Firebase Admin SDK for PHP

Release

Jan 23, 2018

Contents

Interact with Google Firebase from your PHP application.

The source code can be found at https://github.com/kreait/firebase-php/

```
<?php

require __DIR__.'/vendor/autoload.php';

use Kreait\Firebase\Factory;
use Kreait\Firebase\ServiceAccount;

$serviceAccount = ServiceAccount::fromJsonFile(__DIR__.'/google-service-account.json
↪');

$firebase = (new Factory)
    ->withServiceAccount($serviceAccount)
    ->withDatabaseUri('https://my-project.firebaseio.com')
    ->create();

$database = $firebase->getDatabase();

$newPost = $database
    ->getReference('blog/posts')
    ->push([
        'title' => 'Post title',
        'body' => 'This should probably be longer.'
    ]);

$newPost->getKey(); // => -KVr5eu8gcTv7_AHb-3-
$newPost->getUri(); // => https://my-project.firebaseio.com/blog/posts/-KVr5eu8gcTv7_
↪AHb-3-

$newPost->getChild('title')->set('Changed post title');
$newPost->getValue(); // Fetches the data from the realtime database
$newPost->remove();
```

CHAPTER 1

User Guide

1.1 Overview

1.1.1 Requirements

- PHP >= 7.0
- The mbstring PHP extension
- A Firebase project - create a new project in the Firebase console, if you don't already have one.
- A Google service account, follow the instructions in the official Firebase Server documentation and place the JSON configuration file somewhere in your project's path.

1.1.2 Installation

The recommended way to install the Firebase Admin SDK is with Composer. Composer is a dependency management tool for PHP that allows you to declare the dependencies your project needs and installs them into your project.

```
# Install Composer
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

You can add the Firebase Admin SDK as a dependency using the composer.phar CLI:

```
php composer.phar require kreait/firebase-php ^3.0
```

Alternatively, you can specify the Firebase Admin SDK as a dependency in your project's existing composer.json file:

```
 {
   "require": {
      "kreait/firebase-php": "^3.0"
```

```
    }
}
```

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

```
<?php

require __DIR__.'/vendor/autoload.php';
```

You can find out more on how to install Composer, configure autoloading, and other best-practices for defining dependencies at getcomposer.org.

1.1.3 Issues/Support

- Github issue tracker
- Join the Firebase Community Slack at https://firebase-community.appspot.com, join the #php channel and look for @jeromegamez.

1.1.4 Roadmap

The following planned features are not in a particular order:

- Integration of Firebase Storage
- Automatic updates of Firebase Rules
  - Background: Data must be indexed to be queriable or sortable. If you try to query a yet unindexed dataset, the Firebase REST API will return an error. With this feature, the SDK could execute an error, and if an error occurs, update the Firebase Rules as needed and retry.
- Support for listening to the Firebase event stream
- PHP Object Serialization and Deserialization
- Use parallel requests where possible to speed up operations

1.1.5 License

Licensed using the MIT license.

1.1.6 Contributing

Guidelines

1. The SDK utilizes PSR-1, PSR-2, PSR-4, and PSR-7.
2. This SDK has a minimum PHP version requirement of PHP 7.0. Pull requests must not require a PHP version greater than PHP 7.0 unless the feature is only utilized conditionally.
3. All pull requests must include unit tests to ensure the change works as expected and to prevent regressions.

Running the tests

The SDK is unit tested with PHPUnit. Run the tests using the Makefile:

```
make tests
```

Coding standards

The SDK uses the PHP Coding Standars Fixer to ensure a uniform coding style. Apply coding standard fixed using the Makefile:

