Erstellen eines Shopware 6 Plug-Ins für den Import von Produkten und Kategorien über RabbitMQ

Art des Dokuments:	Projektdokumentation
Ausbildungsberuf:	Fachinformatiker Anwendungsentwicklung
Identnummer:	3244395
Prüflingsnummer:	(155) 21501
Autor:	Luca Wlcek

saws
software
engineering

Hauptstraße 29
83533 Edling
fon: 08071/510312
fax: 08071/510311
mail: info@saws.de
internet: www.saws.de

Edling, 2022.

Hauptstraße 29
83533 Edling
Tel: 08071/510312
Fax: 08071/510311
Internet: www.saws.de
E-Mail: info@saws.de

SAWS GmbH & Co. KG
Sitz der Gesellschaft: Edling
Amtsgericht Traunstein, HRA 8211
UST-ID: DE230638335

Persönlich haftende Gesellschafterin:
SAWS Verwaltungsgesellschaft
Sitz der Gesellschaft: Edling
Amtsgericht Traunstein, HRB 15276
Geschäftsführer: Hagen Schneider

- Inhaltsverzeichnis -

1. Abkürzungsverzeichnis

AGPLv3 ..... GNU Affero General Public License version 3
AMQP ..... *Advanced Message Queueing Protocol*
GUI ..... *Graphical User Interface*
HTTP ..... *HyperText Transfer Protocol*
JSON ..... *JavaScript Object Notation*
MQ ..... *Message Queue*
MVC-Pattern ..... Model View Controller Pattern
QM ..... *Quality Managment*
REST ..... *Representational State Transfer*

2. Einleitung

2.1. Projektbeschreibung

Unternehmen, welche ihre Produkte unter anderem im E-Commerce vertreiben, speichern zugehörige Marketingdaten gerne in einem „Product-Information-Managment-System“ (PIM). Dort können diese Daten zentral verwaltet werden, was die Datenpflege vereinfacht und beschleunigt.

Ein Vertreter in dieser Branche ist das Unternehmen Contentserv GmbH mit der gleichnamigen Software Contentserv. Dieses System ist als Webanwendung umgesetzt und zeichnet sich durch seine Modularität sowohl in der Organisation der Daten, als auch innerhalb der Programmstruktur aus.

Das Unternehmen SAWS GmbH & Co. KG entwickelt eine Sammlung an Modulen und Plug-Ins namens „SAWSConnector“, welche das automatisierte Ausleiten dieser Daten ermöglicht. Eines der meistgenutzten Zielsysteme ist dabei Shopware 6, eine moderne E-Commerce Plattform und Shopsystem.

Die Übertragung von Contentserv nach Shopware geschieht mittels einer HTTP-REST-Schnittstelle, wobei der Connector auf die Antwort von Shopware warten muss. Dies ist problematisch, da Shopware für den Import dieser Daten eine Menge Ressourcen und Zeit braucht. Dadurch dauern eigentlich schnelle Exporte oft deutlich länger.

Um dieses Problem zu umgehen soll eine Asynchrone Verbindung aufgebaut werden. Das soll auf Basis eines Message-Queue-Servers geschehen. Dort sollen Befehle für den Import abgelegt und anschließend die Rückmeldungen von Shopware abgeholt werden. Ein solches System ist bereits in mehreren SAWSConnector-Modulen für andere Zielsysteme implementiert. Dort wird ein Message-Queue-Server namens RabbitMQ verwendet.

Werden die Befehle zwischengespeichert, kann der Connector den Export mit voller Geschwindigkeit durchführen anstatt auf ein Ergebnis zu warten. Das verringert die Systemauslastung von Contentserv und verringert das Zeitfenster für Systemausfälle mit Datenverlusten während des Exports.

Auf Seiten Shopwares kann die Auslastung besser gesteuert werden, indem die Import-Geschwindigkeit angepasst wird. Auch hier wird die Sicherheit der Daten zusätzlich erhöht. Fällt das Shopware-System ausfällt, bleiben die Daten dennoch auf dem MQ-Server vorhanden.

Das Projekt umfasst 3 zentrale Punkte. Diese sind im Folgenden erklärt. Die Reihenfolge spielt zu diesem Zeitpunkt noch keine Rolle.

Der erste Punkt ist die Modifikation des SAWSConnectors. Dort muss der bereits vorhandene Code des sogenannten „Transmitters“ angepasst werden. Transmitter sind die Komponenten des SAWSConnectors, welche die Verbindung zu einem Zielsystem, in diesem Fall Shopware, herstellen und die Rohdaten in das benötigte Format (z.B.: JSON) bringen.

Des Weiteren soll eine entsprechende Technik und Software für die Übertragung gewählt werden. Zwar ist bereits eine Verbindung zum Shopsystem „Magento“ über RabbitMQ und ein Protokoll namens AMQP möglich, allerdings gilt es diese Wahl zu überprüfen und Alternativen auszuloten.

Der letzte Teil des Projektes ist die Programmierung der Verbindung von der Message-Queue zu Shopware selbst. Genauere Details werden im Laufe dieser Dokumentation erklärt.

Ein besonderes Augenmerk gilt bei diesem Projekt der Praktikabilität. Das Umrüsten einer bereits genutzten Verbindung soll schnell und einfach gehen. Gleichzeitig muss die neue Verbindung zuverlässiger sein und darf keine Fehler in der Übertragung erzeugen.

2.2. Projektumfeld

2.2.1. Contentserv

Wie bereits erwähnt ist Contentserv im Kern ein PIM-System, es speichert also Produktdaten für Marketing und Verkauf. Allerdings bietet Contentserv eine enorm große Menge an Plug-In Schnittstellen und auch Erweiterungen. Besonders ist dabei, wie sehr sich die Software mittels Plug-Ins erweitern lässt.

Zusätzlich zur Kern-Funktionalität besitzt die Software viele weitere Fähigkeiten, wie das Speichern und Verwalten von Mediendaten, das Automatische Erstellen von HTML-Elementen oder PDFs, sowie viele Funktionen für das Automatisieren und Strukturieren der Datenpflege.

2.2.2. SAWS

„SAWS GmbH & Co. KG“ ist ein in Edling bei Wasserburg ansässiges Unternehmen. Es ist spezialisiert auf E-Commerce Lösungen und die Verbindung von Systemen, wobei die spezifischen Anforderungen der Kunden im Vordergrund stehen.

Einen großen Teil nimmt hierbei die Entwicklung von auf den Kunden zugeschnittenen Modulen für die Contentserv PIM-Plattform ein. Viele dieser Projekte schließen eine Gruppe selbst entwickelter Erweiterungen für Contentserv namens „SAWSConnector“ ein. Der „SAWSConnector“ ermöglicht die Ausleitung von Produktdaten an andere Systeme. Dabei stehen sowohl spezialisierte Ziele, wie etwa die Webshops „Magento“ und „Shopware“, als auch allgemeinere Datenformate wie JSON, XML, CSV und XLS zur Verfügung.

2.2.3. Shopware 6

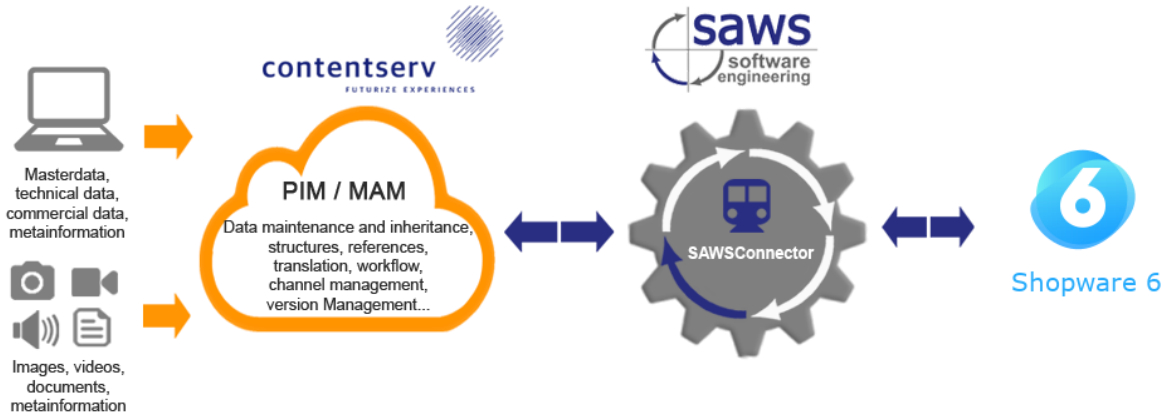
Shopware ist ein Online Shopsystem. Mithilfe von Shopware lassen sich mit geringem Aufwand Online-Shops betreiben. Das System bietet eine ansprechende Benutzeroberfläche und umfassende Einstellungsmöglichkeiten.

Shopware unterstützt das Speichern und Anzeigen von umfangreichen Produktdaten und Bildern, sowie detaillierte Statistiken zu Verkäufen. Die Erweiterbarkeit wird unter anderem durch Plug-Ins gewährleistet.

Shopware veröffentlicht eine Community-Edition welche unter der AGPLv3 lizensiert ist. Die AGPL ist eine Open-Source und so genannte Copy-Left-Lizenz. Eigene Veränderungen am Quellcode von Shopware müssen ebenfalls unter dieser Lizenz veröffentlicht werden. Plug-ins sind von dieser Regelung ausgenommen.

3. Analysephase

3.1. Ist-Analyse



Der SAWSConnector exportiert Produkte indem er die Daten direkt an die Shopware-API via HTTP-Requests schickt.

Dort werden alle geschickten Daten nacheinander importiert. Ist der Import abgeschlossen, antwortet Shopware in einer HTTP-Response. Dort zu finden sind Informationen zu den importierten Datensätzen, wie etwa eine Fehlermeldung oder der Bestätigung des Imports.

Der SAWSConnector wird seit Jahren optimiert um große Mengen an Daten möglichst schnell zu verarbeiten und zu senden. Eine einzelne Anfrage an Shopware kann mehrere hundert Produkte umfassen.

Die Webserver beider Systeme besitzen nur eine begrenzte Anzahl an Prozessen. Sind alle in Prozesse in Benutzung, werden weitere Anfragen nicht verarbeitet bis wieder freie Prozesse zur Verfügung stehen.

Shopware benötigt für den Import von Daten oft eine signifikante Zeitspanne. Während dieser Zeit ist der importierende Prozess beschäftigt und kann keine weiteren Anfragen von z.B. dem Front-End des Shops entgegennehmen.

Der cURL-Aufruf des Connectors blockiert währenddessen den exportierenden Prozess. Dieser kann während dieser Zeit weder das nächste Paket mit Daten vorbereiten, noch ist der Prozess für Web-Anfragen an das Interface von Contentserv verfügbar.

Ergebnis dieses Vorganges ist eine zuweilen problematische Beeinträchtigung beider Systeme.

Contentserv:

- Contentserv verliert an Reaktionsgeschwindigkeit aufgrund weniger freier Prozesse.
- Aufgrund der längeren Laufzeit ist ein Systemausfall während des Exports wahrscheinlicher. In diesem Fall werden nicht alle Daten übertragen.

Shopware:

- Shopware wird oft bis zum Limit belastet und Seiten bauen sich nur langsam oder gar nicht auf. Ein schlecht funktionierender Online-Shop führt oft zu geringerem Umsatz.
- Bei einem Ausfall von Shopware bricht der Connector irgendwann ab und die zu exportierenden Daten müssen neu geschickt werden.

3.2. Technische Möglichkeiten der Umsetzung

3.2.1. Erklärung von Message-Queue-Systemen

Ein Message-Queue-System (zu Deutsch: Nachrichtenwarteschlange) ist eine Technik für die Asynchrone Kommunikation zwischen Programmen oder Programmteilen.

