ISDStore — Panoramica del backend

Studente: Luca Strano Matricola: 1000069298 Professore: Emiliano Tramontana

Questo documento spiega in dettagli la struttura del codice backend del progetto ISDStore per la materia "Ingegneria dei Sistemi Distribuiti". Il progetto segue un approccio didattico, applicando JWT, RBAC e Redis come cache di sessione e carrello, vari Design Pattern trattati nel corso, insieme ad un'architettura RESTful con Spring Boot.

Indice

Avvio applicazione

- `com.isdstore.ISDStoreApplication` Classe main annotata con `@SpringBootApplication`. È il punto di ingresso che avvia l'intero backend Spring Boot.

Esecuzione locale: il profilo di default usa Postgres e Redis su `localhost` (override via env). Con Docker è possibile buildare l'immagine dal `Dockerfile` in `backend/` e collegarla ai container DB/Redis (compose root fornisce db/redis/pgadmin).

Configurazione e infrastruttura

- `com.isdstore.config.AppConfig`
  - Espone bean riutilizzabili:
    - `PasswordEncoder` (BCrypt) per hash/verify della password.
    - `StringRedisTemplate` per interagire con Redis (cart e token list).
- `src/main/resources/application.yml`
  - Config delle connessioni (PostgreSQL, Redis), JPA/Hibernate, logging e parametri JWT:
    - `app.jwt.secret`, `app.jwt.accessTtlSeconds`, `app.jwt.refreshTtlSeconds`.
  - Variabili override via env: `DB_URL`, `DB_USERNAME`, `DB_PASSWORD`, `REDIS_HOST`, `REDIS_PORT`, `JWT_SECRET`.

Bean comuni (BCrypt + Redis):

```
// AppConfig.java
@Bean PasswordEncoder passwordEncoder() { return new
BCryptPasswordEncoder(); }
@Bean StringRedisTemplate stringRedisTemplate(RedisConnectionFactory f) {
return new StringRedisTemplate(f); }
```

Sicurezza — Reference Monitor (JWT + RBAC)

- `com.isdstore.security.SecurityConfig`
  - Definisce la `SecurityFilterChain`:
    - CORS aperto, CSRF disabilitato, sessioni stateless.
    - Regole di accesso: `/api/auth/**` e `/api/products/**` pubblici; `/api/admin/**` richiede ruolo `ADMIN`; il resto è autenticato tramite validazione JWT.
    - Inserisce il filtro `JwtAuthFilter` prima del `UsernamePasswordAuthenticationFilter`.
- `com.isdstore.security.JwtAuthFilter`
  - Filtro "Once per request" che:
    - Legge `Authorization: Bearer <token>`.
    - Usa `JwtService` per validare/parsing del token.
    - Verifica che il token sia di tipo `access` e presente in allow-list su Redis tramite `TokenService`.
    - Se valido, popola lo `SecurityContext` con principal = `userId` (UUID) e authority `ROLE_<ROLE>`.
- `com.isdstore.security.JwtService`
  - inizializza la chiave HMAC anche se il segreto non è Base64 (fallback su bytes raw + padding SHA-256 a 256 bit).
  - Genera e valida JWT. Claim usati:
    - `sub` = UUID utente, `role` = ruolo, `typ` = `access`/`refresh`.
  - TTL configurabili per access e refresh.
- `com.isdstore.security.TokenService`
  - Gestisce allow-list in Redis dei token emessi:
    - Chiave: `auth:{typ}:{userId}:{token}` con TTL.
    - Operazioni: store, check, invalidate.

Punti chiave implementativi

```
// SecurityConfig.java (regole e filtro)
http
  .cors(Customizer.withDefaults())
  .csrf(csrf -> csrf.disable())
  .sessionManagement(sm ->
sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
  .authorizeHttpRequests(auth -> auth
    .requestMatchers("/api/auth/**").permitAll()
    .requestMatchers("/api/products/**").permitAll()
    .requestMatchers("/api/admin/**").hasRole("ADMIN")
```

```
    .anyRequest().authenticated()
  )
  .addFilterBefore(jwtAuthFilter,
UsernamePasswordAuthenticationFilter.class);
```

```
// JwtAuthFilter.java (estratto core)
String auth = request.getHeader(HttpHeaders.AUTHORIZATION);
if (auth != null && auth.startsWith("Bearer ")) {
  String token = auth.substring(7);
  Claims claims = jwtService.parseToken(token);
  if ("access".equals(claims.get("typ", String.class))) {
    UUID userId = UUID.fromString(claims.getSubject());
    String role = claims.get("role", String.class);
    if (tokenService.isTokenValid("access", userId, token)) {
      var authorities = List.of(new SimpleGrantedAuthority("ROLE_" +
role.toUpperCase()));
      var authToken = new
UsernamePasswordAuthenticationToken(userId.toString(), null, authorities);
      SecurityContextHolder.getContext().setAuthentication(authToken);
    }
  }
}
```

