ItkWebGains

Entwicklerdokumentation

André Fischer

30.03.2026

Inhaltsverzeichnis

1 Paket und Namespace **2**

2 Symfony / Shopware kurz erklärt **2**

3 Architekturüberblick **3**

4 Alle PHP-Klassen **4**

5 ConfigService **5**

5.1 Wichtige Konfigurationsfelder aus `config.xml` 6

6 StorefrontSubscriber **6**

7 OrderSubscriber **7**

8 FeedController **8**

9 ExportController (Admin API) **9**

10 Logging **10**

11 TransactionExportService **10**

12 Storefront-Template-Ausgabe **10**

13 Lokales Testen **10**

14 Erweiterungspunkte **11**

1 Paket und Namespace

- **Plugin-Klasse:** `Itk\\WebGains\\ItkWebGains`
- PSR-4: `Itk\\WebGains\\` → `src/`
- **Shopware:** `shopware/core` ~6.6.0

Es gibt **keine** Plugin-eigenen DAL-Entities und **keine** Migrationen im Repository – Zustand steckt in **Order Custom Fields** und **Cookies**.

---

2 Symfony / Shopware kurz erklärt

- **Event Subscriber** (**EventSubscriberInterface**): Du registrierst dich mit `getSubscribedEvents()` bei bestimmten Events (z. B. **CheckoutFinishPageLoadedEvent**). Symfony ruft deine Methode auf, wenn das Event gefeuert wird – ohne den Core zu ändern.

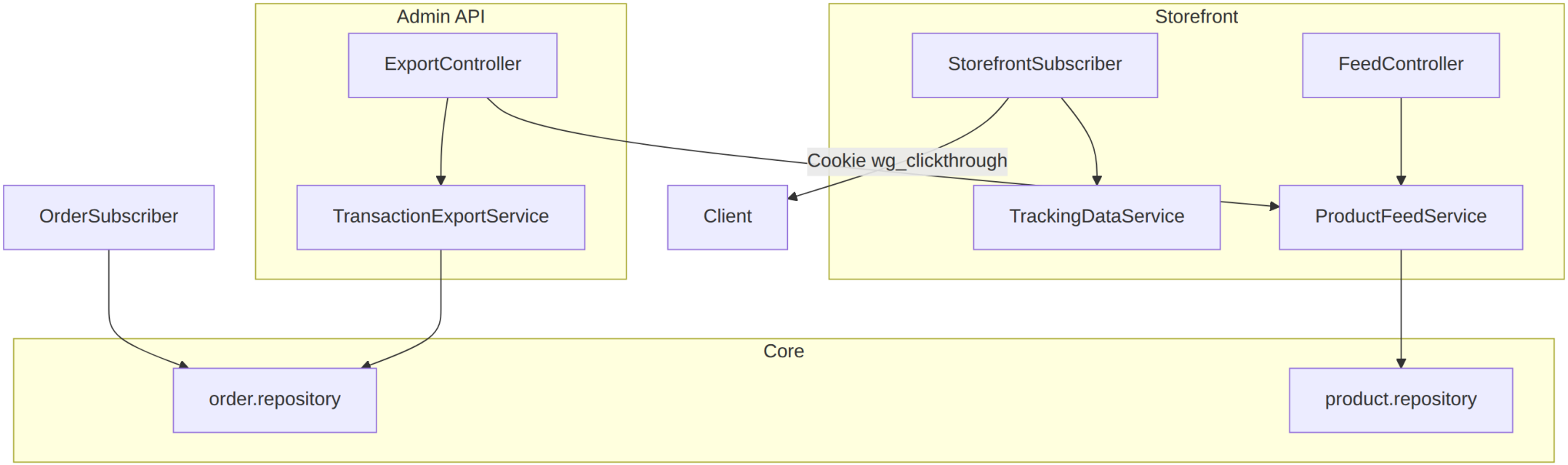
- **RequestStack:** Gibt dir im Subscriber den aktuellen **Request** (Cookies, Query-Parameter). In CLI oder async-Kontext kann `getCurrentRequest()` **null** sein – der Code prüft das.

