www.phparch.com

php[architect]

March 2020
Volume 19 - Issue 3

How Magento is Evolving

The State of Magento

Alternative Checkout Flow

From Monolith to Service Isolated Architecture

Magento Inventory and In-Store Pickup:
The Biggest Community Contribution

Asynchronous Magento

ALSO INSIDE

Education Station:
PHP Development Environments

Community Corner:
AustinPHP

History and Computing:
My NSA Mug Shot

Security Corner:
Mutual TLS

The Workshop:
Managing LAMP with Virtualmin

finally{}:
Enterprise PHP

Adobe

Grow your business. Expand your client network.

The Magento Solution Partner Program and Community Insider Program are now part of the Adobe Solution Partner Program. Are you ready to help your customers build incredible experiences on every screen?

Explore the market-leading ecosystem of Adobe experts, learning resources, exclusive benefits, and much more.

Join the Adobe SPP today for free at solutionpartners.adobe.com

PHP[TEK] 2020

Our 15th annual PHP conference, focusing on Tech Leadership.

Keynote Speakers

Taylor Otwell

Creator of Laravel

Tereza Nemessanyi

CxO & TED Speaker

NASHVILLE

★ TENNESSEE ★

May 18-21, 2020

Nashville, TN — Opryland

tek.phparch.com

Asynchronous Magento

Oleksandr Lyzun

Over the years, Magento architecture has become more sophisticated and complex because merchants' requirements are becoming more complex. This complexity leads to an increase of application infrastructure, release cycles, and the number of features that have to be maintained. For Magento's first decade, it was a monolith application which nicely covered mid-sized merchant needs. But as the system grows, the more difficult it becomes to maintain this monolith architecture.

Beginning in mid-2018, the Magento architecture team began work to achieve a full-service decomposition of the Magento core. The idea is to split the Magento framework into separate services, where each service is fully isolated and independent and deployable as an independent application. One critical idea with this approach is the communication between services. Services must be able to communicate with other services, or with the monolith seamlessly. Following one of the guidelines of service decomposition, communication between services must be efficient and able to process asynchronously, if possible. This article explains why and how asynchronous communication was chosen and which mechanisms Magento uses to achieve this.

Let's have a look at how communication between services can be processed. It can be done in either a synchronous or asynchronous manner. Synchronous communication means all requests have to be processed in the same order that they were sent, and the sender system must wait for a response from the receiver. This also means the connection is kept open between the two services. If we are talking about a complex system with many different services, the sequence of the operations may become quite long and lead to painful performance issues. Because of the high risk for potential process locks, such a system is tough to maintain and scale. For example, on complex systems, after placing a new order, the system may have to execute a long series of post-actions, as seen in Figure 1.

The potential user experience impact is clear: Processing all of these actions takes time and depends on all operations completing without error—problems, in either case, would be a conversion killer. Another common consideration is third-party integrations, which may not be available or are having performance issues. This could mean that the purchase process is entirely blocked or lasts so long that the customer may again cancel their purchase.

Let's proceed with the same example, but this time asynchronously. We can charge the customer right away, place the order, and process all required operations to complete the order by using background processes. In case of a problem or delay with an operation, we can report this information to the user or retry an operation in the background. The only inconvenience we have left in this case is failed operations, which require customer input, but this can be solved with a clearly implemented error-handling business process.

The final picture with asynchronous communication looks like Figure 2.

Service decomposition is only one example where asynchronous communication can be used, but this is currently the most common example.

Now, let's have a look at the work Magento has already made in:

- The direction of asynchronous communication
- Which use cases are already covered
- Which user cases we are planning to integrate

Event-Driven Architecture, CQRS, And Event Sourcing

Asynchronous Magento requires implementing a few well-known architectural paradigms. The first is event-driven architecture[1] (EDA). One