Feature: Autenticazione

- `com.isdstore.auth.AuthController` (Base path: `/api/auth`)
  - `POST /register`
    - Registra un nuovo utente con ruolo `user` (hash password con BCrypt). Usa `UserRepository`, `RoleRepository`.
  - `POST /login`
    - Verifica credenziali, genera `accessToken` e `refreshToken` con `JwtService`.
    - Salva entrambi in Redis via `TokenService` con i rispettivi TTL.
    - Risponde con `AuthResponse`.
- DTO
  - `auth.dto.AuthRequest` — email e password, validati con `jakarta.validation`.
  - `auth.dto.AuthResponse` — payload di risposta con `accessToken` e `refreshToken`.
- Registrazione (`POST /api/auth/register`):
  - Normalizza email a lowercase, verifica unicità, risolve ruolo `user`, salva password hash con BCrypt.
  - Ritorna 200 con stringa "registered"; 400 se email duplicata.

Punti chiave implementativi

```
// AuthController.register (estratto)
Role userRole = roleRepository.findByName("user").orElseThrow();
User user = new User();
user.setEmail(email);
```

```
user.setPasswordHash(passwordEncoder.encode(req.getPassword()));
user.setRole(userRole);
userRepository.save(user);
```

```
// AuthController.login (estratto)
String role = user.getRole().getName();
String access = jwtService.generateAccessToken(user.getId(), role);
String refresh = jwtService.generateRefreshToken(user.getId(), role);
tokenService.storeToken("access", user.getId(), access,
jwtService.getAccessTtlSeconds());
tokenService.storeToken("refresh", user.getId(), refresh,
jwtService.getRefreshTtlSeconds());
return ResponseEntity.ok(new AuthResponse(access, refresh));
```

Feature: Catalogo prodotti (pubblico)

- `com.isdstore.products.ProductController` (Base: `/api/products`)
  - `GET /` — restituisce lista di `ProductDTO`.
  - `GET /{id}` — dettaglio prodotto per UUID.
- `com.isdstore.products.AdminProductController` (Base: `/api/admin/products`, protezione RBAC: necessario ruolo ADMIN)
  - `POST /` — crea prodotto da `ProductDTO` (validazioni basilari).
  - `PUT /{id}` — aggiorna campi presenti nel DTO.
  - `DELETE /{id}` — elimina per UUID.
- Entrambi convertono `Product` → `ProductDTO` per non esporre entity direttamente.

Punti chiave implementativi

```
// ProductController.list (semplificato)
List<Product> products = productRepository.findAll();
return products.stream().map(this::toDto).toList(); //map a ProductDTO
```

```
// AdminProductController.create (validazioni basilari)
if (dto.getTitle()==null || dto.getTitle().isBlank()) return
badRequest("Title is required");
// ... check su description/priceCents/stock ...
Product p = new Product();
p.setTitle(dto.getTitle());
// ... mapping resto campi ...
return ResponseEntity.ok(toDto(productRepository.save(p)));
```

Feature: Carrello (Redis)

- `com.isdstore.cart.CartController` (Base: `/api/cart`, autenticato)

    - Estrae l'`userId` dallo `SecurityContext` (principal = UUID).
    - `GET /` — ritorna `CartViewDTO` (items arricchiti con dettagli prodotto e totali).
    - `POST /items` — aggiunge un item (`productId`, `quantity`) e ritorna la vista aggiornata.
    - `DELETE /items/{productId}` — rimuove l'item e ritorna la vista aggiornata.
- `com.isdstore.cart.CartService`
    - Salva/legge il carrello da Redis come JSON (`cart:{userId}`, TTL 7 giorni).
    - Calcola i totali e costruisce una `CartViewDTO` con `CartViewItemDTO` (include `ProductDTO`).
    - Fornisce `clearCart(userId)` usato dopo il checkout.

