



NET2GRID

Platform Engineer Assignment:

Eleta Ilias

March 12, 2021

Table of Contents

1. INTRODUCTION	3
1.1 PURPOSE OF THIS DOCUMENT	3
2. SOFTWARE PREREQUISITES	3
2.1 PROGRAMS USED	3
3. ASSIGNMENT EXPLANATION	4
3.1 THE PROGRAM WORKFLOW	4
3.2 PRODUCER CODE EXPLANATION (producer.php)	4
Connection	4
API Endpoint	6
Creating the Routing Key	6
Message Body	7
Queue Bind	9
Basic Publish	9
3.3 CONSUMER CODE EXPLANATION (consumer.php)	10
Connection	10
\$callback variable	11
The \$callback variable	13
4. UNRESOLVED ISSUES	15
5. TESTING	16
6. PROBLEM SOLVING PROCESS	22

1. INTRODUCTION

1.1 PURPOSE OF THIS DOCUMENT

The purpose of this document is to provide all the basic necessary info regarding the Platform Engineer Assignment

2. SOFTWARE PREREQUISITES

2.1 PROGRAMS USED

- For IDE I used [PHPSTORM](#).
- I only used [Symfony 4](#) but only for the project set up.
- I downloaded [XAMPP](#).
- [RabbitMQ](#) and [Erlang](#).

3. ASSIGNMENT EXPLANATION

3.1 THE PROGRAM WORKFLOW

The workflow of the program, consists of two main levels:

1. The **producer** who takes an API endpoint, creates a routing key with the given Id's and publishes the message to a RabbitMQ instance.
2. The **consumer** who takes the message from the RabbitMQ instance and inserts it to a database named `cand_e2ro`.

The related configuration and functionality are separated into three classes.

1. The class **config.php** where I set up my configuration variables and I retrieve them through "*get-type*" functions.
2. The class **messageQueueService.php** which contains all the functions related to the Message Queue set up and connectivity.
3. The class **dbService.php** which contains all the functions related to the Database calls my implementation needs to make.
4. The class **helpers.php** which contains all the functions related to data handling - formatting - set up for clearance in the basic implementation files.

3.2 PRODUCER CODE EXPLANATION (producer.php)

Before we take our data from the API endpoint, we must create a connection first to the RabbitMQ service and set up the message queue settings.

Connection

The functions to set up the connection are in the class named **messageQueueService.php** and the message queue details (including the API endpoint URL) are declared to the class named **config.php**.

- The message queue details are created in the function **get_amqp_connection** where it returns the constant details of the message queue (*host, port, username, password*)
- After that, the **set_up_channel** function is used to set up the connection to the RabbitMQ channel and declare the queue and the exchange of that channel.

```
function get_amqp_connection(): AMQPStreamConnection
{
    $config = new config();
    return new AMQPStreamConnection($config->get_host(), $config->get_port(), $config->get_user(), $config->get_password());
}

function set_up_channel($config, $connection)
{
    // Open the connection channel
    $channel = $connection->channel();

    // Declare the queue configuration settings
    $channel->queue_declare($config->get_queue(), false, true, false, false);

    // Declare the exchange configuration settings
    $channel->exchange_declare($config->get_exchange(), 'direct', false, true, false);

    return $channel;
}
```

In order to create the connection in our **producer.php** file

1. We initialize an object of the **config** class and the **messageQueueService** into two variables in our scope. (*\$config*, *\$messageQueueService*)
2. We set up our connection in a variable through the function **get_amqp_connection** from our service instance. (*\$connection*)
3. We set up our communication channel in a variable through the function **set_up_channel** from our service instance. (*\$channel*)

```
<?php
require_once(__DIR__ . '/vendor/autoload.php');
include 'helpers.php';
include 'messageQueueService.php';

$config = new config();
$messageQueueService = new messageQueueService();

/**
 * Set up connection
 */
$connection = $messageQueueService -> get_amqp_connection();

/**
 * Set up channel
 */
$channel = $messageQueueService -> set_up_channel($config, $connection);
```

Now that the connection is set up, we need to retrieve the necessary information from the API endpoint and create a routing key and the message body based on the assignment specification.

API Endpoint

- Since it is a URL, I used the **Client URL Library**¹ (cURL) functions. In my **helpers.php** file I created a custom function were
 - I pass the URL as a variable.
 - I retrieve the response body.
 - I return the decoded message through **json_decode**.²

```
/**
 * Get the api results from url and return decoded message body
 */
function get_api_results($URL)
{
    $client = curl_init($URL);
    curl_setopt($client, option: CURLOPT_RETURNTRANSFER, value: true);
    $response = curl_exec($client);
    $decoded_message = json_decode($response);
    return($decoded_message);
}
```

- In the **producer.php** file I use the function to retrieve my decoded message in a local variable named **\$result**.

```
// Connect to the API and get the messages payload
$result = get_api_results($config->get_apiUrl());
```

Creating the Routing Key

To create a valid routing key (*according to assignment specification*) I needed to get the decimal equivalent id's (profileId, endpointId etc) from my payload, since the API result was providing the info in hexadecimal format (0x). After that I needed to correctly format the routing key in the specified format (X.X.X.X)

¹ <https://www.php.net/manual/en/book.curl.php>

² <https://www.php.net/manual/en/function.json-decode.php>

- In order to do that, I created two functions in the **helpers.php** file named **hex_to_dec** and **set_formated_routing_key**.
 - The **hex_to_dec** function takes a hexadecimal number and converts it to a decimal.
 - The **set_formated_routing_key** returns a string message in the following format: **<gateway eui>. <profile>. <endpoint>. <cluster>. <attribute>**.

```
// Convert hexadecimal to decimal
//
function hex_to_dec($hex): int|string
{
    $dec = 0;
    $len = strlen($hex);
    for ($i = 1; $i <= $len; $i++) {
        $dec = bcadd($dec, bcmul(strval(hexdec($hex[$i - 1])), bcpow($mod, '10', strval($mod * $len - $i))));
    }
    return $dec;
}

// Format routing key x.x.x.x
//
function set_formated_routing_key($result): string
{
    return print_r(hex_to_dec($result->gatewayEui) . " " . hex_to_dec($result->profileId) . " " . hex_to_dec($result->endpointId) . " " . hex_to_dec($result->clusterId) . " " . hex_to_dec($result->attributeId) . " ", true);
}
```

In our **producer.php** file we call the **set_formated_routing_key** to retrieve our formatted routing key and assign its value in the local variable **\$routing_key**.

```
// Connect to the API and get the messages payload
$result = get_api_results($config->get_apiUrl());

// Set up the routing key from response
$routing_key = set_formated_routing_key($result);
```

Message Body

The first thing I noticed when retrieving the payload, is that the **timestamp** was in Unix/Epoch format³ and should be converted to “*human-friendly*” date.