```
make cs
```

from the root of the project.

1.1.7 Acknowledgements

- The structure and wording of this documentation is loosely based on the official Firebase documentation at https://firebase.google.com/docs/.
- The index and overview page are adapted from Guzzle's documentation.

1.2 Setup

1.2.1 Google Service Account

In order to access a Firebase project using a server SDK, you must authenticate your requests to Firebase with a Service Account.

Follow the steps described in the official Firebase documentation to create a Service Account for your Firebase application (see Add the Firebase Admin SDK to your Server) and make sure the Service Account has the *Project -> Editor* or *Project -> Owner* role.

With autodiscovery

By default, the SDK is able to autodiscover the Service Account for your project in the following conditions:

1. The path to the JSON key file is defined in one of the following environment variables
   - `FIREBASE_CREDENTIALS`
   - `GOOGLE_APPLICATION_CREDENTIALS`
2. The JSON Key file is located in Google's "well known path"
   - on Linux/MacOS: `$HOME/.config/gcloud/application_default_credentials.json`
   - on Windows: `$APPDATA/gcloud/application_default_credentials.json`

If one of the conditions above is met, creating a new Firebase instance is as easy as this:

```php
use Kreait\Firebase\Factory;

$firebase = (new Factory)->create();
```

A more explicit alternative:

```php
use Kreait\Firebase\Factory;
use Kreait\Firebase\ServiceAccount;

$serviceAccount = ServiceAccount::discover();

$firebase = (new Factory)
    ->withServiceAccount($serviceAccount)
    ->create();
```

Manually

You can also pass the path to the Service Account JSON file explicitly:

```php
use Kreait\Firebase\Factory;
use Kreait\Firebase\ServiceAccount;

$serviceAccount = ServiceAccount::fromJsonFile(__DIR__.'/firebase_credentials.json');
$firebase = (new Factory)
    ->withServiceAccount($serviceAccount)
    ->create();
```

Use your own autodiscovery

You can use your own, custom autodiscovery methods as well:

```php
use Kreait\Firebase\Factory;
use Kreait\Firebase\ServiceAccount\Discoverer

$discoverer = new Discoverer([
    function () {
        $serviceAccount = ...; // Instance of Kreait\Firebase\ServiceAccount

        return $serviceAccount;
    }
]);

$firebase = (new Factory)
    ->withServiceAccountDiscoverer($myDiscoverer)
    ->create();
```

1.2.2 Custom Database URI

If the project ID in the JSON file does not match the URL of your Firebase application, or if you want to be explicit, you can configure the Factory like this:

```
use Kreait\Firebase\Factory;

$firebase = (new Factory)
    ->withDatabaseUri('https://my-project.firebaseio.com')
    ->create();
```

1.2.3 Enable user management features

To be able to use user management features, you have to provide the Firebase Web API key to the factory. You can find the key in the settings area of your Firebase project.

```
use Kreait\Firebase\Factory;
use Kreait\Firebase\ServiceAccount;

$serviceAccount = ServiceAccount::fromJsonFile(__DIR__.'/google-service-account.json
↪');

$firebase = (new Factory)
    ->withServiceAccount($serviceAccount)
    ->create();
```

1.3 Realtime Database

You can work with your Firebase application's Realtime Database by invoking the `getDatabase()` method of your Firebase instance:

```
use Kreait\Firebase;

$firebase = (new Firebase\Factory())->create();
$database = $firebase->getDatabase();
```

1.3.1 Retrieving data

Every node in your database can be accessed through a Reference:

```
$reference = $database->getReference('path/to/child/location');
```

Note: Creating a reference does not result in a request to your Database. Requests to your Firebase applications are executed with the `getSnapshot()` and `getValue()` methods only.

You can then retrieve a Database Snapshot for the Reference or its value directly:

```
$snapshot = $reference->getSnapshot();

$value = $snapshot->getValue();
// or
$value = $reference->getValue();
```

Database Snapshots

Database Snapshots are immutable copies of the data at a Firebase Database location at the time of a query. The can't be modified and will never change.