Dabei werden, dem Namen entsprechend, Warteschlangen genutzt, welche die gesendeten Daten zwischenspeichern. Das Zielsystem holt sich dabei die Nachrichten selbstständig von der entsprechenden Warteschlange ab. Auf diese Weise können alle verbundenen Systeme mit ihrer eigenen Geschwindigkeit arbeiten ohne auf einander warten zu müssen.

Speziell für diesen Zweck gibt es diverse Server und Netzwerkprotokolle. Ein sich bei SAWS im Einsatz befindendes System ist beispielsweise „RabbitMQ“. Dieser Server unterstützt das Senden und Empfangen von Nachrichten mittels dem Protokoll „AMQP“.

Sender von Nachrichten werden „Producer“ (zu Deutsch: Hersteller) genannt.

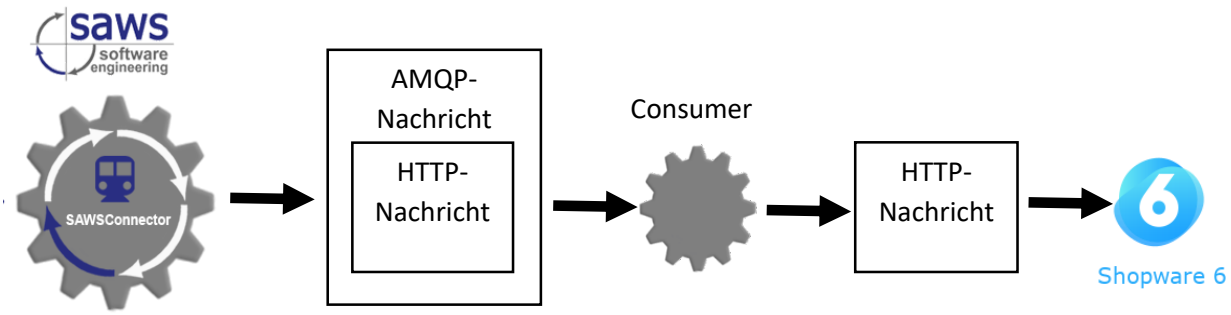
Empfänger von Nachrichten werden „Consumer“ (zu Deutsch: Verbraucher) genannt.

3.2.2. Grundlegendes Konzept

Da der primäre nutzen die Entlastung von Contentserv sein soll, muss die Kommunikation mit Shopware ausgelagert werden. Dafür müssen die API-Anfragen in einem unabhängigen System zwischengelagert werden.

Anschließend müssen diese Anfragen zu Shopware gelangen. Die Antworten von Shopware müssen auf dem umgekehrten Weg wieder zurück zu Contentserv. Da der Connector bereits eine Funktionalität für das senden via AMQP mithilfe der Bibliothek PHP-amqplib bietet, liegt es nahe dieses Protokoll zusammen mit einem MQ-Server zu nutzen.

Der SAWSConnector kann bereits mit der HTTP-Schnittstelle von Shopware kommunizieren. Dieser bereits erprobte Code soll weiterhin verwendet werden. Deshalb sollen diese HTTP-Nachrichten gekapselt und über AMQP geschickt werden. Am Zielort werden Sie vom Consumer wieder ausgepackt und an Shopware übergeben.



3.2.3. Wahl des Consumers (Empfängers)

Für den Consumer (Empfänger) wurden Ursprünglich 2 Möglichkeiten in Betracht gezogen.

Ein externes Relay

Ein eigenständiges Programm, welches unabhängig in Betrieb genommen werden kann. Es lädt einfach die Nachrichten des MQ-Servers und führt entsprechende HTTP-Anfragen aus.

Ein separates Relay hätte den Vorteil, dass es mit jeder HTTP-API kompatibel ist, da es lediglich die API-Requests aus der MQ lädt und abschickt. Nachteilig sind jedoch der zusätzliche Betriebs- und Wartungsaufwand.

Code auf Shopware Seite

Eine Shopware Erweiterung wäre nicht für andere Systeme nutzbar und müsste zusätzlich mit Shopware aktualisiert werden. Vorteil dieser Implementierung ist jedoch, dass kein Zusätzliches Programm benötigt wird und das Modul lediglich in Shopware eingebunden werden muss.

Fazit

Bei einer Implementierung auf Shopware-Seite ist die benötigte Infrastruktur bereits vorhanden, dadurch sind die Einrichtung und Administration ressourcenschonender. Die Möglichkeit der Asynchronen Übertragung soll sich vor allem an bisherige Kunden wenden, die ihren vorhandenen Export beschleunigen wollen. Eine Umstellung ohne zusätzlich benötigte Ressourcen wird daher bevorzugt.

3.2.4. Wahl der Shopware Schnittstelle

Shopware bietet für das Einbinden eigener Logik zwei Systeme an. Es galt zu entscheiden, welches dieser Systeme passender ist.

Shopware Plug-In

Plug-Ins nutzen klassische PHP-Dateien. Sie können interne Funktionen verwenden und beliebiges Verhalten implementieren. Mittels der von Shopware verwendeten Dependency Injection kann das System fast beliebig stark angepasst werden.

Shopware App

Shopware Apps setzen auf ein zweiseitige REST-API und eine Template-Engine namens „Twig“. Da die Logik auf Shopware-Seite implementiert werden soll und Shopware mit „Twig“ nicht die entsprechende Funktionalität bietet, fällt diese Möglichkeit weg.

Fazit

Auf Basis des Ausschluss-Prinzips wird ein Shopware-Plugin verwendet.

3.2.5. Wahl des MQ-Servers

Folgende Lösungen wurden in Betracht gezogen:

Apache ActiveMQ

Dieser MQ-Server ist open-source und bietet das AMQP Protokoll an. Allerdings ist die verwendete Protokollversion inkompatibel mit php-amqplib.

Apache Kafka

Kafka wurde für Event Streaming entwickelt und bietet enorm hohe Bandbreite und Performance. Daten können persistent gespeichert und bei Bedarf wieder abgerufen werden.

RabbitMQ

RabbitMQ wird im Betrieb bereits für den Export an andere Zielsysteme verwendet. Für dieses System ist also bereits Expertise vorhanden. Shopware selbst unterstützt bereits ein Consumer-Modell welches eine RabbitMQ nutzen kann.

RabbitMQ hat ein integriertes Webinterface für die Administration, was den Installations- und Wartungsaufwand reduziert.

Fazit

Dank bereits vorhandener Kompatibilität, Erfahrung und dem potentiellen Nutzen für eine Shopware-Installation wird RabbitMQ verwendet.

3.3. Wirtschaftlichkeitsanalyse

3.3.1. Kostenplanung

Die Kosten des Projekts wurden anhand der Arbeitsstunden und benötigten Hardware berechnet.

Vorgang	Mitarbeiter	Zeit	Personal	Ressourcen	Gesamt
Entwicklung	Auszubildender	64h	64h * 7 EUR/h = 448 EUR	64h * 10 EUR/h = 640 EUR	1088 EUR
Code-Review	Ausbilder	3h	3h * 40 EUR/h = 120 EUR	3h * 10 EUR/h = 30 EUR	150 EUR
Abnahme	Projektleiter	2h	2h * 40 EUR/h = 80 EUR	2h * 10 EUR/h = 20 EUR	100 EUR
					1338 EUR

3.3.2. Amortisationsrechnung

Eine direkte Amortisationsrechnung ist in diesem Fall nicht möglich, da das Shopware-Plug-In weder separat verkauft wird noch auf Seiten von SAWS signifikante Ersparnisse bringt.

Der Nutzen des Projekts liegt vor allem in der erhöhten Kundenzufriedenheit durch schnellere und zuverlässigere Exporte. Zufriedene Kunden wechseln seltener Anbieter und sorgen sowohl durch Lizenzkäufe als auch durch Programmier- und Supportaufträge für Einnahmen. SAWS bietet außerdem auch RabbitMQ-Systeme an, welche für die Nutzung der beschriebenen Verbindung gebucht werden können.

Es wäre möglich, das Plug-In im Shopware-Plug-In-Shop anzubieten um auf den SAWSConnector aufmerksam zu machen und so Verkäufe zu generieren. Die ist nicht mehr Teil des Projektes.

3.3.3. Make-or-Buy Entscheidung

Shopware 6 bietet bereits eine API an, über welche der SAWSConnector importiert, jedoch erlaubt diese keine Anbindung an einen Message Queue Server. Trotz eines großen Kataloges an Erweiterungen für Shopware existiert zum Stand März 2022 keine entsprechende Erweiterung welche das Anbinden der externen Shopware-API an eine Message Queue erlaubt.

Aus diesem Grund wurde sich dafür entschieden selbst eine solche Lösung selbst zu entwickeln.

4. Entwurfsphase

4.1. Entwurf des Shopware-Plugins

Die folgenden Erläuterungen beziehen sich auf das Klassendiagram im Anhang.

Das Shopware-Plug-In soll aus 3 logischen Teilen bestehen.

- Die nötige Logik um Shopware per GUI für eine Verbindung zu RabbitMQ zu konfigurieren.
- Das Empfangen und Verarbeiten von Anfragen über AMQP sowie das Zurücksenden der Antwort.
- Das Frontend für die Eingabe der nötigen Daten der Konfiguration.

Shopware basiert auf Symfony, einem PHP-Framework für Webapplikationen. Symfony ist auf den Einsatz in großen Softwareprojekten zugeschnitten und bietet viele Module, die den Umfang erweitern und den Entwicklungsaufwand reduzieren.

Symfony nutzt das so genannte MVC-Pattern. Dabei wird der Programmcode in Datenmodell, Benutzeroberfläche und Verarbeitungslogik getrennt.

Symfony bietet unter anderem eine auf Routen-Controllern basierende Verarbeitung von HTTP-Anfragen.

Das Front-End kann wahlweise mithilfe einer Template-Engine namens „Twig“, dem Javascript-Gui-Framework „Vue.js“ oder einer Kombination aus beidem erstellt werden.

Symfony baut stark auf ein Pattern namens „Dependency-Injection“ (kurz: DI) auf. Dabei werden Klassen nicht direkt benutzt. Stattdessen fordert ein Programmteil seine Abhängigkeiten als Interfaces und empfängt diese in Objektform.

Durch diesen Ansatz können einzelne Klassen ohne Probleme ausgetauscht werden solange die neue Klasse das benötigte Interface implementiert.

4.1.1. Verbindungssteuerung

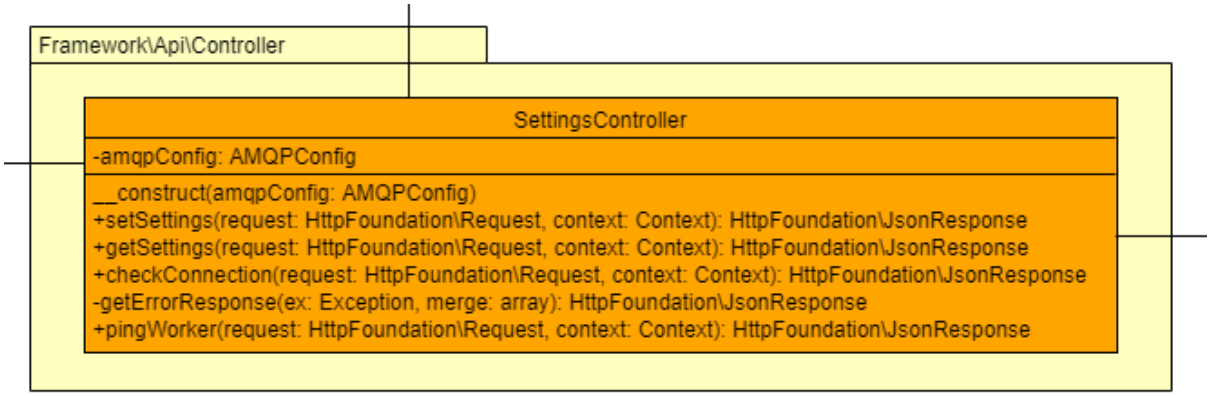
Die Planung wurde mit dem Verbindungsaufbau begonnen.