- I created a new function in the **helper.php** file named **unix_to_utc_formatted** which return a formatted date string (Year-Month-Day Hour: Minutes: Seconds) in UTC⁴.

³ https://en.wikipedia.org/wiki/Unix_time

⁴ <https://www.w3.org/TR/NOTE-datetime>

```
/**
 * Format to UTC date
 */
function unix_to_utc_formatted($result): string
{
    date_default_timezone_set( timezoneid: 'UTC');
    return print_r(date( format: "Y-m-d H:i:s", substr($result->timestamp, offset: 0, length: 10)), return: true);
}
```

- I decided to post the values (*timestamp*, *value*) in json so that it would be easier to consume later. In order to encode them in json, I added the values into an array and assigned the array in the local variable **\$result_message**.

```
// set up the message payload
$result_message = array("Date" => $datetime, "Value" => $result->value);
```

- Before we publish the message to RabbitMQ we must set the AMQP⁵ message. I created a function in my services file (*messageQueueService.php*) where we return a new instance of **AMQPMessage** class with⁶
 - **content_type**: application/json => since our message is in json format
 - **delivery_mode**: 2 (persistent) => to make our message persistent

```
/**
 * Set up a new AMQP message
 */
function set_new_amqp_message($result_message)
{
    return new AMQPMessage(json_encode($result_message), array(
        'content_type' => 'application/json',
        'delivery_mode' => 2 #make message persistent.
    ));
}
```

⁵ <https://github.com/php-amqplib/php-amqplib/blob/master/PhpAmqpLib/Message/AMQPMessage.php>

⁶ <https://www.rabbitmq.com/tutorials/amqp-concepts.html>

Queue Bind

After that we go and bind our queue with the exchange based on our routing key. The **queue_bind** function belongs to the **AMQPStreamConnection**⁷.

```
// bind the queue and the exchange with a routing key.  
$channel->queue_bind($config->get_queue(), $config->get_exchange(), $routing_key);
```

Basic Publish

After that we publish our message in the channel (queue - exchange) through the function **basic_publish** of the channel which belongs to the **AMQPStreamConnection** library.

```
// publish the message  
$channel->basic_publish($message, $config->get_exchange(), $routing_key);
```

The Whole Process of the producer.php (*without the connection setup*)

```
$limit = 0; // The variable limit is used so that we can control how many messages we want to publish  
$id = 1;  
echo " [*] The message will now start being published".PHP_EOL;  
sleep( seconds: 1 );  
while ( $limit < 10 )  
{  
    // Connect to the API and get the messages payload  
    $result = get_api_results($config->get_apiUrl());  
  
    // Set up the routing key from response  
    $routing_key = set_formatted_routing_key($result);  
  
    // Get the formatted datetime from response  
    $datetime = timestamp_to_human_date($result);  
  
    // set up the message payload  
    $result_message = array("Date" => $datetime, "Value" => $result->value);  
  
    // set the AMQP message  
    $message = $messageQueueService->set_new_amqp_message($result_message);  
  
    // bind the queue and the exchange with a routing key.  
    $channel->queue_bind($config->get_queue(), $config->get_exchange(), $routing_key);  
  
    // publish the message  
    $channel->basic_publish($message, $config->get_exchange(), $routing_key);  
  
    print ' [x] ' . $id . ' messages are published.' . PHP_EOL;  
    $id++;  
    $limit++;  
}  
  
print ' [*] All messages are published to the queue' . PHP_EOL;
```

⁷ <https://github.com/php-amqplib/php-amqplib/blob/master/PhpAmqpLib/Connection/AMQPStreamConnection.php>

3.3 CONSUMER CODE EXPLANATION (consumer.php)

Connection

The connection process is the same as the one in the **producer.php** file.

```
<?php
require_once(__DIR__ . '/vendor/autoload.php');

include 'helpers.php';
include 'messageQueueService.php';
include 'dbService.php';

$config = new config();
$messageQueueService = new messageQueueService();
$dbService = new dbService();

/**
 * Set up connection
 */
$connection = $messageQueueService -> get_amqp_connection();

/**
 * Set up channel
 */
$channel = $messageQueueService -> set_up_channel($config, $connection);

/**
 * Bind the queue and the exchange together.
 */
$channel->queue_bind($config->get_queue(), $config->get_exchange());
```

The idea here is that since we are binding to the same exchange, we can consume the information published from the producer with the **basic_consume** function which belongs to the **AMQPStreamConnections** library.

```
/**
 * Initialize consuming of the callback implementation in the queue
 */
$channel->basic_consume($config->get_queue(), $consumerTag= '', false, false, false, false, $callback);
```

\$callback variable

- **\$callback** is the function “doing all the work”. We declare with the published message as a parameter. The message is the one that we had set in the **producer.php** which is now consumed with the help of the **basic_consume** function.
- Since we have the message body “json encoded” from the producer.php, we decode with **json_decode** and assign the result to the **\$messageBody** variable.
- In order to insert all the values in the database, we need all the values from the API, but we only have the **routing key** and the **message body**.
- Since the routing key is a formatted combination of the related identification values, we created a function in the **helpers.php** file named **get_routing_key** to retrieve initially it's value from the **delivery_info** object of the message. (name of object variable => delivery_tag)

```
/**
 * Get the routing key from the message payload
 */
function get_routing_key($message)
{
    $message->delivery_info['channel']->basic_ack($message->delivery_info['delivery_tag']);
    return $message->get('routing_key');
}
```