- **EntityWrittenEvent:** Feuert, wenn DAL eine Entität schreibt. **OrderSubscriber** nutzt `OrderEvents::ORDER_WRITTEN_EVENT`, filtert auf **entityName === 'order'** und aktualisiert dieselbe Order erneut per `order.repository->update` mit **customFields**.

- **CacheInterface (`cache.app`):** Symfony-Cache; **FeedController** nutzt ihn optional für den CSV-String.

---

3 Architekturüberblick



4 Alle PHP-Klassen

Klasse	Pfad unter `src/`	Zweck
Itk\WebGains\ItkWebGains	ItkWebGains.php	Plugin-Bootstrap (Install/Uninstall-Hooks, ohne eigene Business-Logik im aktuellen Stand).
Itk\WebGains\Service\ConfigService	Service/ConfigService.php	Zentraler Lesezugriff auf ItkWebGains.config.* (aktiv, Program-/Event-/Container-/Campaign-IDs, Feed-/Cache-Flags usw.).
Itk\WebGains\Service\TrackingDataService	Service/TrackingDataService.php	Erzeugt WebGains-**Container-/Tracking-Skripte** und zugehörige Daten (`ntm`) je **Seitentyp** (Produkt, Kategorie, Checkout, …).
Itk\WebGains\Service\ProductFeedService	Service/ProductFeedService.php	Baut den **Produkt-Feed als CSV** für den öffentlichen Feed-Endpunkt (Felder wie ID, Name, Preis, URLs).
Itk\WebGains\Service\TransactionExportService	Service/TransactionExportService.php	Exportiert **Bestellungen/Transaktionen** als CSV für WebGains (Zeiträume, Aktionen wie new/approve/reject/adjust).

Klasse	Pfad unter `src/`	Zweck
Itk\WebGains\Subscriber\StorefrontSubscriber	Subscriber/StorefrontSubscriber.php	Hängt **Tracking-Daten** an Storefront-Pages, setzt u. a. **Clickthrough-Cookie** und reagiert auf Checkout-/Response-Events.
Itk\WebGains\Subscriber\OrderSubscriber	Subscriber/OrderSubscriber.php	Bei **Order written**: liest **WebGains-Clickthrough** aus dem Cookie und schreibt es in **Order-Custom-Fields**.
Itk\WebGains\Storefront\Controller\FeedController	Storefront/Controller/FeedController.php	Öffentlicher **GET**-Endpunkt für products.csv inkl. optional **Cache** und Aktiv-/Feed-Prüfung.
Itk\WebGains\Controller\Admin\ExportController	Controller/Admin/ExportController.php	**Admin-API**-Aktionen: CSV-Downloads für Transaktions-Exporte und ggf. Produktfeed aus dem Backend.
Itk\WebGains\Struct\WebGainsTrackingExtension	Struct/WebGainsTrackingExtension.php	**Page-Extension** (webgains_tracking): transportiert das generierte **Script-Markup** zur Storefront-API/Page.

5 ConfigService

Konstante: CONFIG_PREFIX = 'ItkWebGains.config.'
Alle öffentlichen Getter kapseln **systemConfigService->get($prefix . $key, $salesChannelId)** mit Defaults wo nötig.

Im Alltag ist **ConfigService** die fachliche Wahrheit für die Plugin-Konfiguration. Wer neue Tracking- oder Exportlogik einbaut, sollte deshalb über die Getter gehen und nicht quer im Code rohe Config-Keys verteilen.