Shopware kann bereits intern einen MQ-Server nutzen um zu erledigende Aufgaben in eine Warteschlange einzutragen und sie später abzuarbeiten. Es galt eine solche Verbindung testweise aufzubauen.

Hierzu nutzt Shopware Konfigurationsdateien. Im ersten Schritt wurde diese Datei also manuell angepasst und anschließend getestet. Nachdem die Verbindung wie gewollt aufgebaut wurde, begann die Planung für das automatisierte Erstellen einer solchen Datei.

Das Backend dieser Logik ist in 5 Klassen unterteilt.

SettingsController



Der *SettingsController* implementiert eine HTTP-API. Er nimmt Anfragen gegenüber drei Endpunkten in der Shopware Admin-Api entgegen. Die Anfragen werden mittels speziell formatierten Kommentaren an die richtige Methode geleitet.

„/api/saws/settings/set“ (setSettings) erlaubt das Einspielen von Einstellungen. Erwartet wird eine JSON mit den Verbindungsdaten.

„/api/saws/settings/get“ (getSettings) ermöglicht das Auslesen der gespeicherten Verbindungsdaten. Es wird das Selbe JSON zurückgeliefert, mit dem Unterschied, dass nur ein Boolean anstatt des Passworts gesendet wird. Damit ist es zwar möglich festzustellen ob das Passwort fehlt, allerdings nicht dieses auszulesen.

„/api/saws/settings/ping_worker“ (pingWorker) erstellt eine *WorkerPingMessage* und sendet diese in die Ping-Queue. Der momentane Pong-Wert kann im *amqp_check* empfangen werden.

„/api/saws/settings/amqp_check“ (checkConnection) prüft die gespeicherten Daten und versucht eine Verbindung zum MQ-Server herzustellen. Zurückgegeben werden ein numerischer Verbindungsstatus, ein Array fehlender Einstellungen, die Letzte Antwort des Workers auf den Ping sowie ein Fehlercode, falls ein Fehler auftritt.

AMQPConfig

Die AMQP-Verbindung wird von der Klasse *AMQPConfig* verwaltet. Auf der folgenden Darstellung wurden der Verständlichkeit halber die Getter und Setter für die meisten Attribute weggelassen.

Die 7 öffentlichen Methoden zur Steuerung und Überwachung werden im Folgenden erklärt:

applyConfig()
Diese Methode erstellt eine Konfigurationsdatei namens framework.yaml und speichert diese im lokalen Plug-In Verzeichnis. Das Plug-In ist darauf programmiert diese Datei einzulesen, wenn sie gefunden wird. Falls die Datei aufgrund fehlender Einstellungen einen Syntax-Fehler erzeugt, wird die Konfigurationsdatei stattdessen geleert. Die eingetragenen Daten gehen dabei nicht verloren, da sie in der *Settings Klasse hinterlegt sind.*

connect()
Diese Methode veranlasst einen Verbindungsaufbau zum AMQP-Server und konfiguriert dort alle nötigen Ressourcen, wie etwa die benötigten Queues (Warteschlangen). Diese Methode wirft diverse Exceptions, welche der Fehleranalyse dienen.

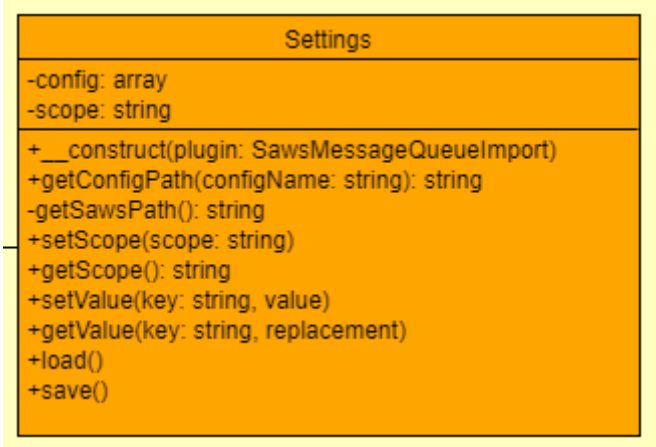
countRequestMessages()
countResponseMessages()
Diese beiden Methoden Verbinden zum AMQP-Server und prüfen die Anzahl der jeweilig wartenden Pakete.

isDeployed()
Diese Methode prüft ob eine aktive Framework-Konfiguration gespeichert ist.

getPong()
setPong()
Diese Methoden setzen und Laden den Wert des letzten empfangenen Pings.

Settings

Zur Speicherung der momentanen Einstellungen wird die Klasse *Settings* verwendet. Für einen besseren Schutz des Passworts werden die Einstellungen YAML-Enkodiert im Dateisystem abgelegt anstatt sie in der Datenbank zu speichern.



Folgende Methoden stehen zur Verfügung:

getConfigPath()
Erstellt den passenden Dateipfad um auf die gegebene Konfigurationsdatei zugreifen zu Können.

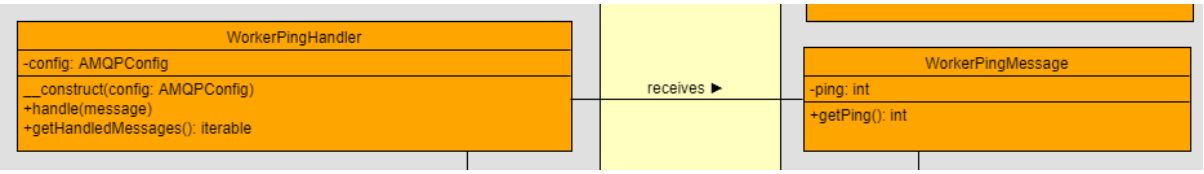
setScope()
getScope()
Mithilfe dieser beiden Methoden können eigenständige Einstellungsbereiche aufgerufen werden. Im Fall der momentanen Implementierung wird lediglich „amqp“ verwendet. Diese Funktionen dienen also der zukünftigen Erweiterbarkeit.

setValue()
Diese Funktion setzt einen Einstellungswert, speichert jedoch noch nicht.

getValue()
Mit dieser Methode kann ein Einstellungswert ausgelesen werden. Falls die Einstellung noch nicht gesetzt wurde, kann ein Standardwert mitgegeben werden.

load()
save()
Erlaubt das Speichern und erneute Laden der Einstellungen.

WorkerPingMessage & WorkerPingHandler



Diese beiden Klassen dienen der Umsetzung einer Worker-Überwachung. Sie nutzen die Messenger-Komponente des Symfony-Frameworks.

Der Worker wird als eigenes Programm gestartet. Als Argumente bekommt er dabei die Namen der Shopware-Queues die er überwachen soll. Empfängt er ein Objekt liefert er dieses an alle für die Klasse eingetragenen Handler.

Dadurch kann jeder Teil der AMQP-Verbindung tatsächlich einmal getestet werden. Zusätzlich macht die Anzeige in der GUI auf die Existenz und Notwendigkeit des Workers aufmerksam, da der Worker der einzige Teil der Verbindung ist, der mehr als nur einen Webzugang zum System erfordert und von Kunden potentiell übersehen wird.

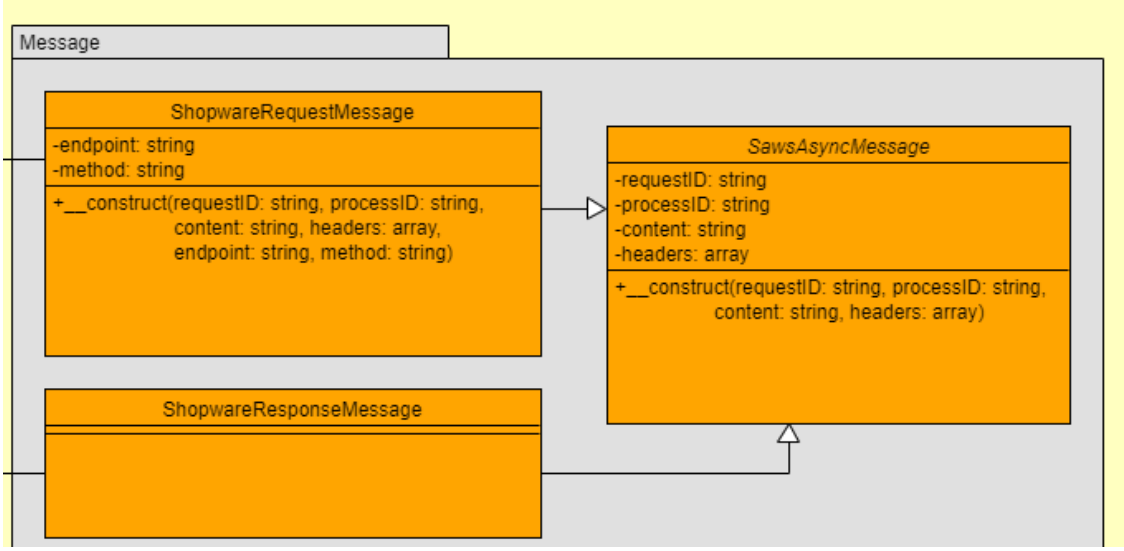
Die Message ist in diesem Fall das Datenmodell, welches verschickt wird. Läuft ein Worker, so holt dieser die Message aus der Queue und ruft den zugehörigen Handler auf, welcher die Message dann abarbeitet. Das hier gezeigte Ping-System nutzt einen Zeitstempel um festzustellen, wann der letzte Ping verarbeitet wurde.

Die Verwendung des Pings ist jedoch dem entsprechenden Client, wie etwa dem Frontend überlassen.

4.1.2. Anfragenverarbeitung

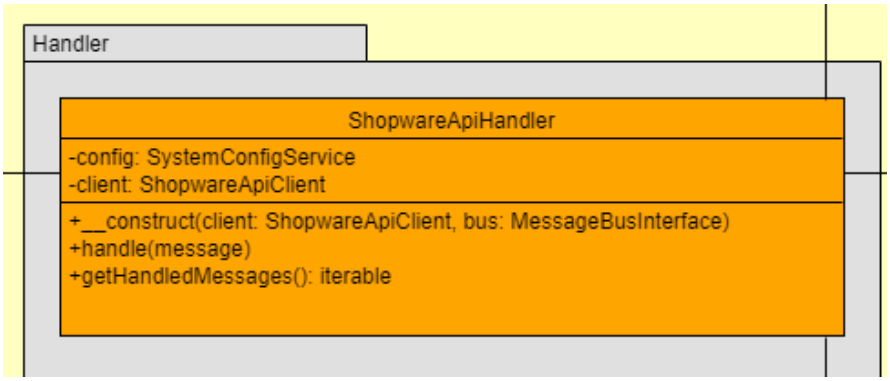
In der Analyse wurde ein Relay-System für die Übertragung gewählt. Das bedeutet, dass die HTTP-Anfragen über AMQP gekapselt und umgeleitet werden.

ShopwareRequestMessage & ShopwareResponseMessage



Diese Klassen entsprechen einer HTTP-Anfrage und HTTP-Antwort. Da sie viele Attribute teilen, erben beide von einer Klasse namens *SawsAsyncMessage*. Die Anfrage benötigt hierbei allerdings noch einen Endpunkt, also einen Pfad der Shopware Admin-API, und das verwendete HTTP-Verb (method). Zusätzlich mitgeschickt wird immer eine Prozess-ID für die Zuordnung zu einem Exportvorgang sowie eine Anfrage-ID um eine Antwort der entsprechenden Anfrage zuordnen zu können.

