bepado SDK API Documentation

This document describes how to use the SDK in a shop, or write an extension / module for a shop system using the API.

Requirements

The bepado SDK has the following requirements:

- PHP 5.3+
- A database (PDO MySQL and MySQLi drivers are currently included, custom implementations are possible)

Basic Concepts

The basic idea is, that a shop exports a set of products, which are then synchronized with bepado. Other shops may import those products and then customers in those shops may buy these products. This results in three basic operations:

- Product Export

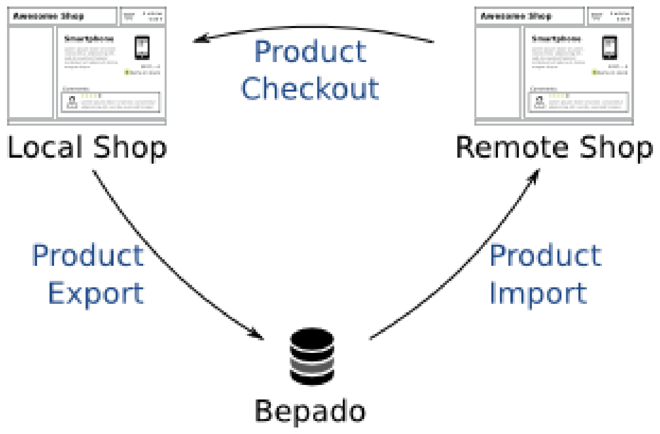
  A set of local products is exported to bepado. This allows other shops to subscribe and sell in their own shops.

- Product Import

  A set of products, which the shop owner subscribed to in bepado are made available as products in the local shop.

- Product Checkout

  Another shops sells your product in their remote shop, and the transaction is acknowledged and completed in your local shop.



Each shop can act as both an importer or exporter of products or just in one of those roles. Product Export and Product Import are both optional.

Using The SDK

Using the SDK in your shop requires you to do three things:

1. Create an instance of the SDK

   When creating a new instance of the SDK to use in your shop, it requires several parameters. Find more about these parameters below.

2. Create a service endpoint URL for the SDK.

3. Call methods on the SDK depending on the actions inside your shop.

Construct the SDK

The SDK requires several parameters. The parameters in more detail:

- The `$apiKey`

  You get the API key for your shop instance after registering with the bepado website. If you develop a SDK for a certain e-commerce system you probably want to provide the shop owner with a simple user interface to configure the API key to use and pass this one to the SDK.

  The `$apiKey` will be validated on the first usage of the SDK.

  The API key is like a password and has be kept secret to outside parties. You should never give the bepado API key to anyone, not even bepado support staff.

- The `$apiEndpointUrl`

  The URL MUST be an absolute URL under which your shop can be reached from bepado. You extension must handle requests to this URL and pass the data on to the SDK. More on that below.

- The `Gateway`

  The `Gateway` is used by the SDK to store various data. There are default implementation using the PHP `mysqli` and `PDO` with MySQL extensions, which you might want to use. If your shop does not use these databases you should implement a custom `Gateway` for this. If you do this, we would love to this contributed back into the SDK.

- The `ProductToShop` interface

  This interface is used when products are imported into your shop. You must implement this interface to work with your shop. The interface is fairly simple and contains two methods. One to create or update a remote product, and one to remove a remote product again.

  ```
  <?php
  /**
   * This file is part of the Bepado SDK Component.
   *
   * @version $Revision$
   */
  ```

```
namespace Bepado\SDK;

/**
 * Interface for product importers
 *
 * Implement this interface with shop specific details to update products in
 * your shop database, which originate from bepado.
 *
 * @version $Revision$
 * @api
 */
interface ProductToShop
{
    /**
     * Import or update given product
     *
     * Store product in your shop database as an external product. The
     * associated sourceId
     *
     * @param Struct\Product $product
     */
    public function insertOrUpdate(Struct\Product $product);

    /**
     * Delete product with given shopId and sourceId.
     *
     * Only the combination of both identifies a product uniquely. Do NOT
     * delete products just by their sourceId.
     *
     * You might receive delete requests for products, which are not available
     * in your shop. Just ignore them.
     *
     * @param string $shopId
     * @param string $sourceId
     * @return void
     */
    public function delete($shopId, $sourceId);

    /**
     * Start transaction
     *
     * Starts a transaction, which includes all insertOrUpdate and delete
     * operations, as well as the revision updates.
     *
     * @return void
     */
    public function startTransaction();

    /**
     * Commit transaction
     *
     * Commits the transactions, once all operations are queued.
     *
     * @return void
     */
    public function commit();
}
```

