Kadala Documentation

Release 0.1.0

Finlay Beaton

May 15, 2015

Contents

Greetings Nephalem,

Kadala is a PHP 5.4+ library that allows you to easily communicate with the Battle.net Diablo 3 Web API. This library provides a simple PHP class-based interface with methods representing API data, and makes use of features such as compression and caching to optimize your requests.

CHAPTER 1

User manual

1.1 Overview

1.1.1 Requirements

PHP

The Kadala library is tested on the following versions of PHP:

- PHP 5.4
- PHP 5.5
- PHP 5.6
- PHP 7.0
- HHVM

It additionally requires the use of Composer for dependency management.

Dependencies

If you use composer, skip this section as it will automatically download the following additional required libraries for you:

- Guzzle PHP - HTTP Client
- Stash PHP - Caching Client
- Version - Github version helpers

Battle.net API Key

You will need an API key from the Battle.net API Portal.

1. **Register for an account on the *Battle.net API Portal*.** This will be different from your regular battle.net account.
2. **Login on the *Battle.net API Portal*.** This is different from logging into the main Battle.net website or the Battle.net desktop launcher.
3. **Create a new application on the My Applications page.** You will need to know where on the internet your website will function, don't worry if your website is just an information page about your application.

4. **Obtain your key from the *My Applications* page.** You will not need the *secret* listed on other pages, just your *key*. This is your *API key* and should be used whenever you are asked for an `$apiKey` or `'my-bnet-api-key'`.

1.1.2 Installing via Composer

The recommended way to install Kadala is through Composer.

```
# Install Composer
curl -sS https://getcomposer.org/installer | php
```

Next, run the Composer command to install the latest stable version:

```
composer.phar require kadala/kadala
```

After installing, you need to require Composer's autoloader in your code:

```
// yourphpfile.php
// at the very top before everything
//  except namespaces and use statements
require 'vendor/autoload.php';
```

1.1.3 Quickstart

Item retrieval

To obtain the `P1_CruShield_norm_unique_02` item type from the *US* battle.net:

```
use \Kadala\Client;
use \Kadala\Region;

// create a new Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::US);

// retrieve the item data
$itemId = 'P1_CruShield_norm_unique_02';
$data = $client->getItem($itemId);

// $data now contains the item type information
```

Player Career retrieval

List of characters and career information for the european player battle tag wudijo#2228:

```
use \Kadala\Client;
use \Kadala\Region;

// create a new client
$apiKey = 'my-bnet-api-key';
// region must match where the player plays
$client = new Client($apiKey, Region::EUROPE);

// retrieve the career data
$battleTag = 'wudijo#2228';
```

```
$data = $client->getCareer($battleTag);

// $data now contains the career information
```

1.2 Creating a Client

1.2.1 Selecting a Region

Player Career and Hero data is specific to a region, so it is important to connect to the right one.

Here is the list of all valid regions for *Diablo 3*:

- `Region::CHINA`
- `Region::EUROPE`
- `Region::KOREA`
- `Region::TAIWAN`
- `Region::US`

You will need a `Client` for each region you want to access, when in doubt pick your own:

```
use \Kadala\Client;
use \Kadala\Region;

// create a Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::US);
```

You could then call methods on `$client` to look up player data in `Region::US`. Create additional `Client` to access data from different regions at the same time.

1.2.2 Choosing a Locale

It is important to note that Item and player information is provided very differently for each locale. A locale is a language as spoken in a specific country. The information is different even in the case of the same language spoken in two different countries. Battle.net only supports specific locales for each _region_.

The list of locales by region is:

- Region::CHINA
  - `'zh_CN'` - China's Chinese
- Region::EUROPE
  - `'en_GB'` - Great Britain's English
  - `'es_ES'` - Spain's Spanish
  - `'fr_FR'` - France's French
  - `'ru_RU'` - Russia's Russian
  - `'de_DE'` - Germany's German
  - `'pt_PT'` - Portugal's Portuguese
  - `'it_IT'` - Italy's Italian

- Region::KOREA
  - `'ko_KR'` - South Korean's Korean
- Region::TAIWAN
  - `'zh_TW'` - Taiwan's Chinese
- Region::US
  - `'en_US'` - United State's English
  - `'es_MX'` - Mexico's Spanish
  - `'pt_BR'` - Brazil's Portugese

If you do not specify a locale when creating your `Client` (as above), then the first one in this list for the region will be chosen as a default.

Here is an example of creating a `Client` specifying the *Europe* region and *France French* locale:

```
use \Kadala\Client;
use \Kadala\Region;

// create a Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::EUROPE, 'fr_FR');
```

1.3 Get Artisan

You can retrieve general information about an *artisan*, what recipes could be learned, and trained. This is a good way to obtain the names and descriptions for a specific *locale*.

