Fun

James W. Dunne

Feb 10, 2022

CONTENTS:

CHAPTER

ONE

TUTORIALS

CHAPTER

TWO

HOW-TO GUIDES

CHAPTER

THREE

FUN API REFERENCE

3.1 Operators

Operators are what you would expect in the mathematical sense e.g addition, subtraction, equality.

Fun provides a large number of operators as PHP functions and callable constants. These are designed to:

1. Accept primitive PHP types
2. Be used as first-class functions

3.1.1 When to use Fun operators

In simple cases, e.g when comparing two values known to be primitive, you should use PHP's built in operators. But, for comparing your own types or Fun types, use these operators.

3.1.2 Using operators

To use them as functions, import as functions:

```
use function Fun\eq;
```

And to use them as first-class values, import as constants:

```
use const Fun\eq;
```

Note: This is because PHP has separate symbol tables for functions and constants. This may change in a near-future version of PHP.

3.1.3 Equivalence

eq

```
eq :: Eq a -> Eq a -> Boolean
```

For PHP primitives, the `eq` operator is equivalent to === such that the following holds:

```
eq(1, 1) && 1 === 1
```

Unlike ===, `eq` compares equivalence of objects that implement the type `Fun\Types\Eq`.

`eq` expects implementations to hold the properties of equivalence relations, namely:

- Reflexivity i.e `eq($x, $x) === true`
- Symmetry i.e `eq($x, $y) === eq($y, $x)`
- Transitivity i.e `!(eq($x, $y) && eq($y, $z)) || eq($x, $z)`

Implementations that *do not* have these properties are errorneous.

neq

```
neq :: Eq a -> Eq a -> Boolean
```

`neq` is defined as the complement of `neq`. It is implemented in kind. Any type that implements `Fun\Types\eq` gains a working `neq` operator.

It is equivalent to PHP's `!==` operator.

3.1.4 Ordering

- lt
- lte
- gt
- gte
- min
- max
- compare

3.1.5 Booleans

- _and
- _or
- _not
- _if
- when
- unless
- complement

3.1.6 Sets

- subset
- proper_subset
- union
- intersect
- diff
- symmetric_diff

3.1.7 Numerical

- add
- sub
- negate
- sum
- product
- mul
- div
- power

3.2 Strings

3.3 Arrays

3.4 Maps

3.5 Sets

CHAPTER

FOUR

API DESIGN

This document describes the design principles and use cases for this library.

4.1 Design Principles

1. *Self-documenting*
2. *Ease of use*
3. *Stability*
4. *Fail-fast*
5. *Example driven*
6. *Clear naming*
7. *Consistency*
8. *Immutability*
9. *Type-safety*

4.1.1 Self-documenting

It should be immediately obvious what an interface does just from looking at:

1. The name
2. Usage

4.1.2 Ease of use

Prioritise ease of use. Favour decisions that maintain or improve how easy an interface is to use.

4.1.3 Stability

Backwards incompatible changes cost the user time. And probably someone's money. Favour backwards compatability.

If we must break BC, do so in a way in which migration can be automated.

In addition, support major versions to allow for said migration.

4.1.4 Fail-fast

PHP is a dynamic language. To fail fast, we must:

1. Support static analysis tools
2. Fail-fast at run-time

This assumes extensive run-time checks of pre-conditions, post-conditions and invariats.

4.1.5 Example driven

We must first produce examples of use cases for an interface before implementing that interface. Those examples then become core to our documentation and drive the implementation.

4.1.6 Clear naming

Naming is crucial. Naming interfaces expands the implementation language. Adhere to consistent naming conventions - treat naming interfaces as the expansion of a language.

Use linguistics to assess and verify naming conventions.

4.1.7 Consistency

PHP evolved organically. And the standard library API shows for it. Fun's raison d'etre is to smooth these inconsistencies.

Consistency forms patterns. The human mind loves patterns. Solid, enforced consistency allows users to predict usage without looking up documentation.

