Product Bundles and SKUs

Namazu Elements Core Features · https://namazustudios.com/docs/namazu-elements-core/product-bundles-and-skus/

Namazu Element Version 3.8+

Overview

The Commerce feature provides two cooperating resources that map payment-provider purchases to in-game rewards.

- **Product SKU Schemas** — a registry of payment-provider identifiers (e.g. `com.apple.appstore`) used across the system. The four built-in IAP providers are seeded automatically on startup.
- **Product Bundles** — per-application configurations that map a provider SKU or product id to a set of items to award when a verified purchase is received. All read and write operations are superuser-only; only `processVerifiedPurchase` is executed in a user context (called internally by the receipt services).

When an IAP receipt is verified, the receipt service calls `ProductBundleService.processVerifiedPurchase(schema, productId, originalTransactionId)`, which looks up the matching bundle for the user's application and issues the configured `RewardIssuance` records idempotently.

Product SKU Schema model

Class: `dev.getelements.elements.sdk.model.goods.ProductSkuSchema`

Field	Type	Description
`id`	`String`	Database id. Assigned on creation; null on write.
`schema`	`String`	**Required.** Reverse-DNS payment-provider identifier, e.g. `com.apple.appstore`.

Product Bundle model

Class: `dev.getelements.elements.sdk.model.goods.ProductBundle`

Field	Type	Description
`id`	`String`	Database id. Null on create, required on update
`schema`	`String`	**Required.** Reverse-DNS payment-provider identifier.
`application`	`Application`	The owning application. Required on create.
`productId`	`String`	**Required.** Product id as it appears in the provider's catalog.
`displayName`	`String`	Optional display title shown to end users.
`description`	`String`	Optional description shown to end users.
`display`	`boolean`	Whether the frontend should surface this bundle to users. Defaults to `false`.
`productBundleRewards`	`List<ProductBundleReward>`	Rewards issued on purchase. See below.
`metadata`	`Map<String, Object>`	Arbitrary key-value metadata for application use.
`tags`	`List<String>`	Searchable tags for filtering.

ProductBundleReward

Class: `dev.getelements.elements.sdk.model.application.ProductBundleReward`

Field	Type	Description
`itemId`	`String`	**Required.** Database id of the `Item` to award.
`quantity`	`Integer`	Quantity to award. Omit (or `null`) for `DISTINCT` items; positive integer for `FUNGIBLE` items. If `null` on a `FUNGIBLE` item the service defaults to `1`.

The compound key **(application, schema, productId)** is unique — only one bundle may exist for a given product in a given application.

DAO interface (ProductBundleDao)

Interface: `dev.getelements.elements.sdk.dao.ProductBundleDao`

Key methods

```
// Paginated listing — all filters are optional; null/blank values are ignored
Pagination<ProductBundle> getProductBundles(int offset, int count);
Pagination<ProductBundle> getProductBundles(String applicationNameOrId, int offset, int count);
Pagination<ProductBundle> getProductBundles(String applicationNameOrId, String schema, int offset,
int count);
Pagination<ProductBundle> getProductBundles(String applicationNameOrId, String schema,
                                            String productId, List<String> tags, int offset, int
count);
Pagination<ProductBundle> getProductBundlesByTag(String tag, int offset, int count);
```

```
// Lookup
ProductBundle getProductBundle(String id);
ProductBundle getProductBundle(String applicationNameOrId, String schema, String productId);
```

```
// Writes — always use inside a transaction
ProductBundle createProductBundle(ProductBundle bundle);  // throws DuplicateException on conflict
ProductBundle updateProductBundle(ProductBundle bundle);
void deleteProductBundle(String id);                      // throws NotFoundException if missing
```

Transaction usage (recommended)

Always perform DAO writes inside a transaction for consistency and automatic retry on failure.

```
@Inject
private Provider<Transaction> transactionProvider;

final var created = transactionProvider.get().performAndClose(tx -> {
    return tx.getDao(ProductBundleDao.class).createProductBundle(bundle);
});
```

DAO interface (ProductSkuSchemaDao)

Interface: `dev.getelements.elements.sdk.dao.ProductSkuSchemaDao`

Key methods

```
Pagination<ProductSkuSchema> getProductSkuSchemas(int offset, int count);
ProductSkuSchema getProductSkuSchema(String id);

// Idempotent create — returns existing record if the schema string already exists
ProductSkuSchema createProductSkuSchema(ProductSkuSchema productSkuSchema);

// Upsert by schema string — safe to call repeatedly; used by seeder and provider plugins
ProductSkuSchema ensureProductSkuSchema(String schema);

void deleteProductSkuSchema(String id)
```

Service interfaces

ProductBundleService

Interface: `dev.getelements.elements.sdk.service.goods.ProductBundleService`

Annotations: `@ElementPublic`, `@ElementServiceExport`, `@ElementServiceExport(name = UNSCOPED)`

All CRUD methods (`getProductBundles`, `getProductBundle`, `createProductBundle`, `updateProductBundle`, `deleteProductBundle`) are SUPERUSER only. Regular USER level receives a `ForbiddenException` if they call them directly.