To learn what level a player's artisan is, see *Get Career* instead.

There is currently no way to know what recipes that drop (*rare*, *set* and *legendary*) the player has learned on their *artisans*.

1.3.1 Methods

Available methods on `Client` to get *artisan* data are:

- `getBlacksmith()`
- `getJeweler()`
- `getMystic()`
- `getArtisan(<artisan>)`

There are also constants on the `Artisan` object to provide to `getArtisan()`:

- `BLACKSMITH`
- `JEWELER`
- `MYSTIC`
- `$artisans` array list of all of the above

Special Case: The *mystic* as of *May 2015* does not have any trainable recipes. You cannot guarantee this will always be the case, and if want to make your code robust, it is recommend you handle the future addition of recipes like any other artisan. Certainly the API treats her as such. Until then the `"tiers"` item in the `ArtisanData` structure will always be empty `[]`.

1.3.2 Code Examples

Blacksmith

```
use \Kadala\Client;
use \Kadala\Region;

// create a new Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::US);

// retrieve the blacksmith data
$data = $client->getBacksmith();

// $data now contains the artisan data structure
```

Jeweler

```
use \Kadala\Client;
use \Kadala\Region;

// create a new Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::US);

// retrieve the jeweler data
$data = $client->getJeweler();

// $data now contains the artisan data structure
```

Mystic

```
use \Kadala\Client;
use \Kadala\Region;

// create a new Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::US);

// retrieve the mystic data
$data = $client->getMystic();

// $data now contains the artisan data structure
```

All

You could use the `getArtisan()` method to retrieve them all in quick succession.

```php
use \Kadala\Client;
use \Kadala\Region;
use \Kadala\Artisan;

// create a new Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::US);

// loop through each available artisan
foreach (Artisan::$artisans as $artisan) {
  $data = $client->getArtisan($artisan);

  // $data now contains the artisan data structure
  // do some processing on the data
}
```

1.3.3 Data Structures

For your reference only, not authorative.

In the event that Blizzard changes the data structures returned by the API, Kadala will still work 100%. The library does not do any post-processing and returns the data to you using `json_decode()` only. You will likely have to change your processing code however.

We have listed the Data Structures as we saw them returned by the 'API <https://dev.battle.net/io-docs>'_ in May 2015 for your convenience only.

The data you get back is organized in very specific ways using associative arrays.

Localized strings are marked with `// lc`.

Array items that may repeat are marked with `...`.

Complex array items are sometimes split into their own Data Structure, which may also repeat where marked.

Kadala inserts the `api` key into the results with important request and response parameters.

ArtisanData

```
[
   "slug" => string,
   "name" => string, // lc
   "portrait" => string,
   "training" => [
      "tiers" => [
         [
            "tier" => int,
            "levels" => [
               [
                  "tierLevel" => int,
                  "percent" => int,
                  "trainedRecipes" => [
                     RecipeData,
                     ... // another RecipeData
                  ], // trainedRecipes
                  "upgradeCost" => int // in gold
                  "upgradeItems" => []
               ], // level
```

```
                ... // another level
            ], // levels
        ], // tier
        ... // another tier
    ] // tiers
], // training
"api" => [
    "region" => string,
    "locale" => string,
    "retrieved" => string,
    "lastModified" => string
], // api
]
```

RecipeData

```
[
    "id" => string,
    "slug" => string,
    "name" => string, // lc
    "cost" => int, // in gold
    "reagents" => [
        [
            "quantity" => int,
            "item" => ReagentItemData
        ], // reagent
        ... // another reagent
    ], // reagents
    "itemProduced" => ProducedItemData,
]
```

ReagentItemData

```
[
    "id" => string,
    "name" => string, // lc
    "icon" => string,
    "displayColor" => string,
    "tooltipParams" => string
]
```

ProducedItemData

Because two of `ProducedItemData` properties refer back up to `RecipeData`, the 2nd time the `ProducedItemData` is listed the `recipe` and `craftedBy` properties are omitted to avoid an invite-loop.

```
[
    "id" => string,
    "name" => string, // lc
    "icon" => string,
    "displayColor" => string,
    "tooltipParams" => string,
    "recipe" => RecipeData, // 2nd pass omitted
```

```
    "craftedBy" => RecipeData // 2nd pass omitted
]
```

RecipeData for `Mystic` without `RecipeData`

The *mystic* as of *May 2015* does not have any trainable recipes. It is recommended you deal with the mystic like you would any other artisan. However, it is worth highlighting how simpler her data structure is as a result.

```
[
    "slug" => "mystic",
    "name" => "Mystic", // lc
    "portrait" => "pt_mystic",
    "training" => [
        "tiers" => [ ] // empty for now
    ] // training
]
```

1.4 Get Follower

You can retrieve general information about a *follower*, including a list of their possible skills. This is a good way to obtain the names and descriptions for a specific *locale*.