ShopwareApiHandler



Der *ShopwareApiHandler* nimmt Anfragen aus der AMQP-Queue entgegen und leitet diese an den *ShopwareApiClient* weiter. Anschließend sendet er die Antwort, welche der Client empfangen hat in die Antwort-Queue des AMQP-Servers. Die Logik dieser Klasse ist minimal um Dependency-Injection für den Client möglichst leicht zu machen.

ShopwareApiClient

Der *ShopwareApiClient* ermöglicht das Kommunizieren mit der Shopware-API mittels *ShopwareRequestMessages*. An dieser Stelle wäre normalerweise eine Implementierung über die cURL Bibliothek notwendig.

Glücklicherweise ist es möglich ein von Symfony genutztes HTTP-Request-Objekt zu erstellen und an den „*HTTPKernel*“ von Symfony zu übergeben. Die Anfrage wird verarbeitet, als wäre sie über den Webserver eingegangen und beachtet alle entsprechenden Routing-Regeln. Der Client kümmert sich also um die Umwandlung zwischen AMQP und HTTP Objekten.

Dies reduziert einerseits die benötigten Netzwerkressourcen, zum anderen ermöglicht Shopware so das Überspringen der Authentifizierung, indem an das HTTP-Request-Objekt Metadaten gehängt werden.

sendRequest()
Diese Methode verarbeitet eine *ShopwareRequestMessage* mithilfe der nativen Shopware-API.

convertRequest()
convertResponse()
Diese Methoden werden verwendet um zwischen den Eigenen und den Symfony Objekten zu

konvertieren. Für die Rückkonvertierung der Antwort wird aus der Anfrage die Prozess-ID und Anfrage-ID kopiert und in der *ShopwareResponseMessage* gesetzt.

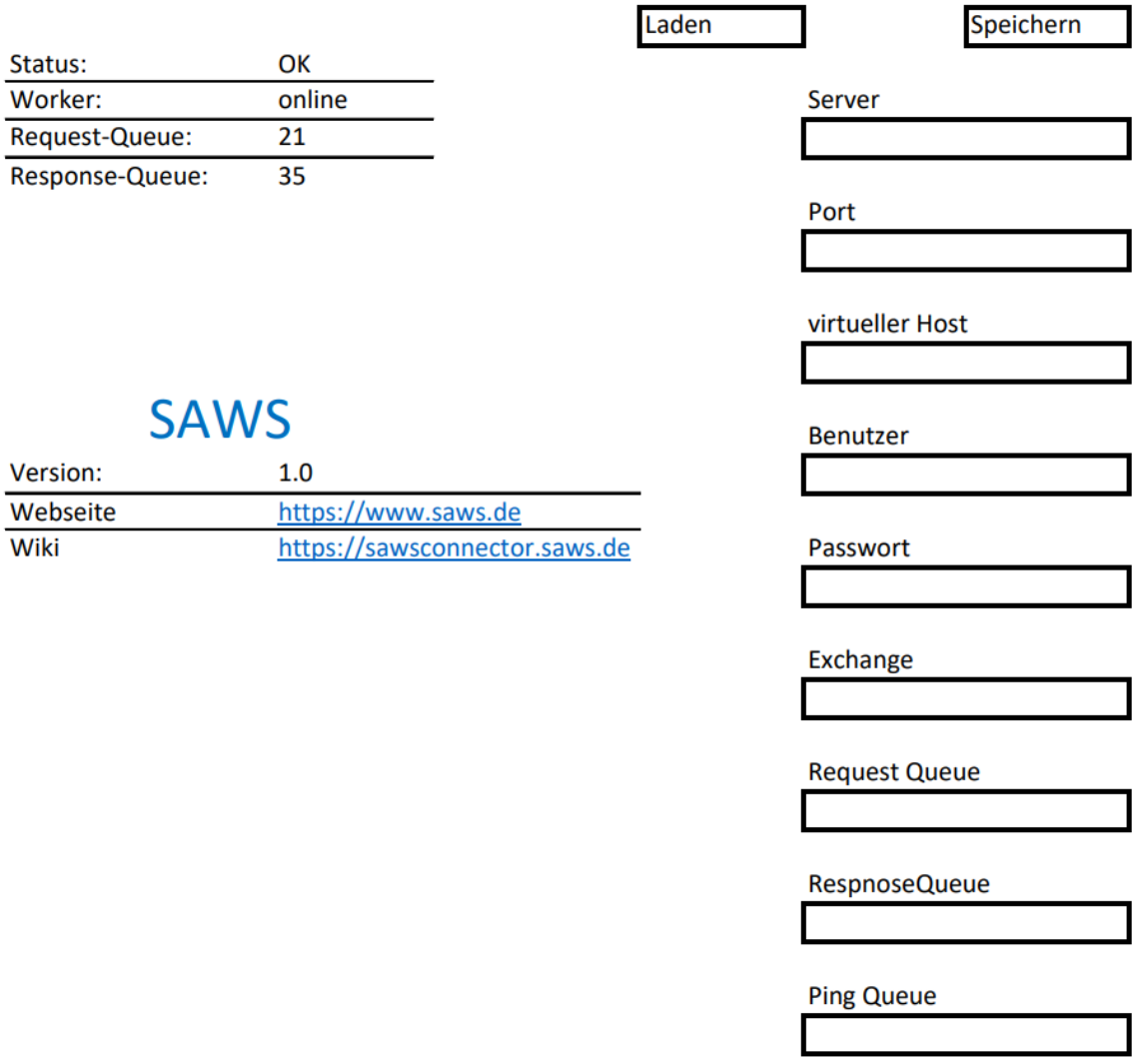
4.1.3. Administrations-Front-End

Für die nötige Konfiguration wurde nun ein Front-End benötigt, das dem Nutzer erlaubt alle nötigen Einstellungen zu treffen. Zusätzlich soll der momentane Zustand der Verbindung angezeigt werden.

Shopware nutzt „Vue.js“ für diese Oberfläche. Dementsprechend wurde ein Modul für das Shopware-Front-End geplant, welches in Zusammenarbeit mit dem PHP-Backend die komplette Anbindung an einen MQ-Server ermöglicht.

Um eine grobe Vorlage zu haben wurde diese Skizze angefertigt. Der Status auf der linken Seite soll sich in Echtzeit aktualisieren um das Diagnostizieren von Fehler zu vereinfachen.

Die Einstellungen können gespeichert aber auch auf den momentan gespeicherten Status zurückgesetzt werden.



Im Folgenden sind die geplanten Einzelkomponenten für das „Vue.js“ Interface aufgelistet:

saws-amqp-status

Diese Komponente prüft in einem Intervall von einigen Sekunden den momentanen Status der AMQP-Verbindung und des Workers. Dazu wird regelmäßig eine Anfrage mit dem momentanen Zeitstempel an den Endpunkt *„/api/saws/settings/ping_worker“* durchgeführt. Die letzte empfangene Antwort des Workers sowie der Verbindungstatus und gegebenenfalls ein Fehlercode werden von *„/api/saws/settings/amqp_check“* abgefragt und abgespeichert.

Die gesammelten Daten werden visuell aufbereitet und Fehlercodes in entsprechende Meldungen übersetzt.

Zusätzlichen werden sowohl fehlende Einstellungen als auch der Fehlercode als Vue-Events abgegeben. Dies ermöglicht dem Darüber liegenden Dialog die entsprechenden Felder zu markieren.

saws-branding

Diese Komponente liefert entsprechendes Branding mit dem Firmenlogo, der Version des Plug-Ins sowie Links zu Wiki und Website von SAWS.

saws-mq-settings

Dies ist die Hauptkomponente. Sie beinhaltet sowohl die anderen beiden, als auch die benötigten Eingabefelder. Sie kümmert sich außerdem um Standardwerte.

SawsSettings.service

Dieser Service übersetzt Funktionsaufrufe in die tatsächlichen Anfragen an Shopware und kümmert sich um die Authentifizierung. Er besitzt die folgenden Methoden:

save(settings)
load()
Speichert oder Lädt die momentanen Einstellungen mithilfe des REST-Service.

checkConnection()
Holt den Verbindungsstatus, fehlende Felder und potentielle Fehlercodes.

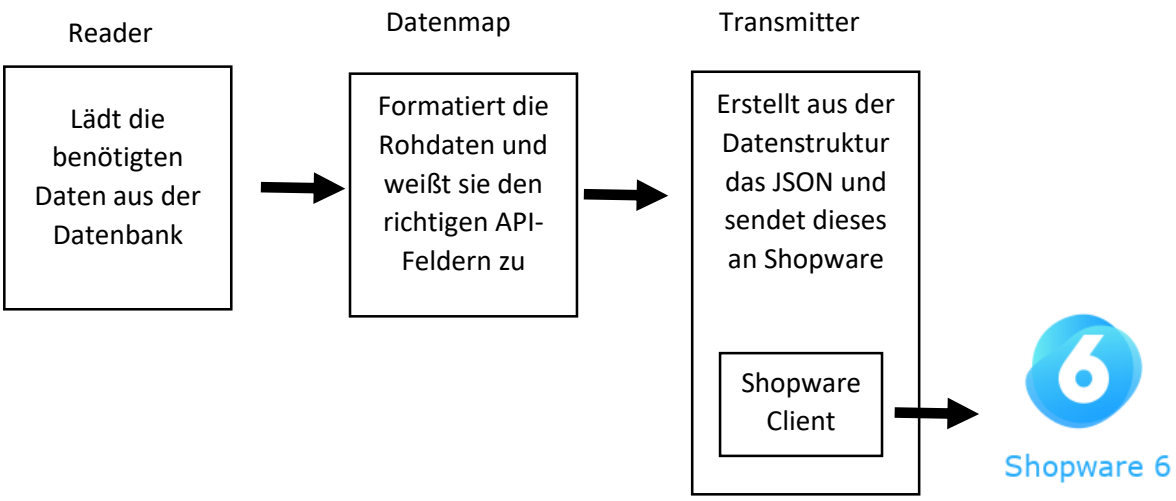
pingWorker(ping)
Ruft den ping_worker REST-Service auf und übergibt den angegebenen Zeitstempel.

getInfo()
Frägt die Infos des installierten Plug-Ins mithilfe der Shopware API ab. Darin enthalten sind unter anderem Name und Version.

4.2. Entwurf der connectorseitigen Übertragung

Zu guter Letzt gilt es, den Connector für die Verbindung zu erweitern. Die Hauptkomponente, welche verändert wird, ist dabei der so genannte Transmitter für Shopware 6.

Transmitter werden im Connector verwendet um die Rohdaten welche der Export ausgibt in das richtige Format zu bringen und an das Zielsystem zu schicken.



Client und AmqpClient

Der Shopware 6 Transmitter nutzt für die eigentliche Kommunikation eine Klasse namens Client im Namespace „sawsshopware\api\shopwarevix“. Es gilt eine Kind-Klasse zu erstellen, welche die entsprechenden Methoden überschreibt und diese über AMQP umleitet. Allerdings ist hier zu beachten, dass einige Anfragen während des Exports durchgeführt werden. Diese müssen weiterhin Synchron laufen.

Um Antworten abzuholen wird die Methode *basic_get()* eingeführt, analog zur darunterliegenden Methode der AMQP-Bibliothek.

Folgende Methoden werden überschrieben um die Anfragen über AMQP umzuleiten:

sync(…)

Diese Methode lädt Produkte oder Kategorien zu Shopware hoch. Der Rest-Service funktioniert dabei wie ein „upsert“. Er prüft ob ein Produkt vorhanden ist und führt entweder ein Update durch oder legt das Objekt an.

upload(…)

*D*iese Methode fordert Shopware auf die entsprechende Datei von Contentserv herunterzuladen und in die eigene Datenstruktur einzupflegen.

SAWSShopwareVIxDefaultTransmitterPlugin

Diese Klasse registriert den Transmitter im Connector und baut die GUI für die nötigen Einstellungen auf. Hier gilt es die neuen Felder für die Einstellungen hinzuzufügen.

