QPush Bundle Documentation

Release 1.1.3

Keith Kirk

Jun 07, 2017

Contents

The QPush Bundle relies on the Push Queue model of Message Queues to provide asynchronous processing in your Symfony application. This allows you to remove blocking processes from the immediate flow of your application and delegate them to another part of your application or, say, a cluster of workers.

This bundle allows you to easily consume and process messages by simply tagging your service or services and relying on Symfony's event dispatcher - without needing to run a daemon or background process to continuously poll your queue.

CHAPTER 1

Content

Installation

The bundle should be installed through composer.

Add the bundle to composer

```
{
    "require": {
        "uecode/qpush-bundle": "~2.3.0",
    }
}
```

Update AppKernel.php of your Symfony Application

Add the `UecodeQPushBundle` to your kernel bootstrap sequence, in the `$bundles` array

```
public function registerBundles()
{
    $bundles = array(
        // ...
        new Uecode\Bundle\QPushBundle\UecodeQPushBundle(),
    );

    return $bundles;
}
```

Configure the Bundle

The bundle allows you to specify different Message Queue providers - however, Amazon AWS and IronMQ are the only ones currently supported. Blocking, synchronous queues are also supported through the `sync` driver to aid development and debugging.

We are actively looking to add more and would be more than happy to accept contributions.

Providers

This bundle allows you to configure and use multiple supported providers with in the same application. Each queue that you create is attached to one of your registered providers and can have its own configuration options.

Providers may have their own dependencies that should be added to your `composer.json` file.

For specific instructions on how to configure each provider, please view their documents.

AWS Provider

The AWS Provider uses SQS & SNS to create a Push Queue model. SNS is optional with this provider and its possible to use just SQS by utilizing the provided Console Command (`uecode:qpush:receive`) to poll the queue.

Configuration

This provider relies on the AWS SDK PHP library, which needs to be required in your `composer.json` file.

This bundle will support both v2 and v3 of the AWS SDK.

```
{
    require: {
        "aws/aws-sdk-php": : "2.*" #OR "3.*"
    }
}
```

From there, the rest of the configuration is simple. You need to provide your credentials in your configuration.

```
#app/config.yml

uecode_qpush:
    providers:
        my_provider:
            driver: aws
            key:    <aws key>
            secret: <aws secret>
            region: us-east-1
    queues:
        my_queue_name:
            provider: my_provider
            options:
                push_notifications: true
                subscribers:
                    - { endpoint: http://example.com/qpush, protocol: http }
```

You may exclude the aws key and secret if you are using IAM role in EC2.

Using SNS

If you set `push_notifications` to `true` in your queue config, this provider will automatically create the SNS Topic, subscribe your SQS queue to it, as well as loop over your list of `subscribers`, adding them to your Topic.

This provider automatically handles Subscription Confirmations sent from SNS, as long as the HTTP endpoint you've listed is externally accessible and has the QPush Bundle properly installed and configured.

Overriding Queue Options

It's possible to override the default queue options that are set in your config file when sending or receiving messages.

Publishing

The `publish()` method takes an array as a second argument. For the AWS Provider you are able to change the options listed below per publish.

If you disable `push_notifications` for a message, it will skip using SNS and only write the message to SQS. You will need to manually poll the SQS queue to fetch those messages.

Option	Description	Default Value
`push_notifications`	Whether or not to POST notifications to subscribers of a Queue	`false`
`message_delay`	Time in seconds before a published Message is available to be read in a Queue	`0`

```
$message = ['foo' => 'bar'];

// Optional config to override default options
$options = [
    'push_notifications' => 0,
    'message_delay'      => 1
];

$this->get('uecode_qpush.my_queue_name')->publish($message, $options);
```

Receiving

The `receive()` method takes an array as a second argument. For the AWS Provider you are able to change the options listed below per attempt to receive messages.