- Now that we have our routing key, we can parse it to retrieve the original Id's and turn them to integers. We are doing that with the help of the **get_routing_ids_in_array** function in the **helpers.php**.
 - There, we split the routing_key string by the separator “.”
 - Populate an array with the values and return the integer array with our Id's.

```
/**
 * Parse the routing identifiers into an array
 */
function get_routing_ids_in_array($routing): array
{
    $Ids = explode( separator: ".", $routing);
    for($i=1;$i < count($Ids); $i++)
    {
        $Ids[$i] = intval($Ids[$i]);
    }
    return $Ids;
}
```

- I use the **create_db** function from the **dbService.php** class where we create the database with the given details

```
function create_db()
{
    $conn = new mysqli( hostname: 'candidatends.n2g-dev.net', username: 'cand_e2ro', password: 'awTAS6m1hjJzVRg4', database: 'cand_e2ro');
    echo " [x] Trying to connect to the database".PHP_EOL;
    if ($conn == FALSE) {
        die(" [-] ERROR: Could not connect.");
    }
    echo " [x] Connected successfully", "\n";

    $sql = "CREATE TABLE queue_messages(filtered_messages_Id int NOT NULL AUTO_INCREMENT,
    gatewayEui varchar(20) NOT NULL,
    profileId int(10) NOT NULL,
    endpointId int(10) NOT NULL,
    clusterId int(10) NOT NULL,
    attributeId int(10) NOT NULL,
    value bigint(30) NOT NULL,
    date datetime NOT NULL,
    routing_key varchar(100) NOT NULL,
    PRIMARY KEY (filtered_messages_Id))";
    echo "[x] Table created successfully";
    $conn->query($sql);
}
}
```

- The Database schema shown below.

cand_e2ro queue_messages	
🔑	filtered_Messages_Id : int(20)
📄	gatewayEui : varchar(20)
#	profileId : int(10)
#	endpointId : int(10)
#	clusterId : int(10)
#	attributeId : int(10)
#	value : bigint(30)
📅	date : datetime
📄	routing_key : varchar(100)

- Lastly, I created the **insert_to_database** function which belongs to the **dbService.php** class as well, and its job is to insert our consumed data to the created database.

```

const hostname = 'candidaterds.n2g-dev.net';
const username = 'cand_e2ro';
const password = 'awTAS6m1hjJzVRg4';
const database = 'cand_e2ro';
const hostname = 'localhost';
const username = 'root';
const password = '';
const database = 'cand_e2ro';

/**
 * Open a connection to the database
 * Insert the error message in the database
 */
function insert_to_database($messageIds, $messageBody, $routing)
{
    $conn = new mysqli(hostname: self::hostname, username: self::username, password: self::password, database: self::database);
    // $conn = new mysqli(self::hostname, self::username, self::password, self::database);
    echo " [x] Trying to connect to the database".PHP_EOL;
    if ($conn === FALSE) {
        die(" [-] ERROR: Could not connect.");
    }
    echo " [x] Connected successfully", "\n";
    $sql = "INSERT INTO queue_messages(gatewayEui, profileId, endpointId, clusterId, attributeId, value, date, routing_key)
VALUES($messageIds[0], $messageIds[1], $messageIds[2], $messageIds[3], $messageIds[4], $messageBody->Value, $messageBody->Date, $routing)";

    if (mysqli_query($conn, $sql)) {
        echo " [x] New record created successfully" . PHP_EOL;
    } else {
        echo "Error: " . $sql . "<br>" . mysqli_error($conn);
    }

    echo " [x] Done", "\n";
    $conn->query($sql);

    $conn->close();
}

```

The \$callback variable

```

/**
 * Set up the callback function to be consumed
 * Implementation: Decode the message and insert it into the database
 */
$callback = function($message)
{
    $dbService = new dbService();

    echo " [x] Received ", $message->body, "\n";

    // Decode the consumed message body
    $messageBody = json_decode($message->body);

    // Set up the routing key
    $routing = get_routing_key($message);

    // Insert messages in DB
    $dbService->create_db();
    $dbService->insert_to_database(get_routing_ids_in_array($routing), $messageBody, $routing);
};

```

After the setup of the consumer, it will keep waiting for messages indefinitely. (unless we press Ctrl + C).

- In order to make the consumer stop automatically when there are no messages in the feed, I created a function named **initialize_channel** inside the **helpers.php** file. This function is basically an “exception handler” when there are not any messages to consume, we make the consumer to wait (sleep⁸) for as long the **\$timeout** variable is set and then it will automatically stop.

```
function initialize_channel($channel)
{
    try {
        while (count($channel->callbacks)) {
            print " [*] Waiting for more messages..." . PHP_EOL;
            //Variable $timeout counts in seconds so now he will wait 3 seconds before closing
            $channel->wait($allowed_methods = null, $nonBlocking = true, $timeout = 3);
        }
    } catch (Exception $e) {
        print " [*] There are no more tasks in the queue." . PHP_EOL;
        sleep( seconds: 1);
        print " [*] The consumer will automatically close.";
        sleep( seconds: 1);
    }
}
```

The Whole Process of the consumer.php (without the connection setup)

⁸ <https://www.php.net/manual/en/function.sleep.php>

```
/**
 * Set up the callback function to be consumed
 * Implementation: Decode the message and insert it into the database
 */
$callback = function($message)
{
    $dbService = new dbService();

    echo " [x] Received ", $message->body, "\n";

    // Decode the consumed message body
    $messageBody = json_decode($message->body);

    // Set up the routing key
    $routing = get_routing_key($message);

    // Insert messages in DB
    $dbService->create_db();
    $dbService ->insert_to_database(get_routing_ids_in_array($routing),$messageBody,$routing);
};

/**
 * Initialize consuming of the callback implementation in the queue
 */
$channel->basic_consume($config->get_queue(), $consumerTag= '', false, false, false, false, $callback);

/**
 * Initialized the channel
 */
initialize_channel($channel);

$channel->close();
$connection->close();
```