1 *event-driven architecture:* *https://phpa.me/wikip-eda*

Figure 1

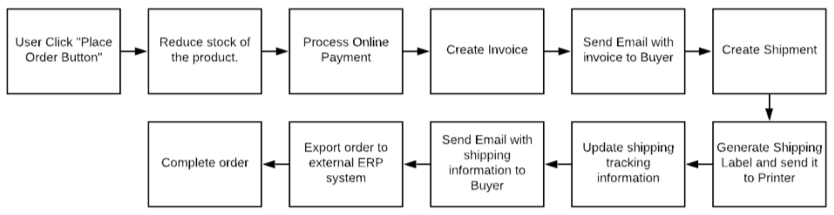
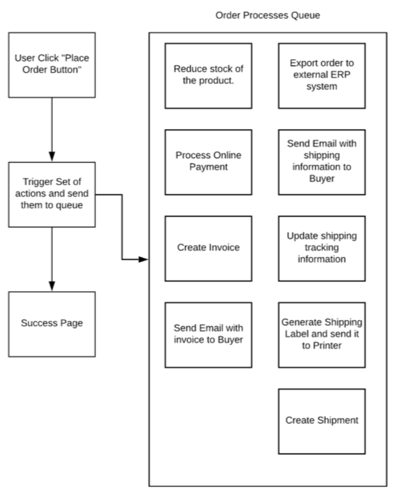

User Click "Place Order Button"
Reduce stock of the product.
Process Online Payment
Create Invoice
Send Email with invoice to Buyer
Create Shipment
Generate Shipping Label and send it to Printer
Update shipping tracking information
Send Email with shipping information to Buyer
Export order to external ERP system
Complete order

Figure 2


Order Processes Queue
User Click "Place Order Button"
Trigger Set of actions and send them to queue
Success Page
Reduce stock of the product.
Export order to external ERP system
Process Online Payment
Send Email with shipping information to Buyer
Create Invoice
Update shipping tracking information
Send Email with invoice to Buyer
Generate Shipping Label and send it to Printer
Create Shipment

of the main advantages of an event-oriented approach is that we don't need to change the currently existing implementation (module) when we want to add or change the functionality of the system. Implemented properly, developers can add and remove functionality without affecting the stability of the system.

The second approach is Command Query Responsibility Segregation[2] (CQRS), which involves separating database writes and reads. In Magento, this means commands are sent to Magento (writes), but the result of these commands never contain data from the objects being communicated with. On the other side, we have queries which receive information from persistent storage. In this approach, asynchronous communication introduces the possibility of inconsistency, as the execution order may be different than in a serial execution flow, or in the case of processing delays, depending on current system loads.

Service-oriented architectures may implement different methods of communication between services. As an example, let's have a look at the enterprise service bus (ESB) architectural approach. The ESB provides a centralized messaging between services. The idea is that all service communication goes through a single central point. An ESB implementation, in combination with message queueing, may be quite powerful for implementing a service-oriented architecture. But this also means all services have a strict dependency on the ESB implementation and cannot communicate directly with each other. As all requests are routed through the ESB, this leads to slower communication between services. Another disadvantage is that the ESB is always a single point of failure. A failure of the ESB causes failures across the application. This approach brings additional complexity to the architecture and reduces scalability.

Event Sourcing is the approach of storing the data whenever we make a change to the state of a system. We record that state change as an event, and we can confidently rebuild the system state by reprocessing the events at any time in the future.

2 *Command Query Responsibility Segregation: https://phpa.me/mfowler-cqrs*

The main ideas around Event Sourcing are:

- There are no limitations on the number of events that can happen with an object.
- All commands are immutable; the state of events can never change after creation.

With Event Sourcing, you always know what happened with every object since its creation. As an example, imagine a product's stock value, represented with one cell in a database. Each time a new order is placed, the stock is changed based on third-party, or the admin adjusts the quantity manually, the value of this cell changes without record. With Event Sourcing, you always see when and what happened with the stock value.

CQRS and Event Sourcing are usually tightly connected. And, as we are talking about full-service decomposition in Magento, Event Sourcing currently looks like the correct path. Magento's functionality requires a large number of processes and objects, and currently, there is a limited transactional record of the values which change and the processes involved. Event Sourcing, correctly implemented, provides a complete record of the what/why/how/when of changes, which will:

- give us control over all system processes,
- facilitate their planning and development, and
- allow us to monitor and track the state of all objects and persist each write operation.

Challenges to Change

Until now, Magento has mostly used a read-write approach. If you made a `POST` request, you directly received information about that object. Even with this behavior, you can't say the Magento API was RESTful. In recent releases, Magento has tried to change these requests, but could not wholly rewrite all of them. The main reason is that such changes are not backward compatible and may break third-party integrations. At the same time, if we follow CQRS and let go of the RESTful approach, such requests have to be eliminated. If you dive deeply into Magento operations, you can see it is not always necessary to receive a response immediately after sending the request.