- The `ProductFromShop` interface

This interface is used when products are exported from your shop. You must implement this interface to work with your shop. The interface contains two methods to verify the products, which the shop currently exports, and two methods to perform an actual buy.

```php
<?php
/**
 * This file is part of the Bepado SDK Component.
 *
 * @version $Revision$
 */

namespace Bepado\SDK;

/**
 * Interface for product providers
 *
 * @version $Revision$
 * @api
 */
interface ProductFromShop
{
    /**
     * Get product data
     *
     * Get product data for all the product IDs specified in the given string
     * array.
     *
     * @param string[] $ids
     * @return Struct\Product[]
     */
    public function getProducts(array $ids);

    /**
     * Get all IDs of all exported products
     *
     * @return string[]
     */
    public function getExportedProductIDs();

    /**
     * Reserve a product in shop for purchase
     *
     * @param Struct\Order $order
     * @return void
     * @throws \Exception Abort reservation by throwing an exception here.
     */
    public function reserve(Struct\Order $order);

    /**
     * Buy products mentioned in order
     *
     * Should return the internal order ID.
     *
     * @param Struct\Order $order
     * @return string
     *
     * @throws \Exception Abort buy by throwing an exception,
     *                    but only in very important cases.
     *                    Do validation in {@see reserve} instead.
```

```
         */
        public function buy(Struct\Order $order);
    }
```

There are also some optional parameters:

- The `ErrorHandler` can be used to get more information about failures inside the SDK. This includes exceptions, RPC error and shutdowns (Fatal Error) during RPC calls.
- The `$pluginSoftwareVersion` contains information about the plugin you are developing. This information will be required in the future.

Use the `\Bepado\SDK\SDKBuilder` to create an instance of the SDK. This handles all the complexity of creating the different dependencies. The constructor may change in the future, using the SDKBuilder is required to implement a supported plugin.:

```
<?php

$builder = new \Bepado\SDK\SDKBuilder();
$builder
    ->setApiKey($yourBepadoApiKey)
    ->setApiEndpointUrl($urlWhereBepadoCanReachYourPlugin)
    ->configurePDOGateway($pdoConnection)
    ->setProductToShop($productToShop)
    ->setProductFromShop($productFromShop)
    ->setPluginSoftwareVersion('my Plugin v1.2.4')
;
$sdk = $builder->build();
```

Creating a Service Endpoint

The service endpoint URL is used by bepado and other shops to interact with your shop. All other instances execute `POST` requests against this URL. The URL itself does not matter, but the absolute URL must be provided to the SDK using the `$apiEndpointUrl` constructor parameter.

When receiving a request to this URL the XML body of the request must be dispatched to the `handle()` method on the SDK object. The `handle()` method will then again return a XML string, which should be echo'd. You must not echo anything but the raw XML returned by the method.

A simple PHP file, which does this could look as simple as:

```
<?php

$sdk = $builder->build();

echo $sdk->handle(file_get_contents('php://input'));

?>
```

You can of course integrate this call into frameworks, plugins or modules of your shop system or application.

Datastructures

There are four datastructures that you need to create from your own product and order data to work with the bepado SDK.

- `Bepado\SDK\Struct\Product` representing a product.
- `Bepado\SDK\Struct\Order` representing an order containing bepado/remote products.
- `Bepado\SDK\Struct\OrderItem` representing an order line-item containing products.
- `Bepado\SDK\Struct\Address` for the delivery address of your customer.

