**Shopix**

Fejlesztői és bemutató dokumentáció

*Madarász Márk - 2025*

**Általános leírás**

A **Shopix** egy Java alapú, Spring Boot keretrendszerre épülő webalkalmazás, amely egy e-kereskedelmi (webáruház) rendszer háttérrészét valósítja meg. A projekt célja egy moduláris felépítésű, könnyen bővíthető online áruház backend biztosítása, amely kezeli a felhasználók regisztrációját és bejelentkezését, a termékek és kategóriák nyilvántartását, a vásárlói kosár kezelését, a rendelések létrehozását, valamint a rendelések fizetésének feldolgozását. A Shopix RESTful API-kat kínál JSON formátumú válaszokkal, így bármilyen front-end alkalmazás (pl. webes felület vagy mobil kliens) kiszolgálására alkalmas.

**Technológiai stack**

*A Shopix a következő főbb technológiákat és könyvtárakat használja a megvalósításhoz:*

* Java 17
* Spring Boot
* Spring Data JPA
* MySQL
* Lombok
* Spring Security (JWT)
* Jakarta Bean Validation
* Springdoc OpenAPI (Swagger)
* Maven
* JUnit 5

**Modulok és rétegek**

*A Shopix alkalmazás kódja több rétegre és modulra bontva követi a jól bevált többrétegű architektúrát. Minden rétegnek megvan a maga felelőssége, ami növeli a kód olvashatóságát és karbantarthatóságát. Az alábbi fő komponensek különülnek el:*

**Controller réteg:** A *REST API végpontokért* felelős modulok. Minden fontos funkcióhoz tartozik egy controller osztály (pl. ProductController, CartController, OrderController, stb.), melyek a megfelelő URL útvonalakra (@RequestMapping) érkező HTTP kéréseket kezelik. A vezérlők tipikusan @RestController annotációval vannak ellátva, a metódusaik pedig HTTP műveletekhez (GET, POST, stb.) kötöttek. Feladatuk a bejövő adatok (@RequestBody, path variable-ek stb.) fogadása, a **Service réteg** meghívása az üzleti logika végrehajtására, majd a kapott eredmény (általában egy DTO) visszaadása HTTP válaszként.

**Service réteg:** Az üzleti logika rétege. Minden fontosabb funkcionális területhez tartozik egy service osztály (pl. ProductService, OrderService, CartService, stb.), amely összefogja az adott domain logikát. A service réteg tranzakciókezelést is alkalmaz (jellemzően @Transactional annotációval a metódusokon), és itt történik az adatlekérdezések és -módosítások üzleti szabályainak érvényesítése. A service hívja a Repository réteg metódusait az adatok kiolvasására vagy mentésére, és használja a Mapper osztályokat az entitások és DTO-k közti konverzióra.

**Repository réteg:** Az adat-hozzáférési réteg, amely közvetlenül kommunikál az adatbázissal. Minden entitáshoz tartozik egy repository interfész (pl. ProductRepository, UserRepository, stb.), általában a Spring Data JPA JpaRepository kiterjesztéseként. A repository-k segítségével egyszerű CRUD műveletek (create, read, update, delete) végezhetők az egyes entitásokon, valamint szükség esetén egyedi lekérdezések is definiálhatók metódusnév alapján vagy @Query annotációval.

**Model réteg:** A domain modell osztályai, melyek az adatbázisból tárolt információkat reprezentálják objektumként. Ezek a JPA entitások (@Entity) tipikusan a hu.shopix.main.model csomagban találhatók (pl. User, Product, Order stb.).

**DTO-k:** Mivel a belső entitásokat nem célszerű közvetlenül kitenni a kliens felé, a Shopix DTO osztályokat definiál a hu.shopix.main.dto csomagban. A DTO-k olyan egyszerű adattároló osztályok (általában Lombok @Builder, @Getter használatával), amelyek meghatározzák, milyen adatokat adunk át a kliensnek, illetve várunk el tőle.

**Mapper-ek:** A mapper osztályok (hu.shopix.main.mapper csomag) felelősek az entitások és DTO-k közti konvertálásért. A projektben minden fontosabb entitáshoz tartozik egy mapper (pl. ProductMapper, OrderMapper, stb.), amelyek tipikusan @Component annotációval ellátott osztályok, és tartalmaznak metódusokat mint toResponse(Entity) vagy toEntity(Dto).

**Főbb entitások és kapcsolataik**

*A Shopix rendszer domain modellje a következő főbb entitásokból áll:*

**User:** A regisztrált felhasználókat írja le, egyedi e-mail címmel, BCrypt-tel hash-elt jelszóval és szerepkörrel (USER/ADMIN). Egy userhez több cím, kosár és rendelés is tartozhat. A kapcsolatok cascade és orphanRemoval beállítással működnek, így például a címek automatikusan törlődnek a felhasználó törlésekor.

