FormatPHP

Release stable

Skillshare, Inc

2022-06-15

CONTENTS

For skillshare/formatphp stable. Updated on 2022-06-15.

CHAPTER

ONE

WELCOME!

Inspired by FormatJS and ECMAScript 2023 Internationalization API (ECMA-402), FormatPHP provides an API to format dates, numbers, and strings, including pluralization and translation. FormatPHP is powered by PHP's intl extension and integrates with Unicode CLDR and ICU Message syntax standards. It requires libicu version 69.1 or higher.

CHAPTER

TWO

CONTENTS

2.1 Getting Started

FormatPHP helps you internationalize and localize your PHP applications. Through the use of its API, you can format messages for any locale, including localization of dates and times, numbers, and pluralization. FormatPHP includes tools for extracting messages for translation and loading translated messages to render, based on a locale.

The general steps when working with FormatPHP are:

1. Load translation strings.
2. Configure FormatPHP for a locale.
3. Format strings in your application with `formatMessage()` and localize data with the other formatter methods (e.g., `formatNumber()`, `formatDate()`, `formatTime()`, etc.).

2.1.1 Requirements

FormatPHP requires:

- PHP 7.4+
- ext-intl
- ext-json
- ext-mbstring
- libicu 69.1 or later

2.1.2 Install With Composer

The only supported installation method for FormatPHP is Composer. Use the following command to add FormatPHP to your project dependencies:

```
composer require skillshare/formatphp
```

2.1.3 Using FormatPHP

The following example shows a complete working implementation of FormatPHP:

```
use FormatPHP\Config;
use FormatPHP\FormatPHP;
use FormatPHP\Intl;
use FormatPHP\Message;
use FormatPHP\MessageCollection;

// Translated messages in Spanish with matching IDs to what you declared.
$messagesInSpanish = new MessageCollection([
    new Message('hello', '¡Hola {name}! Hoy es {today}.'),
]);

$config = new Config(
    // Locale of the application (or of the user using the application).
    new Intl\Locale('es-ES'),
);

$formatphp = new FormatPHP($config, $messagesInSpanish);

echo $formatphp->formatMessage([
    'id' => 'hello',
    'defaultMessage' => 'Hello, {name}! Today is {today}.',
], [
    'name' => 'Arwen',
    'today' => $formatphp->formatDate(new DateTimeImmutable()),
]); // e.g., ¡Hola Arwen! Hoy es 31/1/22.
```

2.2 Formatting Strings

2.2.1 Formatting Messages

The standard way of working with FormatPHP is:

1. Determine the user's locale (there are a variety of ways to do this, outside the scope of this documentation).
2. *Load translated messages* for the locale.
3. Configure FormatPHP with the locale and translated messages.
4. Wrap strings you wish to translate and localize in your application with calls to `formatMessage()`.

Listing 1: Using a simple message for translation.

```
use FormatPHP\Config;
use FormatPHP\FormatPHP;
use FormatPHP\Intl;
use FormatPHP\MessageLoader;

// Your custom method to determine the user's locale.
$userLocale = determineUserLocale();
```

(continues on next page)

(continued from previous page)

```
$config = new Config(new Intl\Locale($userLocale));
$messageLoader = new MessageLoader('/path/to/app/locales', $config);
$formatphp = new FormatPHP($config, $messageLoader->loadMessages());

echo $formatphp->formatMessage([
    'id' => 'hello',
    'defaultMessage' => 'Hello! Welcome to the website!',
]);
```

formatMessage()

The `formatMessage()` method facilitates translation and localization of string messages in your applications. Its method signature is:

```
public function FormatPHP::formatMessage(array $descriptor, array $values = []) string
```

The `$descriptor` argument is an array with the following properties:

id A unique message identifier used to locate translated versions of this message in the message collection provided to FormatPHP.

The `id` is required if `defaultMessage` is not present.

defaultMessage The message to format and translate according to the locale. This may be a simple string, or it may have placeholders and other complex arguments.

The `defaultMessage` is required if `id` is not present. In case the `id` is not present, FormatPHP will auto-generate an identifier for the message.

description An optional description that you may use to provide additional context to translators and developers. This is especially useful for translation management systems, if using the FormatPHP *extraction tools*.

The optional `$values` argument is an array of key/value pairs, where the key refers to either a *named placeholder* in the message or a *tag for rich-text formatting*.

Placeholders

FormatPHP supports ICU message syntax for placeholders.

In their simplest form, placeholders are names surrounded by curly braces (i.e., `{` and `}`). Translators will not modify these names, but they are able to move them around in the string to a position that makes sense according to the grammar of a given language and locale.

```
echo $formatphp->formatMessage([
    'id' => 'greeting',
    'defaultMessage' => 'Hello, {personName}!',
], [
    'personName' => $user->getName(),
])
```

Pluralization and Complex Arguments

FormatPHP supports ICU message syntax for pluralization and complex argument types.

Using the classic example from the ICU documentation, the following shows how to provide complex argument types to FormatPHP. When translating, translators will properly translate each sub-message of this structure, leaving the complex arguments intact.

```
echo $formatphp->formatMessage([
    'id' => 'party',
    'defaultMessage' => <<<'EOD'
        {hostGender, select,
            female {{numGuests, plural, offset:1
                =0 {{host} does not give a party.}
                =1 {{host} invites {guest} to her party.}
                =2 {{host} invites {guest} and one other person to her party.}
                other {{host} invites {guest} and # other people to her party.}
            }}
            male {{numGuests, plural, offset:1
                =0 {{host} does not give a party.}
                =1 {{host} invites {guest} to his party.}
                =2 {{host} invites {guest} and one other person to his party.}
                other {{host} invites {guest} and # other people to his party.}
            }}
            other {{numGuests, plural, offset:1
                =0 {{host} does not give a party.}
                =1 {{host} invites {guest} to their party.}
                =2 {{host} invites {guest} and one other person to their party.}
                other {{host} invites {guest} and # other people to their party.}
            }}
        }
        EOD,
], [
    'hostGender' => $host->getGender(),
    'host' => $host->getName(),
    'numGuests' => count($party->guests),
    'guest' => $guest->getName(),
]);
```