To learn what specific follower skills and equipment a player uses, see *Get Hero* instead.

1.4.1 Methods

Available methods on `Client` to get *follower* data are:

- `getEnchantress()`
- `getScoundrel()`
- `getTemplar()`
- `getFollower(<follower>)`

There are also constants on the `Follower` object to provide to `getFollower()`:

- `ENCHANTRESS`
- `SCOUNDREL`
- `TEMPLAR`
- `$followers` array list of all of the above

1.4.2 Code Examples

Enchantress

```
use \Kadala\Client;
use \Kadala\Region;

// create a new Client
$apiKey = 'my-bnet-api-key';
```

```php
$client = new Client($apiKey, Region::US);

// retrieve the enchantress data
$data = $client->getEnchantress();

// $data now contains the follower data structure
```

Scoundrel

```php
use \Kadala\Client;
use \Kadala\Region;

// create a new Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::US);

// retrieve the scoundrel data
$data = $client->getScoundrel();

// $data now contains the follower data structure
```

Templar

```php
use \Kadala\Client;
use \Kadala\Region;

// create a new Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::US);

// retrieve the templar data
$data = $client->getTemplar();

// $data now contains the templar data structure
```

All

You could use the `getFollower()` method to retrieve them all in quick succession.

```php
use \Kadala\Client;
use \Kadala\Region;
use \Kadala\Follower;

// create a new Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::US);

// loop through each available follower
foreach (Follower::$followers as $follower) {
  $data = $client->getFollower($follower);

  // $data now contains the follower data structure
  // do some processing on the data
}
```

1.4.3 Data Structures

For your reference only, not authorative.

In the event that Blizzard changes the data structures returned by the API, Kadala will still work 100%. The library does not do any post-processing and returns the data to you using `json_decode()` only. You will likely have to change your processing code however.

We have listed the Data Structures as we saw them returned by the 'API <https://dev.battle.net/io-docs>'_ in May 2015 for your convenience only.

The data you get back is organized in very specific ways using associative arrays.

Localized strings are marked with `// lc`.

Array items that may repeat are marked with `...`.

Complex array items are sometimes split into their own Data Structure, which may also repeat where marked.

Kadala inserts the `api` key into the results with important request and response parameters.

FollowerData

As of *May 2015* the value of `realName` contains something like `d3.follower.templar.realName` which may be a bug.

```
[
   "slug" => string,
   "name" => string, // lc
   "realName" => string, // ?
   "portrait" => string,
   "skills" => [
      "active" => [
         FollowerActiveSkillData,
         ... // another FollowerActiveSkillData
      ], // active
      "passive" => [ ] // not currently used
   ], // skills
   "api" => [
      "region" => string,
      "locale" => string,
      "retrieved" => string,
      "lastModified" => string
   ], // api
]
```

FollowerActiveSkillData

The `description` and `simpleDescription` are displayed to the user depending on the *simple descriptions* in-game option.

```
[
   "slug" => string,
   "name" => string, // lc
   "icon" => string,
   "level" => int,
   "tooltipUrl" => string,
   "description" => string, // lc
```

```
    "simpleDescription" => string, // lc
    "skillCalcId" => string
]
```

1.5 Get Item

You can retrieve general information about an *item*, including some base stats. This is a good way to obtain the names and descriptions for a specific *locale*.

To learn what specific items a player uses, see *Get Hero* instead.

1.5.1 Methods

Available methods on `Client` to get *item* data are:

- `getItem(<itemId>)`

1.5.2 Code Example

Set `$itemId` to the `id` attribute of an item to retrieve it.

```
use \Kadala\Client;
use \Kadala\Region;

// create a new Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::US);

// retrieve the item data
$itemId = 'P1_CruShield_norm_unique_02';
$data = $client->getItem($itemId);

// $data now contains the item data structure
```

1.5.3 Data Structures

For your reference only, not authorative.

In the event that Blizzard changes the data structures returned by the API, Kadala will still work 100%. The library does not do any post-processing and returns the data to you using `json_decode()` only. You will likely have to change your processing code however.

We have listed the Data Structures as we saw them returned by the 'API <https://dev.battle.net/io-docs>'_ in May 2015 for your convenience only.

The data you get back is organized in very specific ways using associative arrays.

Localized strings are marked with `// lc`.