Option	Description	Default Value
`messages_to_receive`	Maximum amount of messages that can be received when polling the queue	`1`
`receive_wait_time`	If supported, time in seconds to leave the polling request open - for long polling	`3`

```
// Optional config to override default options
$options = [
    'messages_to_receive' => 3,
    'receive_wait_time'   => 10
];

$messages = $this->get('uecode_qpush.my_queue_name')->receive($options);

foreach ($messages as $message) {
    echo $message->getBody();
}
```

IronMQ Provider

The IronMQ Provider uses its Push Queues to notify subscribers of new queued messages without needing to continually poll the queue.

Using a Push Queue is optional with this provider and its possible to use simple Pull queues by utilizing the provided Console Command (`uecode:qpush::receive`) to poll the queue.

Configuration

This provider relies on the Iron MQ classes and needs to have the library included in your `composer.json` file.

```
{
    require: {
        "iron-io/iron_mq": "^4.0"
    }
}
```

Configuring the provider is very easy. It requires that you have already created an account and have a project id.

Iron.io provides free accounts for Development, which makes testing and using this service extremely easy.

Just include your OAuth *token* and *project_id* in the configuration and set your queue to use a provider using the *ironmq* driver.

```
#app/config.yml

uecode_qpush:
    providers:
        my_provider:
            driver:     ironmq
            token:      YOUR_TOKEN_HERE
            project_id: YOUR_PROJECT_ID_HERE
            host:       YOUR_OPTIONAL_HOST_HERE
            port:       YOUR_OPTIONAL_PORT_HERE
            version_id: YOUR_OPTIONAL_VERSION_HERE
    queues:
        my_queue_name:
            provider: my_provider
            options:
                push_notifications: true
                subscribers:
                    - { endpoint: http://example.com/qpush, protocol: http }
```

IronMQ Push Queues

If you set `push_notifications` to `true` in your queue config, this provider will automatically create your Queue as a Push Queue and loop over your list of `subscribers`, adding them to your Queue.

This provider only supports `http` and `https` subscribers. This provider also uses the `multicast` setting for its Push Queues, meaning that all `subscribers` are notified of the same new messages.

You can chose to have your IronMQ queues work as a Pull Queue by setting `push_notifications` to `false`. This would require you to use the `uecode:qpush:receive` Console Command to poll the queue.

Overriding Queue Options

It's possible to override the default queue options that are set in your config file when sending or receiving messages.

Publishing

The `publish()` method takes an array as a second argument. For the IronMQ Provider you are able to change the options listed below per publish.

Option	Description	Default Value
`message_delay`	Time in seconds before a published Message is available to be read in a Queue	`0`
`message_timeout`	Time in seconds a worker has to delete a Message before it is available to other workers	`30`
`message_expiration`	Time in seconds that Messages may remain in the Queue before being removed	`604800`

```
$message = ['foo' => 'bar'];

// Optional config to override default options
$options = [
    'message_delay'      => 1,
    'message_timeout'    => 1,
    'message_expiration' => 60
];

$this->get('uecode_qpush.my_queue_name')->publish($message, $options);
```

Receiving

The `receive()` method takes an array as a second argument. For the AWS Provider you are able to change the options listed below per attempt to receive messages.

Option	Description	Default Value
`messages_to_receive`	Maximum amount of messages that can be received when polling the queue	`1`
`message_timeout`	Time in seconds a worker has to delete a Message before it is available to other workers	`30`

```
// Optional config to override default options
$options = [
    'messages_to_receive' => 3,
    'message_timeout'     => 10
];

$messages = $this->get('uecode_qpush.my_queue_name')->receive($options);

foreach ($messages as $message) {
    echo $message->getBody();
}
```

Sync Provider

The sync provider immediately dispatches and resolves queued events. It is not intended for production use but instead to support local development, debugging and testing of queue-based code paths.

Configuration

To designate a queue as synchronous, set the `driver` of its provider to `sync`. No further configuration is necessary.

```
#app/config_dev.yml

uecode_qpush:
    providers:
        in_band:
            driver: sync
    queues:
        my_queue_name:
            provider: in_band
```

File Provider

The file provider uses the filesystem to dispatch and resolve queued messages.