You should look at the three classes in code to get all the information about their structure and requirements.

```
<?php
/**
 * This file is part of the Bepado SDK Component.
 *
 * @version $Revision$
 */

namespace Bepado\SDK\Struct;

use Bepado\SDK\Struct;

/**
 * Struct class, representing products
 *
 * @version $Revision$
 * @api
 */
class Product extends ShopItem
{
    /**
     * Describes the weight of this product, e.g. "4.8" in kilograms Kg
     *
     * May be used for shipping cost weight calculation.
     */
    const ATTRIBUTE_WEIGHT = 'weight';

    /**
     * Describes the volume of this product, e.g. "0.75" in liters (L)
     */
    const ATTRIBUTE_VOLUME = 'volume';

    /**
     * Describes the product dimension, e.g. 40x20x100 (Length, Width, Height)
     */
    const ATTRIBUTE_DIMENSION = 'dimension';

    /**
     * Describes the unit of this product, for example "Kg" or "ml"
     */
    const ATTRIBUTE_UNIT = 'unit';

    /**
     * Reference quantity in the configured unit
     */
    const ATTRIBUTE_REFERENCE_QUANTITY = 'ref_quantity';

    /**
     * Units of the reference quantity the product has
     */
    const ATTRIBUTE_QUANTITY = 'quantity';

    /**
     * Local ID of the product in your shop.
     *
     * ID should never change for one product or be reused for another product.
     *
     * @var string
     */
    public $sourceId;

    /**
     * The European Article Number (EAN) of the product.
     *
     * @var string
     */
    public $ean;

    /**
     * URL to the product in your shop.
     *
     * Used for redirects to the product, or views of the product.
     *
     * @var string
     */
    public $url;

    /**
     * Title / name of the product
     *
     * @var string
     */
    public $title;

    /**
     * A short description of the product
     *
     * May contain simple HTML
     *
     * @var string
     */
    public $shortDescription;

    /**
     * An extensive / full description of the product
     *
     * May contain simple HTML
     *
     * @var string
     */
    public $longDescription;

    /**
     * Name of the product vendor
     *
     * @var string
     */
    public $vendor;

    /**
     * The value added tax for this product. The property must be set as a float
     * value. At the moment only 0.00, 0.07 and 0.19 are supported.
     *
     * @var float
     */
    public $vat = 0.19;

    /**
     * Current price of the product.
     *
     * Provided as a float. This is the selling price of the product
     * that end customers will pay. The price is checked again
     * during transactions and is required to be the same in both
     * shops during a transaction.
     *
     * The price is net and does *NOT* include the VAT.
     *
     * @var float
     */
    public $price;

    /**
     * The purchase price of this product.
     *
     * This is the price the seller (from-shop) of a product offers
     * a reseller (to-shop) for transactions in Bepado.
     * The price - purchasePrice gap is the profit of the reseller.
     *
     * Defining this price is optional and the regular Bepado price groups
     * will take effect if its not given. If this price is given however
     * price groups will NOT be considered to calculate the profit margin.
     *
     * The price is net and does *NOT* include the VAT.
     *
     * @var float
     */
    public $purchasePrice;

    /**
     * Do national laws require the price to be fixed at the suppliers level?
     *
     * This flag covers laws such as "Buchpreisbindung" in Germany.
     * SDK implementors have to force the selling price to customers to
     * be the same as given by the suplier.
     *
     * If this flag is not set, then selling shops are free to change
     * the price in their shops to their wishes and SDK implementors
     * **HAVE** to grant Shop users this possibility.
     *
     * This flag is **ONLY** for national price laws, not to prevent your
     * partners to change the price. Using this flag for preventing partners
     * to change the price is not allowed.
     *
     * @var boolean
     */
    public $fixedPrice = false;

    /**
     * Currency of the price
     *
     * Currently only the default "EUR" is supported.
     *
     * @var string
     */
    public $currency = "EUR";

    /**
     * If this property is set to <b>TRUE</b> this product will be shown as
     * delivery free of charge.
     *
     * @var boolean
     */
    public $freeDelivery = false;

    /**
     * Optional delivery date for this product as a unix timestamp.
     *
     * @var integer
     */
    public $deliveryDate;

    /**
     * Availability of the product
     *
     * Provide an integer with the amount of products currently in stock and
     * ready for delivery. When comparing availability during a transaction
     * Bepado SDK will group the availability into empty, low, medium and high
     * groups based on the interval 0 < 1-10 (low) < 11-100 (medium) < 101 to
     * infinity (high).
     *
     * @var integer
     */
    public $availability;

    /**
     * List of product image URLs
     *
     * @var string[]
     */
    public $images = array();

    /**
     * Product categories.
     *
     * For a full list of currently available product categories call
     * getCategories() on the SDK class.
     *
     * @var string[]
     */
    public $categories = array();

    /**
     * Product Tags
     *
     * A list of tags that can help other shops find your product.
```

```
         * Is limited to 10 tags maximum per product.
         *
         * @var array
         */
        public $tags = array();

        /**
         * Factor that affects the boost of products in search results.
         * Valid values are -1, 0, 1.
         *
         * @var int
         */
        public $relevance = 0;

        /**
         * Contains additional attributes for this product. Use one of the constants
         * defined in this class to specify a an attribute:
         *
         * <code>
         * $product->attributes[Product::ATTRIBUTE_UNIT] = 'kg';
         * $product->attributes[Product::ATTRIBUTE_WEIGHT] = '1.0';
         * $product->attributes[Product::ATTRIBUTE_DIMENSION] = '20x30x40';
         * </code>
         *
         * @var string[]
         */
        public $attributes = array();

        /**
         * Workdays until this product can be delivered.
         *
         * @var int
         */
        public $deliveryWorkDays;

        /**
         * Restores a product from a previously stored state array.
         *
         * @param array $state
         * @return \Bepado\SDK\Struct\Product
         */
        public static function __set_state(array $state)
        {
            return new Product($state);
        }
    }
```