Array items that may repeat are marked with `...`.

Complex array items are sometimes split into their own Data Structure, which may also repeat where marked.

Kadala inserts the `api` key into the results with important request and response parameters.

ItemData

Depending on the type of item requested, some fields may not be present. For example `recipe` is only returned when the item can be crafted. Additionally, some fields only hold data when this structure appears in `getHero()`.

```
[
   "id" => string,
   "name" => string, // lc
   "icon" => string,
   "displayColor" => string,
   "tooltipParams" => string,
   "requiredLevel" => int,
   "itemLevel" => int,
   "bonusAffixes" => int,
   "bonusAffixesMax" => int,
   "accountBound" => bool,
   "flavorText" => string, // lc
   "typeName" => string, // lc
   "type" => [
      "twoHanded" => bool,
      "id" => string,
   ], // type
   "dps" => MinMaxData,
   "attacksPerSecond" => MinMaxData,
   "minDamage" => MinMaxData,
   "maxDamage" => MinMaxData,
   "armor" => MinMaxData,
   "slots" => [
      string,
      ... // another string (slot name)
   ], // slots
   "attributes" => AttributeListData,
   "attributesRaw" => AttributesRawListData,
   "randomAffixes" => [
      "oneOf" => [
         "attributes" => AttributeListData,
         "attributesRaw" => AttributesRawListData,
      ] // oneOf
   ], // randomAffixes
   "gems" => [], // in getHero()
   "socketEffects" => [
      SocketEffectData,
      ... // another SocketEffectData
   ], // socketEffects
   "recipe" => RecipeData,
   "craftedBy" => [
      RecipeData,
      ... // another RecipeData
   ], // craftedBy
   "seasonRequiredToDrop" => int,
   "isSeasonREquiredtoDrop" => bool,
   "blockChance" => MinMaxData,
   "api" => [
      "region" => string,
      "locale" => string,
      "retrieved" => string,
      "lastModified" => string
   ], // api
]
```

AttributeListData

```
[
   "primary" => [
      AttributeData,
      ... // another AttributeData
   ], // primary
   "secondary" => [
      AttributeData,
      ... // another AttributeData
   ], // secondary
   "passive" => [
      AttributeData,
      ... // another AttributeData
   ] // passive
]
```

AttributeData

```
[
   "text" => string, // lc
   "color" => string,
   "affixType" => string
]
```

AttributesRawListData

```
[
   string => MinMaxData,
   ... // another string => MinMaxData pair
]
```

SocketEffectData

```
[
   "attributes" => AttributeListData,
   "itemTypeName" => string, // lc
   "itemTypeId => string,
   "attributesRaw" => AttributesRawListData
]
```

RecipeData

```
[
   "id" => string,
   "slug" => string,
   "name" => string, // lc
   "cost" => int, // in gold
   "reagents" => [
```

```
      [
         "quantity" => int,
         "item" => ReagentItemData
      ], // reagent
      ... // another reagent
   ], // reagents
   "itemProduced" => ProducedItemData,
]
```

ReagentItemData

```
[
   "id" => string,
   "name" => string, // lc
   "icon" => string,
   "displayColor" => string,
   "tooltipParams" => string
]
```

ProducedItemData

Unlike in *artisan* data, `recipe` and `craftedBy` are never listed on this structure.

```
[
   "id" => string,
   "name" => string, // lc
   "icon" => string,
   "displayColor" => string,
   "tooltipParams" => string,
]
```

MinMaxData

```
[
   "min" => float,
   "max" => float
]
```

1.6 Get Career

You can retrieve information about a player, called a career. This includes paragon information, the list and some information about their heroes, and their global account progression.

To learn what specific items and skills a player uses on a hero, see *Get Hero* instead.

There is currently no way to know what recipes that drop (*rare*, *set* and *legendary*) the player has learned on their *artisans*.

Remember that player and hero data is specific to each region. You must specify the correct region for the player you wish to retrieve or you will not get the data.

1.6.1 Methods

Available methods on `Client` to get player *career* data are:

- `getCareer(<battleTag>)`

1.6.2 Code Example

Notice how since `wudijo#2228` plays in *europe*, we had to specify `Region::EUROPE` to our `Client`.

```
use \Kadala\Client;
use \Kadala\Region;

// create a new Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::EUROPE);

// retrieve the item data
$battleTag = 'wudijo#2228';
$data = $client->getCareer($battleTag);

// $data now contains the item data structure
```

1.6.3 Data Structures

For your reference only, not authorative.

In the event that Blizzard changes the data structures returned by the API, Kadala will still work 100%. The library does not do any post-processing and returns the data to you using `json_decode()` only. You will likely have to change your processing code however.

