ChatGPT

Dokumentacja systemu ERP do zarządzania magazynem

Backend – struktura bazy danych i modele

Część serwerowa systemu oparta jest o framework Laravel (PHP) z wykorzystaniem Eloquent ORM. Baza danych zawiera tabele odpowiadające głównym encjom systemu magazynowego. Poniżej przedstawiono kluczowe tabele wraz z polami (kolumnami) oraz powiązane modele ORM:

- **Produkty (`products`)** – reprezentowane przez model `Product`. Tabela zawiera m.in. kolumny: `name` (nazwa produktu), `sku` (kod SKU), `ean` (kod EAN), `pos_code` (kod POS kasowy) [1], `foreign_id` (zewn. identyfikator produktu) [2], `description` (opis, często jako JSON), `status` (status produktu, np. aktywny/archiwalny), `product_type` (typ produktu), `weight` (waga), `dimensions` (wymiary, JSON) [3], a także klucze obce `category_id`, `manufacturer_id`, `supplier_id` wskazujące na powiązane encje. W modelu `Product` zdefiniowano te pola jako *fillable* oraz odpowiednie *typowanie* (np. `weight` jako *decimal*, `description` jako *array*, flagi `manage_stock` i `variants_share_stock` jako booleany) [4]. Model korzysta z mechanizmu miękkiego usuwania (SoftDeletes) – czyli posiada kolumnę `deleted_at` [5].

- **Warianty produktów (`product_variants`)** – model `ProductVariant`. Każdy produkt może posiadać wiele wariantów (np. różne rozmiary lub kolory). Tabela wariantów zawiera kolumny: `product_id` (powiązanie z produktem), `name` (nazwa wariantu), `sku`, `ean`, `barcode` (kody wariantu), `position` (kolejność wariantu), `is_default` (flaga czy to wariant domyślny) oraz pola opcjonalne pozwalające nadpisać atrybuty odziedziczone z produktu: np. `description_override`, `weight_override`, `attributes_override` i odpowiadające im flagi `override_product_description` itp. (typu boolean) [6] [7]. Wariant ma również pole `slug` (unikalny identyfikator tekstowy) generowane automatycznie na podstawie nazwy produktu i wariantu [8] [9]. Model `ProductVariant` korzysta z SoftDeletes (kolumna `deleted_at`) [10] [11] i może mieć własne multimedia (zdjęcia) dzięki integracji ze Spatie MediaLibrary.

- **Ceny produktów (`product_prices`)** – model `ProductPrice` przechowuje ceny powiązane z wariantem produktu (np. cena detaliczna, hurtowa, zakupu, promocyjna). Pola w tabeli to m.in. `variant_id` (powiązanie z wariantem), `type` (typ ceny, np. 'retail', 'wholesale'), `price_net`, `price_gross` (wartości netto i brutto), `currency` (waluta), okres obowiązywania (`valid_from`, `valid_to`) oraz `tax_rate_id` (stawka VAT dla ceny) [12]. Każdy rekord cen ma przypisany wariant (relacja `belongsTo` w modelu) [13] oraz opcjonalnie stawkę VAT (model `TaxRate`).

- **Kategorie produktów (`categories`)** – model `Category`. Struktura kategorii jest hierarchiczna (kategoria może mieć podkategorie). Tabela `categories` zawiera kolumny: `name` (nazwa kategorii), `slug` (unikalny identyfikator tekstowy generowany ze slugowaniem nazwy), `parent_id` (opcjonalne ID kategorii nadrzędnej) oraz `baselinker_category_id` (opcjonalne powiązanie kategorii z systemem zewnętrznym BaseLinker) [14]. Model kategorii

wykorzystuje SoftDeletes oraz definiuje relacje: `children` (HasMany – podkategorie), `parent` (BelongsTo – kategoria nadrzędna) i `products` (HasMany – produkty przypisane do danej kategorii) [15] [16]. Przykładowo, próba usunięcia kategorii, która ma przypisane podkategorie lub produkty, jest blokowana w logice aplikacji [17].

- **Producenci (`manufacturers`)** – model `Manufacturer` zawiera np. kolumnę `name` (nazwa producenta) [18]. Każdy producent może mieć wiele produktów (relacja `HasMany` w modelu) [19].

- **Dostawcy (`suppliers`)** – model `Supplier`. Tabela dostawców zawiera informacje takie jak `name` (nazwa dostawcy/firma), `tax_id` (np. NIP), `email`, `phone` (telefon), `address` oraz `notes` (notatki) [20]. Dostawca może być powiązany z wieloma produktami (jeśli jest domyślnym dostawcą produktu) oraz dokumentami zakupowymi i kosztami. W modelu zdefiniowano relacje: `products` (HasMany produktów danego dostawcy), `documents` (HasMany – dokumenty magazynowe gdzie dostawca jest przypisany, np. PZ) oraz `expenses` (HasMany – koszty/faktury powiązane z dostawcą) [21] [22]. Dostawcy, podobnie jak producenci, wspierają SoftDeletes [23].

- **Magazyny (`warehouses`)** – model `Warehouse`. System obsługuje wiele magazynów fizycznych. Tabela `warehouses` zawiera m.in. `name` (nazwa magazynu), `symbol` (skrót/oznaczenie), `address` (adres), flagę `is_default` (czy to magazyn domyślny) oraz `baselinker_storage_id` (ID magazynu w systemie BaseLinker, jeśli integracja) [24]. Model posiada relacje: `stockLevels` (HasMany – stany magazynowe wszystkich produktów w tym magazynie), `sourceDocuments` i `targetDocuments` (HasMany dokumentów, gdzie dany magazyn jest odpowiednio magazynem źródłowym lub docelowym w dokumencie) [25] [26].

- **Stany magazynowe (`stock_levels`)** – model `StockLevel` reprezentuje aktualny stan danego wariantu produktu w konkretnym magazynie. Kolumny w tabeli to: `product_variant_id`, `warehouse_id`, `quantity` (ilość fizyczna na stanie), `reserved_quantity` (ilość zarezerwowana, np. pod zamówienia) oraz `incoming_quantity` (ilość w dostawie – oczekiwana) [27]. Są też dodatkowe informacje: `location` (lokalizacja/skład w magazynie) i `last_stocktake_date` (data ostatniej inwentaryzacji) [28]. Model `StockLevel` posiada relacje do `ProductVariant` oraz `Warehouse` (oba BelongsTo) [29]. Ważną częścią tego modelu są *akcesory* (właściwości wyliczane) – np. `available_quantity` (ilość dostępna = quantity - reserved_quantity) i `expected_quantity` (ilość spodziewana = quantity + incoming_quantity) [30] [31]. Stan magazynowy jest kluczowy dla operacji przyjęć i wydań – zmiany są dokonywane za pomocą metod statycznych modelu, opisanych w dalszej części.

- **Partie dostaw (FIFO) (`stock_batches`)** – model `StockBatch` śledzi partie towaru przyjęte do magazynu, co umożliwia wydawanie metodą FIFO (pierwsze przyszło, pierwsze wyszło). Każda partia ma kolumny: `product_variant_id`, `warehouse_id`, `quantity_total` (ilość przyjęta w partii), `quantity_available` (ilość pozostała niewydana), `purchase_price` (cena zakupu), `purchase_date` (data zakupu/dostawy) oraz pola `source_document_type` i `source_document_id` do opcjonalnego powiązania partii z dokumentem źródłowym (np. numer PZ) [32] [33]. Partie są wykorzystywane we frontowym mechanizmie FIFO – np. przy przyjęciu dostawy tworzy się nowy `StockBatch` i dodaje do magazynu [34].

- **Dokumenty magazynowe (`documents`)** – centralna encja reprezentująca operacje na magazynie (przychody, rozchody itp.). Tabela `documents` zawiera: `number` (numer

dokumentu, generowany wg schematu), `type` (typ dokumentu – np. PZ, WZ, MM, RW, PW, ZW, ZRW, INW, FS, FVZ itp.) [35], `document_date` (data dokumentu), `user_id` (użytkownik wystawiający), `supplier_id` (dostawca – dla przychodów zewnętrznych PZ lub zwrotów do dostawcy), `customer_id` (klient – pole przygotowane pod przyszłą integrację, obecnie może być NULL) [36], `source_warehouse_id`, `target_warehouse_id` (magazyn źródłowy i docelowy – zależnie od typu dokumentu) [37], `related_document_id` (opcjonalne powiązanie do innego dokumentu, np. dokument finansowy powiązany z magazynowym) [38], `total_net`, `total_gross` (łączna wartość netto/brutto dokumentu) oraz `notes` (dodatkowe uwagi) [39]. Dokumenty również mają mechanizm SoftDeletes (kolumna `deleted_at`) [40]. Model `Document` definiuje liczne relacje: `items` (HasMany pozycji dokumentu), `inventoryItems` (HasMany pozycji inwentaryzacji, jeśli dokument to INW), `user` (wystawiający użytkownik), `supplier`, `sourceWarehouse`, `targetWarehouse`, a także `parentDocument` oraz `childDocument` dla powiązań między dokumentami (np. WZ → FS) [41] [42]. Dzięki tym relacjom można łatwo pobrać komplet danych dokumentu – np. dokument WZ zna powiązaną fakturę sprzedaży (FS) przez `childDocument`.

- **Pozycje dokumentu (`document_items`)** – model `DocumentItem` zawiera szczegóły pozycji dokumentu magazynowego: `document_id` (referencja do dokumentu), `product_variant_id` (wariant produktu), `quantity` (ilość), `price_net`, `price_gross` oraz `tax_rate` (stawka podatku VAT) [43]. Każda pozycja ma relacje `document` (należy do dokumentu) i `productVariant` (należy do wariantu produktu) [44]. Ilości i ceny są przechowywane jako wartości dziesiętne z dokładnością do 2 miejsc (decimal(10,2) w bazie) [45].

- **Pozycje inwentaryzacji (`inventory_items`)** – używane tylko dla dokumentów typu INW (inwentaryzacja). Model `InventoryItem` zawiera: `document_id` (ID dokumentu INW), `product_variant_id`, `expected_quantity` (ilość wg stanu ewidencyjnego przed inwentaryzacją), `counted_quantity` (ilość policzona faktycznie) oraz wyliczane pole `difference` (różnica, obliczana jako counted - expected) [46]. Różnica jest kolumną *generated* w bazie danych (stored as expression) i nie jest ręcznie ustawiana w kodzie [47]. Pozycje inwentaryzacji nie mają znaczników czasowych (timestamps wyłączone) [48] i są powiązane z dokumentem oraz wariantem produktu (relacje BelongsTo) [49].