Punti chiave implementativi

```
// CartService.persist (serializza su Redis con TTL)
String json = objectMapper.writeValueAsString(items);
redis.opsForValue().set("cart:" + userId, json, Duration.ofDays(7));
```

```
// CartController.getCart (userId dal SecurityContext)
UUID userId = currentUserId();
CartViewDTO view = cartService.getCartView(userId);
return ResponseEntity.ok(view);
```

Feature: Checkout e Ordini

- `com.isdstore.orders.OrderController` (Base: `/api`)
    - `POST /checkout` (autenticato)
        - L'endpoint è `@Transactional`: decrementi stock + creazione ordine avvengono in una singola transazione DB.
        - Legge carrello da Redis via `CartService`.
        - Per ogni item: valida stock, riduce `stock` prodotto, accumula `totalCents`.
        - Serializza gli items confermati come JSON (LOB JSONB su Postgres) dentro `Order.items`.
        - Crea `Order` con `status = completed`, salva su DB, svuota il carrello.
        - Risponde con `OrderDTO` (include una vista degli items con titoli tramite `OrderItemViewDTO`).
    - `GET /orders` (autenticato)
        - Lista ordini dell'utente corrente (ordinati per `createdAt` desc) come `OrderDTO`.
    - `GET /admin/orders` (RBAC: ADMIN)
        - Lista tutti gli ordini di tutti gli utenti come `OrderDTO`.

Punti chiave implementativi

```
// OrderController.checkout (estratto)
CartDTO cart = cartService.getCart(userId);
for (CartItemDTO it : cart.getItems()) {
```

```
    Product p = productRepository.findById(it.getProductId()).orElse(null);
    int qtyToBuy = Math.min(requested, Math.max(0, p.getStock()));
    p.setStock(p.getStock() - qtyToBuy);
    productRepository.save(p);
    totalCents += (p.getPriceCents() * qtyToBuy);
    adjusted.add(new CartItemDTO(pid, qtyToBuy));
  }
  order.setItems(objectMapper.writeValueAsString(adjusted));
  order.setStatus("completed");
  cartService.clearCart(userId);
```

Modello dati — Entity e Repository

Entity (`com.isdstore.common.entity`)

- `User` — utenti; relazionato a `Role` (molti-a-uno). Campi audit `createdAt`, `updatedAt`.
- `Role` — ruoli (`user`, `admin`).
- `Product` — prodotti con `title`, `description`, `priceCents`, `image`, `stock`, `createdAt`.
- `Cart` — rappresentazione persistita del carrello (non essenziale al flusso, ma definita) con `items` JSONB e `updatedAt`.
- `Order` — ordini: `user`, `items` JSONB (serialized LOB), `totalCents`, `status`, `createdAt`.

Repository (`com.isdstore.common.repo`)

- `UserRepository` — CRUD + `findByEmail`.
- `RoleRepository` — CRUD + `findByName`.
- `ProductRepository` — CRUD prodotti.
- `OrderRepository` — CRUD + `findByUserIdOrderByCreatedAtDesc`.
- `CartRepository` — CRUD carrelli.

DTO condivisi (`com.isdstore.common.dto`)

- `ProductDTO` — proiezione sicura di `Product` per API pubbliche.
- Carrello:
  - `CartItemDTO` — item grezzo (productId, quantity).
  - `CartDTO` — lista grezza + `totalCents`.
  - `CartViewItemDTO` — item arricchito con `ProductDTO` e `itemTotalCents`.
  - `CartViewDTO` — lista arricchita + `totalCents`.
- Ordini:
  - `OrderDTO` — id, userId/email, lista `OrderItemViewDTO`, `totalCents`, `status`, `createdAt`.
  - `OrderItemViewDTO` — quantità e titolo del prodotto al momento della visualizzazione.
- `UserDTO` — (presente per estensioni future; non esposto oggi dai controller forniti).

Flusso tipico

1. Registrazione/Login => emissione token + allow-list Redis.
2. Navigazione prodotti (pubblica).
3. Carrello (autenticato) su Redis => aggiungi.

4. Checkout => validazione stock, serializzazione items in `orders.items` (JSONB), reduce stock, ordine completato, svuota carrello.
5. Consultazione ordini personali o (admin) lista completa.

Logging, errori e codici di stato

- Logging consistente con `log.info/warn/error` in punti chiave: login, CRUD, cart, checkout.
- Errori comuni:
  - 400 Bad Request: validazioni DTO (admin create/update prodotto), carrello vuoto, `productId` invalido.
  - 401 Unauthorized: credenziali non valide.
  - 404 Not Found: prodotto inesistente in `GET /api/products/{id}`.
  - 500 Internal Server Error: serializzazione JSON ordine fallita.