```
$snapshot = $reference->getSnapshot();
$value = $snapshot->getValue();

$value = $reference->getValue(); // Shortcut for $reference->getSnapshot()->
↪getValue();
```

Snapshots provide additional methods to work with and analyze the contained value:

- `exists()` returns true if the Snapshot contains any (non-null) data.
- `getChild()` returns another Snapshot for the location at the specified relative path.
- `getKey()` returns the key (last part of the path) of the location of the Snapshot.
- `getReference()` returns the Reference for the location that generated this Snapshot.
- `getValue()` returns the data contained in this Snapshot.
- `hasChild()` returns true if the specified child path has (non-null) data.
- `hasChildren()` returns true if the Snapshot has any child properties, i.e. if the value is an array.
- `numChildren()` returns the number of child properties of this Snapshot, if there are any.

Queries

You can use Queries to filter and order the results returned from the Realtime Database. Queries behave exactly like References. That means you can execute any method on a Query that you can execute on a Reference.

Note: You can combine every filter query with every order query, but not multiple queries of each type. Shallow queries are a special case: they can not be combined with any other query method.

Shallow queries

This is an advanced feature, designed to help you work with large datasets without needing to download everything. Set this to true to limit the depth of the data returned at a location. If the data at the location is a JSON primitive (string, number or boolean), its value will simply be returned.

If the data snapshot at the location is a JSON object, the values for each key will be truncated to true.

Detailed information can be found on the official Firebase documentation page for shallow queries

```
$db->getReference('currencies')
    // order the reference's children by their key in ascending order
    ->shallow()
    ->getSnapshot();
```

A convenience method is available to retrieve the key names of a reference's children:

```
$db->getReference('currencies')->getChildKeys(); // returns an array of key names
```

Ordering data

The official Firebase documentation explains How data is ordered.

Data is always ordered in ascending order.

You can only order by one property at a time - if you try to order by multiple properties, e.g. by child and by value, an exception will be thrown.

By key

```
$db->getReference('currencies')
    // order the reference's children by their key in ascending order
    ->orderByKey()
    ->getSnapshot();
```

By value

Note: In order to order by value, you must define an index, otherwise the Firebase API will refuse the query.

```
{
    "currencies": {
        ".indexOn": ".value"
    }
}
```

```
$db->getReference('currencies')
    // order the reference's children by their value in ascending order
    ->orderByValue()
    ->getSnapshot();
```

By child

Note: In order to order by a child value, you must define an index, otherwise the Firebase API will refuse the query.

```
{
    "people": {
        ".indexOn": "height"
    }
}
```

```
$db->getReference('people')
    // order the reference's children by the values in the field 'height' in␣
↪ascending order
    ->orderByChild('height')
    ->getSnapshot();
```

Filtering data

To be able to filter results, you must also define an order.

limitToFirst

```
$db->getReference('people')
    // order the reference's children by the values in the field 'height'
    ->orderByChild('height')
    // limits the result to the first 10 children (in this case: the 10 shortest␣
↪persons)
    // values for 'height')
    ->limitToFirst(10)
    ->getSnapshot();
```

limitToLast

```
$db->getReference('people')
    // order the reference's children by the values in the field 'height'
    ->orderByChild('height')
    // limits the result to the last 10 children (in this case: the 10 tallest␣
↪persons)
    ->limitToLast(10)
    ->getSnapshot();
```

startAt

```
$db->getReference('people')
    // order the reference's children by the values in the field 'height'
    ->orderByChild('height')
    // returns all persons taller than or exactly 1.68 (meters)
    ->startAt(1.68)
    ->getSnapshot();
```

endAt

```
$db->getReference('people')
    // order the reference's children by the values in the field 'height'
    ->orderByChild('height')
    // returns all persons shorter than or exactly 1.98 (meters)
    ->endAt(1.98)
    ->getSnapshot();
```

equalTo