We have listed the Data Structures as we saw them returned by the 'API <https://dev.battle.net/io-docs>'_ in May 2015 for your convenience only.

The data you get back is organized in very specific ways using associative arrays.

Localized strings are marked with `// lc`.

Array items that may repeat are marked with `...`.

Complex array items are sometimes split into their own Data Structure, which may also repeat where marked.

Kadala inserts the `api` key into the results with important request and response parameters.

CareerData

The `"battleTag"` returned will not always be the same as the one requested. There are many situations where a player *battleTag* changes, however requesting their old battleTag retrieves their new profile with the new *battleTag* listed instead.

```
[
   "battleTag" => string,
   "paragonLevel" => int,
   "paragonLevelHardcore" => int,
   "paragonLevelSeason" => int,
   "paragonLevelSeasonHardcore" => int,
   "heroes" => [
```

```
      HeroCareerData,
      ... // another HeroCareerData
   ], // heroes
   "lastHeroPlayed" => int,
   "lastUpdated" => int,
   "kills" => [
      "monsters" => int,
      "elites" => int,
      "hardcoreMonsters" => int
   ], // kills
   "highestHardcoreLevel" => int,
   "timePlayed" => [
      string (class id) => float,
      ... // another string (class id) => float pair
   ], // timePlayed
   "progression" => [
      string (act id) => bool,
      ... // another string (act id) => bool pair
   ], // progression
   "fallenHeroes" => [
      HeroCareerData,
      ... // another HeroCareerData
   ], // fallenHeroes
   string (artisan id+Season+Hardcore) => ArtisanCareerData,
   // another string (artisan id+Season+Hardcore) => ArtisanCareerData pair,
   "seasonalProfiles" => [
      string (season name) => SeasonalProfileData,
      // another string (season name) => SeasonalProfileData pair
   ], // seasonalProfiles
   "api" => [
      "region" => string,
      "locale" => string,
      "retrieved" => string,
      "lastModified" => string
   ], // api
]
```

HeroCareerData

```
[
   "paragonLevel" => int,
   "seasonal" => bool,
   "name" => string,
   "id" => int,
   "level" => int,
   "hardcore" => bool,
   "last-updated" => int,
   "gender" => int,
   "dead" => bool,
   "class" => string
]
```

ArtisanCareerData

```
[
   "slug" => string,
   "level" => string,
   "stepCurrent" => int,
   "stepMax" => int
]
```

SeasonalProfileData

```
[
   "seasonId" => int,
   "paragonLevel" => int,
   "paragonLevelHardcore" => int,
   "kills" => [
      "monsters" => int,
      "elites" => int,
      "hardcoreMonsters" => int
   ], // kills
   "highestHardcoreLevel" => int,
   "timePlayed" => [
      string (class id) => float,
      ... // another string (class id) => float pair
   ], // timePlayed
   "progression" => [
      string (act id) => bool,
      ... // another string (act id) => bool pair
   ] // progression
]
```

1.7 Get Hero

You can retrieve detailed information about a player *hero*, including the skills and gear loadout for themselves and their followers.

For general player account information, like their artisan progress, call *Get Career* instead.

1.7.1 Methods

Available methods on `Client` to get _item_ data are:

- `getHero(<battleTag>, <heroId>)`

1.7.2 Code Example

Notice how since wudijo#2228 plays in *europe*, we had to specify `Region::EUROPE` to our `Client`.

```
use \Kadala\Client;
use \Kadala\Region;

// create a new Client
$apiKey = 'my-bnet-api-key';
```

```php
$client = new Client($apiKey, Region::EUROPE);

// retrieve the item data
$battleTag = 'wudijo#2228';
$heroId = 35436981;
$data = $client->getHero($battleTag, $heroId);

// $data now contains the hero data structure
```

1.7.3 Data Structures

For your reference only, not authorative.

In the event that Blizzard changes the data structures returned by the API, Kadala will still work 100%. The library does not do any post-processing and returns the data to you using `json_decode()` only. You will likely have to change your processing code however.

We have listed the Data Structures as we saw them returned by the 'API <https://dev.battle.net/io-docs>'_ in May 2015 for your convenience only.

The data you get back is organized in very specific ways using associative arrays.

Localized strings are marked with `// lc`.

Array items that may repeat are marked with `...`.

Complex array items are sometimes split into their own Data Structure, which may also repeat where marked.

Kadala inserts the `api` key into the results with important request and response parameters.