Configuration

To designate a queue as file, set the `driver` of its provider to `file`. You will need to configure a readable and writable path to store the messages.

```
#app/config_dev.yml

uecode_qpush:
    providers:
        file_based:
            driver: file
            path: [Path to store messages]
    queues:
        my_queue_name:
            provider: file_based
```

Custom Provider

The custom provider allows you to use your own provider. When using this provider, your implementation must implement `Uecode\Bundle\QPushBundle\Provider\ProviderInterface`

Configuration

To designate a queue as custom, set the `driver` of its provider to `custom`, and the `service` to your service id.

```
#app/config_dev.yml

uecode_qpush:
    providers:
        custom_provider:
            driver: custom
            service: YOUR_CUSTOM_SERVICE_ID
    queues:
```

```
        my_queue_name:
            provider: custom_provider
```

Caching

Providers can leverage a caching layer to limit the amount of calls to the Message Queue for basic lookup functionality - this is important for things like AWS's ARN values, etc.

By default the library will attempt to use file cache, however you can pass your own cache service, as long as its an instance of `Doctrine\Common\Cache\Cache`.

The configuration parameter `cache_service` expects the container service id of a registered Cache service. See below.

```
#app/config.yml

services:
    my_cache_service:
        class: My\Caching\CacheService

uecode_qpush:
    cache_service: my_cache_service
```

Note: *Though the Queue Providers will attempt to create queues if they do not exist when publishing or receiving messages, it is highly recommended that you run the included console command to build queues and warm cache from the CLI beforehand.*

Queue Options

Each queue can have their own options that determine how messages are published or received. The options and their descriptions are listed below.

Option	Description	Default Value
`queue_name`	The name used to describe the queue on the Provider's side	`null`
`push_notifications`	Whether or not to POST notifications to subscribers of a Queue	`false`
`notification_retries`	How many attempts notifications are resent in case of errors - if supported	`3`
`message_delay`	Time in seconds before a published Message is available to be read in a Queue	`0`
`message_timeout`	Time in seconds a worker has to delete a Message before it is available to other workers	`30`
`message_expiration`	Time in seconds that Messages may remain in the Queue before being removed	`604800`
`messages_to_receive`	Maximum amount of messages that can be received when polling the queue	`1`
`receive_wait_time`	If supported, time in seconds to leave the polling request open - for long polling	`3`
`fifo`	If supported (only aws), sets queue into FIFO mode	`false`
`content_based_deduplication`	If supported (only aws), turns on automatic deduplication id based on the message content	`false`
`subscribers`	An array of Subscribers, containing an `endpoint` and `protocol`	`empty`

Symfony Application as a Subscriber

The QPush Bundle uses a Request Listener which will capture and dispatch notifications from your queue providers for you. The specific route you use does not matter.

In most cases, it is recommended to just list the host or domain for your Symfony application as the `endpoint` of your subscriber. You do not need to create a new action for QPush to receive messages.

Logging with Monolog

By default, logging is enabled in the Qpush Bundle and uses Monolog, configured via the MonologBundle. You can toggle the logging behavior by setting `logging_enabled` to `false`.

Logs will output to your default Symfony environment logs using the 'qpush' channel.