5.1 Wichtige Konfigurationsfelder aus `config.xml`

Key	Typ	Zweck
active	bool	Hauptschalter des Plugins
programId	text	WebGains Program ID
eventId	text	Event-ID für Transaktions-Tracking
containerId	text	Container-Tag-ID
campaignId	text	optionale Campaign-ID
enableContainerTag	bool	Container-Script auf Storefront-Seiten
enableTransactionTracking	bool	Tracking auf der Finish-Seite
respectGdpr	bool	Consent berücksichtigen
gdprDefault	single-select	Fallback-Wert `-1 / 0 / 1`
enableProductFeed	bool	CSV-Feed aktiv
enableProductFeedCache	bool	Feed über Cache ausliefern
productFeedCacheLifetime	int	Cache-Laufzeit in Minuten
enableTransactionExport	bool	Admin-CSV-Exporte aktiv
transactionValidationDays	int	Validierungsfenster für Transaktionen
trackClickthrough	bool	Landing-Parameter in Cookie sichern
clickthroughCookieLifetime	int	Cookie-Laufzeit in Tagen

---

6 StorefrontSubscriber

Datei: `Subscriber/StorefrontSubscriber.php`
Cookie-Name: `wg_clickthrough`. Gespeichert wird ein JSON-Objekt mit genau diesen Feldern:

- **siteid**: WebGains-Site-ID aus dem Landing-Request
- **clickref**: Click-Referenz für die Attribution

- `utm_source`: typischerweise **webgains**
- **timestamp**: Unix-Zeitpunkt der Erfassung

Subscribed Events:

- **NavigationPageLoadedEvent** – Homepage vs. Kategorie, Scripts: Landing + Container
- **ProductPageLoadedEvent**
- **SearchPageLoadedEvent**
- **WishlistPageLoadedEvent**
- **CheckoutCartPageLoadedEvent**, **OffcanvasCartPageLoadedEvent**
- **CheckoutConfirmPageLoadedEvent**
- **CheckoutFinishPageLoadedEvent** – Transaktions-Tracking
- `KernelEvents::RESPONSE` – Setzen des Clickthrough-Cookies (wenn **trackClickthrough** und aktive Route)

Erweiterung der Page: Über **WebGainsTrackingExtension** (Struct) an die Page gehängt; Templates/Themes müssen die Extension auslesen, falls ihr Skripte nicht rein über globale Listener injiziert.

Die Struct ist absichtlich schlank: Kernstück ist das Script-Snippet aus **TrackingDataService**, das in Twig über den API-Alias `webgains_tracking` bereitliegt.

Wichtig: Jede Methode prüft früh `isTrackingEnabled($salesChannelId)` (Plugin aktiv + ggf. weitere Flags).

---

7 OrderSubscriber

Event: `OrderEvents::ORDER_WRITTEN_EVENT`
Zweck: Beim Schreiben einer Bestellung Cookie `wg_clickthrough` lesen und **Order Custom Fields** setzen:

- `webgains_clickthrough_siteid`
- `webgains_clickthrough_clickref`
- `webgains_clickthrough_utm_source`
- `webgains_clickthrough_timestamp`

Endlosschleifen-Schutz: Wenn **customFields** im Payload bereits `webgains_clickthrough_siteid` enthalten, wird nicht erneut aktualisiert.

Voraussetzung: Die Custom Field **Definitionen** müssen in Shopware existieren (das Plugin legt sie in **`ItkWebGains::install` nicht** an – die **install**-Methode ist praktisch leer außer **`parent::install`**). In der Praxis müssen die Felder per Migration, Admin oder Baseline-Projekt angelegt sein.

Wichtig für Debugging: Der Subscriber prüft nicht zusätzlich noch einmal „Plugin aktiv“, sondern verarbeitet den Cookie, sobald ein **order**-Write-Event mit Request-Kontext ankommt. Wenn also trotz deaktiviertem Tracking Felder beschrieben werden, zuerst den Cookie- und Request-Pfad untersuchen.

---

8 FeedController

Route: GET /webgains/feed/products.csv
Name: `frontend.webgains.feed.products`
Scope: storefront

Ablauf:

1. **`configService->isActive`** und **isProductFeedEnabled** – sonst 404.
2. **getOrGenerateFeed**: ohne Cache → **`ProductFeedService::generateProductFeed`**; mit Cache → Key `webgains_product_feed_ + md5($salesChannelId)`, TTL aus Config (Minuten).
3. Response: CSV, **text/csv; charset=utf-8**, **Content-Disposition: attachment**, **Cache-Control: max-age=...**.