HeroData

```
[
   "id" => int (heroId)
   "name" => string,
   "class" => string,
   "gender" => int,
   "level" => int,
   "paragonLevel" => int,
   "hardcore" => bool,
   "seasonal" => bool,
   "seasonCreated" => int,
   "skills" => [
   "active" => [
   HeroActiveSkillData,
      ... // another HeroActiveSkillData
   ], // active
   "passive" => [
      HeroPassiveSkillData,
      ... // another HeroPassiveSkillData
   ], // skills
   "items" => [
      string (slot) => HeroItemData,
      // another string (slot) => HeroItemData,
   ], // items
   "followers" => [
      HeroFollowerData,
      // another HeroFollowerData
```

```
    ],
    "stats" => [
        "life" => int,
        "damage" => float,
        "toughness" => float,
        "healing" => float,
        "attackSpeed" => float,
        "armor" => int,
        "strength" => int,
        "dexterity" => int,
        "vitality" => int,
        "intelligence" => int,
        "physicalResist" => int,
        "fireResist" => int,
        "coldResist" => int,
        "lightningResist" => int,
        "poisonResist" => int,
        "arcaneResist" => int,
        "critDamage" => float,
        "blockChance" => float,
        "blockAmountMin" => int,
        "blockAmountMax" => int,
        "damageIncrease" => float,
        "critChance" => float,
        "damageReduction" => float,
        "thorns" => float,
        "lifeSteal" => float,
        "lifePerKill" => float,
        "goldFind" => float,
        "magicFind" => float,
        "lifeOnHit" => float,
        "primaryResource" => int,
        "secondaryResource" => int
    ], // stats
    "kills" => [
        "elites": int
    ], // kills
    "progression" => [
        string (act) => ActProgressData,
        ... // another string (act) => ActProgressData pair
    ], // progression
    "dead" => bool,
    "last-updated" => int,
    "api" => [
        "region" => string,
        "locale" => string,
        "retrieved" => string,
        "lastModified" => string
    ], // api
]
```

HeroActiveSkillData

The `description` and `simpleDescription` are displayed to the user depending on the *simple descriptions* in-game option.

```
[
   "skill" => [
      "slug" => string,
      "name" => string, // lc
      "icon" => string,
      "level" => int,
      "categorySlug" => string,
      "tooltipUrl" => string,
      "description" => string, // lc
      "simpleDescription" => string, // lc
      "skillCalcId" => string,
   ], // skill
   "rune" => [
      "slug" => string,
      "type" => string,
      "name" => string, // lc
      "level" => int,
      "description" => string, // lc
      "simpleDescription" => string, // lc
      "tooltipParams" => string,
      "skillCalcId" => string,
      "order" => int
   ] // rune
]
```

HeroPassiveSkillData

```
[
   "skill" => [
      "slug" => string,
      "name" => string, // lc
      "icon" => string,
      "level" => int,
      "tooltipUrl" => string,
      "description" => string, // lc
      "flavor" => string, // lc
      "skillCalcId" => string,
   ] // skill
]
```

HeroItemData

Depending on the type of item requested, some fields may not be present. For example `recipe` is only returned when the item can be crafted. When the `transmogItem` property refers to a new `HeroItemData`, `transmogItem` property is omitted on the item it points to. This is a special `tooltipParams` field that contains a string you can use to lookup the particular item attributes, sockets, gems, etc. It is different than the general `tooltipParam` you usually get, including the one from `getItem()`.

```
[
   "id" => string,
   "name" => string, // lc
   "icon" => string,
   "displayColor" => string,
   "tooltipParams" => string,
   "recipe" => RecipeData,
```

```
   "transmogItem" => HeroItemData, // omitted on transmogItem HeroItemData
   "craftedBy" => [
      RecipeData,
      // ... another RecipeData
   ],
]
```

HeroFollowerData

```
[
   "slug" => string,
   "level" => int,
   "items" => [
      string (slot) => HeroItemData,
      ... // another string (slot) => HeroItemData pair
   ], // items
   "stats" => [
      "goldFind" => int,
      "magicFind" => int,
      "experienceBonus" => int
   ], // stats
   "skills" => [
      FollowerActiveSkillData,
      ... // another FollowerActiveSkillData
   ] // skills
]
```

FollowerActiveSkillData

The `description` and `simpleDescription` are displayed to the user depending on the *simple descriptions* in-game option.

```
[
   "slug" => string,
   "name" => string, // lc
   "icon" => string,
   "level" => int,
   "tooltipUrl" => string,
   "description" => string, // lc
   "simpleDescription" => string, // lc
   "skillCalcId" => string
]
```

ActProgressData

```
[
   "completed" => bool,
   "completedQuests" => [
      QuestData,
      ... // another QuestData
   ], // completedQuests
]
```