- **Zamówienia sprzedaży (`orders`)** – model `Order` reprezentuje zamówienia klientów, zaciągane np. z zewnętrznych kanałów poprzez integrację (np. BaseLinker). Tabela `orders` ma kolumny: `baselinker_order_id` (ID zamówienia z BaseLinkera, unikalne) [50], `external_order_id` (ew. inny zewnętrzny identyfikator zamówienia) [51], `order_source` (źródło zamówienia, np. kanał sprzedaży lub sklep), `order_date` (data zamówienia), `customer_details` (dane klienta, przechowywane jako JSON) [52], `status` (status zamówienia, np. "pending", "shipped" itp.), `total_gross` (łączna wartość brutto zamówienia) [53], `sync_status` (status synchronizacji, np. pending/done) oraz `last_synced_at` (znacznik czasu ostatniej synchronizacji) [54]. Dodatkowo `related_wz_id` (jeśli do zamówienia wystawiono dokument WZ, to tutaj jest referencja do niego) [55]. Model Order korzysta z SoftDeletes [56]. Relacje: `items` (HasMany – pozycje zamówienia) oraz `document` (BelongsTo – powiązany dokument wydania WZ, jeśli istnieje) [57] [58].

- **Pozycje zamówienia (`order_items`)** – model `OrderItem` zawiera pozycje sprzedażowe powiązane z zamówieniem. Tabela ma kolumny: `order_id` (ID zamówienia), `product_variant_id` (powiązany wariant produktu lub NULL, jeśli np. produkt nie jest zmapowany), `sku`, `name` (nazwa pozycji), `quantity` (zamówiona ilość) oraz `price_gross`

(cena brutto jednostkowa) [59]. Pozycje zamówienia mają relacje `order` (należą do zamówienia) i `productVariant` (należą do wariantu produktu, jeśli przypisano) [60].

- **Stawki VAT (`tax_rates`)** – model `TaxRate` prawdopodobnie przechowuje dostępne stawki podatku (np. 23%, 8% itp.), używane przy produktach i cenach. Tabela zawiera np. `name` (opis stawki) i `rate` (wartość procentowa). Choć nie przytoczono kodu migracji, w modelu `ProductPrice` widać klucz `tax_rate_id` oraz relację `taxRate` do modelu TaxRate [61] [13].

- **Tagi produktów (`tags` i pivot `product_tag`)** – system pozwala tagować produkty. Model `Tag` (tabela `tags`) zawiera co najmniej nazwę tagu `name` (unikalną) [62], a w kodzie modelu przewidziano również pole `slug` dla tagu [63] (co sugeruje możliwość generowania slugów tagów, choć migracja bazowa go nie zawiera). Relacja wiele-do-wielu z produktami jest realizowana przez tabelę pośredniczącą `product_tag` z kolumnami `product_id` i `tag_id` (oba klucze obce, unikatowa para) [64]. W modelu `Product` relacja `tags` jest zdefiniowana przez `belongsToMany(Tag::class, 'product_tag')` [65], a w modelu `Tag` przez `belongsToMany(Product::class)` zgodnie z konwencją Laravel [66].

Podsumowując, **struktura bazy danych** jest typowa dla systemu ERP: posiada encje produktów wraz z wariantami i cenami, kategorie, kontrahentów (dostawcy, producenci), magazyny i stany, dokumenty magazynowe i powiązane pozycje, a także obsługę zamówień sprzedaży i kosztów. Wszystkie modele ORM mają zdefiniowane *fillable fields* zgodnie z kolumnami tabel – co zapewnia spójność nazewnictwa między kodem a bazą. Większość encji wspiera mechanizm miękkiego usuwania (SoftDeletes), co oznacza, że rekordy nie są fizycznie kasowane z bazy, a jedynie oznaczane (pole `deleted_at`) [40].

Backend – relacje między modelami

Relacje między modelami odzwierciedlają powiązania logiczne w systemie magazynowym (większość została już wspomniana przy opisach modeli). Poniżej zestawiono najważniejsze relacje:

- **Relacje produkt–warianty**: Model `Product` ma wiele wariantów (`hasMany` do `ProductVariant`) – każdy produkt może posiadać kolekcję wariantów, sortowanych według pola `position` [67]. Jeden z wariantów może być oznaczony jako domyślny (`is_default`), a model `Product` eksponuje relację `defaultVariant` (hasOne filtrujące wariant o is_default=true) [68]. Wariant posiada relację odwrotną `product` (należy do produktu) [69].

- **Relacje produkt–inne encje**: Produkt należy do kategorii (`category` – *belongsTo Category*) [70], do producenta (`manufacturer` – *belongsTo Manufacturer*) [71] oraz do domyślnego dostawcy (`supplier` – *belongsTo Supplier*) [72]. Tym samym kategoria ma wiele produktów (relacja `hasMany` w modelu Category) [73], producent ma wiele produktów [19], a dostawca wiele produktów [21].

- **Relacje produkt–tagi**: Produkty i tagi są połączone relacją wiele-do-wielu. W modelu `Product` zdefiniowano `tags(): BelongsToMany` odwołujące się do tabeli pivot `product_tag` [65]. W modelu `Tag` analogicznie `products(): BelongsToMany` [74]. Dzięki temu produkt może mieć dowolną liczbę tagów, a tag może być przypisany do wielu produktów.

- **Relacje produkt–powiązania (linki i zestawy)**: Model `Product` posiada dodatkowe relacje: `links` (hasMany `ProductLink`) oraz `bundleItems` (hasMany `ProductBundleItem`) [75]. `ProductLink` prawdopodobnie służy do powiązań między produktami (np. linkowanie

produktu z innym w kontekście sklepu, akcesoriów itp.), a `ProductBundleItem` to relacja produkt–komponenty zestawu (bundle/kit). W relacji bundle, `bundleItems` w modelu Product wskazuje komponenty, a każde `ProductBundleItem` odnosi się do wariantu składowego (`componentVariant`). W `DEFAULT_PRODUCT_LOAD` z modelu Product widać, że przy pobieraniu produktu ładowane są m.in. `bundleItems` wraz z `componentVariant` i jego cenami, stanami i mediami [76] – co potwierdza istnienie relacji zestawów produktowych.

- **Relacje wariant–ceny i stany**: Wariant produktu ma wiele cen (`prices`: hasMany do `ProductPrice`) [77] oraz wiele rekordów stanu w magazynach (`stockLevels`: hasMany do `StockLevel`) [78]. Dodatkowo, wariant ma wiele partii magazynowych (`stockBatches`: hasMany do `StockBatch`) [79] – co służy do FIFO. Relacje odwrotne: `ProductPrice` wskazuje na wariant (`belongsTo variant`) [80], `StockLevel` wskazuje na wariant [29], podobnie `StockBatch` ma `belongsTo productVariant` [81].

- **Relacje magazyn–stany i dokumenty**: Magazyn posiada wiele rekordów stanu (`stockLevels` – każdy wariant ma swój stan w magazynie) [82] oraz może mieć wiele dokumentów jako magazyn źródłowy (`sourceDocuments`) lub docelowy (`targetDocuments`) [25] [83]. Dzięki temu łatwo sprawdzić np. wszystkie dokumenty wydania (WZ, RW, ZRW) z danego magazynu (sourceDocuments) lub przyjęcia (PZ, PW, ZW) do danego magazynu (targetDocuments).

- **Relacje dokument–składniki**: Dokument magazynowy ma listę pozycji `items` (pojedyncze pozycje towarowe z ilościami i cenami) [84]. Każda pozycja (DocumentItem) zna swój dokument i wariant produktu [44]. Dla inwentaryzacji, dokument INW ma relację `inventoryItems` (pozycje inwentaryzacji) [85], gdzie każda pozycja zawiera oczekiwany i policzony stan wariantu [86].

- **Relacje dokument–powiązane dokumenty**: Jeśli dokument ma dokument powiązany (np. faktura sprzedaży FS powiązana z wydaniem WZ, lub odwrotnie dokument magazynowy powiązany z finansowym), to `parentDocument` (BelongsTo) i `childDocument` (HasOne) łączą te rekordy [42]. W praktyce, utworzenie dokumentu finansowego FS/FVZ dla danego WZ/PZ jest realizowane przez utworzenie nowego dokumentu z `related_document_id` wskazującym na dokument magazynowy [87] [88]. Relacja parent/child pozwala nawigować między nimi.

- **Relacje dokument–dostawca/użytkownik**: Dokument może mieć przypisanego dostawcę (`supplier`: BelongsTo Supplier) [89] albo klienta (`customer_id`, choć klient nie jest jeszcze w pełni zaimplementowany jako encja – traktowany jako placeholder). Dokument należy także do użytkownika (`user`: BelongsTo User) który go wystawił [90].

- **Relacje dostawca–dokumenty/koszty**: Dostawca ma wspomniane relacje do dokumentów (np. PZ, ZRW) i kosztów. W modelu `Supplier`, metoda `documents()` zwraca wszystkie dokumenty, gdzie `supplier_id` to ID danego dostawcy [91]. Natomiast metoda `expenses()` zwraca listę kosztów (faktur zakupu) powiązanych z dostawcą [92]. Model `Expense` z kolei ma `supplier` (BelongsTo) i `category` (BelongsTo ExpenseCategory) – wskazuje to, że koszt jest przypisany do dostawcy i do kategorii kosztów (np. „Transport”, „Materiały” itp.) [93].

- **Relacje zamówienie–pozycje**: Zamówienie sprzedaży (`Order`) ma wiele pozycji (`items`: hasMany OrderItem) [94]. Każda pozycja zamówienia zna swoje zamówienie (`order`: BelongsTo) i opcjonalnie wariant produktu (`productVariant`: BelongsTo) [60]. Dodatkowo zamówienie może mieć powiązany dokument WZ (`document`: BelongsTo Document) jeśli zostało zrealizowane wydanie towaru do tego zamówienia [95].

Ogólnie, model danych jest silnie powiązany relacjami, co pozwala w zapytaniach Eloquent łatwo załadować potrzebne powiązane informacje. W wielu kontrolerach i serwisach widać stosowanie eager loadingu (`with(...)`) celem pobrania powiązań – np. lista dokumentów ładuje od razu magazyny, dostawcę, użytkownika i pozycje [96]. Takie podejście zapewnia kompletny kontekst biznesowy obiektów.