Magento already has all the necessary software to be able to integrate CQRS as a standard. The Magento Message Queue Framework gives us the beginnings of an Event Sourcing implementation. All asynchronous operation logs are currently stored in the Magento database, where you can see all the operations, request, and response messages. But it's just a first step in the integration of Event Sourcing.

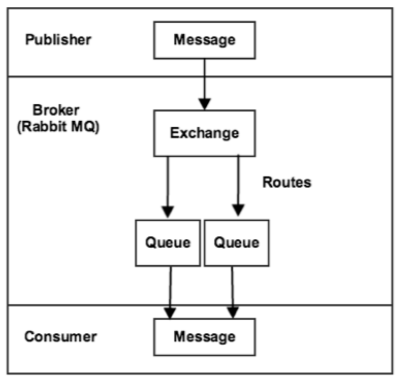
Magento Message Queue Framework

The main idea of message queues is to provide asynchronous communication between the sender and receiver, without having them communicate directly. The sender places the message into the queue, and the receiver gets the message only when the consumer processes it.

Magento Commerce 2.1 and Magento Open Source since version 2.3 provide the Message Queue Framework (MQF).

According to Figure 3:

Figure 3



- Publisher sends messages to the exchange.
- Exchange receives messages and sends them to the appropriate queue. Magento currently uses "`topic exchanges`", where "`topic`" is a routing key for the message. The topic is a string key consisting of dot-separated parts, which also can contain wildcard signs: * for replacing one of the topic parts, or # to replace all parts after the current.
- The queue receives and stores messages.
- Consumer picks up messages from the queue and executes them.

Current Use of MQF

Until recently, the most common use for MQF comes from shared catalogs in B2B. It is possible to set custom prices for each customer or to give customer permissions only for specific categories. Since such operations can take time and require many system resources, they are performed asynchronously.

The next big step forward for MQF use was a community project which provided Asynchronous and Bulk APIs for Magento. This API is based on a message queue implementation and nicely demonstrates this new approach to queues. Currently, the MQF is used in the development of new Magento modules, as a way for different modules to communicate with each other. A good example is the Magento Multi-source Inventory module (MSI), which uses asynchronous communication to assign, unassign, or transfer stock.

How to Use MQF

The out-of-box AMQP implementation for Magento is RabbitMQ[3]. If you read Magento DevDocs:

> *"RabbitMQ is an open-source message broker that offers a reliable, highly available, scalable, and portable messaging system. Message queues provide an asynchronous communication mechanism in which the sender and the receiver of a message do not contact each other. Nor do they need to communicate with the message queue at the same time. When a sender places a message onto a queue, it is stored until the recipient receives them."*

RabbitMQ has to be installed and running, and we must configure Magento to communicate with RabbitMQ. This can be done by adding a configuration block for the queue to `app/code/env.php` shown in Listing 1.

To integrate message queueing for your use cases, you must create four different configuration files inside of your module. Here's a short overview of each of them.

`communication.xml`: This file (Listing 2) contains information about messages the system executes, defined by "topics" and "handlers," which are responsible for message processing. Developers have to define a topic name for the message and define a handler to process this message.

`queue_topology.xml`: Defines the routing rules. In this file, define the destination queue for topics and transfer some custom arguments that are passed to the message broker for processing (Listing 3 on the next page).

`queue_publisher.xml`: Defines the exchange where the specific topic is published. Currently, Magento supports only `db` and `amqp` connections. Be careful here, as only one exchange can be enabled for one topic, see Listing 4 on the next page.

3 RabbitMQ: https://www.rabbitmq.com

Listing 1

```php
<?php
'queue' => [
  'amqp' => [
    'host' => 'rabbitmq.example.com',
    'port' => '11213',
    'user' => 'magento',
    'password' => 'magento',
    'virtualhost' => '/',
    'ssl' => 'true',
    'ssl_options' => [
       'cafile' => '/etc/pki/tls/certs/DigiCertCA.crt',
      'certfile' => '/path/to/magento/app/etc/ssl/test-rabbit.crt',
       'keyfile' => '/path/to/magento/app/etc/ssl/test-rabbit.key'
    ],
  ],
],
```

Listing 2

```xml
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation=
      "urn:magento:framework:Communication/etc/communication.xsd">
  <topic name="synchronous.rpc.test"
         request="string"
         response="string">
     <handler name="processRpcRequest"
              type="Magento\Test\Model\Handler"
              method="process"/>
  </topic>
</config>
```

queue_consumer.xml: Defines the consumer that processes messages from the queue (Listing 5).