The following validation rules apply to products:

- `sourceId`, `price`, `purchasePrice`, `currency`, `availability`, `vat` are required attributes.
- `title`, `shortDescription`, `longDescription`, `vendor` are required and must be UTF-8 strings.
- `purchasePrice` and `price` must be non-zero floats.
- `vat` must be a float of the Value-Added-Tax percentage between 0 and 1, for example 0.07 or 0.19 in Germany.
- `tags` and `categories` must be a numerically-indexed arrays of values.
- `relevance` must be either -1, 0 or 1 and represents the importance of the products in your own search

overview. It cannot be used to affect the relevance of products in global searches.
- `deliveryWorkDays` must be a non-zero integer.
- The optional Dimension-Attribute has to be in the format "XxYxZ" (20x40x60)
- The optional Weight Attribute has to be a number.

Calling the SDK

The SDK class has a set of methods, which provides you with the means to interact with bepado. The available methods can be related to the following tasks:

1. Registering products for export
2. Checkout of remote products
3. bepado Cloud Search
4. Update Order Status
5. Unsubscribe Products

The different tasks and the methods are described in more detail in the following sections.

On top of that there are several helper methods:

1. `verifySDK()` checks if the API key is valid and registers the API endpoint URL with bepado.
2. `isVerified()` checks if the SDK was previoiusly verified successfully.
3. `getCategories()` provides you with a list of valid bepado categories to associate your products with.
4. `getShop($shopId)` provides you with information about your connected shops such as the name of the shop and an URL to the shop. This can be useful to implement for product detail views or during the checkout.

Register Products for Export

For product export bepado must know about the products, which your shop wants to make available and must be informed about all changes to those products.

The first step you need to implement, which is entirely dependent on the shop you are targeting is a method to configure the set of exported products. This is independent from the SDK itself.

Once the list list of exported products is configurable, the SDK must be informed about the products (and later of changes to the listed products, or changes to the list). There are two ways to inform the SDK about this:

Update notifications

The SDK is called for every update to a product or the list. This method does require far less processing power in the SDK, but, depending on your shop, might be hard to implement.

Using this method the SDK must also be informed about all changes, which might be applied by third party tools, like ERP systems etc.

Use the `recordInsert()`, `recordUpdate()` and `recordDelete` methods on the SDK object for this. Just call the respective method every time a change to a product or the product list occurs.:

```
<?php

$product = MyDatabase::getProduct($_GET['id']);

if (!$product) {
    throw new \RuntimeException("No product found.");
}

$sdk->recordInsert($product->getId());
```

Change evaluation

The SDK can itself verify which changes were made to the list or products. For large amounts of products this check might consume lots of processing power. It usually is far easier to implement with existing shop systems, though.

Simply call the `recreateChangesFeed()` method for this, and the SDK will do everything else. But remember: This way costs far more processing power.

You should either call this method every time the currently exported products are requested by bepado using the service endpoint, or every time any change happens to the product database.

Checkout of Remote Products

If, during checkout, the shopping cart contains products from a remote shop the SDK provides you with a set of methods to verify the integrity of the remote products and perform the actual checkout in the remote shops.

Check Products

The method `checkProducts()` allows you to verify that the remote products still have the same price and availability, as stored in your local database. You should call this method when you want to verify an order.

The method receives an array of all products to verify. The SDK will verify that the structure is valid.:

```
<?php

use Bepado\SDK\Struct;

$converter = new YourBepadoConverter();

$yourProductObject = Product::get($productId);
$bepadoProduct = $converter->toBepadoProduct($yourProduct);

$order = new Struct\Order(array(
    'address' => new Struct\Address(
        'firstName' => 'Max',
        'lastName' => 'Mustermann',
        // more
    ),
    'orderItems' => array(
        new Struct\OrderItem(array(
            'count' => 1,
            'product' => $bepadoProduct
        ))
    )
));

$results = $sdk->checkProducts(array($bepadoProduct));

foreach ($results as $shopId => $result) {
    if ($result === true) {
        // everything alright
    } else {
        foreach ($result as $message) {
            var_dump($message);
        }
    }
}
```

You can recieve the following kind of messages:

- Price has changed, old and new values are available in the message and the end-user should be able to confirm to buy the products as the new price.
- Availability decreases to zero. No purchase possible anymore.

Calculate Shipping Costs

The method `SDK#calculateShippingCosts()` will return an object containing the shipping costs your shop has to pay the supplying shop for shipping.

