Implementare da zero un sistema di validazione in PHP (stile Laravel)

In questo articolo creiamo un validatore minimale ma potente in PHP che usa regole in formato stringa come `title => "required|max:255"`. Supporta dot-notation (`user.email`), messaggi personalizzati, regole comuni (*required*, *max*, *min*, *email*, *numeric*, *in*, *regex*, …), comportamenti tipo *sometimes*/*nullable* e persino estensioni personalizzate, il tutto senza dipendenze esterne.

Risultato finale

```
// Esempio d'uso
$input = [
    'title' => 'Hello',
    'email' => 'john@example.com',
    'age'   => '21',
    'user'  => ['name' => 'Jane', 'birthdate' =>
'2000-01-01'],
];

$rules = [
   'title'        => 'required|string|max:255',
   'email'        => 'required|email',
   'age'          =>
'nullable|numeric|min:18|max:120',
   'user.name'    =>
'sometimes|required|string|min:2',
   'user.birthdate' => 'date|after:1990-01-01',
];
```

```
$messages = [
    'title.required' => 'Il titolo è obbligatorio.',
    'age.min'        => 'Devi avere almeno :min
anni.',
];

$validator = new Validator();
if ($validator->validate($input, $rules, $messages))
{
    // dati validi
} else {
    // stampa errori
    print_r($validator->errors());
}
```