When the MQF was first created, it contained only the queue.xml configuration file. But working with message queues requires flexibility in customization, so the next step was to split the configuration such that message queueing is easy to configure and easy to extend. In the case of isolated services, one goal is making it possible for a Publisher to be deployed and configured in one environment and the consumer on another—with either knowing nothing about the other!

After creating all required configurations, your message is basically ready to be processed. To send a message to the queue, you have to execute the publish() method of \Magento\Framework\MessageQueue\PublisherInterface interface:

```
$publisher->publish($topic, $message);
```

After that, your message is published in the queue.

To execute a Magento queue consumer, run the command:

```
bin/magento queue:consumers:start \
 CONSUMER_NAME
```

To get a list of consumers, use the following command, which returns a list of all consumers registered in the system.

```
bin/magento queue:consumers:list
```

You can technically run the same consumer several times. This leads to a situation where each consumer takes messages from the queue and process them simultaneously. From one point of view, that's good, because you can process multiple messages at the same time and speed up the whole process. All cases using the MQF are aimed at implementing asynchronous communication between application parts, which does not affect the performance of the processes themselves but improves the scalability of the entire application.

However, there are always boundaries to consider.

Listing 3

```
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation=
           "urn:magento:framework-message-queue:etc/topology.xsd">
   <exchange name="magento-topic-based-exchange1" type="topic"
             connection="db">
     <binding id="topicBasedRouting2" topic="anotherTopic"
              destinationType="queue" destination="topic-queue1">
       <arguments>
         <argument name="argument1"
                   xsi:type="string">value</argument>
       </arguments>
     </binding>
   </exchange>
</config>
```

Listing 4

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation=
           "urn:magento:framework-message-queue:etc/publisher.xsd">
   <publisher topic="asynchronous.test">
      <connection name="amqp" exchange="magento" disabled="false"/>
      <connection name="db" exchange="exch1" disabled="true"/>
   </publisher>
</config>
```

Listing 5

```
<?xml version="1.0"?>
<config
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation=
          "urn:magento:framework-message-queue:etc/consumer.xsd">
   <consumer name="basic.consumer"
             queue="basic.consumer.queue"
             handler="LoggerClass::log"/>
</config>
```

- **Mysql Deadlocks.** In case the message processing requires a lot of data modifications on persistent storage, this means the same operation may try to create or modify data in the same tables at the same point in time. Depending on the tables and system architecture, this may lead to an issue where the first message locks the table. When another one tries to write data on it, the system throws an exception, and the operation cannot be completed. The Magento team has eliminated much of these, but some ultra-high-scale situations still require investigation and thoughtful design.
- **Performance.** Multiple simultaneous message execution requires more system resources. For example, creating hundreds of products simultaneously requires far more resources than serving hundreds of customer requests.

An additional consideration: All messages that the system creates and transfers to the message queue have to be idempotent. Idempotence is perhaps familiar to those of you who have worked with or built RESTful services. It means the same operation can be executed multiple times, but the result always stays the same. This property facilitates using multiple queues or swapping out queue implementations.

It's also essential to find the sweet spot of running consumers, with a preference for messages that trigger bulk operations, which partially resolve problems with locking. Time of day is also vital to consider; for example, it makes sense to run import operations at night—or whenever customer impact is lowest.

Running Consumers

Cron jobs are the default mechanism to start consumers. In the configuration, it is possible to define the number of messages that are processed by a consumer. After reaching this number, the consumer is terminated. Re-running cron restarts the consumer.

To enable consumers for your system, you have to add following configuration in your `/app/etc/env.php` file:

```
'cron_consumers_runner' => [
   'cron_run' => false,
   'max_messages' => 20000,
   'consumers' => [
      'consumer1',
      'consumer2',
   ]
]
```

In the "consumers" section, you should define a list of consumers you want your cron to execute.

RabbitMQ Messages

Let's have a look at the RabbitMQ messages the system sends to the queue.

Packing of messages happens in this method:

```
\Magento\AsynchronousOperations\Model\MassPublisher
::publish($topicName, $data)
```

It method creates envelopes for each message and sends them to the queue.

RabbitMQ messages have a predefined list of message properties, but the current Magento implementation uses only a couple of them, shown in Listing 6.

- `body` is our message body which contains `ID`, `topic_name`, `bulk_uuid`, and `serialized_data`.
- `delivery_mode` is a constant and means this message has a persistent delivery mode.

Listing 6

