Date)
Between
Boolean
Confirmed
Current Password
Date
Date Equals
Date Format
Decimal
Declined
Declined If
Different
Digits
Digits Between
Dimensions (Image Files)
Distinct
Doesnt Start With
Doesnt End With
Email
Ends With
Enum
Exclude
Exclude If
Exclude Unless
Exclude With
Exclude Without
Exists (Database)
Extensions
File
Filled
Greater Than
Greater Than Or Equal
Hex Color
Image (File)
In
In Array
Integer
IP Address
JSON
Less Than
Less Than Or Equal
List
Lowercase
MAC Address
Max
Max Digits
MIME Types
MIME Type By File Extens...
Min
Min Digits
Missing
Missing If
Missing Unless
Missing With
Missing With All
Multiple Of
Not In
Not Regex
Nullable
Numeric
Present
Present If
Present Unless
Present With
Present With All
Prohibited
Prohibited If
Prohibited Unless
Prohibits
Regular Expression
Required
Required If
Required If Accepted
Required If Declined
Required Unless
Required With
Required With All
Required Without
Required Without All
Required Array Keys
Same
Size
Sometimes
Starts With
String
Timezone
Unique (Database)
Uppercase
URL
ULID
UUID

Request Validation

- Example of validation in a controller action
  - Validation should be the first thing to execute.
  - It only makes sense to execute the remaining code of the controller, when all request data is correct.
- Uses method **validate** of the request object

```
public function store(Request $request) {
  $disciplina = $request->validate([
    'ano' =>'required|integer|between:1,3',
    'semestre' => 'required|in:1,2',
  ],
  [         // Custom Messages
     'ano.required' => 'Ano é obrigatório',
  ]);
  // Only executes next line if all data is valid …
  Disciplina::create($disciplina);
}
```

Laravel Validation – how does it work?

- When all values are considered **valid**:
  - Validate method returns an array with all valid values
    - It only includes values that have validation rules
    - If the form being validated has other fields, these values are not passed on to the array of valid values.
  - Remaining code is executed
    - If everything is OK, we continue the planned operation

Laravel Validation – how does it work?

- When at least one value is considered **invalid**:
  - **Execution** of the controller is **interrupted**
    - Remaining code is NOT executed
    - If something is wrong, it does not make sense to continue the planned operation
  - **Automatically** redirects to the previous location
    - Previous location is usually the form we are validating
  - When redirecting, all the input values and validation errors messages will be flashed to the session
    - Input values to fill the **old** field values - function old()
    - Validation error messages - **$errors** variable on the view

Error Messages

- **$errors** (MessageBag) support multiple error message per field.

```
Illuminate\Support\ViewErrorBag {#398 ▼
  #bags: array:1 [▼
    "default" => Illumin…\MessageBag {#399 ▼
      #messages: array:2 [▼
        "ano" => array:1 [▼
          0 => "Ano é obrigatório"
        ]
        "semestre" => array:1 [▼
          0 => "The semestre field is required."
        ]
      ]
      #format: ":message"
    }
  ]
}
```

Error Messages on Blade

- **@error** directive checks if validation error messages exist for a given field
- Within an @error directive, you have access to the **$message** variable with given field error message (the first error message)

```
@error('year')
  <div class="alert alert-danger">
     {{ $message }}
  </div>
@enderror
```

Form Requests

- Form requests are **custom request classes** that contain validation and authorization logic
  - The controller is cleaner – all validation logic will be centralized on the Form Request.
  - Validation logic can be reused on several controller actions (methods)
  - Form request extends from the Request class

    A form request represents a type of HTTP request that has a specific format (a set of input fields that must follow a set of validation rules)

- **Preferred alternative** for validation with Laravel

Validation with Form Requests

- Instead of this:

```
public function store(Request $request, $id)  {
  $newProd = $request->validate([
     'name' => 'required|regex:/^[\pL\s]+$/u',
     'discount'=> 'required|integer|between:0,50',
  ]);
  Product::create($newProd);
  . . .
```