Localization

FormatPHP supports ICU message syntax for formatting numbers, including currency and units, as well as dates and times.

```
echo $formatphp->formatMessage([
    'id' => 'hello',
    'defaultMessage' => <<<'EOD'
        On {actionDate, date, ::dMMMM} at {actionDate, time, ::jmm},
        they walked {distance, number, ::unit/kilometer unit-width-full-name .#}
        to pay only {amount, number, ::currency/EUR unit-width-short
        precision-currency-standard/w} in the
        {percentage, number, ::percent precision-integer} off sale
        on furniture.
```

(continues on next page)

(continued from previous page)

```
        EOD,
], [
    'actionDate' => new DateTimeImmutable('now'),
    'distance' => 5.358,
    'amount' => 150.00123,
    'percentage' => 0.25,
]);
```

In the en-US locale, this produces a message similar to:

> On June 10 at 6:18 PM, they walked 5.4 kilometers to pay only €150 in the 25% off sale on furniture.

In a locale for which we have no translations, this message will still have localization features specific to the locale. For example, in ja-JP, the message produced is similar to:

> On 610 at 18:21, they walked 5.4 to pay only €150 in the 25% off sale on furniture.

Note: According to ECMA-402, section 15.5.4 (specifically step 5.b.), if the `style` is *percent*, then the number formatter must multiply the value by 100. This means the formatter expects percent values expressed as fractions of 100 (i.e., 0.25 for 25%, 0.055 for 5.5%, etc.).

Since FormatJS also applies this rule to `::percent` number skeletons in formatted messages, FormatPHP does, too.

For example:

```
echo $formatphp->formatMessage([
    'id' => 'discountMessage',
    'defaultMessage' => 'Take {discount, number, ::percent} off the retail price!',
], [
    'discount' => 0.25,
]); // e.g., "Take 25% off the retail price!"
```

Hint: See the sections on *Formatting Dates & Times* and *Formatting Numbers & Currency* for other methods for localizing values.

Rich Text Formatting (Use of Tags in Messages)

While the ICU message syntax does not prohibit the use of HTML tags in formatted messages, HTML tags provide an added level of difficulty when it comes to parsing and validating ICU formatted messages. By default, FormatPHP does not support HTML tags in messages.

Instead, like FormatJS, we support embedded rich text formatting using custom tags and callbacks. This allows developers to embed as much text as possible so sentences don't have to be broken up into chunks.

Attention: These are not HTML or XML tags, and attributes are not supported.

```
echo $formatphp->formatMessage([
    'id' => 'priceMessage',
    'defaultMessage' => <<<'EOD'
```

(continues on next page)

(continued from previous page)

```
        Our price is <boldThis>{price}</boldThis>
        with <link>{discount, number, ::percent} discount</link>
        EOD,
], [
    'price' => $formatphp->formatCurrency(29.99, 'USD', new Intl\NumberFormatOptions([
        'maximumFractionDigits' => 0,
    ])),
    'discount' => .025,
    'boldThis' => fn ($text) => "<strong>$text</strong>",
    'link' => fn ($text) => "<a href=\"/discounts/1234\">$text</a>",
]);
```

For an `en-US` locale, this will produce a string similar to the following:

```
Our price is <strong>$30</strong> with <a href="/discounts/1234">2.5% discount</a>
```

For rich text elements used throughout your application, you may provide a map of tag names to rich text formatting functions, when configuring FormatPHP.

```
$config = new Config(
    new Intl\Locale('en-US'),
    null,
    [
        'em' => fn ($text) => "<em class=\"myClass\">$text</em>",
        'strong' => fn ($text) => "<strong class=\"myClass\">$text</strong>",
    ],
);
```

Using this approach, consider the following formatted message:

```
$formatphp->formatMessage([
    'id' => 'welcome',
    'defaultMessage' => 'Welcome, <strong><em>{name}</em></strong>',
], [
    'name' => 'Sam',
]);
```

It will produce a string similar to the following:

```
Welcome, <strong class="myClass"><em class="myClass">Sam</em></strong>
```

2.2.2 Formatting Dates & Times

You may use the methods `formatDate()` and `formatTime()` to format dates and times according to the locale.

```
use FormatPHP\Config;
use FormatPHP\FormatPHP;
use FormatPHP\Intl;

$config = new Config(new Intl\Locale('es-ES'));
$formatphp = new FormatPHP($config);
```

(continues on next page)

(continued from previous page)

```
$date = new DateTimeImmutable('now');

echo $formatphp->formatDate($date); // e.g., "10/6/22"
echo $formatphp->formatTime($date); // e.g., "19:06"
```

Tip: See the *Intl\DateTimeFormatOptions* reference for more information on the options available.

Formatting Dates

The `formatDate()` method facilitates localization of dates in your applications. Its method signature is:

```
public function FormatPHP::formatDate(
    DateTimeInterface | int | string | null $date,
    ?Intl\DateTimeFormatOptions $options = null
) string
```

The `$date` argument is the value you wish to localize. It uses the configured locale's preferred formatting to localize the date. You may provide a DateTimeInterface instance, a Unix timestamp, a date string in one of the supported date and time formats, or `null` to use the current date and time.

To customize the display of the localized date, you may provide a *Intl\DateTimeFormatOptions* instance as the `$options` argument.

```
$date = new DateTimeImmutable('now');

echo $formatphp->formatDate($date, new Intl\DateTimeFormatOptions([
    'day' => 'numeric',
    'month' => 'short',
    'weekday' => 'short',
    'year' => 'numeric',
])); // e.g., "vie, 10 jun 2022"
```

Formatting Times

The `formatTime()` method facilitates localization of times in your applications. It differs from `formatDate()` by using *numeric* as the default value for the `hour` and `minute` options. Otherwise, it functions identical to `formatDate()`.