Backend – pełny opis endpointów API

Aplikacja udostępnia bogaty zestaw endpointów API (RESTful) zgrupowanych w wersjonowanej przestrzeni URL `/api/v1`. Backend zakłada wykorzystanie tokenowej autoryzacji (Laravel Sanctum) – stąd większość tras API jest chroniona przez middleware `auth:sanctum` [97]. Poniżej wyszczególniono dostępne endpointy i ich funkcjonalności:

- **Uwierzytelnianie** (`/api/auth/...`):
- `POST /api/auth/register` – rejestracja nowego użytkownika na podstawie danych (imię, email, hasło). Tworzy konto użytkownika z domyślną rolą "user" [98] i zwraca JSON z danymi użytkownika oraz tokenem dostępu (Bearer token) [99].
- `POST /api/auth/login` – logowanie istniejącego użytkownika (email + hasło). W przypadku poprawnych danych uwierzytelniających, generowany jest nowy token dostępowy [100] i zwracane są informacje o użytkowniku (w tym rola i uprawnienia) oraz token [101].
- `POST /api/auth/logout` – (chroniony) wylogowanie użytkownika, które polega na usunięciu/b unieważnieniu tokenów powiązanych z użytkownikiem [102]. Zwraca komunikat o pomyślnym wylogowaniu.

- `GET /api/auth/user` – (chroniony) pobranie aktualnych danych zalogowanego użytkownika (profil, rola, uprawnienia). Używane np. do odświeżenia stanu aplikacji po odświeżeniu strony, zwraca te same pola co podczas logowania [103].

- **Zasoby główne API (CRUD)** – zdefiniowane przy pomocy `Route::apiResource`, co oznacza standardowe metody: `index` (lista z opcjonalnym filtrowaniem), `show` (szczegóły pojedynczego), `store` (dodanie), `update` (modyfikacja) i `destroy` (usuniecie). Dostępne zasoby:

- `products` – Produkty. Endpointy:

  - `GET /api/v1/products` – lista produktów (obsługuje filtrowanie po nazwie, SKU, kategorii itp. – logika filtrowania jest zaimplementowana w kontrolerze `ProductController`, np. filtr statusu, producenta, dostawcy, zakresów sprzedaży itd., zgodnie z TODO list) [104]. Domyślnie stronicowana lista z relacjami (warianty, ceny, media).
  - `POST /api/v1/products` – dodanie nowego produktu (walidowane przez `StoreProductRequest` – np. unikalność SKU itd.). Tworzy produkt oraz jego warianty domyślne na podstawie danych.
  - `GET /api/v1/products/{id}` – pobranie szczegółów produktu o danym ID (wraz z załadowanymi wariantami, cenami, stanami, tagami, mediami dzięki `DEFAULT_PRODUCT_LOAD` w modelu) [105].
  - `PUT/PATCH /api/v1/products/{id}` – aktualizacja produktu (oraz ewentualnie powiązanych encji jak warianty – kontroler `ProductController` obsługuje logikę modyfikacji wariantów, stanów itp. w metodzie update, jak wskazano w TODO) [104].

- `DELETE /api/v1/products/{id}` – usunięcie produktu (w praktyce SoftDelete – oznaczenie jako usunięty, ponieważ model używa SoftDeletes). Powiązane warianty również mogą być miękko usunięte (zadbanie o spójność przy usuwaniu jest planowane).

- `categories` – Kategorie produktów. Endpointy CRUD analogiczne:
  - `GET /api/v1/categories` – lista kategorii (z obsługą filtrowania po nazwie, statusie, rodzicu) [106] [107], domyślnie stronicowana. Każda kategoria może być zwrócona z rodzicem (eager load `parent`) [106].
  - `POST /api/v1/categories` – utworzenie nowej kategorii (walidacja w `StoreCategoryRequest`). Po utworzeniu, zwraca kategorię z dołączonym rodzicem (jeśli istnieje) [108].
  - `GET /api/v1/categories/{id}` – szczegóły kategorii (z relacjami `parent` i `children`) [109].
  - `PUT /api/v1/categories/{id}` – aktualizacja kategorii (walidacja w `UpdateCategoryRequest`), zwraca zaktualizowaną kategorię z relacjami [110].
  - `DELETE /api/v1/categories/{id}` – usunięcie kategorii. Implementacja zapobiega usunięciu, jeśli kategoria ma podkategorie lub przypisane produkty – w takim wypadku zwraca błąd 422 z komunikatem [17]. Jeśli brak powiązań, wykonuje SoftDelete kategorii [111].

- `manufacturers` – Producenci:
  - `GET /api/v1/manufacturers` – lista producentów.
  - `POST /api/v1/manufacturers` – dodanie producenta.
  - `GET /api/v1/manufacturers/{id}` – szczegóły producenta.
  - `PUT /api/v1/manufacturers/{id}` – modyfikacja.
  - `DELETE /api/v1/manufacturers/{id}` – usunięcie. (TODO wspomina o dodaniu sprawdzenia, czy nie ma produktów danego producenta przed usunięciem) [112].

- `suppliers` – Dostawcy: analogiczny zestaw endpointów CRUD do zarządzania dostawcami (lista, dodaj, podgląd, edycja, usuń). Przy usuwaniu dostawcy planowane jest sprawdzanie powiązanych produktów/dokumentów (aby nie usuwać aktywnego kontrahenta – wzmianka w TODO) [113].

- `tax-rates` – Stawki VAT: standardowe endpointy CRUD dla stawek podatku (lista dostępnych stawek, dodawanie nowej, edycja, usunięcie – z walidacją by nie usuwać stawki używanej przez produkty, co też jest w planach TODO) [114].

- `warehouses` – Magazyny: umożliwiają zarządzanie listą magazynów (nazwy, adresy, itp.):
  - `GET /api/v1/warehouses` – lista magazynów.
  - `POST /api/v1/warehouses` – utworzenie magazynu.
  - `PUT /api/v1/warehouses/{id}` – edycja magazynu.
  - `DELETE /api/v1/warehouses/{id}` – usunięcie magazynu (zablokowane, jeśli magazyn jest w użyciu lub jest domyślny – do zaimplementowania wg listy TODO) [115].

- `users` – Użytkownicy: zarządzanie użytkownikami systemu (dostępne tylko dla administratora). Pozwala na listowanie użytkowników, tworzenie nowych, edycję (np. nadawanie ról) i usuwanie. W kontrolerze `UserController` planowane jest uniemożliwienie pewnych akcji, np. by użytkownik nie mógł usunąć sam siebie i by tylko uprawniony admin zmieniał role [115].

- `orders` – Zamówienia: endpointy do podglądu i zarządzania zamówieniami sprzedaży (pobierane z integracji lub dodawane ręcznie):
  - `GET /api/v1/orders` – lista zamówień (możliwe filtrowanie po statusie, źródle itp. – w razie implementacji).
  - `GET /api/v1/orders/{id}` – szczegóły zamówienia wraz z pozycjami (eager load pozycji).
  - Tworzenie/aktualizacja zamówień raczej będzie wykonywane automatycznie przez integrację (patrz niżej BaselinekrService), ale API technicznie obsługuje `POST` i `PUT /api/v1/orders` (np. do ręcznej rejestracji zamówienia telefonicznego).
  - `DELETE /api/v1/orders/{id}` – ewentualne anulowanie/usunięcie zamówienia (raczej SoftDelete).
- `documents` – Dokumenty magazynowe: jest to ważny zasób, skupiający różne operacje magazynowe pod jednym modelem:
  - `GET /api/v1/documents` – lista dokumentów magazynowych. Kontroler `DocumentController@index` zaimplementowano ze wsparciem **zaawansowanego filtrowania** według wielu kryteriów: numeru, ID, treści notatki, dostawcy, magazynu (sprawdza zarówno źródłowy jak i docelowy), typu dokumentu (jednego lub wielu jednocześnie), statusu otwarty/zamknięty (jeśli status jest obsługiwany) oraz zakresów kwot netto/brutto [116] [117]. Dzięki temu można w UI tworzyć rozbudowane filtry listy dokumentów. Dane są stronicowane i zawierają powiązania (magazyny, dostawca, wystawca, pozycje itp.) [96].
  - `GET /api/v1/documents/{id}` – szczegóły pojedynczego dokumentu. Zwraca nagłówek dokumentu oraz powiązane pozycje (`items`) lub pozycje inwentaryzacji (`inventoryItems`) w zależności od typu, a także ewentualny powiązany dokument finansowy (childDocument).
  - `POST /api/v1/documents` – utworzenie nowego dokumentu. **Uwaga:** Tworzenie dokumentów odbywa się typowo poprzez różne metody serwisu `DocumentService` (o czym w następnym rozdziale), ponieważ sama struktura dokumentu różni się zależnie od typu. Prawdopodobnie kontroler rozróżnia typ dokumentu w żądaniu (np. `type`: "PZ", "WZ", itp.) i wywołuje odpowiednią metodę serwisową. Walidacja wejścia jest realizowana przez `StoreDocumentRequest` – sprawdzając obecność wymaganych pól w zależności od typu (np. dla PZ wymagany `supplier_id`, dla WZ `source_warehouse_id`, itp.) [118] [119]. Po pomyślnym utworzeniu dokumentu wraz z pozycjami, zwracany jest zasób dokumentu (zapewne używając `DocumentResource`).
  - `PUT /api/v1/documents/{id}` – aktualizacja dokumentu. Pozwala np. edytować notatki lub korekty pozycji przed zamknięciem. W TODO jest wzmianka o zaimplementowaniu bezpiecznego usuwania dokumentu (anulowania) poprzez np. metodę `cancelDocument` w serwisie DocumentService, co sugeruje że aktualizacja dokumentu mogłaby przyjmować akcje typu zamknięcie/anulowanie.
  - `DELETE /api/v1/documents/{id}` – usunięcie dokumentu. Obecnie (wg TODO) działa proste usunięcie, ale planowane jest wdrożenie logiki cofającej operacje magazynowe w przypadku usunięcia (np. anulowanie PZ powinno zdjąć dodane stany, anulacja WZ powinna zwrócić stany, itp.) [120].

- **Zasoby zagnieżdżone**:

- `products/{product}/variants` – zagnieżdżone endpointy wariantów produktu. Ponieważ każdy wariant ma zawsze kontekst produktu nadrzędnego, API zdefiniowano jako zagnieżdżone. Działa to tak, że:
  - `GET /api/v1/products/{productId}/variants` – lista wariantów danego produktu.
  - `POST /api/v1/products/{productId}/variants` – dodanie nowego wariantu do produktu.
  - `GET /api/v1/variants/{variantId}` – (shallow route) pobranie konkretnego wariantu.
  - `PUT /api/v1/variants/{variantId}` – modyfikacja wariantu.
  - `DELETE /api/v1/variants/{variantId}` – usunięcie wariantu (SoftDelete).
  - W kontrolerze `ProductVariantController` znajduje się logika obsługi tych operacji – np. podczas tworzenia nowego wariantu może być generowany automatycznie slug czy weryfikowana unikalność SKU w ramach produktu.