4.1.8 Immutability

Mutability is inherently error-prone and difficult to reason about. A utility belt library should have little need for mutability.

4.1.9 Type-safety

Use static type checkers to:

1. Ensure internal type safety
2. Correct integration with end-user type checking tools

4.2 Use Cases

PHP is a web language. It's commonly used as part of the LAMP stack.

As such, PHP is typically used to:

1. Implement HTTP interfaces, either server-rendered or REST
2. Reading and writing to a database e.g implementing CRUD
3. Render data to some form of document, e.g HTML, JSON, XML, YAML
4. Transform received data for reading from and writing to a database
5. Validating and sanitising received input for safe use
6. Transform fetched data queried from a database for output
7. Validating and sanitising datbase data for safe output
8. Consuming external 3rd party APIs using SDKs and clients
9. Transforming values from internal representation to ones accepted by 3rd party APIs for querying or writing.
10. Transforming received values from 3rd party APIs to internal representations.
11. Validating and sanitising values read from 3rd party APIs
12. Writing tests that interact with DB
13. Writing tests that stub and fake 3rd party services
14. Writing tests that verify data structures

We can reduce these use cases to:

- *Database interaction*
- *HTTP*
- *API Consumption*
- *Template rendering*
- *Serialization*
- *Validation*
- *Sanitisation*
- *Data transformation*
- *Configuration*

4.2.1 Database interaction

For DB interaction, an ORM or DBA layer is typically used. Eloquent, Laravel's ORM, uses the 'active record pattern'. On the other hand, Doctrine uses the data mapper pattern.

4.2.2 HTTP

For defining HTTP interfaces, some form of framework is typically used, allowing the user to define routes and controllers, which respond to consumer requests.

These may be REST routes or implemented to look like static pages.

4.2.3 API Consumption

When interacting with 3rd party APIs, the user typically uses an available SDK or uses a HTTP client.

3rd party SDKs typically accept data in the form of maps. In addition, they return data in maps. Even if objects are used, it is common to support access a la hash map.

Entry-points:

- In response to user input
- In response to a webhook
- In response to a scheduled task

Examples:

1. Leadflo REST API endpoint for actions due
2. Leadflo REST API endpoint for saving a patient
3. Leadflo REST API endpoint for listening on tx type changes
4. IAS Stripe integration on subscription
5. IAS Stripe integration on payment failure
6. IAS Stripe integration on payment success

4.2.4 Template rendering

For server-rendered apps, a templating engine is typically used, as opposed (but not always) to interpolating PHP using tags in HTML documents. Input is typically provided by forms. Output typically interpolates data into a HTML template - using lists and iteration for rendering multiple records.

4.2.5 Serialization

For REST API implementation, JSON is typically used but may support XML. YAML is rarely used to implement REST APIs. Responses are typically restricted to the supported JSON data types - the complex ones being arrays and maps/objects.

In short, REST APIs serialize application data as output. But serialization is not limited to the implementation of REST APIs.

4.2.6 Validation

Validation is often supported by the framework. Frameworks typically provide a means to implement new validation rules. This often leads to string manipulations and regular expression matching and testing.

4.2.7 Sanitisation

Sanitation is often supported by and provided by frameworks. Frameworks typically provide means to implement new sanitisation rules. This involves string manipulation and regular expression matching/replacement.

4.2.8 Data transformation

Transforming values from one format to another typically involve iteration over lists of maps and the transformation of one map into another map. This may also include from one object, such as a domain model object, to a data transfer object or an entity object from a 3rd party SDK.

4.2.9 Configuration

YAML is commonly used for configuration. Symfony uses YAML. But then Symfony allows the ultimate in flexibility and thus supports multiple configuration languages.

Fun makes it easier to write functional code in PHP.

CHAPTER

FIVE

GETTING STARTED

- TODO: Insert getting started instructions here

5.1 Installation

- TODO: Insert installation instructions here

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search