ShopwareVIxDefaultTransmitter

Dies ist der eigentliche Transmitter, welcher die Daten für Shopware aufbereitet und dorthin sendet.

Hier müssen zwei Änderungen durchgeführt werden. Zum einen muss der Client ausgetauscht werden, wenn die AMQP-Verbindung eingeschaltet ist, zum anderen muss die Funktion

syncResponse(…)

aus dem Interface *AsyncTransmitterInterface* implementiert werden, um die Antworten abzuholen und in die entsprechenden Logs einzupflegen.

5. Implementierungsphase

5.1. Erstellen und Einbinden des Plugins

Plug-Ins in Shopware sind Symfony-Pakete. Der erste Schritt war daher eine Manifest-Datei zu erstellen. Diese Datei enthält Informationen für den Paketmanager Composer und heißt „composer.json“.

Diese Datei enthält Abhängigkeiten, Version, Name, Beschreibung und andere grundlegende Informationen des Plug-Ins.

```
{
  "name": "saws/messagequeueimport",
  "description": "",
  "version": "0.1.0a",
  "type": "shopware-platform-plugin",
  "license": "proprietary",
  "authors": [
    {
      "name": "SAWS GmbH & Co. KG"
    }
  ],
  "require": {
    "shopware/core": "6.4.*"
  },
  "extra": {
    "shopware-plugin-class": "Saws\\MessageQueueImport\\SawsMessageQueueImport",
    "label": {
      "de-DE": "Saws MessageQueue Import Plug-In",
```

Auszug aus der Datei composer.json

Zusätzlich zu diesen Metainformationen wird eine Hauptklasse für das Plug-In benötigt. In dieser Hauptklasse muss nun auch die entsprechende Konfigurationsdatei eingebunden werden. Dieser Code ist unabhängig vom Erstellen der Datei. Er bindet sie nur ein, falls sie gefunden wird.

5.2. Erstellen und Anwenden der nötigen Konfigurationsdateien

5.2.1. Speichern der Einstellungen

Im ersten Schritt wurde die *Settings* Klasse implementiert. Dabei wurden assoziative Arrays aus PHP verwendet. Die Klasse erlaubt das nutzen eigener Abschnitte um Konfigurationen voneinander zu trennen.

Alle Methoden, die keinen eigenen Rückgabewert haben, geben das momentane Objekt zurück um Method-Chaining zu ermöglichen.

5.2.2. Anwenden der Einstellung mithilfe der framework.yaml

Darauf aufbauend kam die vermutlich komplexeste Klasse dieses Abschnitts dran: *AMQPConfig*. Zentral ist die Methode *buildConfig()*. Sie erstellt das Innenleben der Konfigurationsdatei und gibt diese als String zurück.

```
$arrConfig = [
    'framework' => [
        'messenger' => [
            'transports' => [
                'saws_request_queue' => [
                    'dsn' => $this->buildAmqpUrl($this->getRequest_queue()),
                    'serializer' => 'messenger.transport.symfony_serializer',
                    'options' => $this->buildOptions($this->getRequest_queue())
                ],
                'saws_response_queue' => [
                    'dsn' => $this->buildAmqpUrl($this->getResponse_queue()),
                    'serializer' => 'messenger.transport.symfony_serializer',
                    'options' => $this->buildOptions($this->getResponse_queue())
                ],
                'saws_ping_queue' => [
                    'dsn' => $this->buildAmqpUrl($this->getPing_queue()),
                    'serializer' => 'messenger.transport.symfony_serializer',
                    'options' => $this->buildOptions($this->getPing_queue())
                ]
            ],
            'routing' => [
                ShopwareRequestMessage::class => 'saws_request_queue',
                ShopwareResponseMessage::class => 'saws_response_queue',
                WorkerPingMessage::class => 'saws_ping_queue'
            ]
        ]
    ]
];
return Yaml::dump($arrConfig, 10);
```

Die Konfiguration für die Queues ist bis auf die Shopware-Namen und die Namen der dahinterliegenden Message-Queues identisch. Dementsprechend werden die URL und Optionen lediglich mit der Namensänderung der Queue gebildet.

```
private function buildOptions(string $queueName): array {
  return [
      'heartbeat' => 0,
      'password' => $this->getPassword(),
      'exchange' => [
          'name' => $this->getExchange(),
          'type' => 'topic',
          'default_publish_routing_key' => $queueName
      ],
      'queues' => [
          $queueName => [
              'binding_keys' => [
                  $queueName
              ]
          ]
      ]
  ];
}
```

Die beiden Klassen wurden als Symfony Services registriert. Dadurch können sie später mithilfe der Dependency-Injection angefordert werden.

```
<service id="Saws\MessageQueueImport\Config\Settings"></service>
<service id="Saws\MessageQueueImport\Config\AMQPConfig"></service>
```

Ausschnitt aus services.xml

5.2.3. Status des Workers

Für die Überwachung des Workers werden jetzt sowohl ein Message-Objekt als auch ein zugehöriger Handler benötigt. Im *WorkerPingHandler* wird der Zeitstempel aus der *WorkerPingMessage* geladen (*getPing*) und mithilfe der *AMQPConfig (setPong)* gespeichert. Der Setter hat hierbei zusätzliche Logik um den Pong-Wert direkt zu speichern.

5.2.4. REST-Service

Für den Abschluss des Backends fehlen jetzt lediglich die entsprechenden HTTP-API-Endpunkte. Um diese zur Verfügung zu stellen musste der *SettingsController* implementiert und bei Symfony registriert werden.

```
<?xml version="1.0" encoding="UTF-8" ?>
<routes xmlns="http://symfony.com/schema/routing"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://symfony.com/schema/routing
        https://symfony.com/schema/routing/routing-1.0.xsd">

    <import resource="../../Framework/Api/Controller/**/*Controller.php" type="annotation" />
</routes>
```

Die Routen werden mittels speziell formatierten Kommentaren über den Funktionen gesteuert. Die Funktionen dienen lediglich als Verbindungsglied und enthalten nur wenig relevante Logik.

```
/**
 * Allows to change the settings.
 * Valid settings will automatically be applied.
 *
 * @Route("/api/saws/settings/set", name="api.saws.settings.set", methods={"POST"})
 *
 */
public function setSettings(Request $request, Context $context): JsonResponse {
```

5.2.5. Implementierung des Front-Ends

Nachdem das Backend implementiert ist, fehlt nun noch der Einstellungsdialog in der Shopware Administration. Im ersten Schritt muss dafür ein Frontend-Plug-In in Shopware registriert werden. Der Dialog kann hier über das Einstellungsmenü aufgerufen werden.

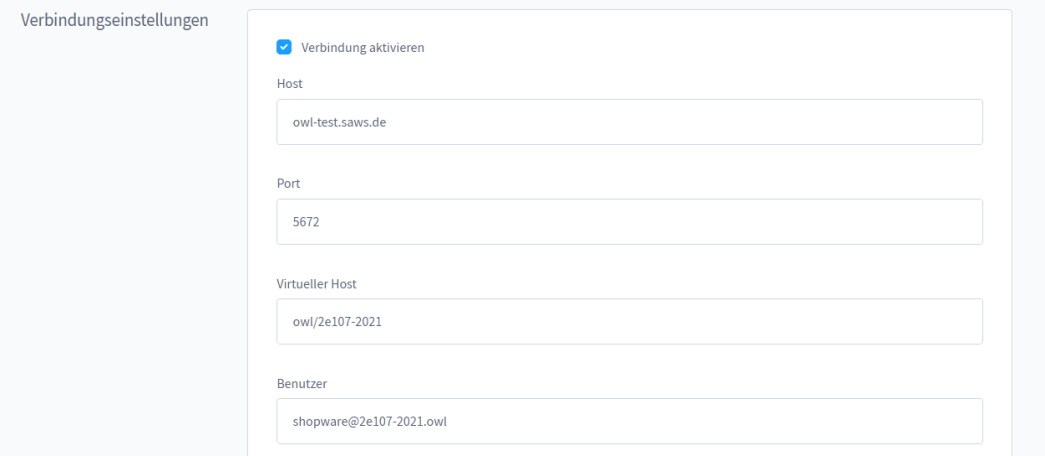
Hier wird die Vue-Komponente „saws-mq-settings“ an die Route /saws/mq/settings angebunden. Anschließend wird dieser Link zu den Systemeinstellungen von Shopware hinzugefügt. Shopware bietet bereits eine Menge Komponenten um in kurzer Zeit eine thematisch passende GUI zu erstellen.

```
Module.register('saws-mq', {
  type: 'plugin',
  name: 'Bundle',
  title: 'saws-mq.general.mainMenuItemGeneral',
  description: 'sw-property.general.descriptionTextModule',
  color: '#FFD700',
  icon: 'default-shopping-paper-bag-product',

  snippets: {
    'de-DE': deDE,
    'en-GB': enGB
  },

  routes: {
    settings: {
      component: 'saws-mq-settings',
      path: 'settings'
    }
  },

  settingsItem: [{
      group: 'system',
      to: 'saws.mq.settings',
      icon: 'default-object-rocket',
      name: 'saws-mq.settings.title'
    }]
});
```

Dank der vielen fertigen Vue-Komponenten von Shopware ist der GUI recht kurz.

```
<template #content>
  <sw-card-view>
    <sw-card :isLoading="isLoading" :title="$t('saws-mq.settings.connection')">
      <sw-checkbox-field :label="$t('saws-mq.settings.enabled')" v-model="enabled"></sw-checkbox-field>
      <sw-text-field :label="$t('saws-mq.settings.host')" v-model="host" :isInvalid="missing.host"></sw-text-field>
      <sw-number-field :label="$t('saws-mq.settings.port')" v-model="port" numberType="int" min="1" max="65535" :isInvalid="missing.port"></sw-number-field>
      <sw-text-field :label="$t('saws-mq.settings.vhost')" v-model="vhost" :isInvalid="missing.vhost"></sw-text-field>
      <sw-text-field :label="$t('saws-mq.settings.user')" v-model="user" :isInvalid="missing.user"></sw-text-field>
      <sw-password-field :label="$t('saws-mq.settings.password')" v-model="password" :passwordToggleAble="false" :isInvalid="missing.password"></sw-password-field>
      <sw-text-field :label="$t('saws-mq.settings.exchange')" v-model="exchange"  :isInvalid="missing.exchange"></sw-text-field>
      <sw-text-field :label="$t('saws-mq.settings.request_queue')" v-model="request_queue"  :isInvalid="missing.request_queue"></sw-text-field>
      <sw-text-field :label="$t('saws-mq.settings.response_queue')" v-model="response_queue" :isInvalid="missing.response_queue"></sw-text-field>
      <sw-text-field :label="$t('saws-mq.settings.ping_queue')" v-model="ping_queue" :isInvalid="missing.ping_queue"></sw-text-field>
    </sw-card>
  </sw-card-view>
</template>
```

Dieser relativ kurze Abschnitt führt zu folgender schönen und funktionalen Anzeige.