QuestData

```
[
   "slug" => string,
   "name" => string // lc
]
```

RecipeData

```
[
   "id" => string,
   "slug" => string,
   "name" => string, // lc
   "cost" => int, // in gold
   "reagents" => [
      [
         "quantity" => int,
         "item" => ReagentItemData
      ], // reagent
      ... // another reagent
   ], // reagents
   "itemProduced" => ProducedItemData,
]
```

ReagentItemData

```
[
   "id" => string,
   "name" => string, // lc
   "icon" => string,
   "displayColor" => string,
   "tooltipParams" => string
]
```

ProducedItemData

Unlike in *artisan* data, `recipe` and `craftedBy` are never listed on this structure.

```
[
   "id" => string,
   "name" => string, // lc
   "icon" => string,
   "displayColor" => string,
   "tooltipParams" => string,
]
```

1.8 Set up Caching

One way to speed up your requests to the API is to cache the responses. That way, when you ask for the resource again and it has not changed, the library will return the cached copy. You can use top-of-the-line caching software like memcached or redis, as well as get-started-quickly simpler caches including Filesystem and SQLite.

All cashing is done through the caching library called Stash.

1.8.1 Filesystem cache

Setting the location of the `Filesystem` cache is *optional*. Notice the additional `use` statements.

```
use \Stash\Driver\Filesystem;
use \Stash\Pool;
use \Kadala\Client;
use \Kadala\Region;

// create a new Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::US);

// create the driver for our cache
$driver = new FileSystem();

// optionally set a custom location for the cache
$driver->setOptions([
  'path' => '/tmp/myCache/',
]);

// create and set the cache
$cache = new Pool($driver);
$client->setCache($cache);

// retrieve the item data
$itemId = 'P1_CruShield_norm_unique_02';
$data = $client->getItem($itemId);

// $data now contains the item type information
```

1.8.2 SQLite cache

Setting the location of the `SQLite` cache is *optional*. Notice the additional `use` statements.

```
use \Stash\Driver\Sqlite;
use \Stash\Pool;
use \Kadala\Client;
use \Kadala\Region;

// create a new Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::US);

// create the driver for our cache
$driver = new Sqlite();

// optionally set a custom location for the cache
$driver->setOptions([
  'path' => '/tmp/myCache/',
]);

// create and set the cache
$cache = new Pool($driver);
```

```php
$client->setCache($cache);

// retrieve the item data
$itemId = 'P1_CruShield_norm_unique_02';
$data = $client->getItem($itemId);

// $data now contains the item type information
```

1.9 Advanced Options

1.9.1 Region

You may want to handle retrieving data in multiple regions or locales.

The simplest way to handle this is as recommended elsewhere in the manual, to create additional `Client` objects for each region or locale you want. Treating the `Client` as emutable is the safest course of action, and will result in the fewest bugs in your code.

Nevertheless, Kadala does provides alternatives to advanced users who want to keep the same `Client` object but retrieve data in multiple regions or locales.

Changing the `Client`

You can permanently change the region and locale on the `Client`.

Methods

Here are the methods to do this:

- `setRegion(<region>)`
- `setRegion(<region>, <locale>)`

Examples

```php
use \Kadala\Client;
use \Kadala\Region;

// create a Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::US);

// change the region permanently to Europe
$client->setRegion(Region::EUROPE);
// locale will default to en_CB
```

```php
use \Kadala\Client;
use \Kadala\Region;

// create a Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::US);
```

```
// change the region permanently to Europe and locale to fr_FR
$client->setRegion(Region::EUROPE, 'fr_FR');
```

Changing the request

You can also use a different region or locale only for the specific request you are issuing. This will *not* change the region or locale of the `Client` object itself.

Methods

Each `getX()` method on the `Client` has an optional parameter for *options*, which can contain *region* and *locale*.

So for example, `getBlacksmith()` is optionally `getBlacksmith(<options>)`

For methods already with parameters, just append the options on the end, like `getHero(<battleTag>, <heroId>, <options>)`.

Example

```
use \Kadala\Client;
use \Kadala\Region;

// create a Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::US);

// because of the extra options,
// this request will temporarily return data in fr_FR from EUROPE
$data = $client->getBlacksmith([
   'region' => Region::EUROPE,
   'locale' => 'fr_FR'
);

// this reqequest does not provide the overrides,
// so data will return as normal from US defaulted to en_US
$data = $client->getBlackSmith();
```

Warning

If this all seems a bit confusing, that is because it is. Changing the region and locale on the fly can make your code very confusing, especially if you change it per-request. It can become very hard to know what region and locale you are really requesting data from, and how it will be returned to you, leading to problems in your application.

Nevertheless, this is a powerful feature that when used properly, can simplify your application.