Its method signature is:

```
public function FormatPHP::formatTime(
    DateTimeInterface | int | string | null $date,
    ?Intl\DateTimeFormatOptions $options = null
) string
```

The `$date` argument is the value you wish to localize. It uses the configured locale's preferred formatting to localize the time. You may provide a DateTimeInterface instance, a Unix timestamp, a date string in one of the supported date and time formats, or `null` to use the current date and time.

To customize the display of the localized time, you may provide a *Intl\DateTimeFormatOptions* instance as the `$options` argument.

```
$date = new DateTimeImmutable('now');

echo $formatphp->formatTime($date, new Intl\DateTimeFormatOptions([
    'timeStyle' => 'full',
    'timeZone' => 'America/Chicago',
])); // e.g., "14:21:50 (hora de verano central)"
```

2.2.3 Formatting Numbers & Currency

You may use the methods `formatNumber()` and `formatCurrency()` to format numbers and currency according to the locale.

```
use FormatPHP\Config;
use FormatPHP\FormatPHP;
use FormatPHP\Intl;

$config = new Config(new Intl\Locale('es-ES'));
$formatphp = new FormatPHP($config);

$number = -12_345.678;

echo $formatphp->formatNumber($number);           // e.g., "-12.345,678"
echo $formatphp->formatCurrency($number, 'USD'); // e.g., "-12.345,68 $"
```

Tip: See the *Intl\NumberFormatOptions* reference for more information on the options available.

Formatting Numbers

The `formatNumber()` method facilitates localization of numbers in your applications. Its method signature is:

```
public function FormatPHP::formatNumber(
    float | int $number,
    ?Intl\NumberFormatOptions $options = null
) string
```

The `$number` argument is the value you wish to localize. It uses the configured locale's preferred formatting to localize the number.

To customize the display of the localized number, you may provide a *Intl\NumberFormatOptions* instance as the `$options` argument.

Listing 2: Distance to Proxima Centauri in kilometers.

```
echo $formatphp->formatNumber(4.2465, new Intl\NumberFormatOptions([
    'notation' => 'scientific',
    'style' => 'unit',
    'unit' => 'kilometer',
    'scale' => 9.46 * (10 ** 12), // Kilometers in a light year
])); // e.g., "4.017E13 km"
```

Formatting Currency

While you may specify *currency* in the `style` property of `NumberFormatOptions` when using `formatNumber()`, you may also use the convenience method `formatCurrency()`, whose signature is:

```
public function FormatPHP::formatCurrency(
    float | int $number,
    string $currencyCode,
    ?Intl\NumberFormatOptions $options = null
) string
```

The `$number` argument is the currency value you wish to localize. Like `formatNumber()`, it uses the configured locale's preferred formatting to localize the currency.

The `$currencyCode` argument is an ISO 4217 currency code to use when formatting currency. For example, *USD*, *EUR*, or *CNY*.

```
$config = new Config(new Intl\Locale('es-ES'));
$formatphp = new FormatPHP($config);

echo $formatphp->formatCurrency(123.0, 'USD', new Intl\NumberFormatOptions([
    'currencyDisplay' => 'symbol',
    'trailingZeroDisplay' => 'stripIfInteger',
])); // e.g., "123 US$"
```

2.2.4 Formatting Display Names

You may use the method `formatDisplayName()` to format the display names of languages, regions, currency, and more. This returns a locale-appropriate, translated string for the type requested.

Tip: See the *Intl\DisplayNamesOptions* reference for more information on the options available.

All of the following examples use the locale es-ES.

Localize a Language Name

Using the *language* `type`, you may format a localized and translated display name for any language tag.

```
echo $formatphp->formatDisplayName('en-US', new Intl\DisplayNamesOptions([
    'type' => 'language',
])); // e.g., "inglés (Estados Unidos)"

echo $formatphp->formatDisplayName('zh-Hans-SG', new Intl\DisplayNamesOptions([
    'type' => 'language',
])); // e.g., "chino (simplificado, Singapur)"
```

Localize a Currency Name

Using the *currency* `type`, you may format a localized and translated display name for any ISO 4217 currency code.

```
echo $formatphp->formatDisplayName('EUR', new Intl\DisplayNamesOptions([
    'type' => 'currency',
])); // e.g., "euro"

echo $formatphp->formatDisplayName('JPY', new Intl\DisplayNamesOptions([
    'type' => 'currency',
])); // e.g., "yen"
```

Localize a Region Name

Using the *region* `type`, you may format a localized and translated display name for any region code.

```
echo $formatphp->formatDisplayName('GB', new Intl\DisplayNamesOptions([
    'type' => 'region',
])); // e.g., "Reino Unido"

echo $formatphp->formatDisplayName('UN', new Intl\DisplayNamesOptions([
    'type' => 'region',
])); // e.g., "Naciones Unidas"
```

Localize a Script Name

Using the *script* `type`, you may format a localized and translated display name for the script part of any language tag.

```
echo $formatphp->formatDisplayName('Latn', new Intl\DisplayNamesOptions([
    'type' => 'script',
])); // e.g., "latino"

echo $formatphp->formatDisplayName('Cyrl', new Intl\DisplayNamesOptions([
    'type' => 'script',
])); // e.g., "cirílico"
```

2.3 Extracting & Loading Strings

2.3.1 Using the Console Command To Extract Messages

The `formatphp extract` console command helps you extract messages from your application source code, saving them to JSON files that your translation management system can use.