However, the key USER-context method is:

```
/**
 * Looks up the ProductBundle for the current user's application matching
 * (schema, productId) and issues a RewardIssuance for each configured reward.
 * Uses originalTransactionId as an idempotency key — repeated calls with the
 * same arguments will not create duplicate rewards.
 *
 * Returns an empty list if no matching bundle is found.
```

```
 */
List<RewardIssuance> processVerifiedPurchase(
        String schema,
        String productId,
        String originalTransactionId);
```

This method is called automatically by the built-in receipt services after a purchase is verified. You should only call it directly if you are implementing a custom payment provider, and only after the receipt has been verified by the payment provider.

ProductSkuSchemaService

Interface: `dev.getelements.elements.sdk.service.goods.ProductSkuSchemaService`

Annotations: `@ElementPublic`, `@ElementServiceExport`, `@ElementServiceExport(name = UNSCOPED)`

```
Pagination<ProductSkuSchema> getProductSkuSchemas(int offset, int count);
ProductSkuSchema getProductSkuSchema(String id);
ProductSkuSchema createProductSkuSchema(ProductSkuSchema productSkuSchema);  // idempotent
ProductSkuSchema ensureProductSkuSchema(String schema);                      // upsert by string
void deleteProductSkuSchema(String id);
```

Seeding

At startup, `ProductSkuSchemaSeeder` automatically calls `ensureProductSkuSchema` for all four built-in provider schemas so they are always available:

Provider	Schema constant	Value
Apple App Store	`AppleIapReceiptService.APPLE_IAP_SCHEME`	`com.apple.appstore`
Google Play	`GooglePlayIapReceiptService.GOOGLE_IAP_SCHEME`	`com.android.vending`
Meta Quest / Oculus	`OculusIapReceiptService.OCULUS_IAP_SCHEME`	`com.oculus.platform`
Meta / Facebook Platform	`FacebookIapReceiptService.FACEBOOK_IAP_SCHEME`	`com.facebook.platform`

Custom schemas are registered the same way — call `ensureProductSkuSchema` from your Element 's startup code.

How processVerifiedPurchase works

Call this method after your payment provider confirms a purchase. It resolves the correct `ProductBundle` for the authenticated user's application and issues the configured rewards in a single transactional step:

```
List<RewardIssuance> issuances = productBundleService.processVerifiedPurchase(
        MYPROVIDER_IAP_SCHEME,   // schema — identifies the payment provider
        providerProductId,        // productId — as it appears in the provider's catalog
        providerTransactionId     // originalTransactionId — used as the idempotency key
);
// issuances is empty if no bundle is configured for this product; never null
```

The method will:

1. Resolve the current user's application from their active `Profile`.
2. Look up the `ProductBundle` matching `(application, schema, productId)`. If none is found, return an empty list — this is not treated as an error.
3. Inside a transaction, create a `RewardIssuance` for each entry in `productBundleRewards`, using `rewardIssuanceDao.getOrCreateRewardIssuance` so repeated calls with the same `originalTransactionId` are safe and will not produce duplicate rewards.
4. Return the list of `RewardIssuance` records that were created or already existed.

The idempotency key for each reward entry is derived from `"product-bundle.<originalTransactionId>.<itemId>.<index>"`, so use a stable, provider-assigned transaction id to get correct deduplication behaviour across retries.