1.9.2 User Cache

Many applications have a data layer on top of a cache. For setting a cache within Kadala, see the *Set up Caching* section.

You may have a database where you store the results from the API. In this case you will want to tell Kadala the `lastModified` for any item you stored on your end.

Simply pass it in like you would to Guzzle,

```
use \Kadala\Client;
use \Kadala\Region;

// create a new Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::EUROPE);

// retrieve the item data
$battleTag = 'wudijo#2228';
$heroId = 35436981;
$data = $client->getHero($battleTag, $heroId, [
   'headers' => [
      'If-Modified-Since' => 'Tue, 07 Apr 2015 16:30:58 GMT',
   ],
]);

// $data now contains the hero data structure if it has changed, or false
```

If you get back `false` then the data has not changed, and you don't need to re-save it.

You can get the `lastModified` in the `api` key of returned data. Some items won't have it, so use the `retrieved` in the `api` key instead.

1.9.3 Guzzle

If you would like to specify custom options to guzzle, the library that retrives your request from the api, you may do so in the `options` array to any request method.

```
use \Kadala\Client;
use \Kadala\Region;

// create a new Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::EUROPE);

// retrieve the item data
$battleTag = 'wudijo#2228';
$heroId = 35436981;
$data = $client->getHero($battleTag, $heroId, [
   'headers' => [
      'User-Agent' => 'My Website',
   ],
]);

// $data now contains the hero data structure
```

1.10 Frequently Asked Questions

1.10.1 Questions

Contents

1.10.2 How do I turn off compression?

By default all communication from Battle.net to the library is compressed. Data is always returned to you fromt he library in an uncompressed state. In vary rare circumstances it can be useful to turn off compressing responses. If you choose to do this, you may see *significant slowdowns* of especially larger responses. The `setSendCompressed` method allows you to turn off compression:

```
use \Kadala\Client;
use \Kalada\Region;

// create a new Client
$apiKey = 'my-bnet-api-key';
$client = new Client($apiKey, Region::US);

// turn off compression
$client->setSendCompressed(false);

// all method calls including `getItem`
//  will communicate with bnet uncompressed
$itemId = 'P1_CruShield_norm_unique_02';
$data = $client->getItem($itemId);

// $data now contains the item type information
```

1.10.3 How do I change the region or locale?

The easiest way is to create a new `Client` object for each region or locale you want.

For more complex uses, see *Advanced Options* section.

1.10.4 I have a database, can I tell Kadala the `lastModified`?

While it's recommended you use our *Set up Caching* feature for temporarily storing requests, a database is a good use case for a user cache.

See the *Advanced Options* section on user caches.

1.10.5 How do I report an issue?

Bug reports and feature requests can be submitted using the Github Issue Tracker.

1.10.6 How do I support you?

Hi, I'm Finlay Beaton (@ofbeaton. This software is only made possible by donations of fellow users like you, encouraging me to toil the midnight hours away and sweat into the code and documentation. Everyone's time should be valuable, please seriously consider donating.

Website Paypal Donate Button | Paypal Email Money Form

1.10.7 How do I get involved?

I'm glad you asked! People interested in getting involved in the development of the library should head to the *Contributing* section of the manual.

1.11 Examples

The `examples/` directory of the library contains stand-alone programs you can run and inspect to learn how to use the library. They are useful if you're looking for complete *A-Z* examples.

1.11.1 Basic Client

In `examples/basic client` is `client.php`.

This program will retrieve all API endpoints and write each to a `.json` file.

Run it using the command: `php client.php my-bnet-api-key`

1.12 Contributing

We're glad you are considering contributing to this project. It takes a lot of work to keep the library up to date, and any help is appreciated.

1.12.1 Where to start

- Report a bug or feature improvement in our issue tracker
- Submit an improvement to the documentation
- Pick an issue and send us a pull request with code to fix it

If you decide to contribute code, please ensure that you add the appropriate unit tests that covers your change. The project must also build properly.

1.12.2 Building the project

```
php composer.phar install
bin/phing dist
bin/phing all
```

If you are fixing an issue, you **MUST** submit an accompanying unit test that fails in the current version, but passes with your fix.

Code quality is important, and things that must run without errors:

- `phpfcb` must *successfully* beautify your code
- `phpcs` must report **Zero** Code Style *errors* or *warnings*. See `build/browser/index.html`
- `phpunit` unit tests must run with **Zero** errors.
- `humbug` mutation testing must pass with **Zero** errors.
- `build/coverage/index.html` must report **100%** code coverage.
- `apigen` and `apigen-dev` must *successfully* generate documentation.

Good luck in your adventures, Nephalem!