Implementazione

Il seguente codice PHP implementa il validatore con: parsing delle regole, dot-notation, gestione messaggi, regole integrate e un meccanismo per aggiungere regole custom.

```
class Validator
{
    protected array $errors = [];
    protected static array $customRules = [];
// ['rule' => callable]
    protected static array $customMessages = [];
// ['rule' => 'messaggio di default']

    public function errors(): array
    {
```

```
        return $this->errors;
    }

    public function validate(array $data, array
$rules, array $messages = []): bool
    {
        $this->errors = [];

        foreach ($rules as $attribute => $ruleDef) {
            $parsed = $this->parseRules($ruleDef);
            $bail = in_array('bail',
array_column($parsed, 'name'), true);

            $exists = false;
            $value = $this->getValue($data,
$attribute, $exists);

            $hasSometimes = $this->hasRule($parsed,
'sometimes');
            if ($hasSometimes && !$exists) {
                continue; // se non c'è il campo, non
validare
            }

            $isNullable = $this->hasRule($parsed,
'nullable');
            if ($isNullable && $exists && $value ===
null) {
                continue; // null è permesso, skip
altre regole
            }

            foreach ($parsed as $r) {
                $name = $r['name'];
```

```
                $params = $r['params'];

                if (in_array($name, ['sometimes',
'nullable', 'bail'], true)) {
                    continue;
                }

                $passed = true;
                $message = null;

                // Regole integrate
                switch ($name) {
                    case 'required':
                        $passed = $exists && !$this-
>isEmpty($value);
                        break;

                    case 'string':
                        $passed = is_string($value);
                        break;

                    case 'array':
                        $passed = is_array($value);
                        break;

                    case 'boolean':
                        $passed = is_bool($value) ||
$value === 0 || $value === 1 || $value === '0' ||
$value === '1';
                        break;

                    case 'integer':
                        $passed = filter_var($value,
FILTER_VALIDATE_INT) !== false;
```

```
                        break;

                    case 'numeric':
                        $passed = is_numeric($value);
                        break;

                    case 'email':
                        $passed = filter_var($value,
FILTER_VALIDATE_EMAIL) !== false;
                        break;

                    case 'min':
                        $min = $this-
>expectParam($attribute, $name, $params, 0);
                        $passed = $this-
>checkMin($value, $min, $parsed);
                        break;

                    case 'max':
                        $max = $this-
>expectParam($attribute, $name, $params, 0);
                        $passed = $this-
>checkMax($value, $max, $parsed);
                        break;

                    case 'between':
                        $min = $this-
>expectParam($attribute, $name, $params, 0);
                        $max = $this-
>expectParam($attribute, $name, $params, 1);
                        $passed = $this-
>checkMin($value, $min, $parsed) && $this-
>checkMax($value, $max, $parsed);
                        break;
```

```
                    case 'in':
                        $passed =
in_array((string)$value, array_map('strval',
$params), true);
                        break;

                    case 'regex':
                        $pattern = $this-
>expectParam($attribute, $name, $params, 0);
                        $passed =
@preg_match($pattern, (string)$value) === 1;
                        break;

                    case 'date':
                        $passed = $this-
>parseDate($value) instanceof DateTimeInterface;
                        break;

                    case 'after':
                        $other = $this-
>expectParam($attribute, $name, $params, 0);
                        $passed = $this-
>compareDates($data, $attribute, $value, $other,
'>');
                        break;

                    case 'before':
                        $other = $this-
>expectParam($attribute, $name, $params, 0);
                        $passed = $this-
>compareDates($data, $attribute, $value, $other,
'<');
                        break;
```

```
                    default:
                        // Regola custom
                        if
(isset(self::$customRules[$name])) {
                            $callback =
self::$customRules[$name];
                            $result =
$callback($attribute, $value, $params, $data);
                            if ($result === true) {
                                $passed = true;
                            } elseif ($result ===
false) {
                                $passed = false;
                            } elseif
(is_string($result)) {
                                $passed = false;
                                $message = $result;
// messaggio diretto dalla regola
                            }
                        } else {
                            throw new
InvalidArgumentException("Regola sconosciuta:
{$name}");
                        }
                }

                if (!$passed) {
                    $this->addError(
                        $attribute,
                        $name,
                        $message ?? $this-
>makeMessage($attribute, $name, $params, $messages,
$value),
```

```
                        $params
                    );

                    if ($bail) {
                        break; // interrompi alla
prima violazione per questo campo
                    }
                }
            }
        }

        return empty($this->errors);
    }

    /** ------------------------------ Helper:
parsing & accesso ------------------------------ */

    protected function parseRules(string|array
$ruleDef): array
    {
        if (is_array($ruleDef)) {
            $chunks = $ruleDef;
        } else {
            // Supporta | e : escapati
            $chunks = preg_split('/(?<!)|/',
$ruleDef) ?: [];
            $chunks = array_map(fn($c) =>
str_replace(['|', ':'], ['|', ':'], trim($c)),
$chunks);
        }

        $out = [];
        foreach ($chunks as $chunk) {
            if ($chunk === '') continue;
```

```
            $parts = preg_split('/(?<!):/', $chunk,
2);
            $name = strtolower(trim($parts[0]));
            $params = [];
            if (isset($parts[1])) {
                $params = array_map('trim',
explode(',', $parts[1]));
            }
            $out[] = ['name' => $name, 'params' =>
$params];
        }
        return $out;
    }

    protected function hasRule(array $parsed, string
$rule): bool
    {
        foreach ($parsed as $r) {
            if ($r['name'] === $rule) return true;
        }
        return false;
    }

    protected function getValue(array $data, string
$key, ?bool &$exists = null)
    {
        $segments = explode('.', $key);
        $cur = $data;
        foreach ($segments as $seg) {
            if (is_array($cur) &&
array_key_exists($seg, $cur)) {
                $cur = $cur[$seg];
            } else {
                $exists = false;
```

```
                return null;
            }
        }
        $exists = true;
        return $cur;
    }

    protected function isEmpty($value): bool
    {