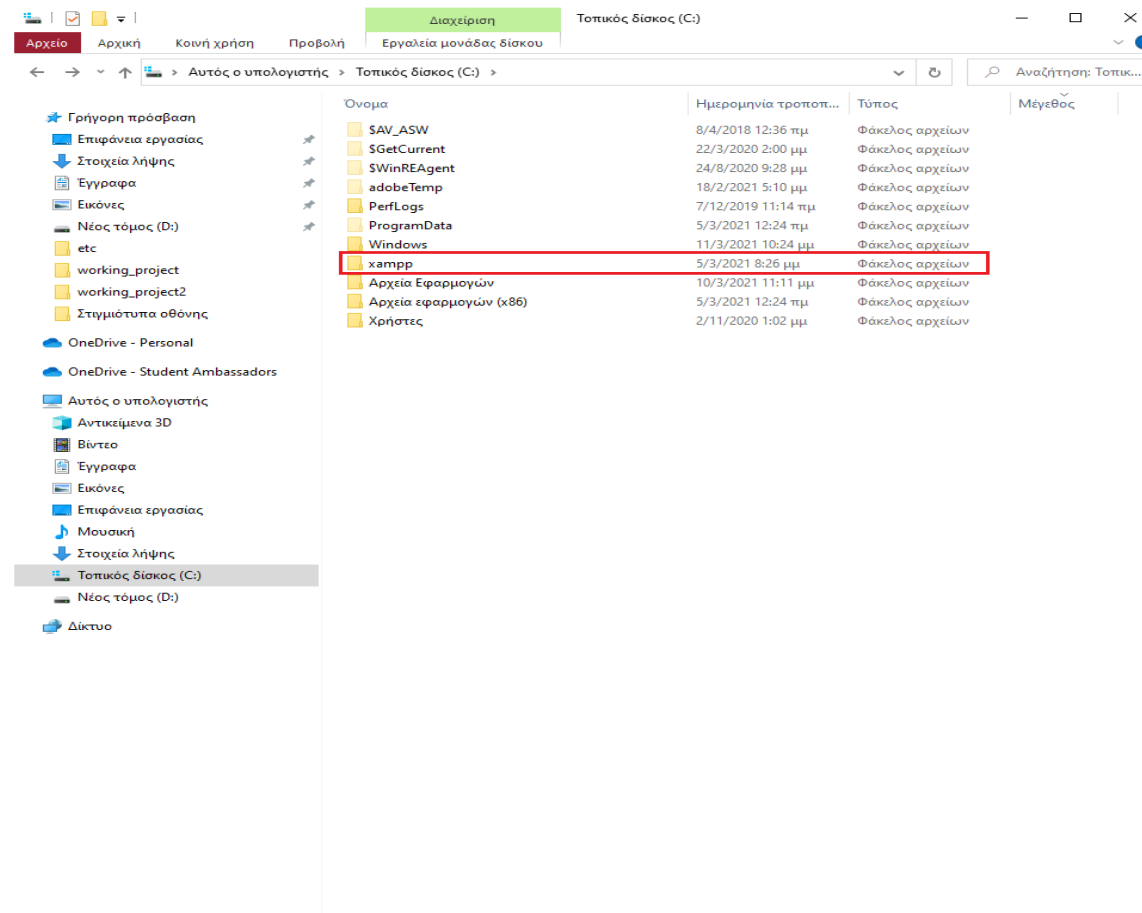
4. UNRESOLVED ISSUES

The problem that I encountered, and I couldn't overcome was that I couldn't change the node name on RabbitMQ, so the hostname is '127.0.0.1' and not 'rabbitmq.candidatemq.n2g-dev.net', so I am running RabbitMQ locally.

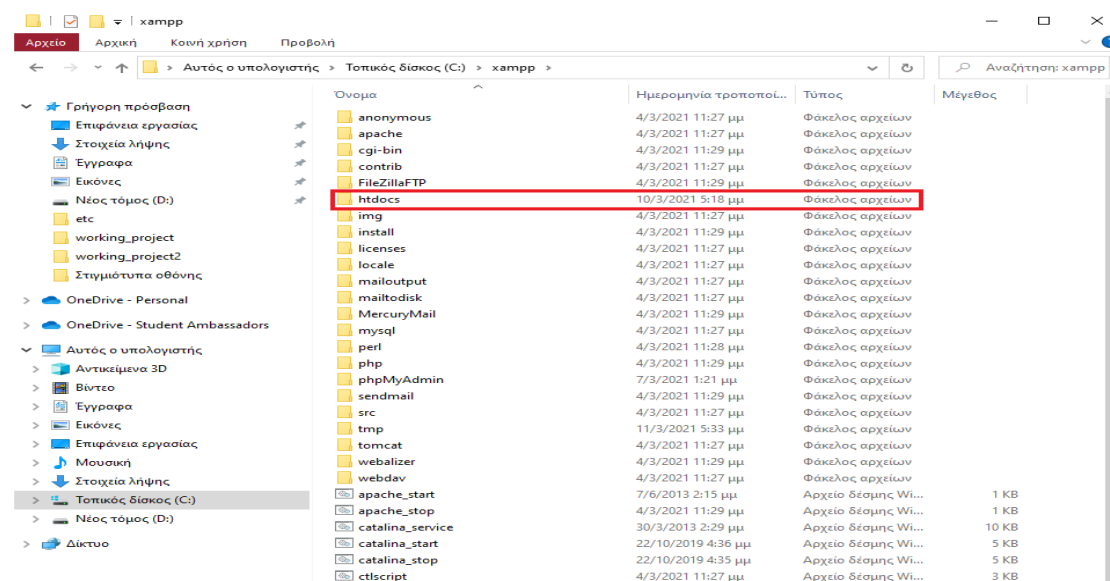
```
const host = '127.0.0.1';
const port = '5672';
const user = 'cand_e2ro';
const password = 'awTAS6m1hjJzVRg4';
const exchange = 'cand_e2ro';
const queue = 'cand_e2ro_results';
```