```
$db->getReference('people')
    // order the reference's children by the values in the field 'height'
    ->orderByChild('height')
    // returns all persons being exactly 1.98 (meters) tall
    ->equalTo(1.98)
    ->getSnapshot();
```

1.3.2 Saving data

Set/replace values

For basic write operations, you can use set() to save data to a specified reference, replacing any existing data at that path. For example a configuration array for a website might be set as follows:

```
$db->getReference('config/website')
   ->set([
       'name' => 'My Application',
       'emails' => [
           'support' => 'support@domain.tld',
           'sales' => 'sales@domain.tld',
       ],
       'website' => 'https://app.domain.tld',
      ]);

$db->getReference('config/website/name')->set('New name');
```

Note: Using `set()` overwrites data at the specified location, including any child nodes.

Update specific fields[1]

To simultaneously write to specific children of a node without overwriting other child nodes, use the update() method.

When calling `update()`, you can update lower-level child values by specifying a path for the key. If data is stored in multiple locations to scale better, you can update all instances of that data using data fan-out.

For example, in a blogging app you might want to add a post and simultaneously update it to the recent activity feed and the posting user's activity feed using code like this:

[1] This example and its description is the same as in the official documentation: Update specific fields.

```php
$uid = 'some-user-id';
$postData = [
    'title' => 'My awesome post title',
    'body' => 'This text should be longer',
];

// Create a key for a new post
$newPostKey = $db->getReference('posts')->push()->getKey();

$updates = [
    'posts/'.$newPostKey => $postData,
    'user-posts/'.$uid.'/'.$newPostKey => $postData,
];

$db->getReference() // this is the root reference
   ->update($updates);
```

Writing lists[2]

Use the `push()` method to append data to a list in multiuser applications. The `push()` method generates a unique key every time a new child is added to the specified Firebase reference. By using these auto-generated keys for each new element in the list, several clients can add children to the same location at the same time without write conflicts. The unique key generated by `push()` is based on a timestamp, so list items are automatically ordered chronologically.

You can use the reference to the new data returned by the `push()` method to get the value of the child's auto-generated key or set data for the child. The `getKey()` method of a `push()` reference contains the auto-generated key.

```php
$postData = [...];
$postRef = $db->getReference('posts')->push($postData);

$postKey = $postRef->getKey(); // The key looks like this: -KVquJHezVLf-lSye6Qg
```

Server values

Server values can be written at a location using a placeholder value which is an object with a single *.sv* key. The value for that key is the type of server value you wish to set.

Firebase currently supports only one server value: `timestamp`. You can either set it manually in your write operation, or use a constant from the `Firebase\Database` class.

The following to usages are equivalent:

```php
$ref = $db->getReference('posts/my-post')
          ->set('created_at', ['.sv' => 'timestamp']);

$ref = $db->getReference('posts/my-post')
          ->set('created_at', Database::SERVER_TIMESTAMP);
```

Delete data[3]

The simplest way to delete data is to call remove() on a reference to the location of that data.

[2] This example and its description is the same as in the official documentation: Append to a list of data.
[3] This example and its description is the same as in the official documentation: Delete data.

```
$db->getReference('posts')->remove();
```

You can also delete by specifying null as the value for another write operation such as *set()* or *update()*.

```
$db->getReference('posts')->set(null);
```

You can use this technique with *update()* to delete multiple children in a single API call.

1.3.3 Debugging API exceptions

When a request to Firebase fails, the SDK will throw a `\Kreait\Firebase\Exception\ApiException` that includes the sent request and the received response object:

```
try {
    $db->getReference('forbidden')->getValue();
} catch (ApiException $e) {
    /** @var \Psr\Http\Message\RequestInterface $request */
    $request = $e->getRequest();
    /** @var \Psr\Http\Message\ResponseInterface|null $response */
    $response = $e->getResponse();

    echo $request->getUri().PHP_EOL;
    echo $request->getBody().PHP_EOL;

    if ($response) {
        echo $response->getBody();
    }
}
```

1.3.4 Database rules

Learn more about the usage of Firebase Realtime Database Rules in the official documentation.