        if ($value === null) return true;
        if (is_string($value)) return trim($value)
=== '';
        if (is_array($value)) return count($value)
=== 0;
        return false;
    }

    /** ------------------------------ Helper:
min/max/between ------------------------------ */

    protected function isStringContext(array
$parsed): bool
    {
        return $this->hasRule($parsed, 'string');
    }

    protected function isArrayContext(array $parsed):
bool
    {
        return $this->hasRule($parsed, 'array');
    }

    protected function isNumericContext(array
$parsed): bool
```

```
    {
        return $this->hasRule($parsed, 'numeric') ||
$this->hasRule($parsed, 'integer');
    }

    protected function checkMin($value, $min, array
$parsed): bool
    {
        if ($this->isNumericContext($parsed)) {
            return is_numeric($value) && $value + 0
>= (float)$min;
        }
        if ($this->isArrayContext($parsed) &&
is_array($value)) {
            return count($value) >= (int)$min;
        }
        // default: lunghezza stringa
        return is_string($value) && mb_strlen($value)
>= (int)$min;
    }

    protected function checkMax($value, $max, array
$parsed): bool
    {
        if ($this->isNumericContext($parsed)) {
            return is_numeric($value) && $value + 0
<= (float)$max;
        }
        if ($this->isArrayContext($parsed) &&
is_array($value)) {
            return count($value) <= (int)$max;
        }
        // default: lunghezza stringa
        return is_string($value) && mb_strlen($value)
```

```
<= (int)$max;
    }

    /** ------------------------------ Helper: date
& confronti ------------------------------ */

    protected function parseDate($value): ?
DateTimeInterface
    {
        if ($value instanceof DateTimeInterface)
return $value;
        if (!is_string($value)) return null;

        // prova ISO e fallback a strtotime
        $dt = DateTime::createFromFormat('Y-m-d',
$value) ?: DateTime::createFromFormat(DateTime::ATOM,
$value);
        if ($dt instanceof DateTimeInterface) return
$dt;

        $ts = strtotime($value);
        return $ts ? (new DateTime())-
>setTimestamp($ts) : null;
    }

    protected function compareDates(array $data,
string $attribute, $value, string $other, string
$op): bool
    {
        $left = $this->parseDate($value);
        if (!$left) return false;

        if ($other === 'today') {
            $right = (new DateTime('today'));
```

```
        } elseif (str_contains($other, '-')) {
            $right = $this->parseDate($other);
        } else {
            $exists = false;
            $otherVal = $this->getValue($data,
$other, $exists);
            if (!$exists) return false;
            $right = $this->parseDate($otherVal);
        }

        if (!$right) return false;

        return match ($op) {
            '>' => $left > $right,
            '<' => $left < $right,
            default => false
        };
    }

    /** ------------------------------ Messaggi ----
-------------------------- */

    protected function addError(string $attribute,
string $rule, string $message, array $params): void
    {
        $this->errors[$attribute][$rule][] =
$message;
    }

    protected function makeMessage(string $attribute,
string $rule, array $params, array $overrides,
$value): string
    {
        // Messaggi per-campo
```

```
        $key = "{$attribute}.{$rule}";
        if (isset($overrides[$key])) {
            return $this->interpolate($overrides[$key], $attribute, $params, $value);
        }

        // Messaggi predefiniti (per-regola)
        $defaults = $this->defaultMessages();
        if (isset($defaults[$rule])) {
            return $this->interpolate($defaults[$rule], $attribute, $params, $value);
        }

        // Messaggio custom registrato
        if (isset(self::$customMessages[$rule])) {
            return $this->interpolate(self::$customMessages[$rule], $attribute, $params, $value);
        }

        return "Il campo :attribute non è valido.";
    }

    protected function interpolate(string $message, string $attribute, array $params, $value): string
    {
        // parametri comuni
        $repl = [
            ':attribute' => $attribute,
            ':value'     => is_scalar($value) ? (string)$value : '',
        ];
```

```
        // parametri posizionali :min, :max, :other
ecc.
        $names = [':min', ':max', ':other', ':1',
':2'];
        foreach ($names as $i => $name) {
            if (isset($params[$i])) {
                $repl[$name] = (string)$params[$i];
            }
        }

        return strtr($message, $repl);
    }

    protected function defaultMessages(): array
    {
        return [
            'required' => 'Il campo :attribute è
obbligatorio.',
            'string'   => 'Il campo :attribute deve
essere una stringa.',
            'array'    => 'Il campo :attribute deve
essere un array.',
            'boolean'  => 'Il campo :attribute deve
essere booleano.',
            'integer'  => 'Il campo :attribute deve
essere un intero.',
            'numeric'  => 'Il campo :attribute deve
essere numerico.',
            'email'    => 'Il campo :attribute deve
essere un indirizzo email valido.',
            'min'      => 'Il campo :attribute deve
essere almeno :min.',
            'max'      => 'Il campo :attribute non
```

```
può superare :max.',
            'between'  => 'Il campo :attribute deve
essere tra :min e :max.',
            'in'       => 'Il campo :attribute deve
essere uno dei valori consentiti.',
            'regex'    => 'Il formato di :attribute
non è valido.',
            'date'     => 'Il campo :attribute deve
essere una data valida.',
            'after'    => 'Il campo :attribute deve
essere successivo a :other.',
            'before'   => 'Il campo :attribute deve
essere precedente a :other.',
        ];
    }

    protected function expectParam(string $attribute,
string $rule, array $params, int $index): string
    {
        if (!isset($params[$index])) {
            throw new InvalidArgumentException("La
regola {$rule} per {$attribute} richiede un parametro
alla posizione {$index}");
        }
        return $params[$index];
    }

    /** ------------------------------ Estensioni
custom ------------------------------ */

    public static function extend(string $name,
callable $callback, ?string $defaultMessage = null):
void
    {
```

```
        self::$customRules[$name] = $callback;
        if ($defaultMessage) {
            self::$customMessages[$name] =
$defaultMessage;
        }
    }
}

// ------------------------------ Esempio di
estensione: starts_with ----------------------------
--

Validator::extend('starts_with', function (string
$attribute, $value, array $params) {
    $prefix = $params[0] ?? '';
    if (!is_string($value)) return false;
    return str_starts_with($value, $prefix);
}, 'Il campo :attribute deve iniziare con :1.');

// ------------------------------ Esempio di
utilizzo ------------------------------

$input = [
    'title' => 'Hello world',
    'email' => 'john@example.com',
    'age'   => '17',
    'slug'  => 'post-1',
];

$rules = [
    'title' =>
'required|string|between:3,255|starts_with:Hel',
    'email' => 'required|email',
    'age'   => 'nullable|numeric|min:18', // fallirà
```

```
    'slug'  => 'regex:/^[a-z0-9-]+$/',
];

$messages = [
    'slug.regex' => 'Lo slug può contenere solo
lettere minuscole, numeri e trattini.',
];

$validator = new Validator();
$ok = $validator->validate($input, $rules,
$messages);

if (!$ok) {
    print_r($validator->errors());
}
```