- `products/{product}/bundle-items` – endpointy do zarządzania składnikami zestawu produktowego. Z ograniczeniem do operacji `store`, `update`, `destroy` (wg definicji `->only(['store','update','destroy'])` w trasach) [121]. Czyli można dodać komponent do produktu (tworząc `ProductBundleItem`), zaktualizować go (np. ilość komponentu w zestawie) lub usunąć ze zestawu. Relacja jest shallow, więc np. `DELETE /api/v1/bundle-items/{id}` usuwa dany element zestawu.

• **Obsługa plików (Media)**:

- `POST /api/v1/media/upload` – upload pliku (np. zdjęcia produktu). Wykorzystuje kontroler `MediaController@store` [122], zapewne integrujący się ze Spatie MediaLibrary, aby dołączyć plik do modelu (np. Produkt lub Wariant – w żądaniu może być wskazane do czego przypiąć media).
- `DELETE /api/v1/media/{mediaId}` – usunięcie pliku (MediaController@destroy) [122] – usuwa wskazane medium (zdjęcie).

- `POST /api/v1/media/reorder` – zmiana kolejności zdjęć (MediaController@reorder) [122] – zapewne przyjmuje dane zmiany kolejności i aktualizuje atrybut `order` mediów dla np. galerii produktu.

• **FIFO (partie magazynowe)**:

- `POST /api/v1/stock/fifo/in` – przyjęcie towaru do magazynu wg FIFO [123]. W kontrolerze `StockFifoController@storeStockIn` metoda ta tworzy nową partię (`StockBatch::create(...)`) z podaną ilością i ceną zakupu oraz wywołuje `StockLevel::change` dodając ilość do ogólnego stanu magazynowego [124] [34]. Zwraca kod 201 z komunikatem o przyjęciu towaru [125] [126].

- `POST /api/v1/stock/fifo/out` – rozchodowanie (wydanie) towaru metodą FIFO [123]. `StockFifoController@issueStockOut` pobiera żądanie z wariantem, magazynem i ilością do wydania. Najpierw sprawdza dostępność (czy ilość dostępna >= żądana) – jeśli nie, zwraca błąd 422 "Brak wystarczającej ilości towaru" [127] [128]. Następnie pobiera listę partii (`StockBatch`) dla danego wariantu w magazynie, sortowanych rosnąco po dacie zakupu (i ID) [129]. Iteruje przez partie, zmniejszając ich `quantity_available` zgodnie z wydawaną ilością aż wyczerpie żądaną ilość [130] [131]. Na koniec aktualizuje agregat stanu przez

`StockLevel::change(variant, warehouse, -quantity)` i zwraca sukces z kodem 200 i komunikatem "Towar wydany zgodnie z FIFO" [132].

- **Dashboard**:

- `GET /api/v1/dashboard/document-stats` – zwraca statystyki dokumentów do wyświetlenia na pulpicie (np. ile dokumentów danego typu w bieżącym miesiącu, itp.). Implementacja w `DashboardController@documentStats` prawdopodobnie zbiera liczby PZ/WZ/RW itp. z ostatnich okresów.

- **Select options (listy wyboru)**: Endpointy pomocnicze udostępnione pod `/api/v1/select-options/...` generowane przez `SelectOptionsController`. Służą one do wypełniania list rozwijanych i komponentów autouzupełniających w interfejsie. W ramach tej grupy mamy:

- `GET /api/v1/select-options/categories` – lista kategorii (np. id i nazwa) do wyboru w formularzach.
- `GET /api/v1/select-options/manufacturers` – lista producentów.
- `GET /api/v1/select-options/suppliers` – lista dostawców.
- `GET /api/v1/select-options/warehouses` – lista magazynów.
- `GET /api/v1/select-options/tax-rates` – lista stawek VAT.
- `GET /api/v1/select-options/users` – lista użytkowników (np. do przypisania dokumentu do użytkownika).
- `GET /api/v1/select-options/document-types` – lista typów dokumentów magazynowych (enum DocumentType) dostępnych w systemie.
- `GET /api/v1/select-options/product-variants` – lista wariantów produktów (np. do wyszukiwania po SKU podczas tworzenia dokumentu).
- `GET /api/v1/select-options/tags` – lista tagów produktów.
- `GET /api/v1/select-options/products` – lista produktów.
- `GET /api/v1/select-options/document-mappings` – specjalny endpoint wykorzystywany przez listę dokumentów; prawdopodobnie zwraca mapowanie typów dokumentów na nazwy lub powiązania typów finansowych z magazynowymi (np. że WZ -> FS, PZ -> FVZ). Służy to UI do wyświetlania czy dany dokument ma już powiązaną fakturę.

Wszystkie powyższe trasy (za wyjątkiem logowania/rejestracji) są zabezpieczone i wymagają poprawnego tokenu uwierzytelniającego (nagłówek `Authorization: Bearer <token>`). Jeśli token jest nieprawidłowy lub wygasły, serwer zwróci błąd 401.

API zostało zaprojektowane tak, by front-end (opisany dalej) mógł z niego korzystać do pełnego zarządzania systemem magazynowym. Struktura odpowiedzi zazwyczaj opiera się na formacie JSON Resources Laravel – np. `CategoryResource`, `DocumentResource` formatują dane modeli przed wysłaniem. To oznacza, że klient otrzymuje dane z polami dokładnie tak nazwanymi jak kolumny w bazie (co jest ważne dla wykorzystania zewnętrznego, np. przez modele AI lub integracje).

Backend – główne operacje w systemie magazynowym

Do głównych operacji magazynowych zaliczamy przyjęcia towarów, wydania towarów, przesunięcia między magazynami, inwentaryzacje oraz działania z tym związane (np. zwroty, korekty). W systemie zostały one odwzorowane za pomocą różnych **typów dokumentów magazynowych** (pole `type` w tabeli `documents`). Obsługę logiki biznesowej tych dokumentów skupiono w serwisie

`DocumentService` – jest to klasa, która zawiera metody tworzące poszczególne rodzaje dokumentów i wykonujące odpowiednie akcje (np. zmiany stanów magazynowych):

- **Przyjęcie zewnętrzne (PZ)** – dokument PZ oznacza przyjęcie towaru z zewnątrz (np. od dostawcy) do magazynu. Metoda `DocumentService::createPz($data)` tworzy taki dokument [118]. Wymagane jest podanie `supplier_id` (dostawcy) i `target_warehouse_id` (magazynu docelowego) [133]. Implementacja:
- Oblicza sumy wartości pozycji (`totalNet`, `totalGross`) na podstawie przekazanych pozycji towarowych [134].
- Tworzy dokument bazowy w tabeli `documents` za pomocą metody `createBaseDocument`, ustawiając typ na PZ, podaną datę, użytkownika, dostawcę, magazyn docelowy i wyliczone sumy [135].
- Dla każdej pozycji z `$data['items']` pobiera odpowiedni wariant produktu i wywołuje `StockLevel::change($variant, $warehouse, +quantity)`, aby zwiększyć fizyczny stan magazynowy o przyjmowaną ilość [136].

- Zwraca utworzony dokument PZ. W efekcie w systemie pojawia się nowy dokument przychodu, a stan produktów w wskazanym magazynie rośnie o zadane wartości.

- **Wydanie zewnętrzne (WZ)** – dokument WZ oznacza wydanie towaru na zewnątrz (np. sprzedaż klientowi). Tworzy go `DocumentService::createWz($data)` [137]. Wymagany jest `source_warehouse_id` (magazyn, z którego wydajemy) [138]. Działanie analogiczne do PZ, z tą różnicą, że dla każdej pozycji wywoływane jest `StockLevel::change($variant, $warehouse, -quantity)` – czyli zmniejszenie stanu w magazynie źródłowym o wskazaną ilość [139]. Dokument WZ ma typ `DocumentType::WZ` i może być powiązany później z fakturą sprzedaży (FS).

- **Przesunięcie międzymagazynowe (MM)** – dokument MM służy do przemieszczenia towaru z jednego magazynu do innego (wewnątrz firmy). Tworzy go `createMm($data)` [140], wymagając `source_warehouse_id` i `target_warehouse_id` [141]. Implementacja:

- Tworzy dokument typu MM (magazyn źródłowy, docelowy, użytkownik, data – suma wartości netto/brutto zwykle 0, bo to przesunięcie wewnętrzne) [142].
- Dla każdej pozycji: zmniejsza stan w magazynie źródłowym o daną ilość i zwiększa stan w magazynie docelowym o tę ilość (dwa wywołania `StockLevel::change`) [143].

- Zwraca dokument. W praktyce jeden dokument MM powoduje dwa ruchy magazynowe: rozchód z magazynu A i przychód do magazynu B.

- **Rozchód wewnętrzny (RW)** – dokument RW oznacza wydanie wewnętrzne, np. zużycie towarów na potrzeby firmy (utylizacja, produkcja wewnętrzna). `createRw($data)` tworzy dokument typu RW, bardzo podobnie do WZ, ale zwykle używany w kontekście wewnętrznym [144]. Wymaga `source_warehouse_id` [145] i dla pozycji wywołuje `StockLevel::change` odejmujące stany [146].

- **Przyjęcie wewnętrzne (PW)** – dokument PW to przyjęcie wewnętrzne, np. przyjęcie wyrobu gotowego z produkcji na magazyn. Tworzony przez `createPw($data)` [147], wymagane `target_warehouse_id` [148]. Dla pozycji dodaje stany w magazynie docelowym (podobnie jak PZ) [149]. Zwiększa stan magazynu wewnętrznie, bez dostawcy.

- **Zwrot od klienta (ZW)** – dokument ZW rejestruje zwrot towaru od klienta. Tworzony przez `createZw($data)` [150], podobny do PW/PZ (przyjmujemy towar do magazynu docelowego, np. zwrócony przez klienta, więc wymagane `target_warehouse_id`) [151]. Każda pozycja zwiększa stan magazynowy (`StockLevel::change` dodatni) [152].

- **Zwrot do dostawcy (ZRW)** – dokument ZRW to zwrot do dostawcy (odesłanie towaru). Tworzony przez `createZrw($data)` [153], wymagany `source_warehouse_id` (skąd zwracamy) i `supplier_id` (do kogo zwracamy) [154]. Dla pozycji zmniejsza stan w magazynie źródłowym (podobnie jak WZ/RW) [155]. Ten dokument można traktować jako korektę do PZ (odesłanie wadliwego towaru dostawcy itp.).

- **Dokumenty finansowe (FS, FVZ)** – system przewiduje tworzenie dokumentów finansowych powiązanych z magazynowymi:

- FS (Faktura Sprzedaży) powiązana z wydaniem WZ.