For reference, the core logic looks like this:

```
// Annotated illustration — see UserProductBundleService for the authoritative source
List<RewardIssuance> processVerifiedPurchase(String schema, String productId, String txId) {
    final var applicationId = currentProfileSupplier.get().getApplication().getId();

    final ProductBundle bundle;
    try {
        bundle = productBundleDao.getProductBundle(applicationId, schema, productId);
```

```
    } catch (NotFoundException e) {
        return List.of(); // no bundle configured — not an error
    }

    return transactionProvider.get().performAndClose(tx -> {
        final var rewardIssuanceDao = tx.getDao(RewardIssuanceDao.class);
        final var itemDao = tx.getDao(ItemDao.class);
        final var issuances = new ArrayList<RewardIssuance>();

        for (int i = 0; i < bundle.getProductBundleRewards().size(); i++) {
            final var reward = bundle.getProductBundleRewards().get(i);
            final var item = itemDao.getItemByIdOrName(reward.getItemId());
            final int qty = DISTINCT.equals(item.getCategory()) ? 1
                    : (reward.getQuantity() != null ? reward.getQuantity() : 1);

            final var issuance = new RewardIssuance();
            issuance.setUser(user);
            issuance.setItem(item);
            issuance.setItemQuantity(qty);
            issuance.setType(PERSISTENT);
            issuance.setState(ISSUED);
            issuance.setSource("PRODUCT_BUNDLE." + schema + "." + productId);
            issuance.setContext("product-bundle." + txId + "." + reward.getItemId() + "." + i);

            issuances.add(rewardIssuanceDao.getOrCreateRewardIssuance(issuance));
        }

        return issuances;
    });
}
```

Adding a new payment provider

If you are building a custom Element for a payment provider not already supported, follow these steps.

1. Register a schema string

Use reverse-DNS notation and declare a constant:

```
String MYPROVIDER_IAP_SCHEME = "com.example.myprovider";
```

Call `ensureProductSkuSchema` at Element startup to register it:

```
@Inject
private ProductSkuSchemaService productSkuSchemaService;

// In your startup code / or SYSTEM_EVENT_ELEMENT_LOADED event:
productSkuSchemaService.ensureProductSkuSchema(MYPROVIDER_IAP_SCHEME);
```

2. Create Product Bundles

Use the admin API or the web console to create `ProductBundle` records mapping your provider's product ids to in-game items. The bundle uniquely identifies a product by `(application, schema, productId)`.

3. Verify the purchase and call processVerifiedPurchase

After verifying the purchase with your provider's server-to-server API, call:

```
@Inject
private ProductBundleService productBundleService;

List<RewardIssuance> issuances = productBundleService.processVerifiedPurchase(
        MYPROVIDER_IAP_SCHEME,
        providerProductId,
        providerTransactionId   // used as idempotency key
);
```

The call is safe to retry — repeated calls with the same `originalTransactionId` return the existing issuances without creating duplicates.

> **Tip:** With `dev.getelements.elements.auth.enabled=true` set in your Element attributes you can inject `UserService` and call `userService.getCurrentUser()` to get the authenticated user making the request. See the `example-element` project for a complete example.

Best practices and recommendations

- Always call `processVerifiedPurchase` inside a user-scoped context (i.e. the user whose profile contains the matching application).
- Use `originalTransactionId` values that are truly unique per purchase and stable across retries — provider-assigned transaction ids are ideal.
- Keep `productId` values consistent with what the provider sends in its receipt payload. A mismatch will result in a silent empty-list return, which can be difficult to debug.
- Prefer `ensureProductSkuSchema` over `createProductSkuSchema` in automation and plugin startup code — it is idempotent and will not fail if the schema already exists.
- Use the `tags` field on `ProductBundle` to group bundles (e.g. by season or event) for efficient filtered queries without needing separate endpoints.
- When a bundle is removed or its `productBundleRewards` list is cleared, future calls to `processVerifiedPurchase` for that product will return an empty list. Existing `RewardIssuance` records are not affected.

FAQ

Q: Can a single product id be mapped to different reward sets per application? A: Yes. The compound key is `(application, schema, productId)`, so different applications can configure different rewards for the same provider product.

Q: What happens if `processVerifiedPurchase` is called and no bundle is configured? A: The method returns an empty list and logs a DEBUG message. It does not throw an exception, so the calling receipt service continues normally.

Q: Can I use `processVerifiedPurchase` from a superuser context? A: No — `SuperuserProductBundleService` delegates to the user-scoped service, which requires an authenticated user with a profile and application. Call it only from user-scoped request handlers.

Q: How are DISTINCT vs FUNGIBLE items handled? A: For `DISTINCT` items the quantity is always `1` regardless of what is set on the reward. For `FUNGIBLE` items the quantity from the reward entry is used, defaulting to `1` if null.

Q: Is there an event fired when rewards are issued? A: `processVerifiedPurchase` itself does not fire a dedicated event. Individual `RewardIssuance` records are created via `RewardIssuanceDao.getOrCreateRewardIssuance`, which may fire its own events depending on your configuration. See the Reward Issuance documentation for details.