```
use Kreait\Firebase\Database\RuleSet;

// The default rules allow full read and write access to authenticated users of your␣
↪app
$ruleSet = RuleSet::default();

// This level of access means anyone can read or write to your database. You should
// configure more secure rules before launching your app.
$ruleSet = RuleSet::public();

// Private rules disable read and write access to your database by users.
// With these rules, you can only access the database through the
// Firebase console and the Admin SDKs.
$ruleSet = RuleSet::private();

// You can of course define custom rules
$ruleSet = RuleSet::fromArray(['rules' => [
    '.read' => true,
    '.write' => false,
    'users' => [
        '$uid' => [
            '.read' => '$uid === auth.uid',
            '.write' => '$uid === auth.uid',
        ]
    ]
]]);

$db->updateRules($ruleSet);

$freshRuleSet = $db->getRules(); // Returns a new RuleSet instance
$actualRules = $ruleSet->getRules(); // returns an array
```

1.4 Authentication[1]

Before you can access the Firebase Realtime Database from a server using the Firebase Admin SDK, you must authenticate your server with Firebase. When you authenticate a server, rather than sign in with a user account's credentials as you would in a client app, you authenticate with a service account which identifies your server to Firebase.

You can get two different levels of access when you authenticate using the Firebase Admin SDK:

Administrative privileges: Complete read and write access to a project's Realtime Database. Use with caution to complete administrative tasks such as data migration or restructuring that require unrestricted access to your project's resources.

Limited privileges: Access to a project's Realtime Database, limited to only the resources your server needs. Use this level to complete administrative tasks that have well-defined access requirements. For example, when running a summarization job that reads data across the entire database, you can protect against accidental writes by setting a read-only security rule and then initializing the Admin SDK with privileges limited by that rule.

1.4.1 Authenticate with admin privileges

When you initialize the Firebase Admin SDK with the credentials for a service account with the Editor role on your Firebase project, that instance has complete read and write access to your project's Realtime Database.

```
use Kreait\Firebase\Factory;
use Kreait\Firebase\ServiceAccount;

$serviceAccount = ServiceAccount::fromJsonFile(__DIR__.'/google-service-account.json
↪');

$firebase = (new Factory)
    ->withServiceAccount($serviceAccount)
    ->create();
```

Note: Your service only has as much access as the service account used to authenticate it. For example, you can limit your service to read-only by using a service account with the Reader role on your project. Similarly, a service account with no role on the project is not able to read or write any data.

1.4.2 Authenticate with limited privileges

As a best practice, a service should have access to only the resources it needs.

[1] Google: Introduction to the Admin Database API

To get more fine-grained control over the resources a Firebase app instance can access, use a unique identifier in your Security Rules to represent your service.

Then set up appropriate rules which grant your service access to the resources it needs. For example:

```
{
  "rules": {
    "public_resource": {
      ".read": true,
      ".write": true
    },
    "some_resource": {
      ".read": "auth.uid === 'my-service-worker'",
      ".write": false
    },
    "another_resource": {
      ".read": "auth.uid === 'my-service-worker'",
      ".write": "auth.uid === 'my-service-worker'"
    }
  }
}
```

Then, on your server, when you initialize the Firebase app, use the `asUser($uid)` method with the identifier you used to represent your service in your Security Rules.

```
use Kreait\Firebase\Factory;
use Kreait\Firebase\ServiceAccount;

$serviceAccount = ServiceAccount::fromJsonFile(__DIR__.'/google-service-account.json
↪');

$firebase = (new Factory)
    ->withServiceAccount($serviceAccount)
    ->asUser('my-service-worker')
    ->create();
```

1.4.3 Create custom tokens[2]

The Firebase Admin SDK has a built-in method for creating custom tokens. At a minimum, you need to provide a uid, which can be any string but should uniquely identify the user or device you are authenticating. These tokens expire after one hour.