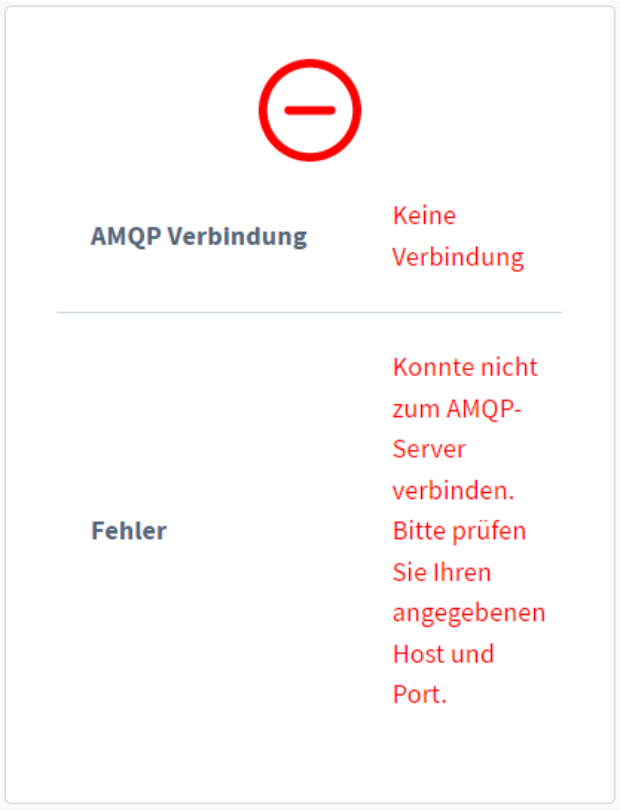


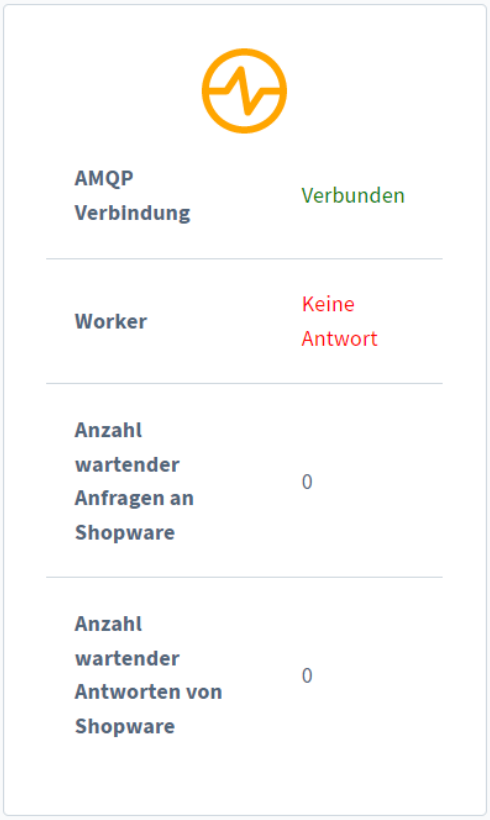
Allerdings fehlt natürlich noch entsprechende Logik um die Daten sowohl zu laden, als auch wieder zu speichern. Der Dialog selbst nutzt dafür ein Service-Objekt. Da ein Großteil der Logik bereits im Backend implementiert ist muss der Code lediglich die Daten verpacken und abschicken.

Die Statusanzeige benötigt im Gegensatz dazu ein wenig komplexere Logik. Hier sollen Status und Probleme auf einen Blick zu sehen sein.

Das Symbol ganz oben gibt mit einem Blick Aufschluss darüber, ob das Plug-In einsatzbereit ist.

Ein rotes Ausfall-Zeichen bedeutet, dass keine Verbindung aufgebaut werden konnte. Zusätzlich werden möglichst präzise Fehlermeldungen angezeigt. Außerdem wird an die Einstellungskomponente gemeldet, welche Einstellungen möglicherweise fehlerhaft sind.





Gelb bedeutet, dass die Verbindung steht aber kein Hintergrundprozess für das Verarbeiten der Nachrichten läuft.

Grün bedeutet, dass alles Einsatzbereit ist.

Wird die Verbindung deaktiviert, ändert sich das Symbol ebenfalls und wird grau eingefärbt.

Bei einer funktionierenden Verbindung wird zusätzlich der Status des Workers und die Anzahl wartender Pakete angezeigt.

Dank „Vue.js“ und fertigen Shopware-Komponenten wird auch hier nur wenig HTML- bzw. Template-Code benötigt.

```
<sw-card class="saws--status" :isLoading="isLoading">
  <div v-if="!isLoading" class="connectionStatus">
    <sw-icon size="60" :name="icon" :color="color"></sw-icon>
    <sw-grid :table="true" :selectable="false" :header="false" :items="infoTable">
      <template #columns="{ item }">

        <sw-grid-column flex="minmax(100px, 1fr)">
          <strong>{{ item.name }}</strong>
        </sw-grid-column>

        <sw-grid-column flex="minmax(120px, 0fr)">
          <span :class="item.class">{{ item.value }}</span>
        </sw-grid-column>

      </template>
    </sw-grid>
  </div>
</sw-card>
```

Das eintragen der passenden Daten erledigt hierbei das JavaScript.

Die Komponente zeigt immer nur die gerade verfügbaren oder relevanten Daten an.

Kann keine Verbindung hergestellt werden, so sind lediglich der Status der Verbindung selbst, sowie der gemeldete Fehler relevant.

Zusätzlich gibt die Statusanzeige Vue-Events nach außen ab um den Fehlercode zu melden. Dadurch können die vermutlich fehlerhaften Einstellungen rot markiert werden.

5.3. Implementierung des Handlers und der Message Objekte

Für das Kapseln der Informationen benötigt Shopware einige Klassen.

Als Basis dient hierbei die abstrakte Klasse *SawsAsyncMessage*. Sie beinhaltet alle grundlegenden Informationen, welche in einer HTTP-Anfrage oder Antwort enthalten sind.

Zusätzlich ist es noch nötig diverse Schlüssel mitzugeben, um die Antwort passend verarbeiten zu können.

Zum einen wird die ID des individuellen Export-Vorgangs benötig. Diese ermöglicht es dem Connector zuzuordnen, in welchem Log er die empfangene Antwort protokollieren soll.

Zum anderen bekommen Anfragen eine individuelle ID. Im Falle einer normalen HTTP-Anfrage wird die Antwort direkt zurückgegeben, wodurch eine manuelle Zuordnung nicht nötig ist. Bei dieser asynchronen Übertragungsart ist dies jedoch nicht der Fall. Aus diesem Grund ist es nötig, jeder Antwort manuell ihre zugehörige Anfrage zuzuordnen.

Eine HTTP-Anfrage besitzt zusätzlich noch einen Endpunkt an den die Anfrage geschickt wird und ein HTTP-Verb wie etwa GET, POST oder PATCH.

Diese werden in einer Kind-Klasse hinzugefügt.

HTTP-Antworten hätten zwar theoretisch einen Status-Code, dieser wird vom Connector allerdings nicht benötigt. Dementsprechend ist die Response-Klasse leer und erbt lediglich von der abstrakten Klasse.

Der Handler leitet die Message-Objekte lediglich an den Client weiter.

5.4. Verarbeitung der API-Anfrage mithilfe der ShopwareApiClient Klasse

Um eine Anfrage zu verarbeiten wird sie als Subrequest durch den internen Shopware-Ablauf geleitet. Dadurch wird der Overhead für Netzwerk und einen neuen PHP-Prozess gespart. Die Authentifizierung wird mithilfe eines Attributs des Request-Objektes übersprungen.

```php
class ShopwareApiClient {

  private HttpKernelInterface $kernel;

  public function __construct(HttpKernelInterface $kernel) {
    $this->kernel = $kernel;
  }

  public function sendRequest(ShopwareRequestMessage $request): ShopwareResponseMessage {
    $objRequest = $this->convertRequest($request);
    $objResponse = $this->kernel->handle($objRequest, HttpKernelInterface::SUB_REQUEST);
    return $this->convertResponse($objResponse, $request);
  }

  private function convertRequest(ShopwareRequestMessage $request): Request {
    $objRequest = Request::create(
                  '/api' . $request->getEndPoint(),
                  $request->getMethod(),
                  [],
                  [],
                  [],
                  [],
                  $request->getContent()
    );
    $objRequest->attributes->set('auth_required', false);
    return $objRequest;
  }

  private function convertResponse(Response $response, ShopwareRequestMessage $amqpRequest): ShopwareResponseMessage {
    return new ShopwareResponseMessage($amqpRequest->getRequestID(), $amqpRequest->getProcessID(), $response->getContent(), []);
  }
```

Nun gilt es lediglich noch, das System auch zu starten.

```
root@68b938743ccc:/app# root@68b938743ccc:/app# bin/console messenger:consume \
> saws_ping_queue saws_request_queue \
> --time-limit=300 --memory-limit=256M


 [OK] Consuming messages from transports "saws_ping_queue, saws_request_queue".


 // The worker will automatically exit once it has exceeded 256M of memory, been
 running for 300s or received a stop
 // signal via the messenger:stop-workers command.

 // Quit the worker with CONTROL-C.

 // Re-run the command with a -vv option to see logs about consumed messages.
```

Der gezeigte Befehl startet den Shopware-Consumer, welcher bevorzugt aus der Ping-Queue Anfragen verarbeitet und alternativ auf die Request-Queue wechselt.

Der Worker beendet sich selbst, wenn das Zeitlimit abgelaufen ist oder das Speicherlimit überschritten wurde. Er kann sich jedoch nicht selbst wieder starten.

Für diese Aufgabe wird ein Überwachungsprozess benötigt. Dieser hat auch die Aufgabe den Worker zu starten nachdem das System neu gestartet wurde.

Ein prominentes Beispiel wäre der „supervisord“, alternativ kann auch „systemd“ verwendet werden.

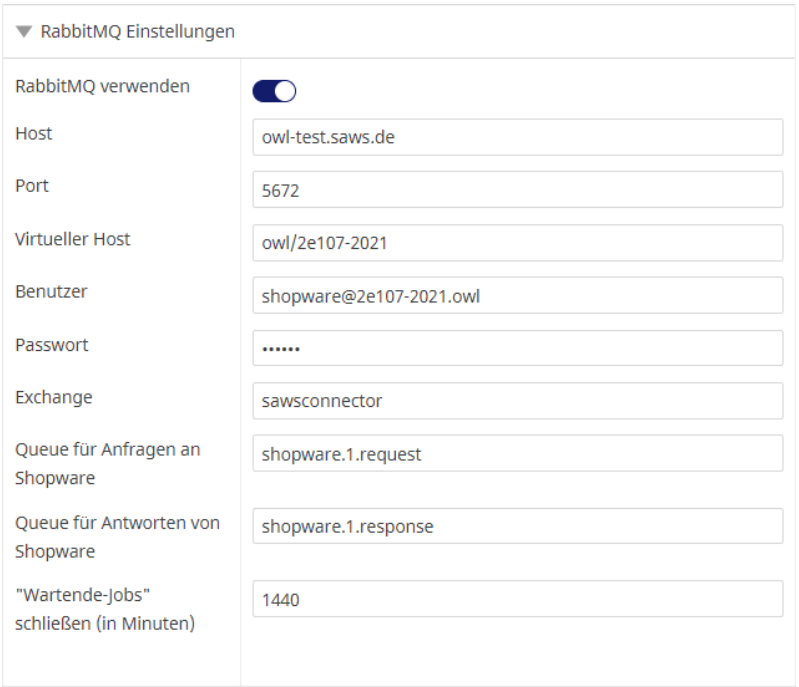
5.5. Umrüsten des Connectors auf Seiten von Contentserv

5.5.1. Erweitern der GUI

Im ersten Schritt wird die GUI um entsprechende Optionen erweitert. Dafür wird eine private Funktion genutzt, welche einfach in die Hauptlogik der GUI eingebunden wird.

```php
private function addAmqpOptions(EditorAdapterInterface $objEditor) {
  $__ = __('sawsshopware', 'sawsconnector');

  if ($objEditor instanceof sawsconnector\api\editor\CSGuiPaneEditorEditorAdapter) {
    $objEditor->setSectionTitle($__('SHOPWARE_AMQP_SECTION_TITLE'));
  }

  $objEditor->addField('amqpEnabled', $__('SHOPWARE_AMQP_ENABLED'), 'checkbox', false);

  $objEditor->addField('amqpHost', $__('SHOPWARE_AMQP_HOST'), 'caption', 'owl.saws.de');
  $objEditor->addField('amqpPort', $__('SHOPWARE_AMQP_PORT'), 'int', 5672);
  $objEditor->addField('amqpVHost', $__('SHOPWARE_AMQP_VHOST'), 'caption', 'owl/xxxxx-xxxx');
  $objEditor->addField('amqpUser', $__('SHOPWARE_AMQP_USER'), 'caption', 'shopware@xxxxx-xxx.owl');
  $objEditor->addField('amqpPassword', $__('SHOPWARE_AMQP_PASSWORD'), 'password', '');
  $objEditor->addField('amqpExchange', $__('SHOPWARE_AMQP_EXCHANGE'), 'caption', 'sawsconnector');
  $objEditor->addField('amqpRequestQueue', $__('SHOPWARE_AMQP_REQUEST_QUEUE'), 'caption', 'shopware.1.request');
  $objEditor->addField('amqpResponseQueue', $__('SHOPWARE_AMQP_RESPONSE_QUEUE'), 'caption', 'shopware.1.response');
  $objEditor->addField('amqpJobTimeout', $__('SHOPWARE_AMQP_JOB_TIMEOUT'), 'int', 1440);
}
```

Das Ergebnis wird folgendermaßen angezeigt:



Die Werte der angelegten Editor-Felder werden automatisch im entsprechenden Datenbank-Eintrag gespeichert und können ohne weitere Arbeit abgerufen werden.