Tip: See *TMS Support* for translation management systems we support out-of-the-box.

```
./vendor/bin/formatphp extract \
    --out-file=locales/en.json \
```

(continues on next page)

(continued from previous page)

```
    'src/**/*.php' \
    'src/**/*.phtml'
```

In order for message extraction to function properly, we have a few rules that must be followed (see comments inline in the following example):

```
// The method name must be exactly `formatMessage` (see Tip below).
// The name of the variable (i.e., `$formatphp`) does not matter.
$formatphp->formatMessage(

    // The message descriptor must be an array literal.
    [
        'id' => 'hello', // ID (if provided) must be a string literal.
        'description' => 'Message to translators', // Description must be a string␣
↪literal.
        'defaultMessage' => 'My name is {name}', // Message must be a string literal.
    ],

    [
        'name' => $userName,
    ],

);
```

At least one of `id` or `defaultMessage` must be present.

Tip: If you wish to use a different function name (e.g., maybe you wish to wrap this method call in a Closure, etc.), you may do so, but you must provide the `--additional-function-names` option to the `formatphp extract` console command. This option takes a comma-separated list of function names for the extractor to parse.

```
--additional-function-names='formatMessage, myCustomFormattingFunction'
```

To see all available options, view the command help with `formatphp help extract`.

2.3.2 Using MessageLoader to Load Messages

We provide a message loader to load translation strings from locale files that have been generated by your translation management system.

```
use FormatPHP\Config;
use FormatPHP\FormatPHP;
use FormatPHP\Intl;
use FormatPHP\MessageLoader;

$config = new Config(new Intl\Locale('es-419'));

$messageLoader = new MessageLoader(

    // The path to your locale JSON files (i.e., en.json, es.json, etc.).
    '/path/to/app/locales',
```

(continues on next page)

(continued from previous page)

```
    // The configuration object created above.
    $config,

);

$formatphp = new FormatPHP($config, $messageLoader->loadMessages());
```

This example assumes a directory of locale JSON files located at `/path/to/app/locales`, which includes an `en.json` file with these contents:

```
{
  "hello": {
    "defaultMessage": "Hello, {name}! Today is {today}."
  }
}
```

and an `es.json` file with these contents:

```
{
  "hello": {
    "defaultMessage": "¡Hola {name}! Hoy es {today}."
  }
}
```

The message loader uses a fallback method to choose locales. In this example, if the configured locale is `es-419` (i.e., Spanish appropriate for the Latin America and Caribbean region), the fallback method will choose `es.json` for the messages.

2.3.3 Testing With Pseudo Locales

Pseudo locales provide a way to test an application with various types of characters and string widths. FormatPHP provides a tool to convert any locale file to a pseudo locale for testing purposes.

Given the English message `my name is {name}`, the following table shows how each supported pseudo locale will render this message.

Table 1: Pseudo locales

Locale	Message
en-XA	ṁẏ ńâṁè íś {name}
en-XB	[!! ṁẏ ńâṁè íííś !!]{name}
xx-AC	MY NAME IS {name}
xx-HA	[javascript]my name is {name}
xx-LS	my name is {name}SSSSSSSSSSSSSSSSSSSSSSSS

To convert a locale to a pseudo locale, use the `formatphp pseudo-locale` console command.

```
./vendor/bin/formatphp pseudo-locale \
    --out-file locales/en-XA.json \
    locales/en.json \
    en-XA
```

You may then *configure and load* the pseudo locale just like with other locales.

Tip: To see all available options, view the command help with `formatphp help pseudo-locale`.

2.4 TMS Support

A translation management system, or TMS, allows translators to use your default locale file to create translations for all the other languages your application supports. To work with a TMS, you will *extract the formatted strings* from your application to send to the TMS. Often, a TMS will specify a particular document format they require.

To extract strings in a specific TMS format, use the `--format` option.

```
./vendor/bin/formatphp extract \
    --format=simple \
    --out-file=locales/en.json \
    'src/**/*.php'
```

Out of the box, FormatPHP supports the following formatters for integration with third-party TMSes. Supporting a TMS does not imply endorsement of that particular TMS.

Table 2: Translation management systems

TMS	`--format=`
Crowdin Chrome JSON	`crowdin`
Lingohub	`simple`
locize	`simple`
Phrase	`simple`
SimpleLocalize	`simple`
Smartling ICU JSON	`smartling`

Our default formatter is `formatphp`, which mirrors the output of the default formatter for FormatJS.

2.4.1 Custom Formatters

You may provide your own formatter using our interfaces. You will need to create a writer for the format. Optionally, you may create a reader, if using our message loader or the `formatphp pseudo-locale` command with the `--in-format` option.

The *writer* must implement `FormatPHP\Format\WriterInterface` or be a callable of the shape:

```
callable(FormatPHP\DescriptorCollection, FormatPHP\Format\WriterOptions): mixed[]
```

The *reader* must implement `FormatPHP\Format\ReaderInterface` or be a callable of the shape:

```
callable(mixed[]): FormatPHP\MessageCollection
```

To use your custom writer with the `formatphp extract` CLI tool, pass the fully-qualified class name to `--format`, or a path to a script that returns the callable.

For example, given the script `my-writer.php` with the following contents:

```php
<?php

use FormatPHP\DescriptorCollection;
use FormatPHP\Format\WriterOptions;

require_once 'vendor/autoload.php';

/**
 * @return mixed[]
 */
return function(DescriptorCollection $descriptors, WriterOptions $options): array {
    // Custom writer logic to create an array of data we will write
    // as JSON to a file, which your TMS will be able to use.
};
```

You may call `formatphp extract` like this:

```
./vendor/bin/formatphp extract \
    --format='path/to/my-writer.php' \
    --out-file=locales/en.json \
    'src/**/*.php'
```

Then, to use a custom reader with the message loader, you may do something like the following:

```php
$messageLoader = new \FormatPHP\MessageLoader(

    // The path to your locale JSON files (i.e., en.json, es.json, etc.).
    '/path/to/app/locales',

    // The configuration object created earlier.
    $config,

    // Pass your custom reader through the formatReader parameter.
    MyCustomReader::class,

);
```