```
$envelopes[] = $this->envelopeFactory->create(
   [
      'body' => $message,
      'properties' => [
         'delivery_mode' => 2,
         'message_id' => $this->messageIdGenerator
                            ->generate($topicName),
      ]
   ]
);`
```

- `message_id` is a unique id for the message.

One new feature released in Magento version 2.3.3 is the use of message "application_headers." Now each message receives a new parameter, `store_id`. This parameter notes the current store for which a request was executed. We need this information to apply the message to the correct system scope.

When a consumer picks up a message from the queue, it receives the same information that was transferred from the publisher. Then based on `communication.xml` and the message body, it can process the correct operation.

Synchronous Web API

Magento provides a REST API giving developers and integrators access to almost all Magento features. It can be used for smooth and efficient communication between Magento and third-party systems. Theoretically, this approach also can be used for communication between Magento and Magento services. However, for this purpose, there are some challenges which you can guess based on the other points. First, all REST requests are synchronous. Doing so leads to the problem where each system executing an API request must wait for the response from the system to be sure that the operation executed successfully. If an operation requires a lot of time to execute, the whole process is slowed down. If we try to send multiple API requests at the same time, this may have a significant impact on system performance. Also, the REST API does not allow us to transfer multiple messages to Magento at the same time.

The reason why this happens is that the initial idea was to support a single point of customization, and it was assumed that if a developer needed to handle `N` entities, they would create a single message. But in the case of large data operations, developers quickly hit deadlocks or resource limits. These points were the main reason why the community started to develop asynchronous and Bulk APIs for Magento, which has moved Magento APIs to the next level.

Asynchronous/Bulk API

In discussing asynchronous Magento, we definitely have to cover the topic of using asynchronous API as the primary communication mechanism between services.

Asynchronous API is implemented on top of the REST API. When the system receives API requests which have to be executed asynchronously, it adds them to the queue. At the same time, the consumer reads messages from the queue and process them.

Bulk API is a community-driven implementation on top of asynchronous API. It allows us to send a single request with multiple objects in the body. The system splits these objects to single messages and executes them asynchronously, providing substantial benefits:

1. The response time of all asynchronous bulk requests are faster than the response time of typical synchronous REST requests, yielding approximately 30% of

performance improvements; in the case of Bulk API, this change is tremendous.

2. Bulk API provides the possibility to send multiple objects in one request. Doing so leads to a reduction in the total number of requests, further reducing process overhead.