5. TESTING

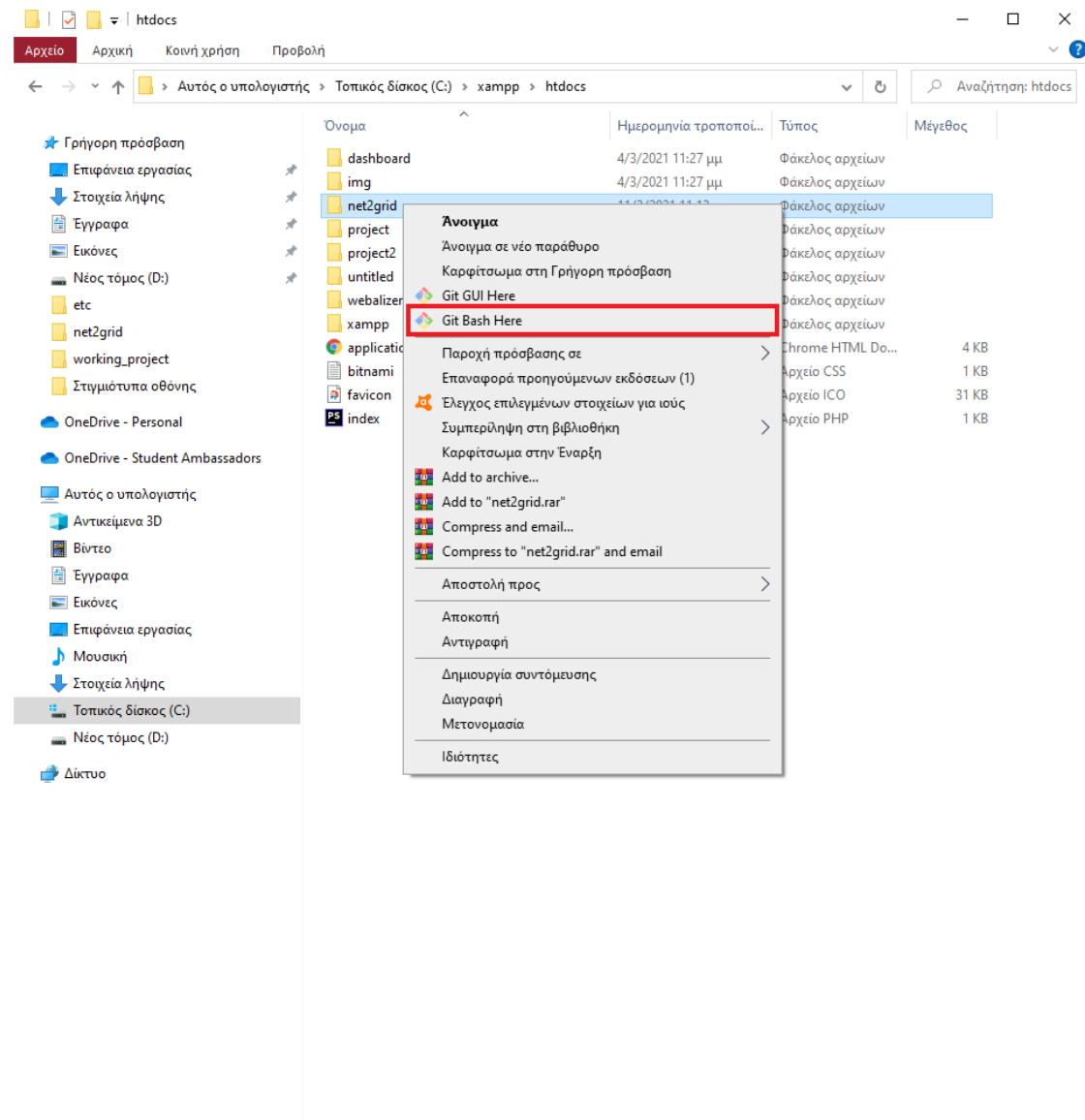
In order to run the project successfully we must insert the project file(**net2grid**) to the correct directory. We go to our downloaded **xampp** file.



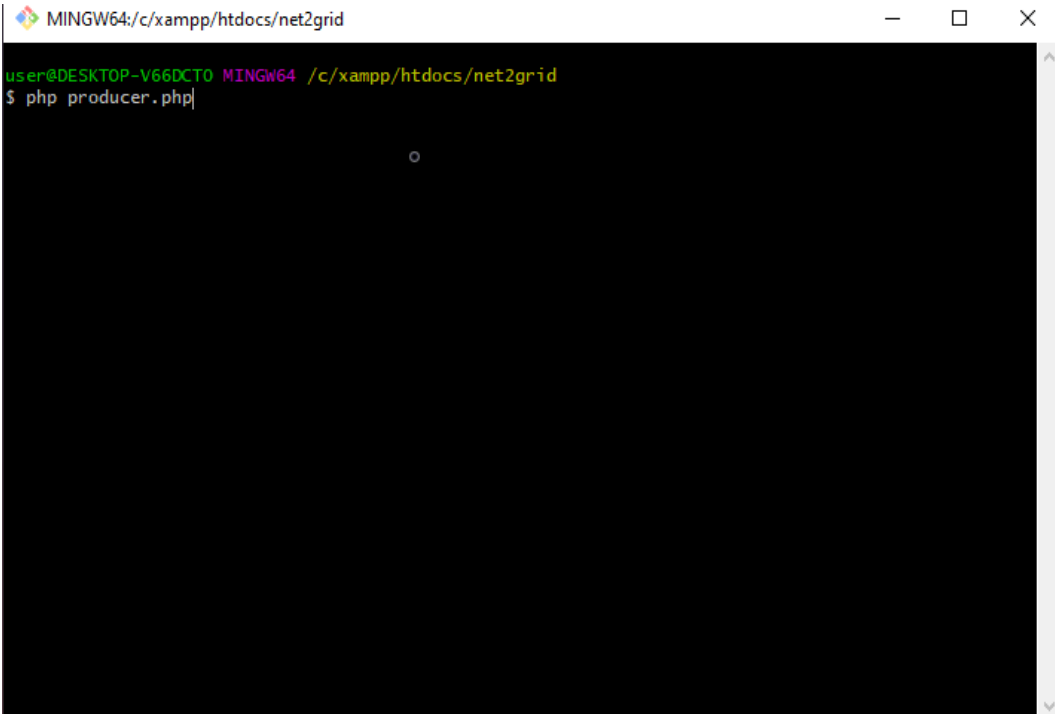
Then, we place the project file(**net2grid**) inside the **htdocs** file.