The `formatReader` parameter of the `MessageLoader` constructor accepts the following:

- Fully-qualified class name for a class that implements `FormatPHP\Format\ReaderInterface`
- An already-instantiated instance object of `FormatPHP\Format\ReaderInterface`
- A callable with the shape `callable(mixed[]): FormatPHP\MessageCollection`
- The path to a script that returns a callable with this shape

2.5 Reference

2.5.1 Intl\DateTimeFormatOptions

You can fine-tune date and time formatting with `DateTimeFormatOptions`.

```
use FormatPHP\Config;
use FormatPHP\FormatPHP;
use FormatPHP\Intl;

$config = new Config(new Intl\Locale('es-ES'));
$formatphp = new FormatPHP($config);

$date = new DateTimeImmutable('now');

echo $formatphp->formatDate($date, new Intl\DateTimeFormatOptions([
    'dateStyle' => 'medium',
])); // e.g., "10 jun 2022"

echo $formatphp->formatTime($date, new Intl\DateTimeFormatOptions([
    'timeStyle' => 'long',
])); // e.g., "0:58:05 UTC"
```

General Formatting

`DateTimeFormatOptions` accepts the following general options to format dates and times:

dateStyle General formatting of the date, according to the locale. Possible values are: *full*, *long*, *medium*, and *short*.

```
echo $formatphp->formatDate($date, new Intl\DateTimeFormatOptions([
    'dateStyle' => 'full',
])); // e.g., "viernes, 10 de junio de 2022"
```

timeStyle General formatting of the time, according to the locale. Possible values are: *full*, *long*, *medium*, and *short*.

```
echo $formatphp->formatDate($date, new Intl\DateTimeFormatOptions([
    'timeStyle' => 'full',
])); // e.g., "1:10:33 (tiempo universal coordinado)"
```

Tip: You may use both `dateStyle` and `timeStyle` together, but if used with any of the specific formatting options (i.e., `weekday`, `hour`, `month`, etc.), FormatPHP will throw `FormatPHP\Exception\InvalidArgumentException`.

Specific Formatting

In addition to the general formatting options, the following options provide more specific control over date and time formatting.

```
echo $formatphp->formatDate($date, new Intl\DateTimeFormatOptions([
    'era' => 'short',
    'year' => '2-digit',
    'month' => 'short',
    'weekday' => 'short',
    'day' => 'numeric',
    'hour' => '2-digit',
    'minute' => '2-digit',
    'second' => '2-digit',
])); // e.g., "vie, 10 jun 22 d. C., 1:24:01"
```

era The locale representation of the era (e.g. “AD”, “BC”). Possible values are: *long*, *short*, and *narrow*.

year The locale representation of the year. Possible values are: *numeric* or *2-digit*.

month The locale representation of the month. Possible values are: *numeric*, *2-digit*, *long*, *short*, or *narrow*.

weekday The locale representation of the weekday name. Possible values are: *long*, *short*, and *narrow*.

day The locale representation of the day. Possible values are: *numeric* or *2-digit*.

hour The locale representation of the hour. Possible values are: *numeric* or *2-digit*.

minute The locale representation of the minute. Possible values are: *numeric* or *2-digit*.

second The locale representation of the seconds. Possible values are: *numeric* or *2-digit*.

Hint: Not all locales support the same formatting. For example, some locales treat *short* and *narrow* eras with the same presentation. Others may treat *numeric* and *2-digit* hours with the same presentation.

These formats are hints for how to display dates and times, according to the given locale. When localizing content, use the locale’s preferred formatting. This is what the underlying ICU library does, and therefore, this what FormatPHP does.

Additional Options

You may use any of the following additional options to further influence date and time formatting.

Hint: While you may use these options with `dateStyle` and `timeStyle`, the `dateStyle` and `timeStyle` general formatting options rigidly stick to the preferences of the locale, so some of these options might not appear to have any effect. For example, setting `hourCycle` to *h23* will not have any effect when used with `timeStyle` in the en-US locale. This is because en-US prefers *h12*. Instead, you may use the specific formatting options with these additional options to achieve the desired results.

---

calendar The calendar system to use. Possible values include: *buddhist*, *chinese*, *coptic*, *dangi*, *ethioaa*, *ethiopic*, *gregory*, *hebrew*, *indian*, *islamic*, *islamic-civil*, *islamic-rgsa*, *islamic-tbla*, *islamic-umalqura*, *iso8601*, *japanese*, *persian*, or *roc*.

```
echo $formatphp->formatDate($date, new Intl\DateTimeFormatOptions([
    'dateStyle' => 'full',
    'calendar' => 'japanese',
])); // e.g., "Friday, June 10, 4 Reiwa" when locale is en-US
```

dayPeriod The formatting style used for day periods like "in the morning", "am", "noon", "n" etc. Keep in mind not all locales may support presentation of day periods.

Possible values are: *narrow*, *short*, or *long*.

```
echo $formatphp->formatDate($date, new Intl\DateTimeFormatOptions([
    'hour' => 'numeric',
    'dayPeriod' => 'long',
])); // e.g., "1 at night" when locale is en-US
```

hour12 If `true`, `hourCycle` will be *h12*, if `false`, `hourCycle` will be *h23*. This property overrides any value set by `hourCycle`.

```
echo $formatphp->formatDate($date, new Intl\DateTimeFormatOptions([
    'hour' => '2-digit',
    'minute' => '2-digit',
    'hour12' => false,
])); // e.g., "13:47"
```

hourCycle The hour cycle to use. Possible values are: *h11*, *h12*, *h23*, and *h24*.

If specified, this property overrides the `hc` property of the locale's language tag. The `hour12` property takes precedence over this value.