Die Einstellungen können einfach vom zuständigen Record-Objekt geladen werden.

Ist die Übertragung über AMQP aktiv, werden die eben erstellten Einstellungen an den Transmitter weitergereicht. Dieser ist für den Verbindungsaufbau und das Senden der Daten zuständig.

5.5.2. Erstellen und Einbinden des neuen Clients

Intern benutzt der Transmitter die Client Klasse um Anfragen zu schicken. Das umleiten über AMQP kann daher einfach mit dem Austauschen des entsprechenden Objektes erreicht werden.

Die Methode „createClient“ wird modifiziert und erstellt bei eingeschalteter AMQP-Verbindung einen „AmqpClient“.

```
private function createClient(): Client {
  if ($this->getOption('amqpEnabled')) {
    $objClient = new AmqpClient(
            $this->getOption('ConnectorUrl'),
            $this->getOption('Username'),
            $this->getOption('Password'),
            $this->getOption('ShopwareApiVersion'),
            $this->getOption('Timeout')
    );
    $objClient->initAmqp(
            $this->getOption('amqpHost'),
            $this->getOption('amqpPort'),
            $this->getOption('amqpVHost'),
            $this->getOption('amqpUser'),
            $this->getOption('amqpPassword'),
            $this->getOption('amqpExchange'),
            $this->getOption('amqpRequestQueue'),
            $this->getOption('amqpResponseQueue')
    );
  }
  else {
    $objClient = new Client(
            $this->getOption('ConnectorUrl'),
            $this->getOption('Username'),
            $this->getOption('Password'),
            $this->getOption('ShopwareApiVersion'),
            $this->getOption('Timeout')
    );
  }
  return $objClient;
}
```

Nun müssen lediglich die gewollten Methoden überschrieben werden.

Dabei ist allerdings zu beachten, dass der originale Client Request-Objekte aus der genutzten cURL-Bibliothek zurückgibt. Um kompatibel zu bleiben muss der AmqpClient dies auch tun.

Da allerdings die Antwort des Shopware-Systems zum Zeitpunkt des Absendens nicht verfügbar ist, muss manuell ein Response-Objekt erstellt werden, welches lediglich eine Markierung enthält, dass die Nachricht Asynchron übertragen wurde.

Der für die Erstellung des Logs zuständige Code wird dabei erweitert, um diese Markierung zu erkennen und einen menschlich lesbaren Logeintrag zu erstellen.

5.5.3. Zuordnen der Antworten zu Logs

Nun werden im Transmitter noch die Shopware-Antworten abgefragt, in passende Logeinträge umgewandelt und in das entsprechende Log geschrieben. Jede Antwort besitzt dafür eine Prozess-ID welche von der Shopware-Seite aus der Anfrage kopiert wird.

```php
private function getAndLogAsyncResponse(AsyncTransmitterProcessCollection $objProcessCollection, LoggerInterface $objLogger): bool {
  /** @var AmqpClient $objClient */
  $objClient = $this->objClient;
  $objResponse = $objClient->basic_get();

  if ($objResponse === null) {
    return false;
  }

  $arrResponse = json_decode($objResponse->getBody(), true);

  if ($objProcessCollection->setCurrentProcess($arrResponse['processID'])) {
    $this->setLogger($objProcessCollection->getLogger());

    $objFakeResponse = new \CSHttpResponse($arrResponse['content'], []);

    $arrLogResponses = $this->objParser->parseResponse(
            $objFakeResponse, 'sync', $this->strTransferEntity, $this->strSendMode
    );
    $this->objResponseLogger->logResponses($arrLogResponses);

    $objJob = $objProcessCollection->getJob();
    $objJob->incRecordCounts();

    if ($objJob->getValue('CurrentRecord') >= $objJob->getValue('RecordCount')) {
      $objJob->finishJob();
    }
  }
  $objResponse->ack();

  return true;
}
```

6. Testphase

Die Testphase ist bei diesem Projekt besonders wichtig, da später Produktdaten und zum Beispiel auch Preise exportiert werden. Fehlerhafte Übertragungen können daher zu größeren Schäden führen.

Um dem vorzubeugen werden intern Code-Reviews durchgeführt. Alle Änderungen an vorhandener Logik, aber auch alle neuen Komponenten werden von Dritten überprüft. Unklare Programmteile werden mit Dokumentation versehen und viele Fehler können bereits im Voraus eliminiert werden.

Bei der Code-Review wurden keine potentiellen Fehler entdeckt. Die Klasse ShopwareClient wurde zur Erklärung der Symfony-Subrequests mit zusätzlicher Dokumentation versehen.

6.1. Testinstallationen auf verschiedenen Shopware-Systemen

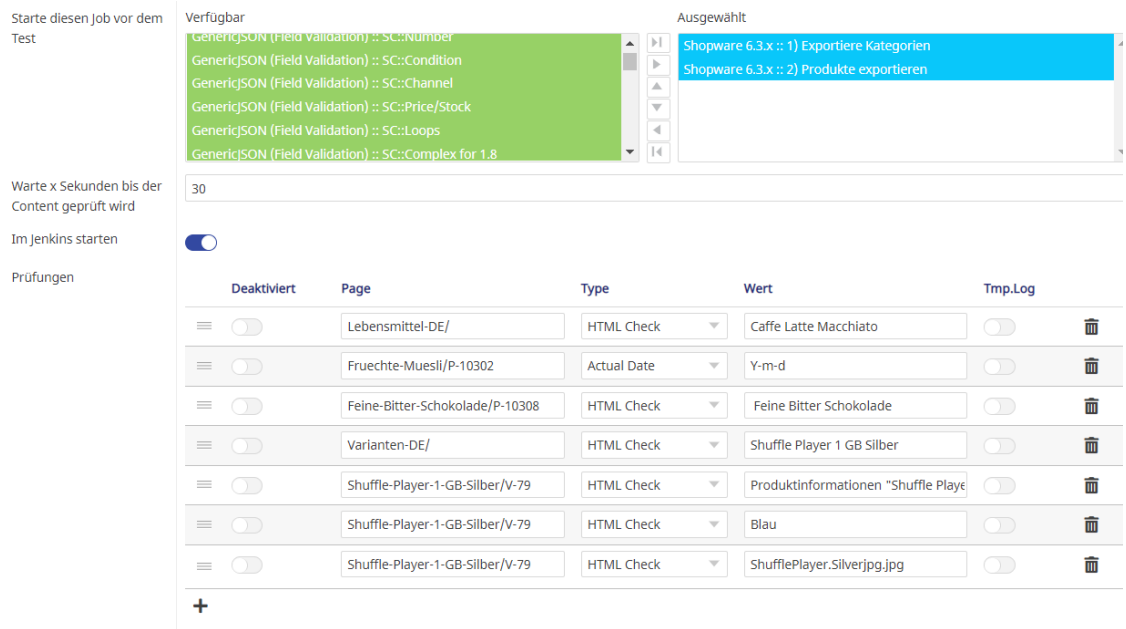
Da in der Praxis eingesetzte Shopware-Systeme sich oftmals in Version und Konfiguration unterscheiden, war es wichtig die Installation auf verschiedenen Instanzen zu testen.

Das Modul wurde auf folgenden Versionen installiert:

- Shopware 6.3.3.1 Stable
- Shopware 6.4.8.1 Stable
- Shopware 6.4.11.1

Intern existieren bereits Exporte und Prüfskripte für einen Quality-Management-Server. Dieser führt automatisch jede Nacht oder nach Bedarf auch manuell einen Export durch und prüft anschließend ob die exportierten Informationen im Shop gelistet sind.

Für die Kompatibilitätstests wurden diese Skripte wiederverwendet, da die genutzten Konfigurationen für den Export möglichst komplex und anspruchsvoll sind. Die momentanen Einstellungen wurden über Jahre verfeinert und erweitern um Fehler oder Abweichungen zu erkennen.



Eines der verwendeten MQ-Skripte für Shopware

Ausgabe eines der MQ-Skripte

Zusätzlich wurden diverse manuelle Tests durchgeführt um das Rückspielen von Fehlermeldungen und Statusinformationen von Shopware zu prüfen.

Bei diesen Tests fiel ein Fehler in der Programmierung der Passwort-Felder auf Seiten von Contentserv auf. Da der entsprechende Code nicht von SAWS ist, konnte das Problem nicht direkt behoben werden. Es reicht jedoch aus das geforderte Passwort zweimal einzugeben.

Das Problem existiert unabhängig und wurde durch das Projekt nur bemerkbarer. Daher wurde ein Ticket erstellt und die Tests fortgesetzt.

Zusätzlich fiel ein Fehler in der Shopware-GUI auf, der besonders langsame Shopware Systeme lahmlegte, indem er das System mit Status-Anfragen überflutet.

Der Fehler konnte schnell durch einen Locking-Mechanismus behoben werden, welcher eine Status-Anfrage erst erlaubt, wenn die vorherige abgeschlossen ist.

Anschließend wurden keine weiteren Fehler gefunden.

6.2. Übertragungstest mit Realdaten von Kunden

Bei SAWS wird zuweilen mit Repliken der Kundensysteme gearbeitet um möglichst nah an der zukünftigen Verwendung des Features zu bleiben und Fehler durch abweichende Konfigurationen zu minimieren.

Eines dieser Systeme wurde für diesen Test mit den neuesten Daten aus dem Produktivsystem des Kunden aktualisiert. Dieser Kunde gehört zu den Extremfällen der bereits beschriebenen Probleme des momentanen Exports.

Die Datenbank umfasst insgesamt mehr als 19.000 Produkte und über eine Million Datenpunkte.

Dieser Test soll zum einen den Export an seine Belastungsgrenze bringen, zum anderen soll er letzte Fehler vor einem Einsatz bei diesem Kunden bemerkbar machen.

Bei diesen Tests wurden einige PHP-8 Kompatibilitätsprobleme der Erweiterungen für Contentserv entdeckt. Diese konnte allerdings schnell behoben werden.

Die Verarbeitung und auch das Zurückliefern der Antworten erfolgten wie erwartet.

Die Worker beendeten sich wie gewollt nachdem das Arbeitsspeicher-Limit überschritten wurde.

Die Warteschlange funktionierte wie geplant.

Wie hier zu sehen, wurden die Verbindungsdaten wie plangemäß in echtzeit aktualisiert. Shopware akzeptierte und verarbeitete, alle Anfragen korrekt und blieb dabei „ansprechbar“.

Die Rückmeldungen wurden anschließend wieder vom SAWSConnector abgeholt und in die normale Antwortverarbeitung gegeben. Die hier gezeigten Logeinträge wurden über RabbitMQ empfangen.