- FVZ (Faktura Zakupu) powiązana z przyjęciem PZ. Tworzenie takiego dokumentu realizuje metoda `linkFinancialDocument($parentDocument, $invoiceData)` [156]. Przyjmuje istniejący dokument magazynowy (`$parentDocument` typu WZ lub PZ) i dane faktury (data, numer, itp.). Tworzy odpowiednio dokument typu FS lub FVZ – rozpoznaje to przez `match` na typie dokumentu źródłowego [87]. Po utworzeniu dokumentu finansowego:

    - Kopiuje on pozycje z dokumentu magazynowego (`items`) do nowego dokumentu finansowego [87] [157].
    - Przenosi też powiązania kontrahenta: dla WZ->FS kopiuje `customer_id`, dla PZ->FVZ kopiuje `supplier_id` [158].
    - Tworzy dokument przez `createBaseDocument` z wyliczonym typem (FS lub FVZ) i sumami netto/brutto identycznymi jak w dokumencie magazynowym [87] [159].
    - Zwraca utworzony dokument finansowy. Ta operacja **nie wpływa na stany magazynowe** (to tylko dokument księgowy) [160], ale wiąże dokumenty przez pole `related_document_id` (magazynowy <-> finansowy).

- **Inwentaryzacja (INW)** – dokument INW służy do przeprowadzenia inwentaryzacji magazynu. Tworzenie i przetwarzanie inwentaryzacji odbywa się w metodzie `processInventory($data)` [161]. Wymagany jest `warehouse_id` (magazyn, w którym przeprowadzono spis z natury) [162]. Działanie:

- Tworzy dokument typu INW poprzez `createBaseDocument` (z flagą, że nie tworzymy pozycji przez standardowy mechanizm, bo będą specjalne `inventoryItems`) [163].
- Iteruje przez przekazane pozycje inwentaryzacyjne (`$data['items']`, gdzie dla każdego wariantu podano policzoną ilość `counted_quantity`). Dla każdej pozycji:
    - Pobiera lub tworzy (`StockLevel::firstOrNew`) bieżący stan magazynowy dla danego wariantu i magazynu [164].
    - Odczytuje `expectedQuantity` (stan przed inwentaryzacją, czyli bieżący `quantity` z tabeli stock_levels) [165].
    - Tworzy pozycję inwentaryzacji w dokumencie: `inventoryItems()->create([...])` z zapisem oczekiwanej i policzonej ilości oraz różnicy [166].
    - Wylicza różnicę = `counted_quantity - expectedQuantity`. Jeśli różnica nie jest zero, to wykonuje korektę stanu magazynowego: `StockLevel::change($variant, $warehouse, $difference)` – tzn. jeśli policzono mniej niż ewidencja (różnica

ujemna), odpisze brak z magazynu, a jeśli policzono nadwyżkę (różnica dodatnia), wprowadzi dodatkową ilość na stan [167].

- Po przeiterowaniu wszystkich pozycji, dokument INW zostaje zapisany, a wraz z nim pozycje inwentaryzacyjne. Zwracany jest dokument załadowany z relacją `inventoryItems` (wraz z powiązanymi wariantami dla podglądu wyników inwentaryzacji) [168].

- W efekcie inwentaryzacja koryguje stany: różnice między stanem księgowym a faktycznym zostają automatycznie naniesione na magazyn poprzez odpowiednie PZ lub RW (ukryte w tej operacji). Po dok. INW stany magazynowe wszystkich spisanych produktów odpowiadają stanom rzeczywistym.

- **Anulowanie/Usuwanie dokumentów**: Choć nie ma dedykowanej metody w DocumentService (w TODO jest plan `cancelDocument`), koncepcyjnie anulowanie dokumentu powinno odwrócić jego skutki. Np. anulowanie PZ – zmniejszyć stan o przyjętą ilość; anulowanie WZ – zwiększyć stan z powrotem; anulowanie MM – dodać z powrotem do magazynu źródłowego i odjąć z docelowego; anulowanie RW/PW – odpowiednio odwrócić zmiany. W aktualnej implementacji `DocumentController@destroy` te czynności nie są zaimplementowane (jest tylko prosty delete) [169], ale plan zakłada wykorzystanie transakcji i operacji `StockLevel::change` by wycofać zmiany. Użytkownik otrzyma komunikat, jeśli nie można usunąć dokumentu (np. bo już powiązany z innym, albo została wystawiona faktura).

Każda z powyższych operacji jest opakowana w transakcję bazodanową (`DB::transaction`), co zapewnia atomowość – np. utworzenie dokumentu PZ i zwiększenie kilku stanów magazynowych musi się wykonać w całości lub zostać wycofane w razie błędu [119] [170]. W metodach DocumentService widać rzucanie wyjątków w sytuacjach niepoprawnych, np. próba zejścia stanu poniżej zera – co jest łapane przez transakcję i powoduje wycofanie zmian [171].

Numery dokumentów są generowane automatycznie w metodzie `generateDocumentNumber(DocumentType $type)` [172]. Schemat jest ustawiony jako: `<kolejny_nr>/<miesiąc>/<rok>/<typ>`, gdzie `<kolejny_nr>` jest liczony w obrębie miesiąca i typu dokumentu (np. pierwszy WZ w marcu 2025 może dostać numer `1/03/2025/WZ`, kolejny `2/03/2025/WZ`) [173]. Implementacja wyszukuje ostatni dokument danego typu w bieżącym miesiącu i zwiększa licznik [174]. Numer jest przechowywany w polu `number` i wymuszona jest jego unikalność na poziomie bazy [35], dzięki czemu nie będzie duplikatów numeracji.

Reasumując, system ERP obsługuje pełny cykl operacji magazynowych: od przyjęcia towaru, przez wewnętrzne przesunięcia i wydania, po okresowe inwentaryzacje, a wszystko to za pomocą dokumentów i automatycznej aktualizacji stanów magazynowych. Główne operacje (PZ, WZ, itp.) są dostępne poprzez wywołania API (dodanie dokumentu odpowiedniego typu z listą pozycji), a logika zapewnia spójność danych magazynowych.

Backend – zarządzanie stanami magazynowymi

Zarządzanie stanami magazynowymi odbywa się dwupoziomowo: 1. **Agregatowy stan na poziomie wariantu i magazynu** – tabela `stock_levels` przechowuje bieżący stan ilościowy każdego wariantu w każdym magazynie. 2. **Szczegółowy stan partii (FIFO)** – tabela `stock_batches` przechowuje szczegóły kolejnych dostaw (partii) towaru dla wariantu, co umożliwia wydawanie wg kolejności dostaw.

Aktualizacja stanów magazynowych jest zawsze wywoływana explicite poprzez metody statyczne modelu `StockLevel`. Gwarantuje to, że każda zmiana przejdzie przez jednolite sprawdzenia

biznesowe: - `StockLevel::change(ProductVariant $variant, Warehouse $warehouse, float $quantity, string $location = null)` – uniwersalna metoda do zmiany *fizycznej* ilości towaru w magazynie [175]. Jeśli dla danego wariantu i magazynu nie istnieje rekord w `stock_levels`, zostanie utworzony (`firstOrCreate`) z domyślnymi zerowymi ilościami [176]. Następnie: - Jeśli podano nową lokalizację `$location` i różni się od już zapisanej, zostanie zaktualizowana (umożliwia to przemieszczenie towaru w obrębie magazynu, np. zmiana półki) [177]. - Obliczana jest nowa ilość: `newQuantity = current_quantity + quantity` [178]. Jeżeli wynik byłby ujemny (np. próba wydania więcej niż jest na stanie), rzucany jest wyjątek (co anuluje transakcję) z informacją, że stan nie może być ujemny [171]. - Następnie aktualizowany jest stan (`quantity = newQuantity`) i zapis do bazy [179]. Metoda zwraca obiekt `StockLevel` z odświeżonym stanem. - Przykład użycia: przy tworzeniu WZ system wywołuje `StockLevel::change(..., -$item['quantity'])` dla każdej pozycji [139] – zmniejszając stan. Przy PZ – analogicznie z wartością dodatnią [136].

- `StockLevel::changeReservation(ProductVariant $variant, Warehouse $warehouse, float $quantity)` – metoda do zmiany *rezerwacji* stanu [180]. Działa podobnie jak powyżej: tworzy rekord stanu jeśli nie istnieje, a następnie zmienia pole `reserved_quantity` o zadaną wartość:
- Jeśli próbujemy zwiększyć rezerwację powyżej dostępnej ilości (tj. quantity > (stan obecny - już zarezerwowane)), rzucany jest wyjątek "Niewystarczający stan dostępny..." [181].
- Jeśli nowa wartość rezerwacji byłaby ujemna (próba zwolnienia więcej niż zarezerwowano), także wyjątek [182].
- W przeciwnym razie aktualizuje `reserved_quantity` i zapisuje rekord [183].

- Rezerwacje są używane np. przy tworzeniu zamówienia sprzedaży – gdy zamówienie jest zintegrowane, system może zarezerwować towar na poczet realizacji zamówienia (aby nie został wydany w innym dokumencie). Zwolnienie rezerwacji nastąpi np. przy realizacji (WZ zmniejszy stan a rezerwację) lub anulacji zamówienia (zdejmie rezerwację).

- `StockLevel::changeIncoming(ProductVariant $variant, Warehouse $warehouse, float $quantity)` – metoda do zmiany *oczekiwanego* stanu (incoming) [184]. Również tworzy rekord jeśli brak, a następnie zmienia pole `incoming_quantity`:

- Nie pozwala, by incoming_quantity spadło poniżej zera (próba odjęcia więcej niż zapisane powoduje wyjątek) [185].
- Służy to np. do odnotowania, że zamówiono towar u dostawcy: wtedy zwiększamy incoming. Gdy towar dotrze i zostanie zarejestrowany PZ, incoming będzie zmniejszone (bo przerzucone do właściwego stanu).
- W praktyce, integracja z BaseLinker może korzystać z tego pola – jeśli moduł zamówień od dostawców będzie dodany, można zaznaczać oczekiwane dostawy.

Dostępne stany obliczeniowe: Model `StockLevel` udostępnia dynamiczne pola (accessors): - `available_quantity` – wolna dostępna ilość = stan fizyczny minus rezerwacje [30]. - `expected_quantity` – stan spodziewany = stan fizyczny + w dostawie [186]. Te wartości nie są przechowywane, ale obliczane w locie, co jest przydatne np. w interfejsie (dostępne do sprzedaży, czy ile będzie po przyjęciu oczekiwanych dostaw).