Dependency: `cache.app` (Symfony default cache pool).

Die eigentliche CSV-Erzeugung sitzt in **ProductFeedService**. Dort werden Produkte über das Repository in Batches geholt und in eine feste Spaltenreihenfolge übersetzt. Wenn Feed-Inhalte falsch sind, ist fast nie der Controller schuld, sondern der Service oder seine Produktkriterien.

---

9 ExportController (Admin API)

Basis: AbstractController (nicht StorefrontController).
Routen (alle **POST**, `_routeScope` **api**):

Name	Pfad
api.action.webgains.export.transactions.new	/api/_action/webgains/export/transactions/new
api.action.webgains.export.transactions.approve	/api/_action/webgains/export/transactions/approve
api.action.webgains.export.transactions.reject	/api/_action/webgains/export/transactions/reject
api.action.webgains.export.transactions.adjust	/api/_action/webgains/export/transactions/adjust
api.action.webgains.export.products	/api/_action/webgains/export/products

Request-Parameter (Auszug):

- **new**: **from**, **to** (DateTime-Strings), optional **salesChannelId**
- **approve**: **orderNumbers** (array), **salesChannelId**
- **reject**: **orders** (array), **salesChannelId**
- **adjust**: **adjustments**, **salesChannelId**
- **products**: **salesChannelId**

Hinweis: In `services.xml` ist **ExportController** als **public="true"** registriert, hat aber **kein** `controller.service_arguments`-Tag – je nach Shopware-Version kann die Route trotzdem funktionieren, wenn der Controller als Service-Controller registriert ist. Wenn 404/Controller not found: mit `debug:router` und Container-Definition abgleichen.

---

10 Logging

- **Service:** `webgains_logger` → Monolog-Channel **webgains**
- **Handler:** `var/log/webgains.log`, Level **100 (DEBUG)**

TrackingDataService erhält diesen Logger injiziert.

---

11 TransactionExportService

Der Service bildet die CSV-Daten für **new / approve / reject / adjust**. Fachlich passiert dort:

- Order-Auswahl nach Zeitraum oder Order-Nummern
- Mapping auf den von WebGains erwarteten Exporttyp
- Nutzung von **transactionValidationDays** als fachliches Zeitfenster
- Lesen vorhandener Order-Custom-Fields wie Clickthrough- und Kampagneninformationen

Wenn Exportdaten fachlich falsch wirken, sitzt die Ursache fast immer hier und nicht im Controller.

---

12 Storefront-Template-Ausgabe

Die eigentliche Ausgabe der Tracking-Snippets passiert nicht im Subscriber selbst, sondern über das Template `@ItkWebGains/storefront/layout/meta.html.twig` bzw. die dort erweiterten Blöcke. Der Subscriber liefert also Daten und Script-Snippets an Twig; das Template entscheidet, an welcher Stelle sie im HTML landen.

---

13 Lokales Testen

1. Plugin aktivieren, Config für einen Kanal setzen.

2. **curl -I https://localhost/webgains/feed/products.csv** (mit Host-Header für den Kanal, falls nötig).

3. Storefront mit **?siteid=…** oder `utm_source`=… aufrufen, Cookie prüfen, Bestellung durchspielen, Order in Admin auf Custom Fields prüfen.

4. Admin-Export mit Postman + OAuth gegen /`api`/`_action`/`webgains`/`...`.

---

14 Erweiterungspunkte

- Zusätzliche **Subscriber** für weitere Page-Events.
- **TrackingDataService** erweitern für neue Parameter (Cookie-Schema und Order-Felder synchron anpassen).
- **ProductFeedService** für zusätzliche CSV-Spalten oder Filter.
- Custom Fields per **Migration** im Plugin nachziehen, damit Fresh-Installs nicht manuell im Admin nacharbeiten müssen.