Not all locales support each of these values.

```
echo $formatphp->formatDate($date, new Intl\DateTimeFormatOptions([
    'hour' => '2-digit',
    'minute' => '2-digit',
    'hourCycle' => 'h12',
])); // e.g., "2:06 p. m." when locale is es-ES
```

numberingSystem Specifies a numbering system to use when representing numeric values. You may specify any numbering system defined within Unicode CLDR and bundled in the ICU library version that is available on your platform. However, numbering systems featuring algorithmic numbers do not yet work.

Possible values include (but are not limited to): *adlm*, *ahom*, *arab*, *arabext*, *bali*, *beng*, *bhks*, *brah*, *cakm*, *cham*, *deva*, *fullwide*, *gong*, *gonm*, *gujr*, *guru*, *hanidec*, *hmng*, *java*, *kali*, *khmr*, *knda*, *lana*, *lanatham*, *laoo*, *latn*, *lepc*, *limb*, *mathbold*, *mathdbl*, *mathmono*, *mathsanb*, *mathsans*, *mlym*, *modi*, *mong*, *mroo*, *mtei*, *mymr*, *mymrshan*, *mymrtlng*, *newa*, *nkoo*, *olck*, *orya*, *osma*, *rohg*, *saur*, *shrd*, *sind*, *sora*, *sund*, *takr*, *talu*, *tamldec*, *telu*, *thai*, *tibt*, *tirh*, *vaii*, *wara*, and *wcho*.

```
echo $formatphp->formatDate($date, new Intl\DateTimeFormatOptions([
    'hour' => '2-digit',
    'minute' => '2-digit',
    'numberingSystem' => 'jpan',
])); // e.g., ":"
```

timeZoneName An indicator for how to format the localized representation of the time zone name. Values are: *long*, *short*, *shortOffset*, *longOffset*, *shortGeneric*, or *longGeneric*.

```
echo $formatphp->formatDate($date, new Intl\DateTimeFormatOptions([
    'hour' => '2-digit',
    'minute' => '2-digit',
    'timeZoneName' => 'long',
])); // e.g., "14:17 " when the locale is ja-JP
```

timeZone The time zone to use. The default is the system's default time zone (see date_default_timezone_set()). You may use the zone names of the IANA time zone database, such as "Asia/Shanghai", "Asia/Kolkata", "America/New_York".

```
echo $formatphp->formatDate($date, new Intl\DateTimeFormatOptions([
    'hour' => '2-digit',
    'minute' => '2-digit',
    'timeZone' => 'America/Chicago',
    'timeZoneName' => 'long',
])); // e.g., "9:21 AM Central Daylight Time" when the locale is en-US
```

2.5.2 Intl\DisplayNamesOptions

When formatting display names, you must provide a `FormatPHP\Intl\DisplayNamesOptions` instance with at least a `type` defined.

type The type of data for which we wish to format a display name. This currently supports *currency*, *language*, *region*, and *script*.

Note: While ECMA-402 also defines *calendar* and *dateTimeField* as additional types, these types are not implemented in Node.js or any browsers. If set, these implementations throw exceptions, so FormatPHP follows the same pattern.

```
echo $formatphp->formatDisplayName('USD', new Intl\DisplayNamesOptions([
    'type' => 'currency',
])); // e.g., "dólar estadounidense" when the locale is es-ES
```

fallback The fallback strategy to use. If we are unable to format a display name, we will return the same code provided if `fallback` is set to *code*. If `fallback` is *none*, then we return `null`.

The default is *code*.

```
var_export($formatphp->formatDisplayName('FOO', new Intl\DisplayNamesOptions([
    'type' => 'currency',
    'fallback' => 'none',
]))); // e.g., NULL
```

style The formatting style to use: *long*, *short*, or *narrow*.

This currently only affects the display name when `type` is *currency*. The default is *long*.

```
echo $formatphp->formatDisplayName('USD', new Intl\DisplayNamesOptions([
    'type' => 'currency',
    'style' => 'narrow',
])); // e.g., "$"
```

languageDisplay This is a suggestion for displaying the language according to the locale's dialect or standard representation. In JavaScript, this defaults to *dialect*. For now, PHP supports only the *standard* representation, so *dialect* has no effect.

2.5.3 Intl\NumberFormatOptions

You can fine-tune number and currency formatting with `NumberFormatOptions`.

```
use FormatPHP\Config;
use FormatPHP\FormatPHP;
use FormatPHP\Intl;

$config = new Config(new Intl\Locale('es-ES'));
$formatphp = new FormatPHP($config);

$number = -12_345.678;

echo $formatphp->formatNumber($number, new Intl\NumberFormatOptions([
    'style' => 'unit',
    'unit' => 'meter',
    'unitDisplay' => 'long',
])); // e.g., "-12.345,678 metros"

echo $formatphp->formatCurrency($number, 'USD', new Intl\NumberFormatOptions([
    'currencySign' => 'accounting',
    'currencyDisplay' => 'symbol',
])); // e.g., "-12.345,68 US$"
```

Notation & Style

`NumberFormatOptions` accepts the following options to specify the style and type of notation desired:

notation The type of number formatting to use. Possible values are: *standard*, *scientific*, *engineering*, and *compact*. The default is *standard*.

```
echo $formatphp->formatNumber(1234.5678, new Intl\NumberFormatOptions([
    'notation' => 'scientific',
])); // e.g., "1.235E3"

echo $formatphp->formatNumber(1234.5678, new Intl\NumberFormatOptions([
    'notation' => 'compact',
])); // e.g., "1.2K"
```

style The style of the number formatting. Possible values are: *decimal*, *currency*, *percent*, and *unit*.