Now in order to run the program we must go inside the **htdocs** file and right click our project (**net2grid**) and click **Git Bash Here**

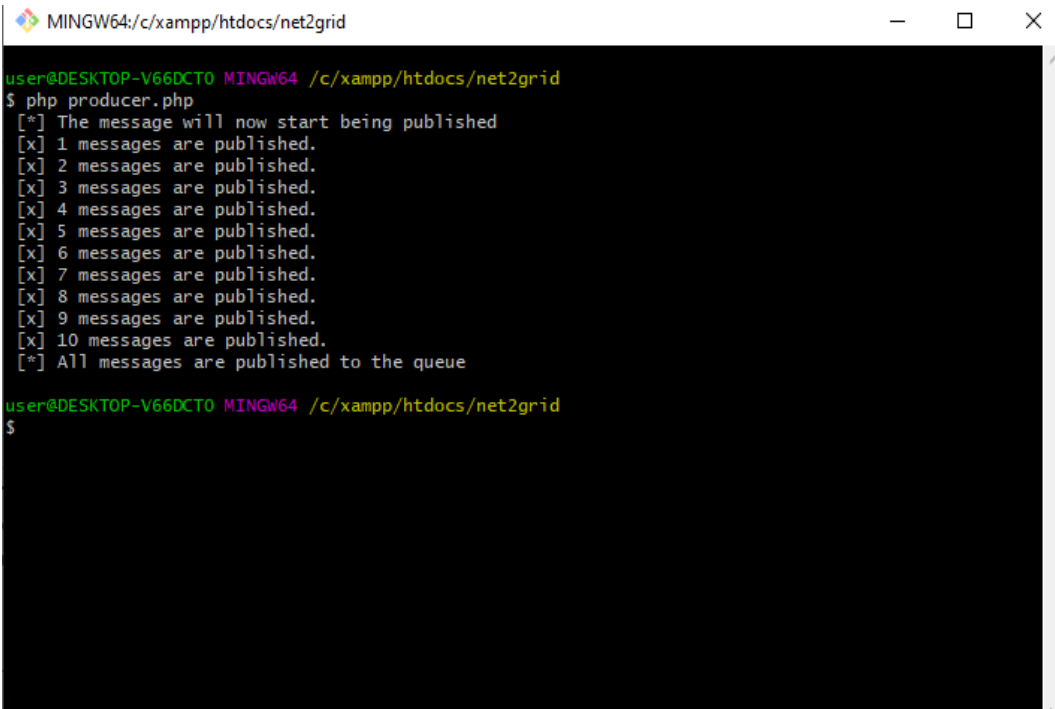


After we do that, a black window will show up where we can run our project where we type inside of it the command (**php producer.php**).



```
MINGW64:/c/xampp/htdocs/net2grid
user@DESKTOP-V66DCT0 MINGW64 /c/xampp/htdocs/net2grid
$ php producer.php
```

When we will hit enter the program will now start running and publish the messages to RabbitMQ



```
MINGW64:/c/xampp/htdocs/net2grid
user@DESKTOP-V66DCT0 MINGW64 /c/xampp/htdocs/net2grid
$ php producer.php
[*] The message will now start being published
[x] 1 messages are published.
[x] 2 messages are published.
[x] 3 messages are published.
[x] 4 messages are published.
[x] 5 messages are published.
[x] 6 messages are published.
[x] 7 messages are published.
[x] 8 messages are published.
[x] 9 messages are published.
[x] 10 messages are published.
[*] All messages are published to the queue
user@DESKTOP-V66DCT0 MINGW64 /c/xampp/htdocs/net2grid
$
```

If we head to the RabbitMQ Management, we can see that a queue and an exchange were made and that the messages were sent with their binding key.

← → ↻ localhost:15672/#/queues

RabbitMQ™ RabbitMQ 3.8.14 Erlang 23.2 Refreshed 2021-03-12 00:25:40 Refresh every 5 seconds

Virtual host All Cluster **rabbit@DESKTOP-V66DCT0** User **cand_e2ro** Log out

Overview Connections Channels Exchanges **Queues**

Admin

Queues

▼ All queues (1)

Pagination

Page 1 of 1 - Filter: ☐ Regexp ? Displaying 1 item , page size up to: 100

Overview				Messages			Message rates			
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
cand_e2ro_results	classic	D	idle	10	0	10	0.00/s			

▼ Add a new queue

Type: Classic

Name:

Durability: Durable

Auto delete: ☐ No

Arguments: = String

Add Message TTL ? Auto expire ? Max length ? Max length bytes ? Overflow behaviour ? Dead letter exchange ? Dead letter routing key ? Single active consumer ? Maximum priority ? Lazy mode ? Master locator ?

Add queue

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub

Changelog

← → ↻ localhost:15672/#/queues/%2F/cand_e2ro_results

RabbitMQ™ RabbitMQ 3.8.14 Erlang 23.2 Refreshed 2021-03-12 00:25:51 Refresh every 5 seconds


Virtual host All Cluster **rabbit@DESKTOP-V66DCT0** User **cand_e2ro** Log out

Overview Connections Channels Exchanges **Queues** Admin

Queue cand_e2ro_results


▼ Overview

Queued messages last minute ?