**Product:** A webshop termékeit reprezentálja: név, leírás, ár (BigDecimal), aktív státusz és felvitel ideje. Minden termék egy kategóriához tartozik, és szorosan kapcsolódik az Inventory-hoz (1-1 kapcsolat a készlet miatt). Kosár- és rendelésoldalon a CartItem/OrderItem entitásokon keresztül jelenik meg.

**Category:** A termékek kategóriákba sorolását biztosítja, opcionálisan hierarchikus (parent kategória kapcsolattal). Egy kategóriához több termék tartozhat. A lekérdezés kategória szerint egyszerűen megoldható a repository rétegen keresztül.

**Cart:** Egy felhasználó aktuális vásárlását tartalmazza, amíg nem jön létre rendelés belőle. A kosár státusza nyitott vagy lezárt lehet, és több CartItem kapcsolódik hozzá. A cascade és orphanRemoval biztosítja, hogy a kosár törlésekor a tételek is eltűnjenek.

**CartItem:** Egy kosárban lévő konkrét termék és mennyiség. Tartalmazza a mennyiséget és a termék pillanatnyi árát (snapshot), hogy ármozgás esetén is következetes maradjon. A Cart és Product entitásokhoz kapcsolódik ManyToOne viszonyban.

**Order:** A leadott kosárból jön létre, tartalmazza a végösszeget, státuszt (CREATED, PAID), valamint a rendeléshez tartozó tételeket. A rendelés a userhez tartozik, és opcionálisan tartalmaz fizetést (Payment). A címadatokat snapshotként menti, hogy historikus adatrögzítés biztosított legyen.

**OrderItem:** Egy rendelésben szereplő termék és mennyiség, az adott időpontban érvényes egységárral. Hivatkozik a termékre és a rendelésre. Lényegében a kosár tételeinek „átmásolt” változata a rendeléshez.

**Address:** A felhasználók által mentett szállítási/számlázási címek. Egy felhasználóhoz több cím tartozhat, az Order snapshotként másolja be a kiválasztott cím adatokat. A cascade miatt a user törlésével a címek is törlődnek.

**Payment:** Egy rendelés fizetési állapotát írja le (SUCCESS, FAILED stb.), providerRef azonosítóval és időbélyeggel. Egy rendeléshez pontosan egy Payment tartozhat. A jelen implementáció szimulált fizetést valósít meg (nincs külső szolgáltató integráció).

**Főbb funkciók és szolgáltatások**

*A Shopix alkalmazás a fent említett struktúrára építve számos e-kereskedelmi funkciót valósít meg:*

* Kategóriák listázása *(/categories)*
* Felhasználói regisztráció és bejelentkezés *(/auth/register) és (/login)*
* Kosárkezelés *(/cart)*
* Rendelések *(/orders)*
* Fizetés *(/orders/{id}/pay)*
* Címek kezelése *(/addresses/me)*
* Profil lekérdezése *(/auth/users/me)*

**Biztonság és hitelesítés**

A Shopix alkalmazásban kiemelt szerepet kap a biztonság, hiszen érzékeny műveletek (pl. rendelés leadása, felhasználói adatok kezelése) csak az arra jogosult felhasználók által végezhetők:

A Shopix alkalmazás **Spring Security + JWT alapú, stateless hitelesítést** használ. A bejelentkezéskor generált token minden kérésnél kötelező, a szerver a JwtAuthenticationFilter segítségével ellenőrzi érvényességét, és csak ekkor engedi a védett végpontokhoz való hozzáférést.

Az **/auth**, Swagger és publikus GET végpontok (termékek, kategóriák) mindenki számára elérhetők, míg a kosár, rendelés, fizetés, címek és profil funkciók kizárólag érvényes JWT birtokában használhatók. Bár jelenleg nincs finomhangolt role-kezelés, a rendszer elő van készítve admin jogosultságokhoz.

A jelszavak **BCrypt hash-elve** tárolódnak, a bejelentkezés során a rendszer a hash ellenőrzésével validálja az adatokat. A CurrentUser segédkomponens gondoskodik arról, hogy egy felhasználó csak a saját erőforrásaihoz férjen hozzá.

A modell megfelel a modern REST API biztonsági gyakorlatnak: session nélkül, HTTPS-en keresztül biztonságosan kezeli a regisztrációt, belépést és adatvédelmet, miközben könnyen bővíthető szerepkörök és további szabályok bevezetésével.

**Példák a működésre:**

**-**