FIFO vs LIFO: System domyślnie traktuje pole `quantity` w `stock_levels` jako sumaryczny stan bez rozróżnienia partii. Jeśli firma nie potrzebuje śledzić partii, może operować tylko na tych sumach. Jednak zaimplementowano szczegółowe śledzenie partii poprzez `StockBatch` i dedykowane endpointy FIFO: - Przy przyjęciu dostawy, oprócz zwiększenia `StockLevel`, tworzona jest nowa partia

`StockBatch` z dokładną ilością i ceną [34]. - Przy wydaniu, wykorzystywane są najstarsze partie (po `purchase_date` rosnąco) – ich `quantity_available` jest redukowane kolejno [129] [130]. Dzięki temu wiadomo, która partia (data zakupu, cena) jest jeszcze na stanie i ile zostało. - **Uwaga:** Aktualnie, mechanizm FIFO jest wywoływany niezależnie od tworzenia dokumentów WZ/PZ. Oznacza to, że np. utworzenie dokumentu WZ **nie** automatycznie uszczupla konkretnych partii – WZ tylko zmniejsza `StockLevel` (sumaryczny stan). Jeśli firma chce prowadzić FIFO, powinna operację wydania wykonywać przez endpoint FIFO (`stock/fifo/out`), który pod spodem również zmniejszy `StockLevel`. Ewentualnie integracja między DocumentService a StockBatch mogłaby zostać dodana (by przy WZ również redukować partie), ale z kodu wynika, że to rozdzielono: DocumentService dba o sumy, a osobne operacje FIFO o partie.

Lokalizacje magazynowe: Każdy rekord `StockLevel` ma pole `location` pozwalające przechować informację o miejscu składowania danego wariantu (np. sektor, półka). Jeśli przy zmianie stanu przekażemy parametr `location`, metoda `change()` zaktualizuje to pole [177]. Domyślnie, jeśli rekord tworzony jest pierwszy raz, przypisuje lokalizację jako `<SKU>-<MAGAZYN>` [187]. Dzięki temu magazynier może potem edytować to pole (np. "A1-3-5" oznaczające regał A1, półka 3, miejsce 5). W integracji z inwentaryzacją, te lokalizacje mogą być użyteczne przy generowaniu arkuszy spisowych.

Podsumowując, system zapewnia **spójność stanów magazynowych** poprzez centralne metody `StockLevel`. Każda transakcja magazynowa wywołuje odpowiednie zmiany stanów. Mechanizmy kontrolne (wyjątki) chronią przed zejściem stanu poniżej zera lub rezerwowaniem towaru, którego nie ma [182]. Dodatkowo, SoftDeletes na kluczowych encjach (Product, Variant, Warehouse) powodują, że rekordy powiązane (stock_levels, stock_batches) nie są usuwane od razu – co zapobiega utracie historii stanów. Inwentaryzacja umożliwia okresowe korekty, a integracja z BaseLinker (poprzez BaselinekrService, np. mapowanie SKU z zamówień na warianty i synchronizację zamówień) dba o to, by stany rezerwowane i dostępne były aktualne także względem zamówień zewnętrznych (pobierane zamówienia mogą automatycznie tworzyć rezerwacje).

Backend – autoryzacja, role i uprawnienia użytkowników

System posiada podstawowy mechanizm uwierzytelniania oparty o Laravel Sanctum – wykorzystywane są **tokeny API** przypisane do użytkowników. Przy rejestracji lub logowaniu generowany jest token (Personal Access Token) zwracany aplikacji klienckiej [99] [101], który następnie jest przesyłany w kolejnych żądaniach. Sanctum weryfikuje token i identyfikuje użytkownika (middleware `auth:sanctum`) [97].

Role użytkowników: Każdy użytkownik ma przypisaną rolę zapisaną w polu `role` w tabeli `users`. Role są zdefiniowane jako proste stringi – w kodzie występują: `"admin"`, `"manager"` oraz domyślna `"user"` [188] [189]. Nowo zarejestrowani użytkownicy dostają rolę `"user"` domyślnie [188]. Struktura bazy została rozszerzona o pole `role` poprzez migrację (domyślna wartość 'user') [190]. Nie ma osobnej tabeli ról czy uprawnień – zastosowano najprostsze podejście polegające na przechowywaniu roli w rekordzie użytkownika.

Uprawnienia (abilities): System wykorzystuje bibliotekę *CASL* (Conditional Access Control) po stronie front-end do kontrolowania dostępu do widoków i akcji. Backend wspiera to, zwracając w danych użytkownika pole `ability` (lista uprawnień) zdefiniowane zależnie od roli [189]. W metodzie `AuthController::formatUserData($user)` widać zmapowanie ról na tzw. zdolności: - Dla roli `admin`: `ability` zawiera `[ { action: 'manage', subject: 'all' } ]` – co oznacza pełny dostęp do wszystkiego [191]. - Dla roli `manager`: `ability` zawiera uprawnienia do odczytu całości (`{ action: 'read', subject: 'all' }`) oraz do zarządzania (create/edit) tylko obiektami typu

`Product` (`{ action: 'manage', subject: 'Product' }`) [192]. Można dopisać tu więcej uprawnień, np. manage Orders itp., jeśli potrzebne. - Dla domyślnego `user` (lub innych nieznanych): `ability` zawiera jedynie uprawnienie odczytu dashboardu (`{ action: 'read', subject: 'Dashboard' }`) [193].

Te informacje są wykorzystywane na froncie do dynamicznego sterowania np. widocznością opcji w menu (porównując wymagane uprawnienie z posiadanym). Przykładowo, w konfiguracji nawigacji frontu widać, że sekcja "Magazyn" i większość elementów wymaga `action: 'manage', subject: 'all'` (czyli tylko admin zobaczy) [194] [195], natomiast zwykły użytkownik może mieć dostęp tylko do niektórych podstron (np. Pulpit). Rolę można zmienić przy edycji użytkownika – jest wspomniane, że w UI jest widok zarządzania rolami użytkowników. Prawdopodobnie tylko administrator może zmienić czyjąś rolę (to powinno być wymuszone także na backendzie – np. w `UserController@update` sprawdzić czy aktualny user jest adminem). Takie warunki są do dopracowania (w TODO zaznaczono potrzebę walidacji uprawnień w UserController) [115].

Autoryzacja żądań: Po stronie backendu nie zaimplementowano rozbudowanego mechanizmu Gate/Policy dla ról – póki co opiera się to na zaufaniu, że front-end nie wyśle nieuprawnionego żądania, lub że w razie czego backend zwróci 403 jeśli coś wykryje. W praktyce: - Middleware sanctum zapewnia, że użytkownik jest zalogowany. - Dalsze ograniczenia (np. tylko admin może tworzyć użytkowników, tylko manager lub admin może dodawać produkty) wymagają albo dedykowanych Policy, albo sprawdzeń w kontrolerach. W aktualnym kodzie brak wyraźnych sprawdzeń, co oznacza, że potrzebne są dopełnienia (np. *policy* dla modelu User, Product itp.) lub wykorzystanie mechanizmu Gate. Możliwe, że w planach jest użycie przypisanych ability również w backendzie – do tego Laravel udostępnia Gate definiowany np. w AuthServiceProvider. Na razie jednak jest to kontrolowane na poziomie UI.

Bezpieczeństwo operacji w API: Tam gdzie istnieje ryzyko, pewne walidacje już dodano: - Np. próba usunięcia kategorii używanej przez produkty jest blokowana [17] niezależnie od roli, z komunikatem. - Próba usunięcia magazynu, który jest oznaczony jako domyślny lub używany, ma być blokowana (to w planach – do zaimplementowania). - Próba usunięcia producenta/dostawcy używanego – również w planach (walidacja w kontrolerach). - Przy operacjach magazynowych, jak wspomniano, walidacja jest na poziomie stanów (nie da się zdjąć za dużo ze stanu – system rzuci wyjątek i zwróci błąd 422) [182]. - Rejestracja i logowanie walidują dane wejściowe (np. unikalność email, minimalna długość hasła, zgodność potwierdzenia) [196] [197].

Podsumowanie autoryzacji: Model uprawnień jest uproszczony, co jednak ułatwia integrację z modelami AI lub zewnętrznymi – każdemu użytkownikowi można przypisać jedną z trzech ról. Administrator ma pełnię władzy, manager ma ograniczoną edycję (np. produktów), zwykły użytkownik ma tylko odczyt pewnych danych. Wszystkie akcje API wymagają bycia zalogowanym (autentykacja tokenem). Na interfejsie dzięki CASL nieprawidłowe akcje są ukryte lub zablokowane, natomiast na backendzie należy założyć rozszerzenie o dodatkowe sprawdzanie w przyszłości. Mimo uproszczeń, mechanizm ten jest wystarczający w kontekście małego zespołu czy firmy, gdzie admin przydziela role i ufa się pracownikom co do zakresu działań.

Frontend – technologie i architektura

Frontend aplikacji został zbudowany jako **jednostronicowa aplikacja webowa (SPA)** z wykorzystaniem **Vue.js 3** (z Composition API i TypeScript). W projekcie użyto scaffoldu Vue 3 + Vite – o czym świadczy domyślna treść README.md i konfiguracje (pliki `vite.config.ts`, `env.d.ts` itp.) [198] [199]. Technologie użyte na frontendzie to m.in.: - **Vue 3** – główny framework UI. Pozwala tworzyć dynamiczne komponenty i reagować na stan aplikacji. - **TypeScript** – kod frontendu jest pisany w TS (rozszerzenia

`.ts` i `.vue` z `<script lang="ts">`). To zapewnia większą niezawodność dzięki statycznemu typowaniu. - **Vue Router** – aplikacja korzysta z routera do obsługi wielu podstron (ścieżek) bez przeładowania strony. Wskazuje na to struktura plików w `resources/ts/pages` oraz definicje tras w `typed-router.d.ts` i użycie komponentów `<RouterView>`. Nawigacja zdefiniowana jest w pliku `navigation/vertical/index.ts` [200], gdzie każda pozycja menu ma przypisany `to: { path: '...' }` – co odpowiada konkretnym komponentom stron. W projekcie jest także plik `generate_vuexy_pages.py`, co sugeruje wykorzystanie gotowego szablonu **Vuexy** (popularny motyw admin w Vue) – zapewne do szybkiego wygenerowania szkieletów stron. - **Pinia (lub Vuex)** – do zarządzania stanem. Ponieważ projekt jest w Vue 3, najpewniej zastosowano Pinia jako store (z folderu `resources/ts/stores` wynika istnienie np. `selectOptionsStore.ts` [201]). Pinia ułatwia przechowywanie stanu globalnego, np. informacji o zalogowanym użytkowniku i jego uprawnieniach, tokenu, czy danych słownikowych pobranych z API (jak select options). - **CASL** – biblioteka kontroli uprawnień na froncie. Jej obecność wynika z tego, że backend zwraca strukturę `ability` w danych użytkownika, oraz w kodzie frontendu (nawigacja) elementy menu mają pola `action` i `subject` [194] zgodne z terminologią CASL. Zapewne zdefiniowano *ability* w momencie logowania użytkownika i używane są komponenty <Can> lub podobne do warunkowego renderowania elementów interfejsu w zależności od uprawnień. - **UI Library**: Sądząc po nazwach ikon (np. `'tabler-*'`) użytych w menu [195], użyto biblioteki ikon Tabler Icons i być może komponentów UI z motywu Vuexy (Vuexy opiera się o framework BootstrapVue lub własne komponenty + stylowanie CSS/SCSS). Możliwe jest także użycie Tailwind CSS, jednak nie widać bezpośrednich oznak; natomiast styl motywu raczej narzuca gotowy styl Vuexy. - **Axios** – do komunikacji HTTP z backendem. Choć kod nie jest przytoczony, standardem w Vue projektach jest korzystanie z axios do wykonywania requestów do API. Prawdopodobnie skonfigurowano instancję axios z domyślnym URL bazowym `/api` i z automatycznym dołączaniem tokenu (np. z localStorage) w nagłówkach.