Ready 10 Unacked 0 Total 10

Message rates last minute ?



Publish 0.00/s

Details

Features	State
Policy	idle
Operator policy	Consumers 0
Effective policy definition	Consumer capacity ? 0%

	Total	Ready	Unacked	In memory	Persistent	Transient, Paged Out
Messages ?	10	10	0	10	10	0
Message body bytes ?	443 B	443 B	0 B	443 B	443 B	0 B
Process memory ?	28 KiB					

► Consumers

► Bindings

► Publish message

▼ Get messages

Warning: getting messages from a queue is a destructive action. ?

localhost:15672/#/queues/%2F/cand_e2ro_results

Refreshed 2021-03-12 00:26:06 Refresh every 5 seconds

Virtual host All

Cluster rabbit@DESKTOP-V66DCT0

User cand_e2ro Log out

Overview Connections Channels Exchanges

Queues Admin

	Messages ?	10	10	0	10	10	0
Message body bytes ?	443 B	443 B	0 B	443 B	443 B	0 B	0 B
Process memory ?	28 kiB						

Consumers

Bindings

From	Routing key	Arguments	
(Default exchange binding)			
cand_e2ro	9574384527092989662.260.10.1794.0		Unbind
cand_e2ro	9574384527254748409.260.11.1794.0		Unbind
cand_e2ro	9574384527297338174.260.10.1794.1024		Unbind
cand_e2ro	9574384527591293622.260.11.1794.0		Unbind
cand_e2ro	9574384527668880908.260.11.1794.0		Unbind
cand_e2ro	9574384527683871101.260.10.1794.1024		Unbind
cand_e2ro	9574384527949597969.260.11.1794.0		Unbind
cand_e2ro	9574384529330636094.260.11.1794.0		Unbind
cand_e2ro	9574384530668262239.260.10.1794.1024		Unbind
cand_e2ro	9574384531104650557.260.10.1794.0		Unbind

↓

This queue

localhost:15672/#/exchanges

Refreshed 2021-03-12 00:25:45 Refresh every 5 seconds

Virtual host All

Cluster rabbit@DESKTOP-V66DCT0

User cand_e2ro Log out

Overview Connections Channels Exchanges Queues

Admin

Exchanges

All exchanges (8)

Pagination

Page 1 of 1 - Filter: ☐ Regex ?

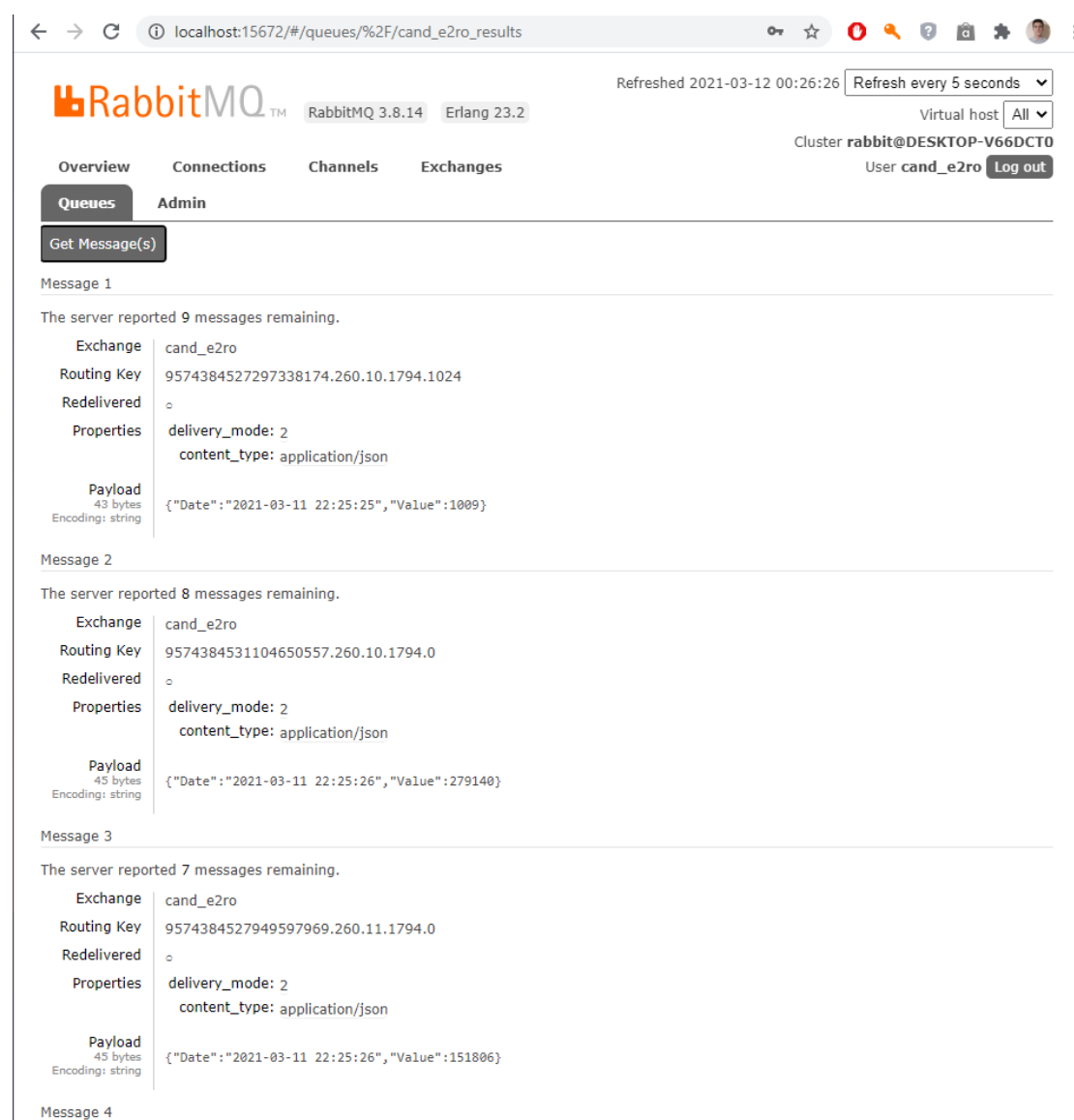
Displaying 8 items , page size up to: 100