- We have this:

Form Request Class:

```
public function store(StoreProduct $request)  {
  Product::create($request->validated());
  . . .
```

Validation with Form Requests

- To create a form request class:

```
php artisan make:request ProductPost
```

- The generated class will be placed in the `app/Http/Requests` folder
- Validation rules and messages will be located on the Form Request class (instead of the controller)

Validation with Form Requests

- Form requests includes a **rules** method that returns an array with validation rules:

```
class ProductPost extends FormRequest
{
   . . .
   public function rules()
   {
      return [
         'name' => 'required|regex:/^[\pL\s]+$/u',
         'discount'=> 'required|integer|between:0,50',
      ];
   }
   . . .
}
```

Validation with Form Requests

- Form requests may include a **messages** method that returns an array with custom validation messages:

```
class ProductPost extends FormRequest
{
 . . .
  public function messages()
  {
    return [
      'name.required'     => 'Name is required',
      'discount.required'=> 'Discount is required',
    ];
  }
  . . .
}
```

Validation with Form Requests

- Form requests may include a **attributes** method – returns an array with custom attributes display name
  - Changes the name of the fields on the default error messages

```
class ProductPost extends FormRequest
{
 . . .
  public function attributes()
  {
    return [
      'name'      => 'Nome',
      'discount' => 'Desconto',
    ];
  }
  . . .
}
```

Form Request - Authorize

- The form request class also contains an **authorize** method.
- Within this method, you may check if the authenticated user has the authority to make the HTTP request

```
class ProductPost extends FormRequest
{
    . . .
    public function authorize()
    {
        // Only "admin" usar can store the product
        return $this->user()->admin;
    }
    . . .
}
```

- By default, this method returns false - **if we don't change it, all requests will be considered as forbidden**

Validation with Form Request

- Using **Form requests** on the controller to validate:
  - Type-hint the request on your controller method
  - Use validated() method of the request to obtain all valid values

```
public function store(ProductPost $request)
{
    $validatedData = $request->validated();
}
```

- Same behavior as before:
  - If everything is OK, valid data from the request is retrieved by $request->validated()
  - If there is any error, Laravel will not enter the controller method (action). It automatically redirect to previous location and flash the error messages and inputted values to the redirect location

Form Request - prepareForValidation

- **prepareForValidation** is executed before any validation occurs
- It can be used, for example, to guarantee that an input value has a default value. Great for checkbox or radio inputs

```
class ProductPost extends FormRequest
{
    . . .
    public function prepareForValidation()
    {
        if(!$this->has('active')) {
          $this->merge([
             'active' => '0'
          ]);
        }
    }
```

If the 'active' input field is not present on the request, then the prepareForValidation will add that field (with a default value) to the list of request fields.

Form Request - after

- **after** method returns an array of callables or closures which will be invoked after validation is complete.
- It can be used, for example, to add **extra validation code** that for some reason is not easily implemented with validation rules

```
class ProductPost extends FormRequest {
  . . .
  public function after() : array {
    return [
      function (Validator $validator) {
        // customValidationCode would be a custom function that
        // returns false is the salary is invalid
        if (!customValidationCode($this->salary)) {
          $validator->errors()->add('salary', 'error message');
        }
      }
    ];
  }
```

Custom Validation Rules

- To create a new Custom Rule

```
php artisan make:rule Uppercase
```

- This creates a class named UpperCase, that extends from "`Illuminate\Contracts\Validation\Rule`"

- Validation Rules Classes are placed on "`App\Rules`" namespace

Custom Validation Rules

- Class for the Custom Validation Rule

```
namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;

class Uppercase implements Rule
{
    public function passes($attribute, $value)
    {
        return strtoupper($value) === $value;
    }
    public function message()
    {
        return 'The :attribute must be uppercase.';
    }
}
```

Custom Validation Rules

- Use a Custom Validation Rule

```
. . .
'abreviatura'   =>
            [
                'required'
                'string',
                new Uppercase
            ]
```

5 – SESSIONS

Session

- Session: time frame for communication between two systems or two parts of a system.