Example Configuration

A working configuration would look like the following

```
uecode_qpush:
    cache_service: null
    logging_enabled: true
    providers:
        aws:
            driver: aws #optional for providers named 'aws' or 'ironmq'
            key: YOUR_AWS_KEY_HERE
            secret: YOUR_AWS_SECRET_HERE
            region: YOUR_AWS_REGION_HERE
        another_aws_provider:
            driver: aws #required for named providers
            key: YOUR_AWS_KEY_HERE
            secret: YOUR_AWS_SECRET_HERE
            region: YOUR_AWS_REGION_HERE
        ironmq:
            driver: aws #optional for providers named 'aws' or 'ironmq'
            token: YOUR_IRONMQ_TOKEN_HERE
            project_id: YOUR_IRONMQ_PROJECT_ID_HERE
        in_band:
            driver: sync
        custom_provider:
            driver: custom
            service: YOUR_CUSTOM_SERVICE_ID
    queues:
        my_queue_key:
            provider: ironmq #or aws or in_band or another_aws_provider
            options:
                queue_name:                 my_actual_queue_name
                push_notifications:         true
                notification_retries:       3
                message_delay:              0
                message_timeout:            30
                message_expiration:         604800
                messages_to_receive:        1
                receive_wait_time:          3
                fifo:                       false
                content_based_deduplication: false
```

```
                subscribers:
                    - { endpoint: http://example1.com/, protocol: http }
                    - { endpoint: http://example2.com/, protocol: http }
        my_fifo_queue_key:
            provider: aws
            options:
                queue_name:                 my_actual_queue_name.fifo
                push_notifications:         true
                notification_retries:       3
                message_delay:              0
                message_timeout:            30
                message_expiration:         604800
                messages_to_receive:        1
                receive_wait_time:          3
                fifo:                       true
                content_based_deduplication: true
                subscribers:
                    - { endpoint: http://example1.com/, protocol: http }
                    - { endpoint: http://example2.com/, protocol: http }
```

Usage

Once configured, you can create messages and publish them to the queue. You may also create services that will automatically be fired as messages are pushed to your application.

For your convenience, a custom `Provider` service will be created and registered in the Container for each of your defined Queues. The container queue service id will be in the format of `uecode_qpush.{your queue name}`.

Publishing messages to your Queue

Publishing messages is simple - fetch your `Provider` service from the container and call the `publish` method on the respective queue, which accepts an array.

```
#src/My/Bundle/ExampleBundle/Controller/MyController.php

public function publishAction()
{
    $message = [
        'messages should be an array',
        'they can be flat arrays' => [
            'or multidimensional'
        ]
    ];

    $this->get('uecode_qpush.my_queue_name')->publish($message);
}
```

Working with messages from your Queue

Messages are either automatically received by your application and events dispatched (setting `push_notification` to `true`), or can be picked up by Cron jobs through an included command if you are not using a Message Queue provider that supports Push notifications.

When the notifications or messages are Pushed to your application, the QPush Bundle automatically catches the request and dispatches an event which can be easily hooked into.

MessageEvents

Once a message is received via POST from your Message Queue, a `MessageEvent` is dispatched which can be handled by your services. Each `MessageEvent` contains the name of the queue and a `Uecode\Bundle\QPushBundle\Message\Message` object, accessible through getters.

```
#src/My/Bundle/ExampleBundle/Service/ExampleService.php

use Uecode\Bundle\QPushBundle\Event\MessageEvent

public function onMessageReceived(MessageEvent $event)
{
    $queue_name = $event->getQueueName();
    $message    = $event->getMessage();
}
```

The `Message` objects contain the provider specific message id, a message body, and a collection of provider specific metadata.

These properties are accessible through simple getters.

The message `body` is an array matching your original message. The `metadata` property is an `ArrayCollection` of varying fields sent with your message from your Queue Provider.

```
#src/My/Bundle/ExampleBundle/Service/ExampleService.php

use Uecode\Bundle\QPushBundle\Event\MessageEvent;
use Uecode\Bundle\QPushBundle\Message\Message;

public function onMessageReceived(MessageEvent $event)
{
    $id         = $event->getMessage()->getId();
    $body       = $event->getMessage()->getBody();
    $metadata   = $event->getMessage()->getMetadata();

    // do some processing
}
```

Tagging Your Services

For your Services to be called on QPush events, they must be tagged with the name `uecode_qpush.event_listener`. A complete tag is made up of the following properties:

Tag Property	Example	Description
`name`	`uecode_qpush.event_listener`	The Qpush Event Listener Tag
`event`	`{queue name}.message_received`	The *message_received* event, prefixed with the Queue name
`method`	`onMessageReceived`	A publicly accessible method on your service
`priority`	`100`	Priority, `1-100` to control order of services. Higher priorities are called earlier

The `priority` is useful to chain services, ensuring that they fire in a certain order - the higher priorities fire earlier.