7. Abschließende Analyse und Fazit

Nach Abschluss der Tests kann das Ziel des Projekts als erreicht gesehen werden. Der SAWSConnector kann Produkte nun mit der maximal möglichen Geschwindigkeit Formatieren und Exportieren. Mithilfe der Worker kann der Import auf Seiten Shopwares jetzt gesondert von der HTTP-API durchgeführt werden und durch die Limits und Anzahl der Worker kann die gewollte Auslastung erreicht werden, ohne den Shop zu überlasten.

Sowohl das Modul als auch der verbesserte SAWSConnector sind nun Einsatzbereit und werden mit dem nächsten Feature-Release veröffentlicht. Erfahrungsgemäß ist hier Trotz aller Tests mit einigen kleineren Fehlern und Problemen zu rechnen.

Das Kapseln der HTTP-Nachrichten hatte den gewollten Effekt. Der SAWSConnector besitzt bestimmte Anpassungen für verschiedene Versionen der Shopware-API. Dank der Kapselung konnte der bereits bewährte Code weiter für die Kommunikation mit Shopware verwendet werden. Das ist einer der Gründe für die geringe Anzahl an gefundenen Fehlern.

Für die Zukunft sind einige Erweiterungen in Überlegung, jedoch noch nicht fest geplant.

Zum einen wäre es möglich die Konfiguration von Shopware vollständig über den Connector auf Seiten Contentservs zu übernehmen. Da Shopware auch intern für alle Aktionen der Admin-Oberfläche eine HTTP-API nutzt, wäre sowohl die Installation als auch das Spiegeln der Einstellungen ein erreichbares Ziel. Eine solche Erweiterung würde den Einrichtungsaufwand weiter reduzieren.

Zusätzlich gäbe es die Möglichkeit eigene Import-Schnittstellen hinzuzufügen, welche einige Validierungen überspringen und so den Import auch auf Seiten Shopwares beschleunigen. Dabei sollte jedoch immer darauf geachtet werden, kompatibel zu Shopware-Systemen ohne das Modul zu bleiben, da manche Kunden keine Möglichkeit haben ein solches Plug-In zu installieren oder dies nicht möchten.

Als dritte potentielle Erweiterung wäre eine AMQP-Authentifizierung mithilfe von Zertifikaten eine gute Möglichkeit die Sicherheit des Systems zu verstärken.

Zu guter Letzt gilt es denn geplanten und gebrauchten Zeitaufwand abzugleichen.

Abschnitt	Geplant	Verwendet	Differenz
Analysephase	10	11	-1
Ist-Analyse	2	1	+1
Technische Möglichkeit der der Umsetzung	7	9	-2
Wirtschaftlichkeitsanalyse	1	1	0
Entwurfsphase	11	11	0
Entwurf des Shopware-Plug-Ins	6	7	-1
Entwurf der Connectorseitigen Übertragung	5	4	+1
Implementierungsphase	35	34	+1
Erstellen und Einbinden des Plug-Ins	5	4	+1
Erstellen und Anwenden der nötigen Configs	7	10	-3
Implementierung des Handlers und der Message Ojekte	5	4	+1
Verarbeitung der API-Anfrage mithilfe der ShopwareApiClient Klasse	8	6	+2
Umrüsten des Connectors	10	10	0
Testphase	7	7	-1
Testinstallation auf verschiedenen Shopware-Systemen	3	4	-1
Übertragungstest mit Realdaten von Kunden	4	4	0
Dokumentation	6	6	0
Code-Doku	3	4	-1
Benutzer-Doku	3	2	+1
Gesamt	69	70	-1

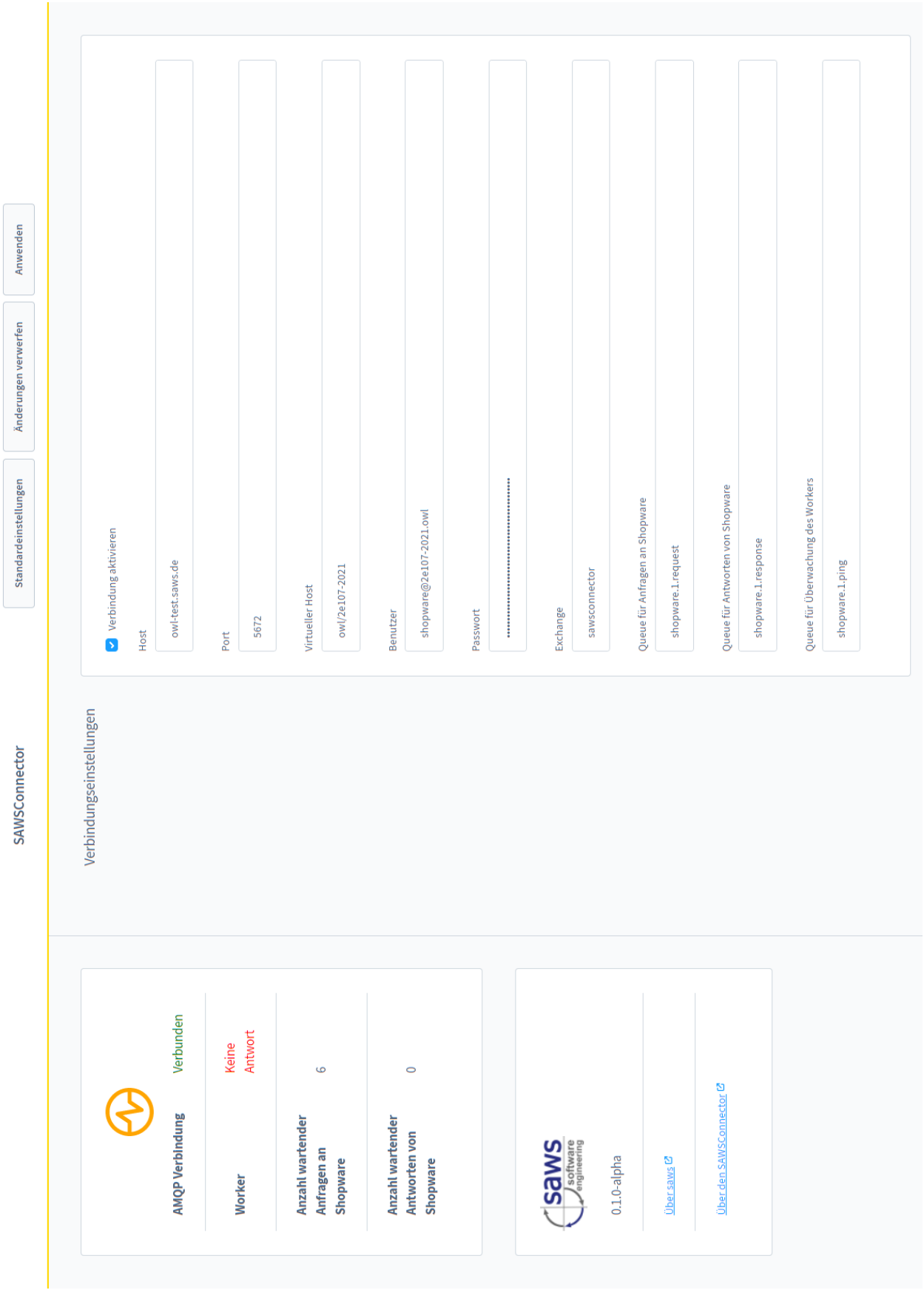
Die meisten Teile der Implementierung benötigten weniger Zeit als geplant. Allerdings war das Erstellen der Konfigurationsdateien deutlich aufwendiger als gedacht, da bisher noch keine Erfahrung mit dem „Vue.js“ Framework vorhanden war.

8. Quellen

Shopware Dokumentation. (04 2022). Von https://developer.shopware.com/docs/guides/plugins abgerufen

Symfony Dokumentation. (04 2022). Von https://symfony.com/doc/current/index.html abgerufen

Anhang


SAWSConnector
Standardeinstellungen
Änderungen verwerfen
Anwenden
Verbindungseinstellungen
Verbindung aktivieren
Host
owl-test.saws.de
Port
5672
Virtueller Host
owl/2e107-2021
Benutzer
shopware@2e107-2021.owl
Passwort
Exchange
sawsconnector
Queue für Anfragen an Shopware
shopware.1.request
Queue für Antworten von Shopware
shopware.1.response
Queue für Überwachung des Workers
shopware.1.ping
AMQP Verbindung
Verbunden
Worker
Keine Antwort
Anzahl wartender Anfragen an Shopware
6
Anzahl wartender Antworten von Shopware
0
saws software engineering
0.1.0-alpha
Über saws
Über den SAWSConnector


SawsMessageQueueImport
Client
ShopwareApiClient
-kernel: HttpKernelInterface
+__construct(kernel: HttpKernelInterface)
+sendRequest(request: ShopwareRequestMessage): ShopwareResponseMessage
-convertRequest(request: ShopwareRequestMessage): HttpFoundation\Request
-convertResponse(response: HttpFoundation\Response
amqpRequest: ShopwareRequestMessage
): ShopwareResponseMessage
Message
SawsAsyncMessage
-requestID: string
-processID: string
-content: string
-headers: array
+__construct(requestID: string, processID: string,
content: string, headers: array)
ShopwareRequestMessage
-endpoint: string
-method: string
+__construct(requestID: string, processID: string,
content: string, headers: array,
endpoint: string, method: string)
ShopwareResponseMessage
WorkerPingMessage
-ping: int
+getPing(): int
Config
Settings
-config: array
-scope: string
+__construct(plugin: SawsMessageQueueImport)
+getConfigPath(configName: string): string
+getSawsPath(): string
+setScope(scope: string)
+getScope(): string
+setValue(key: string, value)
+getValue(key: string, replacement)
+load()
+save()
saves to
AMQPConfig
-settings: Settings
-enabled: boolean
-host: string
-port: int
-vhost: string
-exchange: string
-request_queue: string
-response_queue: string
-user: string
-password: string
+__construct(settings: Settings)
+applyConfig()
-getFrameworkPath()
-buildConfig(): string
-buildAmqpUrl(): string
-buildOptions(queueName: string): array
+connect()
-setup(queueName : string)
-buildConnection(queueName: string): Connection
+countRequestMessages(): int
+countResponseMessages(): int
-countMessages(queueName:string): int
+isDeployed(): boolean
+hasPassword(): boolean
+getPassword(): string
+setPassword(password: string)
+setPong(pong: int)
+getPong(): int
MessageQueue
Handler
ShopwareApiHandler
-config: SystemConfigService
-client: ShopwareApiClient
+__construct(client: ShopwareApiClient, bus: MessageBusInterface)
+handle(message)
+getHandledMessages(): iterable
WorkerPingHandler
-config: AMQPConfig
__construct(config: AMQPConfig)
+handle(message)
+getHandledMessages(): iterable
SawsMessageQueueImport
+build(container: ContainerBuilder)
-loadConfigs(container: ContainerBuilder)
Framework\Api\Controller
SettingsController
-ampqConfig: AMQPConfig
__construct(amqpConfig: AMQPConfig)
+setSettings(request: HttpFoundation\Request, context: Context): HttpFoundation\JsonResponse
+getSettings(request: HttpFoundation\Request, context: Context): HttpFoundation\JsonResponse
+checkConnection(request: HttpFoundation\Request, context: Context): HttpFoundation\JsonResponse
-getErrorResponse(ex: Exception, merge: array): HttpFoundation\JsonResponse
+pingWorker(request: HttpFoundation\Request, context: Context): HttpFoundation\JsonResponse
Shopware\Core\Framework
Plugin
Shopware-Code
MessageQueue\Handler
AbstractMessageHandler
Shopware-Code
Symfony\Bundle\FrameworkBundle\Controller
AbstractController
Symfony-Code
uses
handles
dispatches
receives
dispatches
returns pong
controls