```
$uid = 'some-uid';

$customToken = $firebase->getAuth()->createCustomToken($uid);
```

You can also optionally specify additional claims to be included in the custom token. For example, below, a premiumAccount field has been added to the custom token, which will be available in the auth / request.auth objects in your Security Rules:

```
$uid = 'some-uid';
$additionalClaims = [
    'premiumAccount' => true
];
```

[2] Google: Create custom tokens

```
$customToken = $firebase->getAuth()->createCustomToken($uid, $additionalClaims);
```

1.4.4 Verify a Firebase ID Token[3]

If a Firebase client app communicates with your server, you might need to identify the currently signed-in user. To do so, verify the integrity and authenticity of the ID token and retrieve the uid from it. You can use the uid transmitted in this way to securely identify the currently signed-in user on your server.

Note: Many use cases for verifying ID tokens on the server can be accomplished by using Security Rules for the Firebase Realtime Database and Cloud Storage. See if those solve your problem before verifying ID tokens yourself.

Warning: The ID token verification methods included in the Firebase Admin SDKs are meant to verify ID tokens that come from the client SDKs, not the custom tokens that you create with the Admin SDKs. See Auth tokens for more information.

Use `Auth::verifyIdToken()` to verify an ID token:

```
use Kreait\Firebase\Exception\Auth\InvalidIdToken;

$idTokenString = '...';

try {
    $verifiedIdToken = $firebase->getAuth()->verifyIdToken($idTokenString);
} catch (InvalidIdToken $e) {
    echo $e->getMessage();
}
```

References

1.5 User management

You can enable user management features by providing your project's web API key to the Firebase factory and getting an `Auth` instance:

```
use Kreait\Firebase\Factory;
use Kreait\Firebase\ServiceAccount;

$serviceAccount = ServiceAccount::fromJsonFile(__DIR__.'/google-service-account.json
↪');

$firebase = (new Factory)
    ->withServiceAccount($serviceAccount)
    ->create();

$auth = $firebase->getAuth();
```

[3] Google: Verify ID Tokens

1.5.1 List users

To enhance performance and prevent memory issues when retrieving a huge amount of users, this methods returns a Generator.

```
$users = $auth->listUsers($defaultMaxResults = 1000, $defaultBatchSize = 1000);

foreach ($users as $user) {
    print_r($user);
}
// or
array_map(function (array $userData) {
    print_r($userData);
}, iterator_to_array($users));
```

1.5.2 Get information about a specific user

```
$userInfo = $auth->getUserInfo('some-uid');
```

1.5.3 Create an anonymous user

```
$auth->createAnonymousUser();
```

1.5.4 Create a user with email and password

```
$auth->createUserWithEmailAndPassword('user@domain.tld', 'a secure password');
```

1.5.5 Change a user's password

```
$uid = 'some-uid';

$updatedUser = $auth->changeUserPassword($uid, 'new password');
```

1.5.6 Change a user's email

```
$uid = 'some-uid';

$updatedUser = $auth->changeUserEmail($uid, 'user@domain.tld');
```

1.5.7 Disable a user

```
$uid = 'some-uid';

$updatedUser = $auth->disableUser($uid);
```

1.5.8 Enable a user

```
$uid = 'some-uid';

$updatedUser = $auth->enableUser($uid);
```

1.5.9 Delete a user

```
$uid = 'some-uid';

$auth->deleteUser($uid);
```

1.5.10 Send a password reset email

```
$email = 'user@domain.tld';

$auth->sendPasswordResetEmail($email);
```

1.5.11 Invalidate user sessions[1]

This will revoke all sessions for a specified user and disable any new ID tokens for existing sessions from getting minted. **Existing ID tokens may remain active until their natural expiration (one hour).** To verify that ID tokens are revoked, use `Auth::verifyIdToken()` with the second parameter set to `true`.

If the check fails, a `RevokedIdToken` exception will be thrown.