The default is *decimal* when using `formatNumber()`. When using `formatCurrency()`, this value is always *currency* no matter what value is set on the `NumberFormatOptions` instance.

```
echo $formatphp->formatNumber(0.25, new Intl\NumberFormatOptions([
    'style' => 'percent',
])); // e.g., "25%"

echo $formatphp->formatNumber(42, new Intl\NumberFormatOptions([
```

(continues on next page)

(continued from previous page)

```
    'style' => 'unit',
    'unit' => 'mile',
])); // e.g., "42 mi"
```

General Options

All notation types support the following options to provide more granular control over number formatting:

signDisplay Controls when to display the sign symbol for the number. The default is *auto*.

Possible values are:

- *always*: Always display the sign.
- *auto*: Use the locale to determine when to display the sign.
- *exceptZero*: Display the sign for positive and negative numbers, but never display the sign for zero.
- *never*: Never display the sign.

```
echo $formatphp->formatNumber(-123, new Intl\NumberFormatOptions([
    'signDisplay' => 'never',
])); // e.g., "123"
```

roundingMode Controls the rounding rules for the number. The default is *halfEven*.

Possible values are:

- *ceil*: All values are rounded towards positive infinity (+∞).
- *floor*: All values are rounded towards negative infinity (-∞).
- *expand*: All values are rounded away from zero.
- *trunc*: All values are rounded towards zero.
- *halfCeil*: Values exactly on the 0.5 (half) mark are rounded towards positive infinity (+∞).
- *halfFloor*: Values exactly on the 0.5 (half) mark are rounded towards negative infinity (-∞).
- *halfExpand*: Values exactly on the 0.5 (half) mark are rounded away from zero.
- *halfTrunc*: Values exactly on the 0.5 (half) mark are rounded towards zero.
- *halfEven*: Values exactly on the 0.5 (half) mark are rounded to the nearest even digit. This is often called Banker's Rounding because it is, on average, free of bias.
- *halfOdd*: Similar to *halfEven*, but rounds ties to the nearest odd number instead of even number.
- *unnecessary*: This mode doesn't perform any rounding but will throw an exception if the value cannot be represented exactly without rounding.

```
echo $formatphp->formatNumber(2.0000335, new Intl\NumberFormatOptions([
    'roundingMode' => 'halfEven',
])); // e.g., "2.000034"

echo $formatphp->formatNumber(2.0000335, new Intl\NumberFormatOptions([
    'roundingMode' => 'halfOdd',
])); // e.g., "2.000033"
```

useGrouping Controls display of grouping separators, such as thousand separators or thousand/lakh/crore separators. The default is *auto*.

Possible values are:

- *always*: Always display grouping separators, even if the locale prefers otherwise.
- *auto*: Use the locale's preference for grouping separators.
- *false*: Do not display grouping separators. Please note this is a string value and not a boolean `false` value.
- *min2*: Display grouping separators when there are at least two digits in a group.
- *true*: This is an alias for *always*. Please note this is a string value and not a boolean `true` value.

```
echo $formatphp->formatNumber(1234, new Intl\NumberFormatOptions([
    'useGrouping' => 'min2',
])); // e.g., "1234"

echo $formatphp->formatNumber(12345, new Intl\NumberFormatOptions([
    'useGrouping' => 'min2',
])); // e.g., "12,345"
```

scale A scale by which to multiply the number before formatting it. For example, a value of 100 will multiply the number by 100 first, then apply other formatting options.

```
echo $formatphp->formatNumber(0.042, new Intl\NumberFormatOptions([
    'scale' => 1000,
])); // e.g., "42"
```

minimumIntegerDigits Specifies the minimum number of integer digits to use. The default is 1.

```
echo $formatphp->formatNumber(5, new Intl\NumberFormatOptions([
    'minimumIntegerDigits' => 3,
])); // e.g., "005"
```

maximumFractionDigits Specifies the maximum number of fraction digits to use.

```
echo $formatphp->formatNumber(1.23456, new Intl\NumberFormatOptions([
    'maximumFractionDigits' => 3,
])); // e.g., "1.235"
```

minimumFractionDigits Specifies the minimum number of fraction digits to use.

`minimumFractionDigits` cannot be greater than `maximumFractionDigits`.

```
echo $formatphp->formatNumber(42.1, new Intl\NumberFormatOptions([
    'minimumFractionDigits' => 2,
])); // e.g., "42.10"
```

maximumSignificantDigits Specifies the maximum number of significant digits to use.

```
echo $formatphp->formatNumber(12345.6789, new Intl\NumberFormatOptions([
    'maximumSignificantDigits' => 3,
])); // e.g., "12,300"
```

minimumSignificantDigits Specifies the minimum number of significant digits to use.

`minimumSignificantDigits` cannot be greater than `maximumSignificantDigits`.

```
echo $formatphp->formatNumber(123.45, new Intl\NumberFormatOptions([
    'minimumSignificantDigits' => 10,
])); // e.g., "123.4500000"
```

numberingSystem Specifies a numbering system to use when representing numeric values. You may specify any numbering system defined within Unicode CLDR and bundled in the ICU library version that is available on your platform. However, numbering systems featuring algorithmic numbers do not yet work.

Possible values include (but are not limited to): *adlm*, *ahom*, *arab*, *arabext*, *bali*, *beng*, *bhks*, *brah*, *cakm*, *cham*, *deva*, *fullwide*, *gong*, *gonm*, *gujr*, *guru*, *hanidec*, *hmng*, *java*, *kali*, *khmr*, *knda*, *lana*, *lanatham*, *laoo*, *latn*, *lepc*, *limb*, *mathbold*, *mathdbl*, *mathmono*, *mathsanb*, *mathsans*, *mlym*, *modi*, *mong*, *mroo*, *mtei*, *mymr*, *mymrshan*, *mymrtlng*, *newa*, *nkoo*, *olck*, *orya*, *osma*, *rohg*, *saur*, *shrd*, *sind*, *sora*, *sund*, *takr*, *talu*, *tamldec*, *telu*, *thai*, *tibt*, *tirh*, *vaii*, *wara*, and *wcho*.

```
echo $formatphp->formatNumber(123.456, new Intl\NumberFormatOptions([
    'numberingSystem' => 'tibt',
])); // e.g., "."
```

Fraction Options

The following options affect the formatting of fractional digits (e.g., when using `minimumFractionDigits` or `maximumFractionDigits`).

trailingZeroDisplay Controls the display of trailing zeros when formatting numbers. The default is *auto*.