Let's have a look at some stats—these tests were made on Magento Cloud infrastructure with 16 CPUs, 256 GB RAM, and Magento 2.3.2 Commerce Edition installed. In Figure 4, you can see the performance comparison of different types of APIs when receiving messages.

We performed tests where we started to send simultaneous requests to Magento API product-create endpoints, starting from one request and ending with 150 simultaneous requests. You can see that for synchronous and asynchronous APIs, average response times for the maximal amount of requests are close to 90 and 80 seconds, respectively (each request contains one object). If we have a look at 150 objects in a single request of the BULK API, Magento was capable of receiving 150 objects within approximately 10 seconds for all of them—an eight-fold improvement!

This performance test offers another important metric—incidence of errors, sync versus async—shown in Figure 5.

Each system has its resource limitations and limitations of server software. The system in this test had a limitation of 134 simultaneous processes running. Starting at 135 requests, the system started to throw 50x errors back without processing those messages. You can see how the number of failures grows from the number of requests—red and blue lines are the same here. The same Bulk API does not have such problems because it only sends one request independent from the numbers of objects. The same situation happens with server resources because, with high numbers of requests, CPU and RAM usage is growing, respectively. Definitely, during Bulk API usage, you have to be aware of some other system settings, like the PHP setting `max_post_size`, but those are not dependent on hardware.

As I mentioned, the Bulk and asynchronous API implementation works atop the usual REST API. Developers don't need to take care of any additional development to make their API support Bulk operation. By default, Magento takes all `POST`, `PUT`, or `DELETE` REST endpoints and makes them available as Bulk. In addition, if third-party developers are using PATCH requests, they are also automatically supported by Bulk API.

In the case of asynchronous APIs, everything is quite simple. To the usual REST URL `POST {{URL}}/rest/all/V1/products` the `async` prefix is added: `{{URL}}/rest/all/async/V1/products`. Then, the system recognizes this request as asynchronous and processes it.

Figure 4

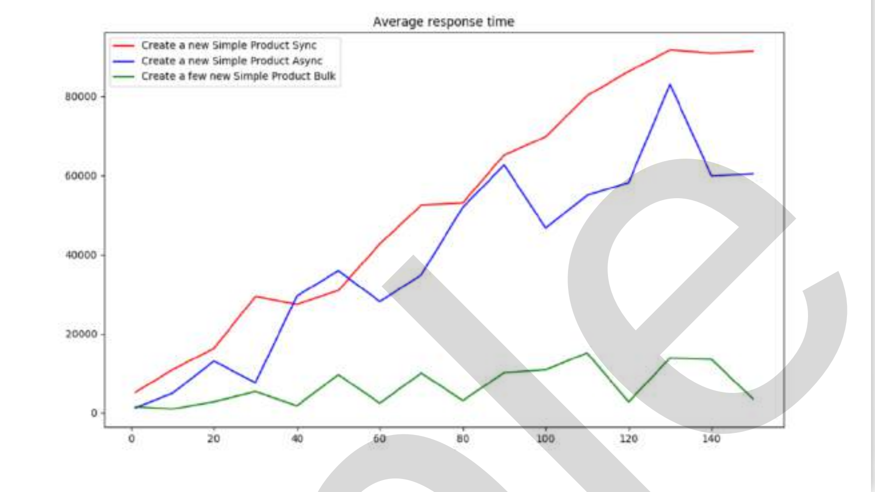
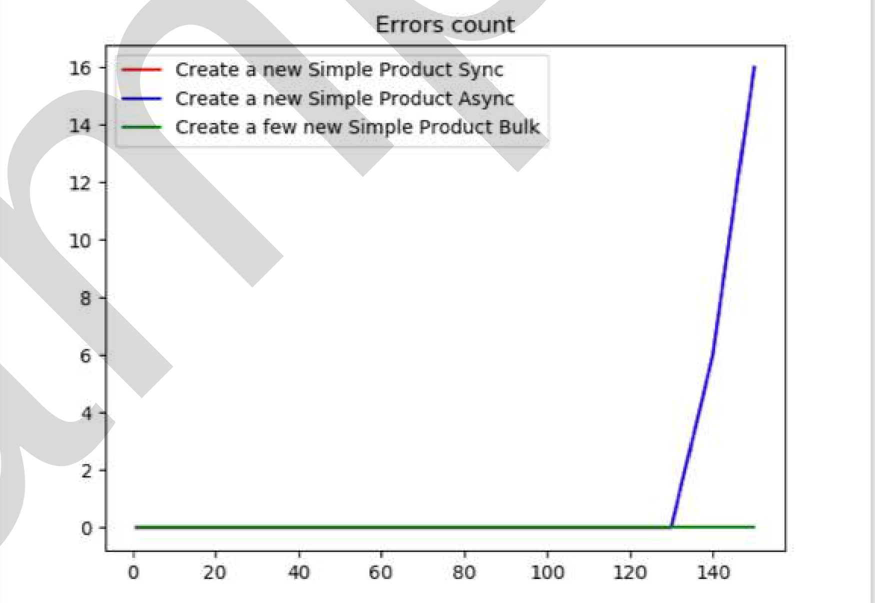


Figure 5



In the case of Bulk API, the usage is a `POST {{URL}}/rest/all/V1/products` with an `async/bulk` prefix: `{{URL}}/rest/all/async/bulk/V1/products`, and also, items inside of the body of request have to be transferred as an array (Listing 7).

Some API requests also contain parameters in URLs, for example: `DELETE /V1/cmsBlock/:blockId`, which expects to have the exact block ID to be transferred as a URL parameter. As you can understand, in case of multiple object transfer, we had the challenge to create some solution that does not require any additional development from the developers' side, and at the same time, give integrators the flexibility in API usage. The idea is to automatically convert those URL parameters to the static part of the URL. For Bulk API usage, URLs have to be converted such that ":" has to be replaced with "by" prefix and next letter converted to uppercase. For example, "/V1/cmsBlock/:blockId" has to be changed to "/V1/cmsBlock/byBlockId". And at the same time, the `block_id` parameter has to be added to the object body.