- On the Web, session refers to a visitor's time browsing a web site. The time between a visitor's first arrival at a page (any page) on the web site and the time they stop using the web site (no page of the web site is accessed during some time – session timeout)

- Session is not the same as login. A new session is created if any page of the site is accessed (even when user is anonymous), not when the user logins.

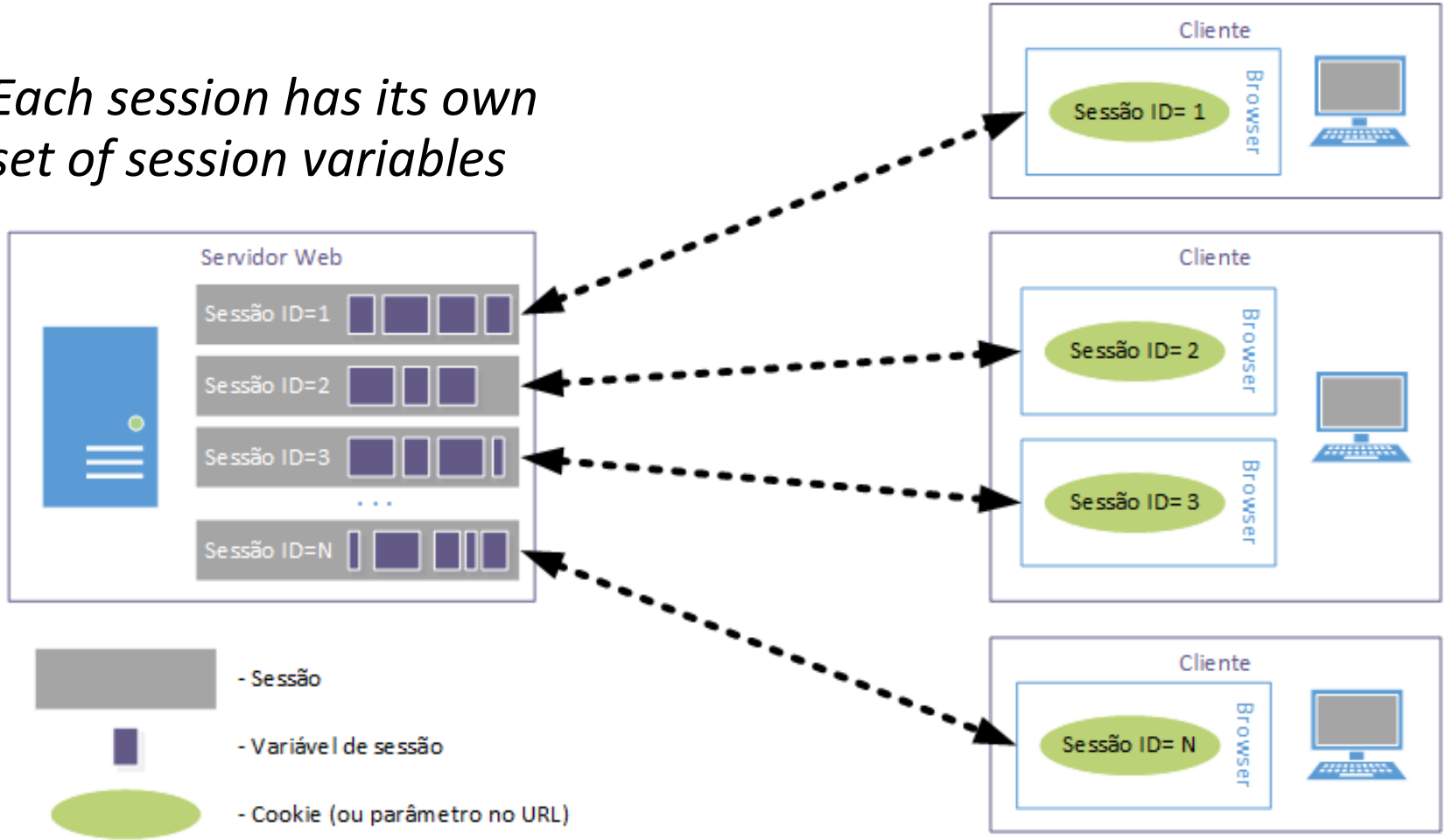
Session State

- Sessions can temporarily store information related to the activities of the user while connected.
  - Examples: Shopping Cart; Visited Products; Values inputted on a form, etc.

- On a typical Web Application:
  - Session state is stored on the server: Memory (non-persistent); Database; File; etc.
  - Each session is identified by a **session ID**. This session ID is passed on from the server to the client (and vice-versa) as a cookie
    - Instead of a cookie, session id can be passed on the URL (query string), headers or hidden field.

Session State on the Web

Each session has its own set of session variables



Session ID

- Since HTTP is a stateless protocol, there is no built-in way of maintaining state between two transactions (request/response pair), so a cookie is used to identify the session
- When the session is created (first page of the application is accessed), the Web server sends the "Session ID" to the client as a cookie
  - On Laravel, the session ID cookie is called "laravel_session"
- Every time the client makes a request to the same site, it will add the session ID cookie to the HTTP request
- When the server receives the session ID cookie, it will lookup the session state on its storage – if no session exists for that session ID, a new session is created

Laravel Sessions

- To access the **session object** in Laravel:
  - From the Request instance:

```
public function show(Request $request, …)
{
    $value = $request->session()->get('key');
    …
}
```

  - Using session helper function:

```
… $value = session('key');
```

- Session object refers to the **current** session

Session Object

- **Read** session data

```
$value = $request->session()->get('key');
$value = $request->session()->get('key', 'default');