Architektura aplikacji frontendu: - Aplikacja jest modułowa – każdy dział (produkty, kategorie, magazyn, dokumenty, zamówienia, itp.) ma własne podstrony widoczne w menu. Menu nawigacji zostało zdefiniowane w sposób deklaratywny w `verticalNavItems`. Przykładowo, sekcja "Magazyn" grupuje elementy: *Produkty*, *Kategorie produktów*, *Dokumenty magazynowe*, *Zamówienia*, *Dostawcy*, *Magazyny*, *Koszty*, *Użytkownicy* itp. (w pliku nawigacji widać te sekcje enumerowane z headingami i ikonami) [202] [203]. Każdy z nich ma listę dzieci – np. *Produkty* ma dzieci "Lista produktów", "Szczegóły produktu", "Edycja produktu", "Dodaj produkt", "Import/Eksport", "Zdjęcia/Media", "Historia zmian", "Masowe akcje" [195] [204]. Oznacza to, że: - Jest strona listy produktów (tabela z produktami, filtrowanie, akcje). - Strona widoku szczegółów (pewnie pod URL `/products/view/:id` – do wyświetlania danych produktu). - Strona edycji produktu (`/products/edit/:id`) z formularzem. - Strona dodawania nowego (`/products/add`). - Strona importu/eksportu (możliwe moduł do masowego importu CSV). - Strona zarządzania zdjęciami produktu. - Strona historii zmian produktu (być może log zmian magazynowych i cen). - Strona operacji masowych (hurtowe zmiany cen, kategorii etc.). Można zauważyć, że część z tych stron może być w fazie szkieletowej (placeholders wygenerowane skryptem – wiele z nich może nie mieć pełnej logiki jeszcze, co pokrywa się z wpisami w TODO, że trzeba je zaimplementować) [205]. - Podobny układ jest dla *Kategorie produktów* (lista, dodaj/edytuj, drzewo kategorii) [206], *Dokumenty magazynowe* (zapewne lista dokumentów, podgląd dokumentu, tworzenie PZ/WZ etc.), *Zamówienia*, *Dostawcy*, *Magazyny*, *Koszty*, *Stawki VAT*, *Producenci*, *Użytkownicy* – wszystkie te moduły pojawiają się w pliku nawigacji pionowej. Dzięki temu menu, użytkownik porusza się po aplikacji, a router przełącza widoki bez przeładowania strony.

- **Komunikacja z backendem**: Gdy komponenty stron się ładują, wykonują żądania do odpowiednich endpointów API, aby pobrać potrzebne dane. Na przykład:
- Strona listy produktów przy montowaniu wywoła `GET /api/v1/products` (z parametrami filtra/paginacji) i wypełni tabelę danymi produktów.

- Formularz dodawania produktu może wywołać `GET /api/v1/select-options/categories` i inne select options, by wypełnić listy wyboru (kategoria, producent, dostawca itp.) zanim użytkownik wprowadzi dane [207].
- Podczas zapisu nowego produktu, komponent wykona `POST /api/v1/products` przekazując dane z formularza, a po sukcesie może przekierować do listy lub do strony szczegółów.
- Podobnie moduł dokumentów: strona tworzenia dokumentu WZ może dać do wyboru magazyn źródłowy (pobiera listę magazynów), listę produktów (może używać endpointu `select-options/product-variants` do wyszukiwania SKU wariantów) [207], a po wypełnieniu pozycji i zatwierdzeniu wykonuje `POST /api/v1/documents` z odpowiednim typem i listą pozycji. Po otrzymaniu sukcesu (201 Created) może przekierować użytkownika na wydruk lub listę dokumentów.
- Mechanizm autoryzacji: po logowaniu front-end zapisuje otrzymany `accessToken` (np. w `localStorage` lub `sessionStorage`). Następnie globalnie konfiguruje axios, by do każdego żądania do API dołączał nagłówek `Authorization: Bearer <token>`. Dzięki temu, gdy router nawigując wchodzi w sekcje chronione, żądania są już autoryzowane.
- Przy starcie aplikacji (np. w komponencie głównym lub store), jeśli jest zachowany token, front może wywołać `GET /api/auth/user` by zweryfikować token i pobrać świeże informacje o użytkowniku (np. aby ustawić jego rolę i ability w stanie aplikacji, co warunkuje co widzi w menu).

- Wszelkie błędy 401 (nieautoryzowane) na globalnym interceptorze axios mogą powodować przekierowanie na ekran logowania – zapewniając, że po wygaśnięciu tokenu użytkownik się zaloguje ponownie.

- **Komponenty i układ**: Z wpisów w repository wynika użycie gotowych komponentów z szablonu Vuexy:

- Pliki w `resources/ts/@core/components` wskazują na komponenty layoutu (np. `AppBarSearch.vue` itp.).
- Navbar, sidebar itp. prawdopodobnie są zaimplementowane zgodnie z dokumentacją Vuexy.
- Stylizacja – Vuexy prawdopodobnie dostarcza zestaw styli (możliwe SCSS), dlatego w projekcie mogą być pliki stylów globalnych.
- Responsywność i UX: Vuexy jest dostosowany do paneli administracyjnych, więc mamy układ z menu bocznym (vertical nav) i główną zawartością, co pokrywa potrzebę ergonomii pracy biurowej.

Stan implementacji frontendu: Należy zauważyć, że nie wszystkie części frontendu mogą być w pełni ukończone. Plik TODO wymienia wiele punktów dotyczących frontendu, np.: - Stworzenie brakujących widoków dla każdej ścieżki zdefiniowanej w nawigacji (skrypt generujący strony utworzył tylko puste placeholders) [208]. - Implementacja logiki wyświetlania, dodawania, edycji, usuwania dla wszystkich modułów (Produkty, Kategorie, Dokumenty, Zamówienia, Dostawcy, Kontrahenci, Użytkownicy, Magazyny, Koszty, Stawki VAT, Producenci) [209]. To oznacza dopisanie kodu w komponentach, który korzysta z wyżej opisanych endpointów. - Dodanie obsługi filtrowania i sortowania na listach w UI zgodnie z możliwościami API (np. lista produktów powinna umożliwiać filtrowanie po nazwie, statusie itp. – skoro API to oferuje) [210]. - Implementacja formularzy z walidacją po stronie klienta, odpowiadającą regułom z backendowych FormRequest (np. unikalność SKU wariantu podczas edycji produktu – już na froncie można to sprawdzać, czy pola wymagane nie są puste, format email itp.) [211]. - Zarządzanie zdjęciami produktów – tj. interfejs do dodawania/ usuwania zdjęć oraz zmiany ich kolejności (wykorzystujący wspomniane endpointy media) [212]. - Interfejs do zarządzania zestawami produktów (bundle items) – możliwość wybrania komponentów produktu-zestawu, ustawienia ich ilości w zestawie itp. [213]. - Interfejs do zarządzania linkami produktów (powiązania, np. powiązane produkty) [213]. - Dodanie obsługi tagów na UI (przypisywanie tagów do produktu, lista tagów dostępnych) [214]. -

Rozbudowa UI modułu Inwentaryzacji – zapewne formularz do generowania dokumentu INW (wybór magazynu, wprowadzenie policzonych ilości, wyświetlenie różnic) [215]. - Interfejs do dokumentów finansowych (FS, FVZ) – utworzenie widoku pozwalającego wygenerować fakturę sprzedaży do WZ lub fakturę zakupu do PZ, a także ich przeglądanie [216].

Te zadania są wskazówką, że choć **fundamenty frontendu są położone** (nawigacja, struktura stron, połączenia z API), to wymaga on dopracowania w szczegółach. Niemniej jednak, architektura jest czytelna i rozbudowa będzie polegała głównie na korzystaniu z przygotowanych endpointów i serwisów backendowych.

Frontend – komunikacja z backendem

Komunikacja między frontendem a backendem odbywa się przez **REST API** opisane wcześniej, w formacie JSON. Kluczowe cechy tej komunikacji: - **Autoryzacja tokenem**: Po poprawnym zalogowaniu, frontend otrzymuje token uwierzytelniający (Bearer token) i przechowuje go (np. w localStorage). Następnie przy każdym wywołaniu API do chronionych endpointów dołącza ten token. Dzięki temu backend rozpoznaje użytkownika. Token jest typu *personal access token* Sanctum – długotrwały, aż do wylogowania (lub wygaśnięcia po określonym czasie, jeżeli ustalono). - **Format danych**: Backend zwraca JSON, zwykle zorganizowany jako obiekt zawierający główne pole danych (np. `data` z listą obiektów lub szczegółami obiektu). Wiele endpointów (zwłaszcza listy) zwraca dane zorganizowane przez Laravel jako *Resource Collections*, np.:

```
{
  "data": [
     { "id": 1, "name": "Produkt X", "sku": "...", ... },
     { "id": 2, "name": "Produkt Y", ... },
     ...
  ],
  "links": { ...paginacja... },
  "meta": { ...paginacja... }
}
```