```
[
    {
        "block_id": "1"
    },
    {
        "block_id": "2"
    }
]
```

By the way, if in the REST API, `DELETE` requests have no body, then in the Bulk API request, the body of such requests contains an array of corresponding object IDs.

The response of all async and Bulk requests is always the same as you can see in Listing 8. It is the UUID of the operation and a list of items (in case of asynchronous API, always one item):

The Bulk UUID can be used later to receive operation status and also to check the response of the operation. The UUID is always unique and always represents one bulk operation.

Request the status of the operation by calling:

```
GET /V1/bulk/:bulkUuid/detailed-status
```

And the answer you receive looks like Listing 9.

Asynchronous and Bulk API functionality become available for developers starting in version Magento 2.3. And we want to thank Magento partners: comwrap GmbH (Frankfurt, Germany) and Balance Internet (Melbourne, Australia) who made this possible.

Working with Queues

To enable the processing of Bulk APIs, you have to specify a consumer to be run. Consumer name for asynchronous and Bulk API is "async.operations.all." This consumer receives and processes all requests that come to the async/Bulk API, which means you have one master consumer for all APIs. If you want to speed up processing, you can run multiple consumers. If you are sure your operations are independent

Listing 7

```
[
    {
        "product": {
            "sku": "SKU1",
            "name": "Simple product 1",
            "attribute_set_id": 4
        }
    },
    {
        "product": {
            "sku": "SKU2",
            "name": "Simple product 2",
            "attribute_set_id": 4
        }
    },
    ........
]
```

Lis`ting 8

```
    {
        "bulk_uuid": "799a59c0-09ca-4d60-b432-2953986c1c38",
        "request_items": [
            {
                "id": 0,
                "data_hash": null,
                "status": "accepted"
            },
            {
                "id": 1,
                "data_hash": null,
                "status": "accepted"
            }
        ],
        "errors": false
    }
```

Listing 9

```
{
  "operations_list": [
    {
      "id": 4,
      "bulk_uuid": "c43ed402-3dd3-4100-92e2-dc5852d3009b",
      "topic_name": "async.magento.customer.api.
accountmanagementinterface.createaccount.post",
      "serialized_data": "{\"entity_id\":null,
\"entity_link\":\"\",\"meta_information\":\"{\\\"customer\\\":
{\\\"email\\\":\\\"mshaw@example.com\\\",\\\"firstname\\\":
\\\"Melanie Shaw\\\",\\\"lastname\\\":\\\"Doe\\\"},
\\\"password\\\":\\\"Password1\\\",\\\"redirectUrl\\\":\\\"
\\\"}\"}",
      "result_serialized_data": null,
      "status": 3,
      "result_message": "A customer with the same email
address already exists in an associated website.",
      "error_code": 0
    }
}
```

of each other and you want to make the whole communication process much faster—for example, checkout process and customer registration—you can configure the system so each consumer processes only its objects.

Currently, the system is configured so all topic names that start with “async.#” are delivered into “async.operations.all” queue. You can find this definition in `app/code/Magento/WebapiAsync/etc/queue_topology.xml`.

```
<exchange name="magento" type="topic" connection="amqp">
   <binding id="async.operations.all"
            topic="async.#"
            destinationType="queue"
            destination="async.operations.all"/>
 </exchange>
```

And only one consumer is defined in `queue_consumer.xml`:

```
<consumer name="async.operations.all"
   queue="async.operations.all"
   connection="amqp"
   consumerInstance=
"Magento\AsynchronousOperations\Model\MassConsumer"
/>
```

For each object(s) you want to process, you can define its queue where all defined objects are sent, and the consumer is defined.

First, you have to forward topics to the queue (`queue_topology.xml`). For example, let’s create one queue for products processing and another for customers and forward the required topics to them.