Each event fired by the Qpush Bundle is prefixed with the name of your queue, ex: `my_queue_name.message_received`.

This allows you to assign services to fire only on certain queues, based on the queue name. However, you may also have multiple tags on a single service, so that one service can handle events from multiple queues.

```
services:
    my_example_service:
    class: My\Example\ExampleService
    tags:
        - { name: uecode_qpush.event_listener, event: my_queue_name.message_received,␣
↪method: onMessageReceived }
```

The method listed in the tag must be publicly available in your service and should take a single argument, an instance of `Uecode\Bundle\QPushBundle\Event\MessageEvent`.

```
#src/My/Bundle/ExampleBundle/Service/MyService.php

use Uecode\Bundle\QPushBundle\Event\MessageEvent;

// ...

public function onMessageReceived(MessageEvent $event)
{
    $queueName  = $event->getQueueName();
    $message    = $event->getMessage();
    $metadata   = $message()->getMetadata();

    // Process ...
}
```

Cleaning Up the Queue

Once all other Event Listeners have been invoked on a `MessageEvent`, the QPush Bundle will automatically attempt to remove the Message from your Queue for you.

If an error or exception is thrown, or event propagation is stopped earlier in the chain, the Message will not be removed automatically and may be picked up by other workers.

If you would like to remove the message inside your service, you can do so by calling the `delete` method on your provider and passing it the message `id`. However, you must also stop the event propagation to avoid other services (including the Provider service) from firing on that `MessageEvent`.

```
#src/My/Bundle/ExampleBundle/Service/MyService.php

use Uecode\Bundle\QPushBundle\Event\MessageEvent;

// ...

public function onMessageReceived(MessageEvent $event)
{
    $id = $event->getMessage()->getId();
    // Removes the message from the queue
    $awsProvider->delete($id);

    // Stops the event from propagating
```

```
    $event->stopPropagation();
}
```

Push Queues in Development

It is recommended to use your `config_dev.yml` file to disable the `push_notifications` settings on your queues. This will make the queue a simple Pull queue. You can then use the `uecode:qpush:receive` Console Command to receive messages from your Queue.

If you need to test the Push Queue functionality from a local stack or internal machine, it's possible to use ngrok to tunnel to your development environment, so its reachable by your Queue Provider.

You would need to update your *config_dev.yml* configuration to use the *ngrok* url for your subscriber(s).

Console Commands

This bundle includes some Console Commands which can be used for building, destroying and polling your queues as well as sending simple messages.

Build Command

You can use the `uecode:qpush:build` command to create the queues on your providers. You can specify the name of a queue as an argument to build a single queue. This command will also warm cache which avoids the need to query the provider's API to ensure that the queue exists. Most queue providers create commands are idempotent, so running this multiple times is not an issue.:

```
$ php app/console uecode:qpush:build my_queue_name
```

Note: *By default, this bundle uses File Cache. If you clear cache, it is highly recommended you re-run the build command to warm the cache!*

Destroy Command

You can use the `uecode:qpush:destroy` command to completely remove queues. You can specify the name of a queue as an argument to destroy a single queue. If you do not specify an argument, this will destroy all queues after confirmation.:

```
$ php app/console uecode:qpush:destroy my_queue_name
```

Note: *This will remove queues, even if there are still unreceived messages in the queue!*

Receive Command

You can use the `uecode:qpush:receive` command to poll the specified queue. This command takes the name of a queue as an argument. Messages received from this command are dispatched through the `EventDispatcher` and can be handled by your tagged services the same as Push Notifications would be.:

```
$ php app/console uecode:qpush:receive my_queue_name
```

Publish Command

You can use the `uecode:qpush:publish` command to send messages to your queue from the CLI. This command takes two arguments, the name of the queue and the message to publish. The message needs to be a json encoded string.:

```
$ php app/console uecode:qpush:publish my_queue_name '{"foo": "bar"}'
```