Frontend musi te dane odpowiednio obsłużyć – np. w komponentach tabel odczytywać `data` i wyświetlać wiersze, uwzględniając meta do np. pokazania numeru strony. - **Walidacja i błędy**: Jeśli backend zwróci błąd walidacji (kod 422) z informacjami o polach, frontend powinien je obsłużyć – np. wyświetlić komunikaty przy odpowiednich polach formularza. W planach jest "spójna i szczegółowa obsługa błędów API" [217], co oznacza, że backend będzie zwracał czytelne komunikaty (już teraz np. błąd usunięcia kategorii zwraca `message: "Nie można usunąć kategorii..."` [17]). Frontend zapewne posiada globalny interceptor do wyłapywania błędów 401 (niezalogowany) i przekierowania do logowania, oraz może wyłapywać błędy 422 by zasygnalizować użytkownikowi (np. podświetlić pola). - **Realtime**: Obecnie komunikacja odbywa się wyłącznie żądanie-odpowiedź. Nie zaimplementowano mechanizmów real-time (np. WebSocket) – i raczej nie są potrzebne w tym kontekście. - **Integracja z BaseLinker**: Aplikacja posiada serwis `BaselinkerService` po stronie backend, który prawdopodobnie jest wywoływany z Cron lub manualnie z poziomu panelu (np. przycisk "Synchronizuj zamówienia"). Jeśli taka akcja jest dostępna w UI, to np. kliknięcie "Synchronizuj" wywoła endpoint (być może `POST /api/v1/orders/sync` lub podobny – nie widzieliśmy go w routes, więc może wywołanie dzieje się automatycznie w tle). W każdym razie, integracja polega na pobraniu zamówień z BaseLinkera i zapisaniu ich w lokalnej bazie (tabele orders, order_items). Mapowanie pól BaseLinker->nasze tabele jest wspomniane w pliku `sql_mapping.txt` [218]. Frontend może np. okresowo

sprawdzać nowe zamówienia (jeśli by to było wymagane). - **Preload danych słownikowych**: Aby ograniczyć liczbę requestów przy interakcji użytkownika, możliwe że pewne rzeczy są buforowane. Np. listy select (kategorie, magazyny itp.) mogą być pobierane raz i trzymane w store (wspomniany `selectOptionsStore.ts`), zamiast pobierać je za każdym razem na każdej stronie. W kodzie widać dedykowany kontroler `SelectOptionsController` z metodami zwracającymi minimalne zestawy danych [207] – to jest idealne do zbuforowania po zalogowaniu. - **Upload plików**: Wspomniany endpoint `media/upload` przyjmuje prawdopodobnie form-data z plikiem. Frontend używa komponentu do uploadu (np. drag&drop lub przeglądaj) i po wybraniu pliku wysyła go axiosem. Ważne, by do form-data dodać też np. `model_type` i `model_id` (by backend wiedział do czego przypisać plik). Późniejsze operacje (reorder, delete) są już zwykłymi zapytaniami JSON z ID mediów. - **Wydruki / eksporty**: ERP magazynowy może wymagać generowania PDFów dokumentów lub eksportów CSV. Na ten moment nie widać implementacji generowania PDF po stronie backend (np. brak pakietu dompdf itp.), więc być może plan jest użyć frontendu do eksportu – np. generować tabelkę i pozwolić użytkownikowi eksportować do CSV/Excel poprzez biblioteki JS. Jest osobna strona "Import/Eksport" dla produktów [219], prawdopodobnie umożliwi wgranie pliku CSV z produktami lub pobranie CSV istniejących. To dopiero będzie implementowane, ale architektura raczej przewiduje wykonanie uploadu pliku do backendu (który go przetworzy i zaktualizuje produkty) oraz generowanie raportów po stronie backend (np. endpoint typu `GET /api/v1/products/export.csv` zwracający plik).

UX i wydajność: - Dzięki SPA użytkownik ma interfejs reagujący bez przeładowań – np. zapis produktu od razu odświeża listę poprzez manipulację stanu lub ponowne pobranie danych. - Czas odpowiedzi zależy od backendu i bazy, ale zapytania są racjonalnie zbudowane (z indeksami w migracjach dla kluczowych pól, np. indeks na (product_variant_id, warehouse_id) w stock_batches [220], unikalność na ważnych polach jak SKU, email, itp.). Przy rosnącej liczbie rekordów można w razie potrzeby dodać mechanizmy cache (Laravel Cache) dla rzadko zmieniających się słowników, jednak na obecną skalę nie jest to konieczne.

Podsumowanie frontendu: Frontend napisany jest w nowoczesnym stosie (Vue 3 + TS + Vite), co czyni go łatwo rozszerzalnym i zrozumiałym dla programistów. Struktura jest przejrzysta: komponenty stron w katalogu `pages`, współdzielone komponenty (np. formularze, modale) w `components`, globalny layout w `@core`. Komunikacja z backendem jest jasno wydzielona poprzez warstwę API (pewnie znajdują się też funkcje w stylu `apiService` lub bezpośrednie wywołania w metodach komponentów). Dzięki nazwom pól zgodnym z backendem, dane przepływają bez komplikacji. Dokumentacja ta wraz z kodem źródłowym może posłużyć do wytrenowania modeli AI – kluczowe jest, że użyto spójnych nazw (np. Model `Product` ma pole `sku` i w JSON też jest `"sku"`, dokument WZ ma typ `"WZ"` itd.), więc model językowy dysponując tym opisem będzie miał pełny kontekst działania aplikacji i terminologii w niej używanej. Wszystkie powyższe informacje oddają aktualny stan systemu ERP magazynowego **iSanto/erp-warehouse** oraz mogą stanowić podstawę do dalszego rozwoju i integracji.

---

1 2025_05_30_215749_add_pos_code_to_products_table.php
https://github.com/iSanto/erp-warehouse/blob/31754218caf671743679cab353547e0879c668f2/database/migrations/2025_05_30_215749_add_pos_code_to_products_table.php

2 2025_06_04_014127_add_foreign_id_to_products_table.php
https://github.com/iSanto/erp-warehouse/blob/31754218caf671743679cab353547e0879c668f2/database/migrations/2025_06_04_014127_add_foreign_id_to_products_table.php

3 2025_06_04_014938_add_dimensions_to_products_table.php
https://github.com/iSanto/erp-warehouse/blob/31754218caf671743679cab353547e0879c668f2/database/migrations/2025_06_04_014938_add_dimensions_to_products_table.php

4 5 65 67 68 70 71 72 75 76 105 Product.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Models/Product.php

6 7 8 10 11 69 77 78 79 ProductVariant.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Models/ProductVariant.php

9 2025_06_03_175007_add_slug_to_product_variants_table.php
https://github.com/iSanto/erp-warehouse/blob/31754218caf671743679cab353547e0879c668f2/database/migrations/2025_06_03_175007_add_slug_to_product_variants_table.php

12 13 61 80 ProductPrice.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Models/ProductPrice.php

14 2025_05_27_213201_create_categories_table.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/database/migrations/2025_05_27_213201_create_categories_table.php

15 16 73 Category.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Models/Category.php

17 106 107 108 109 110 111 169 CategoryController.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Http/Controllers/Api/V1/CategoryController.php

18 2025_05_27_214550_create_manufacturers_table.php
https://github.com/iSanto/erp-warehouse/blob/31754218caf671743679cab353547e0879c668f2/database/migrations/2025_05_27_214550_create_manufacturers_table.php

19 Manufacturer.php
https://github.com/iSanto/erp-warehouse/blob/31754218caf671743679cab353547e0879c668f2/app/Models/Manufacturer.php

20 2025_05_27_213202_create_suppliers_table.php
https://github.com/iSanto/erp-warehouse/blob/31754218caf671743679cab353547e0879c668f2/database/migrations/2025_05_27_213202_create_suppliers_table.php

21 22 23 91 92 Supplier.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Models/Supplier.php

24 25 26 82 83 Warehouse.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Models/Warehouse.php

27 28 29 30 31 171 175 176 177 178 179 180 181 182 183 184 185 186 187 StockLevel.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Models/StockLevel.php

32 33 81 StockBatch.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Models/StockBatch.php

34 124 125 126 127 128 129 130 131 132 StockFifoController.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Http/Controllers/Api/V1/StockFifoController.php

35 36 37 38 39 2025_05_27_214819_create_documents_table.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/database/migrations/2025_05_27_214819_create_documents_table.php

40 2025_06_03_164433_add_soft_deletes_to_documents_table.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/database/migrations/2025_06_03_164433_add_soft_deletes_to_documents_table.php

41 42 84 85 89 90 Document.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Models/Document.php

43 44 45 DocumentItem.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Models/DocumentItem.php

46 2025_05_27_214822_create_inventory_items_table.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/database/migrations/2025_05_27_214822_create_inventory_items_table.php

47 48 49 InventoryItem.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Models/InventoryItem.php

50 52 53 54 55 2025_05_27_214823_create_orders_table.php
https://github.com/iSanto/erp-warehouse/blob/31754218caf671743679cab353547e0879c668f2/database/migrations/2025_05_27_214823_create_orders_table.php

51 2025_06_01_000005_add_external_fields_to_orders_table.php
https://github.com/iSanto/erp-warehouse/blob/31754218caf671743679cab353547e0879c668f2/database/migrations/2025_06_01_000005_add_external_fields_to_orders_table.php

56 57 58 94 95 Order.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Models/Order.php

59 2025_05_27_214824_create_order_items_table.php
https://github.com/iSanto/erp-warehouse/blob/31754218caf671743679cab353547e0879c668f2/database/migrations/2025_05_27_214824_create_order_items_table.php

60 OrderItem.php
https://github.com/iSanto/erp-warehouse/blob/31754218caf671743679cab353547e0879c668f2/app/Models/OrderItem.php

62 64 2025_05_30_215750_create_tags_and_product_tag_tables.php
https://github.com/iSanto/erp-warehouse/blob/31754218caf671743679cab353547e0879c668f2/database/migrations/2025_05_30_215750_create_tags_and_product_tag_tables.php

63 66 74 Tag.php
https://github.com/iSanto/erp-warehouse/blob/31754218caf671743679cab353547e0879c668f2/app/Models/Tag.php

86 87 88 118 119 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 170 172 173 174 DocumentService.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Services/DocumentService.php

93 Expense.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Models/Expense.php

96 116 117 DocumentController.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Http/Controllers/Api/V1/DocumentController.php

97 121 122 123 201 207 api.php
https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/routes/api.php

98 99 100 101 102 103 188 189 191 192 193 196 197 AuthController.php

https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/app/Http/Controllers/AuthController.php

104 112 113 114 115 120 205 208 209 210 211 212 213 214 215 216 217 TO-DO.txt

https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/TO-DO.txt

190 2025_05_29_221354_add_role_to_users_table.php

https://github.com/iSanto/erp-warehouse/blob/31754218caf671743679cab353547e0879c668f2/database/migrations/2025_05_29_221354_add_role_to_users_table.php

194 195 200 202 203 204 206 219 index.ts

https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/resources/ts/navigation/vertical/index.ts

198 199 README.md

https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/README.md

218 sql_mapping.txt

https://github.com/iSanto/erp-warehouse/blob/d610dd1eb0e5cf3964ce851a4834ca7bbe91dc74/sql_mapping.txt

220 2025_06_07_030457_create_stock_batches_table.php

https://github.com/iSanto/erp-warehouse/blob/31754218caf671743679cab353547e0879c668f2/database/migrations/2025_06_07_030457_create_stock_batches_table.php