```
use Kreait\Firebase\Exception\Auth\RevokedIdToken;

$idTokenString = '...';

$verifiedIdToken = $firebase->getAuth()->verifyIdToken($idTokenString);

$uid = $verifiedIdToken->getClaim('sub');

$firebase->getAuth()->revokeRefreshTokens($uid);

try {
    $verifiedIdToken = $firebase->getAuth()->verifyIdToken($idTokenString, true);
} catch (RevokedIdToken $e) {
    echo $e->getMessage();
}
```

[1] Google: Revoke refresh tokens

References

1.6 Troubleshooting

1.6.1 Call to undefined function `openssl_sign()`

You need to install the OpenSSL PHP Extension: http://php.net/openssl

1.6.2 cURL error XX: SSL certificate validation failed

If you receive the above error, make sure that you have a current CA Root Certificates bundle on your system and that PHP uses it.

To see where PHP looks for the CA bundle, check the output of the following command:

```
var_dump(openssl_get_cert_locations());
```

which should lead to an output similar to this:

```
array(8) {
    'default_cert_file' =>
    string(32) "/usr/local/etc/openssl/cert.pem"
    'default_cert_file_env' =>
    string(13) "SSL_CERT_FILE"
    'default_cert_dir' =>
    string(29) "/usr/local/etc/openssl/certs"
    'default_cert_dir_env' =>
    string(12) "SSL_CERT_DIR"
    'default_private_dir' =>
    string(31) "/usr/local/etc/openssl/private"
    'default_default_cert_area' =>
    string(23) "/usr/local/etc/openssl"
    'ini_cafile' =>
    string(0) ""
    'ini_capath' =>
    string(0) ""
}
```

Now check if the file given in the `default_cert_file` field actually exists. Create a backup of the file, download the current CA bundle from https://curl.haxx.se/ca/cacert.pem and put it where `default_cert_file` points to.

If the problem still occurs, another possible solution is to configure the `curl.cainfo` setting in your `php.ini`:

```
[curl]
curl.cainfo = /absolute/path/to/cacert.pem
```

1.7 Migration

1.7.1 3.1 to 3.2

KreaitFirebase::getTokenHandler() has been deprecated

Use `Kreait\Firebase\Auth::createCustomToken()` and `Kreait\Firebase\Auth::verifyIdToken()` instead.

```
# Before
$tokenHandler = $firebase->getTokenHandler();

$tokenHandler->createCustomToken(...);
$tokenHandler->verifyIdToken(...);

# After
$auth = $firebase->getAuth();

$auth->createCustomToken(...);
$auth->verifyIdToken(...);
```

1.7.2 3.0 to 3.1

KreaitFirebaseFactory::withCredentials() has been deprecated

```
# Before
use Kreait\Firebase\Factory;

$firebase = (new Factory)
    ->withCredentials(__DIR__.'/google-service-account.json');

# After
use Kreait\Firebase\Factory;
use Kreait\Firebase\ServiceAccount;

$serviceAccount = ServiceAccount::fromJsonFile(__DIR__.'/google-service-account.json
↪');
$firebase = (new Firebase\Factory)
    ->withServiceAccount($serviceAccount);
```

1.7.3 2.x to 3.0

Database secret authentication

As Database Secret based authentication has been deprecated by Firebase, it has been removed from this library. Use Service Account based authentication instead.

Firebase Factory

Previously, it was possible to create a new Firebase instance with a convenience class in the root namespace. This class has been removed, and `Kreait\Firebase\Factory` is used instead:

```
# Before
$firebase = \Firebase::fromServiceAccount('/path/to/google-service-account.json');

# After
use Kreait\Firebase\Factory;

$firebase = (new Factory())
    ->withCredentials('/path/to/google-service-account.json')
    ->create();
```

Changed namespace

All classes have been moved from the `Firebase` root namespace to `Kreait\Firebase` to avoid conflicts with official Firebase PHP libraries using this namespace.