```

```
$value = session('key');
$value = session('key', 'default');
```

```
$allValues = $request->session()->all();
```

- Check session data

```
if ($request->session()->has('users')) { … }
```

Item is present and not null

```
if ($request->session()->exists('users')) { … }
```

Item is present (item value can be null)

Session Object

- **Write** session data

```
$request->session()->put('key', 'value');
```

```
session(['key' => 'value']);
```

- **Delete** session data

```
// Forget a single item …
$request->session()->forget('name');

// Forget multiple items …
$request->session()->forget(['name', 'status']);

// Clear all session data…
$request->session()->flush();
```

Flash data

- Data stored in sessions using **flash** method will be available **immediately** and during the **next** HTTP request
- **After** the **next** HTTP request, the flashed data will be **automatically deleted**.
- Allows to pass information between **2 consecutive HTTP Requests**
- Particularly useful for passing data when **redirecting** to another page

Flash data

- **Write** session **flash** data

```
$request->session()->flash('key', 'Value');
```

- Redirect with flash data

```
return redirect('…')->with('key', 'Value');
```

- **Read** session **flash** data
  - Reading flashed data is the same as reading any other session data

```
$value = session('key');
$value = session('key', 'default');
```

Flash input values

- **Flashing input** values to the session
  - Maintains the input values between calls
- Flash all inputs:

```
$request->flash();
```

- Flash all input data when redirecting:

```
return redirect('…')->withInput();
```

```
return back()->withInput();
```

This is automatic when using validators

Flash input values

- **Read** flashed input values
  - From the Request Instance

```
$field_value = $request->old('field_name');
$field_value = $request->old('field_name', 'default');
```

  - Using old helper function

```
$field_value = old('field_name');
$field_value = old('field_name', 'default');
```

  - Using old helper function on a blade view

```
<input … value="{{ old('field_name') }}">
<input … value="{{ old('field_name', 'default') }}">
```