```
<exchange name="magento" type="topic" connection="amqp">
   <binding id="async.magento.catalog.api.productrepositoryinterface"
      topic="async.magento.catalog.api.productrepositoryinterface.#"
      destinationType="queue"
      destination="async.magento.catalog.api.product" />
   <binding id="async.magento.customer"
        topic="async.magento.customer.#"
        destinationType="queue"
        destination="async.magento.customer"/>
</exchange>
```

Topic names for asynchronous and BULK API are auto-generated based on service contract names defined for each API request.

Let’s have a look `Magento/Catalog/etc/webapi.xml`:

```
<route url="/V1/products/:sku" method="PUT">
   <service class="Magento\Catalog\Api\ProductRepositoryInterface"
            method="save" />
   <resources>
      <resource ref="Magento_Catalog::products" />
   </resources>
</route>
```

All topic names are generated as PREFIX + service_class + service_method + route.method. Prefixes always equal to `async`. The service class is a class name, where all "/" are replaced with "." Service method and route method are

Using Xdebug to squash bugs, identify bootlenecks, and boost productivity?

Become a Pro or Business supporter to help ongoing development.

Supporters get help via email and elevated issue priority.

https://xdebug.org/support support@xdebug.org taken as-is. And finally, the line is converted to lowercase. So based on the previous example, topic name for `PUT` request for `/V1/products/:sku` looks like:

```
Async.magento.catalog.api.productrepositoryinterface.save.put
```

After that, you have to create a new consumer. One queue for products and one for customers:

```
<consumer name="async.magento.catalog.api.product"
          queue="async.magento.catalog.api.product"
          connection="amqp"
   consumerInstance=
"Magento\AsynchronousOperations\Model\MassConsumer" />
<consumer name="async.magento.customer"
          queue="async.magento.customer"
          connection="amqp"
   consumerInstance=
"Magento\AsynchronousOperations\Model\MassConsumer" />
```

And as a last step, you have to run the two consumers separately. `async.magento.catalog.api.product` and `async.magento.customer`. Be aware of two critical points:

1. If you are creating a new queue with Magento configuration files, you have to run `bin/magento setup:upgrade` for the system to create new empty queues in RabbitMQ.
2. If you are running the `async.operations.all` consumer, it processes both requests to custom-defined queues, which means your messages execute twice. If `async.operations.all` was already created in the RabbitMQ as a new queue, I would recommend deleting it from RabbitMQ.

Next Steps

MQF and Bulk API are just the first steps for asynchronous Magento. There are already multiple projects that are going in this direction.

Cron Improvements

A new Magento community project focuses on changing the Cron Framework in Magento. All jobs which can be moved from cron execution to AMQP are moved. More info about the project will come in the next few months.

Import Service

Another project fully focused on implementing as an isolated service is `Asynchronous Import`. The idea behind it is to build a standalone service that asynchronously communicates with Magento by using Bulk API and will be responsible for importing data into the Magento database. This implementation will replace the Magento Import module. The project is already up and running, and you can find all the information about it on Magento Community Portal.

Service Isolation

The most important topic in the Magento roadmap is Magento Services Decomposition or Service Isolation. The whole of Magento will be split into independent services, and asynchronous communication between those services is a crucial topic for the entire architecture. Magento Services Decomposition is not a project that has some strict MVP definition. It is mostly a set of architectural decisions and guidelines about how services have to be developed. The main goal is to transform the current Magento codebase in such a way where each service can be independent. For more details, see Igor Miniailo's article in this issue.

Conclusion

In the last few years, the evolution of Magento has been incredible. Changes which the Magento team and the Magento community are currently trying to implement are quite ambitious. That is one reason why the Message Queue Framework is growing, and there are still many topics that Magento and its community are trying to integrate.

CQRS and Event Sourcing design principles are the keys to the future of asynchronous Magento. Message Queue Framework is functionality that will be used underneath the Event Sourcing implementation.

The main challenge of building Services Isolation in Magento is how to implement communication between services. One of the best ways to solve this challenge is a "smart endpoints and dumb pipes" approach. Smart endpoints mean that the main business logic happens behind the main endpoints, on the consumer's level. Dumb pipes are communication where no further actions take place, and only carry data across a particular channel.

Building stable and performant communication workflows between all parties is an essential feature of a stable ecommerce solution. If you want to join the community and help us build the future of commerce, you are always welcome.

The greatest value of Magento is its community!

Oleksandr (aka Alex) Lyzun is the Magento Technical Team Lead at comwrap and joined the company in 2015. In the last 11 years of working with Magento, he developed, led and put live numerous Magento projects. In addition to that, he successfully completed the Magento Professional Developer, Cloud Developer, and Solution Specialist Certifications. As Magento Community Maintainer, Magento DevDocs Maintainers and Magento Master 2019 and 2020 he works with passion and love on various Magento Community projects. His passion is to work on complex Magento projects, to provide tailored solutions to the digital challenges of e-commerce customers.