- *auto*: Keep the trailing zeros according to the rules defined in `minimumFractionDigits` and `maximumFractionDigits`.
- *stripIfInteger*: If the formatted number is a whole integer, do not display trailing zeros.

```
echo $formatphp->formatNumber(42.001, new Intl\NumberFormatOptions([
    'maximumFractionDigits' => 2,
    'trailingZeroDisplay' => 'stripIfInteger',
])); // e.g., "42"
```

roundingPriority Specifies how to resolve conflicts between maximum fraction digits and maximum significant digits. The default is *auto*.

- *auto*: The significant digits always win a conflict.
- *morePrecision*: The result with more precision wins the conflict.
- *lessPrecision*: The result with less precision wins the conflict.

```
echo $formatphp->formatNumber(123.4567, new Intl\NumberFormatOptions([
    'maximumSignificantDigits' => 6,
    'maximumFractionDigits' => 2,
    'roundingPriority' => 'morePrecision',
])); // e.g., "123.457"

echo $formatphp->formatNumber(123.4567, new Intl\NumberFormatOptions([
    'maximumSignificantDigits' => 6,
    'maximumFractionDigits' => 2,
    'roundingPriority' => 'lessPrecision',
])); // e.g., "123.46"
```

Currency Options

When formatting currency, you may use the following options.

currency An ISO 4217 currency code to use when formatting currency. For example, *USD*, *EUR*, or *CNY*.

This option is required if the `style` option is *currency*.

```
echo $formatphp->formatNumber(123.456, new Intl\NumberFormatOptions([
    'style' => 'currency',
    'currency' => 'EUR',
])); // e.g., "€123.46"
```

currencySign In accounting, many locales format negative currency values using parentheses rather than the minus sign. You may enable this behavior by setting this property to *accounting*. The default value is *standard*.

```
echo $formatphp->formatNumber(-56.359, new Intl\NumberFormatOptions([
    'style' => 'currency',
    'currency' => 'USD',
    'currencySign' => 'accounting',
])); // e.g., "($56.36)"
```

currencyDisplay How to display the currency.

- *symbol*: Use a localized currency symbol when formatting the currency. This is the default.
- *narrowSymbol*: Use a narrow format for the currency symbol. For example, in some locales (e.g., en-GB), USD currency will default to display as "US$100." When using *narrowSymbol*, it might display as "$100" instead. This behavior is locale-dependent.
- *code*: Use the ISO currency code when formatting currency (e.g., "USD 100").
- *name*: Use a localized name for the currency (e.g., "100 US dollars").

```
echo $formatphp->formatNumber(343.199, new Intl\NumberFormatOptions([
    'style' => 'currency',
    'currency' => 'GBP',
    'currencyDisplay' => 'name',
])); // e.g., "343.20 British pounds"
```

Unit Options

When formatting units, you may use the following options.

unit If the `style` option is *unit*, you must provide a unit identifier as the `unit` property. UTS #35, Part 2, Section 6 defines the core unit identifiers. You may use any unit defined in the CLDR data file.

You may use the following units in these concise forms (without the prefixes defined in CLDR): *acre*, *bit*, *byte*, *celsius*, *centimeter*, *day*, *degree*, *fahrenheit*, *fluid-ounce*, *foot*, *gallon*, *gigabit*, *gigabyte*, *gram*, *hectare*, *hour*, *inch*, *kilobit*, *kilobyte*, *kilogram*, *kilometer*, *liter*, *megabit*, *megabyte*, *meter*, *mile*, *mile-scandinavian*, *milliliter*, *millimeter*, *millisecond*, *minute*, *month*, *ounce*, *percent*, *petabyte*, *pound*, *second*, *stone*, *terabit*, *terabyte*, *week*, *yard*, or *year*.

```
// Displaying hours in Japanese.
$formatphp = new FormatPHP(new Config(new Intl\Locale('ja-JP')));

echo $formatphp->formatNumber(14, new Intl\NumberFormatOptions([
```

(continues on next page)

(continued from previous page)

```
    'style' => 'unit',
    'unit' => 'hour',
    'unitDisplay' => 'long',
])); // e.g., "14 "

// Displaying fluid ounces in French.
$formatphp = new FormatPHP(new Config(new Intl\Locale('fr-FR')));

echo $formatphp->formatNumber(64, new Intl\NumberFormatOptions([
    'style' => 'unit',
    'unit' => 'fluid-ounce',
    'unitDisplay' => 'long',
])); // e.g., "64 onces liquides"
```

unitDisplay How to display the unit. Possible values are *short*, *long*, and *narrow*. The default is *short*.

```
echo $formatphp->formatNumber(14, new Intl\NumberFormatOptions([
    'style' => 'unit',
    'unit' => 'hour',
    'unitDisplay' => 'narrow',
])); // e.g., "14h"
```

Compact Options

If `notation` is *compact*, then you may specify the `compactDisplay` property with the value *short* or *long*. The default is *short*.

```
$formatphp = new FormatPHP(new Config(new Intl\Locale('nl-NL')));

echo $formatphp->formatNumber(123456.789, new Intl\NumberFormatOptions([
    'notation' => 'compact',
    'compactDisplay' => 'long',
])); // e.g., "123 duizend"

echo $formatphp->formatNumber(123456.789, new Intl\NumberFormatOptions([
    'notation' => 'compact',
    'compactDisplay' => 'short',
])); // e.g., "123K"
```

2.6 Copyright

CHAPTER

THREE

INDICES AND TABLES

- genindex
- search