Name	Type	Features	Message rate in	Message rate out	+/-
(AMQP default)	direct	D			
amq.direct	direct	D			
amq.fanout	fanout	D			
amq.headers	headers	D			
amq.match	headers	D			
amq.rabbitmq.trace	topic	D I			
amq.topic	topic	D			
cand_e2ro	direct	D	0.00/s	0.00/s	

Add a new exchange

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub

Changelog



The screenshot shows the RabbitMQ web interface at localhost:15672. The interface displays the status of the RabbitMQ server (3.8.14) and Erlang (23.2). The user is logged in as 'cand_e2ro'. The 'Queues' tab is selected, and the 'Get Message(s)' button is clicked. The interface shows four messages in the queue, each with its own details including Exchange, Routing Key, Redelivered status, Properties, and Payload.

Message	Exchange	Routing Key	Redelivered	Properties	Payload
Message 1	cand_e2ro	9574384527297338174.260.10.1794.1024	0	delivery_mode: 2 content_type: application/json	{"Date": "2021-03-11 22:25:25", "Value": 1009}
Message 2	cand_e2ro	9574384531104650557.260.10.1794.0	0	delivery_mode: 2 content_type: application/json	{"Date": "2021-03-11 22:25:26", "Value": 279140}
Message 3	cand_e2ro	9574384527949597969.260.11.1794.0	0	delivery_mode: 2 content_type: application/json	{"Date": "2021-03-11 22:25:26", "Value": 151806}
Message 4					

In the **Get Message(s)** we can see the messages that were uploaded, we can see routing key and the properties of the Payload.

Now that the **producer.php** runs successfully it's time to consume these data with the **consumer.php** file. Like we did before we go back to our project and click **Git Bash Here** and instead of typing **php producer.php** now we will type **php consumer.php**

```
MINGW64:/c:/xampp/htdocs/net2grid
user@DESKTOP-V66DCT0 MINGW64 /c:/xampp/htdocs/net2grid
$ php producer.php
[*] The message will now start being published
[x] 1 messages are published.
[x] 2 messages are published.
[x] 3 messages are published.
[x] 4 messages are published.
[x] 5 messages are published.
[x] 6 messages are published.
[x] 7 messages are published.
[x] 8 messages are published.
[x] 9 messages are published.
[x] 10 messages are published.
[*] All messages are published to the queue

user@DESKTOP-V66DCT0 MINGW64 /c:/xampp/htdocs/net2grid
$ php consumer.php
```

```
MINGW64:/c:/xampp/htdocs/net2grid
[x] Connected successfully
[x] Table created successfully [x] Trying to connect to the database
[x] Connected successfully
[x] New record created successfully
[x] Done
[*] Waiting for more messages...
[x] Received {"Date":"2021-03-11 22:25:27","Value":237987}
[x] Trying to connect to the database
[x] Connected successfully
[x] Table created successfully [x] Trying to connect to the database
[x] Connected successfully
[x] New record created successfully
[x] Done
[*] Waiting for more messages...
[x] Received {"Date":"2021-03-11 22:25:27","Value":80180}
[x] Trying to connect to the database
[x] Connected successfully
[x] Table created successfully [x] Trying to connect to the database
[x] Connected successfully
[x] New record created successfully
[x] Done
[*] Waiting for more messages...
[x] Received {"Date":"2021-03-11 22:25:28","Value":224431}
[x] Trying to connect to the database
[x] Connected successfully
[x] Table created successfully [x] Trying to connect to the database
[x] Connected successfully
[x] New record created successfully
[x] Done
[*] Waiting for more messages...
[*] There are no more tasks in the queue.
[*] The consumer will automatically close.
user@DESKTOP-V66DCT0 MINGW64 /c:/xampp/htdocs/net2grid
$
```

We can see that the consumer stops by himself when there are not any other messages in the queue, which payloads are we consuming from RabbitMQ and inserting them to the database to the correct format.

6. PROBLEM SOLVING PROCESS

The process I followed for my implementation consists of the steps below

1. **Clarify the Implementation:** The requirement was to publish messages in RabbitMQ, consume those messages from the published queues and save them in a Database.
2. **Breakdown the Implementation:** The implementation consists of the following parts
 - a. Set up RabbitMQ.
 - b. Retrieve my data from the source.
 - c. Understand the type of data that I am using.
 - d. Create the necessary data structure to save my data.
 - e. Create the necessary implementation to publish my data in the Queue. ([producer](#))
 - f. Create the necessary implementation to consume my data from the Queue. ([consumer](#))
 - g. Make the necessary filtering of the messages I am consuming.
 - h. Save my data to my data structure.
3. **Develop/Setup the related parts:** Based on the above breakdown
 - a. I followed the related documentation on the technical parts (RabbitMQ, XAMPP, OOP)
 - b. I analyzed the business parts (data structure, formatting etc.)
 - c. I developed the code to meet the needs of my implementation by running a “try-error-retry” logic
 - d. For each step I performed manual functional testing to make sure my code was working as expected
4. **Complete Testing:** Once I had all my parts in place I performed end to end testing with inline messages that would track my connection – data - output etc. to make sure everything works as expected.
5. **Refactor:** I spend some time to re-evaluate the positioning of my functions, the structure of my code and the “architecture” of the solution and made the changes I felt were “logically” needed to achieve a “separation of concerns”.
6. **Complete Testing:** After my refactoring I performed again end to end testing to make sure nothing was broken from my changes.
7. **Beautify:** While I believe I tried to make the code self-explanatory with the naming of my methods, I added comments in order to make it easier to read and document.
8. **Complete Testing:** I performed one last end to end testing for verification
9. **Documentation:** I documented my implementation with this document.