This information can be used to directly pass on to the customer of your shop or as a first calculation to perform your own shipping cost calculation.

Be aware that you need to pass an `Bepado\SDK\Struct\Order` object to this method. This is required, because shipping costs can depend on items or delivery address. If you don't have all the information about the order in the checkout process yet, just build a "pretend" version of an order, using for example the default delivery country.:

```
<?php

use Bepado\SDK\Struct;

$order = new Struct\Order(array(
    'address' => new Struct\Address(
        'firstName' => 'Max',
        'lastName' => 'Mustermann',
        // more
    ),
    'orderItems' => array(
        new Struct\OrderItem(array(
            'count' => 1,
            'product' => $bepadoProduct
        ))
    )
));

$shippingCosts = $sdk->calculateShippingCosts($order);

if ( ! $shippingCosts->isShippable) {
    // Show your user that the combination of products
    // is not possible to buy for him, because of delivery
    // address or other criteria.
} else {
    echo round($shippingCosts->shippingCosts, 2) . " €";
}
```

Reserve Products

The method `reserveProducts()` should be called when you want to reserve products in the remote shop(s). You must call this method as a part of the checkout process. If relevant product details changed, compared to the data your shop has stored locally, the method will return a message struct. this should be presented to the user. Examples for this are price and availability changes of products.

If the reservation succeeded a set of hashes will be returned, which should be stored in the user session. Those hashes will be used to finalize the checkout process.:

```
<?php

$reservation = $sdk->reserveProducts($order);

if (!$reservation->success) {
    foreach ($reservation->messages as $shopId => $messages) {
        // handle individual error messages here
    }
}
```

The `reserveProducts()` method can fail for all the same reasons that `checkProducts()` can fail for and two more:

- Shipping costs have changed. No purchase possible anymore.
- Products are not shippable to the delivery address of the user, you should detect this early with `calculateShippingCosts()`.

Checkout Products

The final step to buy products is calling the `checkout()` method. This step receives a reservation struct, as returned by the `reserveProducts()` method. Usually the method will just return `true` and the buy process can be considered completed.

If some problem occured while finalizing the checkout, the method will return a message struct. Those messages should again be presented to the user, so the user can take appropriate action. Alternatively you can still present the user a success message and let your support team handle the issue.:

```
<?php

$result = $sdk->checkout($reservation);
```

You can combine `checkout()` and `reserveProducts()` in a single request to reduce the implementation overhead of handling reservations.

Cloud Search

The bepado Cloud Search enables you to search for products in foreign shops. Just call the `search()` method with an appropriate search struct, containing the search terms and optionally additional search configuration. The method will return a search result struct, which you can present as a search result on your website. The structs should be self-documenting.

Update Order Status

To notify other shops about changes in orders that you accepted, you can use the `updateOrderStatus()` method on the SDK.

Unsubscribe Products

When you delete bepado products in your shop system through the backend you can directly notify bepado that you want to unsubscribe these products. Otherwise bepado will push the products to your shop whenever changes happen. Unsubscribe from products by calling `unsubscribeProducts()` on the SDK.

Implementation Hints

Money as floats

Througout the SDK we have to handle money as float objects, because all the big shop systems save money as float themselves. We evaluated handling money as integers of cents, however the conversion between shop system price floats and bepado integers would already be the point where precision is lost through the conversion. Introducing a money object would only make the likelihood of precision errors higher and therefore introduce a risk.

There are some requirements for you to keep in mind: You are not allowed to round the money floats when saving or retrieving them from the database or using them to create products for an order. Only for display purposes should you use `round($money, 2)` to get the value to display to the end user.

If you don't save money as a float in your database then you should save the money values as strings in the database.

Converting Products

For various tasks in combination with the SDK you will need a way to convert the product datastructures of your shop system to Bepado product datastructures. Therefore it makes sense to introduce a dedicated converter object that handles this conversion.:

```
<?php

interface ProductConverter
{
    public function toBepadoProduct($shopProduct);
    public function toShopProduct($bepadoProduct);
}
```

Security

bepado, your SDK and others talk to each other using remote-procedure calls with XML. Message security is handled by using HMAC signatures with a SHA-512 hash. Every combination of parties (two shops or shop-bepado) has a unique shared secret that is generated by bepado and only known to those parties and bepado.

Only the bepado party is allowed to push new products to your shop. The secret key shared between two shops is only allowed to create and confirm remote orders of products between your two shops.

To protect yourself further you should make sure that new products pushed to your shop by bepado are never visible to your customers automatically but should be "activated" or "approved" through a seperate step.