Dettagli importanti del design

- **Ordine delle regole**: viene rispettato; con `bail` interrompi alla prima violazione per quel campo.
- **Contesto di tipo**: `min`/`max` operano su numeri se presenti `numeric`/`integer`, sulla lunghezza se `string`, sul conteggio se `array`.
- **`sometimes`**: valida il campo solo se presente nell'input.
- **`nullable`**: se il valore è `null`, salta le altre regole del campo.
- **Dot-notation**: consente `user.email` senza dover navigare manualmente l'array.
- **Messaggi**: override per *campo.regola* e template predefiniti con placeholder `:attribute`, `:min`, `:max`, `:other`, `:value`.
- **Regole custom**: con `Validator::extend()` aggiungi nuove regole riusabili e un messaggio di default.

Test rapidissimi

Puoi creare dei test veloci in puro PHP per assicurarti che i casi principali siano coperti.

```
// "happy path"
assert((new Validator())->validate(
    ['name' => 'Anna'], ['name' =>
'required|string|min:2']
) === true);

// required fallisce
$v = new Validator();
assert($v->validate(['name' => ''], ['name' =>
'required']) === false);
assert(isset($v->errors()['name']['required']));
```

Estendere il sistema

- Aggiungi *file rules* (dimensione, mime) per upload.
- Supporta wildcard tipo `items.*.id` (si può iterare ricorsivamente sugli array).
- Internazionalizzazione dei messaggi tramite file di lingua.
- Raccogli solo il primo errore per campo quando vuoi errori “compatti”.

Conclusioni

Con poche centinaia di righe abbiamo un validatore flessibile, espressivo e facilmente estendibile, con una sintassi familiare a chi usa Laravel. Puoi inserirlo nei tuoi progetti legacy o micro-servizi dove non vuoi importare l’intero framework ma desideri regole di validazione coerenti e leggibili.